The Supplemental Isabelle/HOL Library

October 24, 2026

Contents

1 Implementation of Association Lists **21**
- 1.1 *update* and *updates* . . . 21
- 1.2 *delete* . . . 23
- 1.3 *update-with-aux* and *delete-aux* . . . 24
- 1.4 *restrict* . . . 26
- 1.5 *clearjunk* . . . 27
- 1.6 *map-ran* . . . 28
- 1.7 *merge* . . . 29
- 1.8 *compose* . . . 30
- 1.9 *map-entry* . . . 32
- 1.10 *map-default* . . . 32

2 Axiomatic Declaration of Bounded Natural Functors **33**

3 Generalized Corecursor Sugar (corec and friends) **33**
- 3.1 Coinduction . . . 34

4 A general "while" combinator **36**
- 4.1 *while-option* . . . 37
- 4.2 *while* . . . 38
- 4.3 Termination, *lfp* and *gfp* . . . 38
- 4.4 *T-while-option* . . . 40
- 4.5 *while-Some* and *while-saturate* . . . 41
- 4.6 Reflexive, transitive closure . . . 42

5 The Bourbaki-Witt tower construction for transfinite iteration **43**
- 5.1 Connect with the while combinator for executability on chain-finite lattices. . . . 46

6 Division with modulus centered towards zero. **48**

7 Order on characters **51**

8 A generic phantom type **52**

9 Cardinality of types **52**
9.1 Cardinalities of types . . . 52
9.2 Classes with at least 1 and 2 . . . 53
9.3 A type class for deciding finiteness of types . . . 54
9.4 A type class for computing the cardinality of types . . . 54
9.5 Instantiations for *card-UNIV* . . . 54

10 Code setup for sets with cardinality type information **58**

11 Eliminating pattern matches **60**

12 Lazy types in generated code **61**
12.1 The type *lazy* . . . 61
12.2 Implementation . . . 63

13 Test infrastructure for the code generator **63**
13.1 YXML encoding for *term* . . . 64
13.2 Test engine and drivers . . . 66

14 A combinator to build partial equivalence relations from a predicate and an equivalence relation **66**

15 Formalisation of chain-complete partial orders, continuity and admissibility **67**
15.1 Continuity . . . 70
15.1.1 Theorem collection *cont-intro* . . . 70
15.2 Admissibility . . . 77
15.3 (=) as order . . . 81
15.4 ccpo for products . . . 81
15.5 Complete lattices as ccpo . . . 86
15.6 Parallel fixpoint induction . . . 89

16 Confluence **93**

17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums **96**
17.1 The datatype universe . . . 96
17.2 Freeness: Distinctness of Constructors . . . 98
17.3 Set Constructions . . . 101

18 Bijections between natural numbers and other types **105**
18.1 Type *nat* × *nat* . . . 105
18.2 Type *nat* + *nat* . . . 107
18.3 Type *int* . . . 108
18.4 Type *nat list* . . . 109
18.5 Finite sets of naturals . . . 110
18.5.1 Preliminaries . . . 110
18.5.2 From sets to naturals . . . 110
18.5.3 From naturals to sets . . . 111
18.5.4 Proof of isomorphism . . . 111

19 Encoding (almost) everything into natural numbers **112**
19.1 The class of countable types . . . 112
19.2 Conversion functions . . . 112
19.3 Finite types are countable . . . 112
19.4 Automatically proving countability of old-style datatypes . . . 112
19.5 Automatically proving countability of datatypes . . . 114
19.6 More Countable types . . . 114
19.7 The rationals are countably infinite . . . 114

20 Infinite Sets and Related Concepts **115**
20.1 The set of natural numbers is infinite . . . 115
20.2 The set of integers is also infinite . . . 116
20.3 Infinitely Many and Almost All . . . 117
20.4 Enumeration of an Infinite Set . . . 119
20.5 Properties of *enumerate* on finite sets . . . 121

21 Countable sets **122**
21.1 Predicate for countable sets . . . 122
21.2 Enumerate a countable set . . . 123
21.3 Closure properties of countability . . . 126
21.4 Misc lemmas . . . 129
21.5 Uncountable . . . 130

22 Countable Complete Lattices **131**
22.0.1 Instances of countable complete lattices . . . 136

23 Type of (at Most) Countable Sets **136**
23.1 Cardinal stuff . . . 136
23.2 The type of countable sets . . . 137
23.3 Additional lemmas . . . 144
23.3.1 *cempty* . . . 144
23.3.2 *cinsert* . . . 144
23.3.3 *cimage* . . . 144

23.3.4 bounded quantification . . . 144
23.3.5 *cUnion* . . . 145
23.4 Setup for Lifting/Transfer . . . 145
23.4.1 Relator and predicator properties . . . 145
23.4.2 Transfer rules for the Transfer package . . . 145
23.5 Registration as BNF . . . 146

24 Debugging facilities for code generated towards Isabelle/ML 148

25 Sequence of Properties on Subsequences 148

26 Common discrete functions 150
26.1 Discrete logarithm . . . 150
26.2 Discrete square root . . . 152

27 Pi and Function Sets 154
27.1 Basic Properties of *Pi* . . . 155
27.2 Composition With a Restricted Domain: *compose* . . . 157
27.3 Bounded Abstraction: *restrict* . . . 157
27.4 Bijections Between Sets . . . 158
27.5 Extensionality . . . 159
27.6 Cardinality . . . 160
27.7 Extensional Function Spaces . . . 160
27.7.1 Injective Extensional Function Spaces . . . 163
27.7.2 Misc properties of functions, composition and restriction from HOL Light . . . 164
27.7.3 Cardinality . . . 164
27.8 The pigeonhole principle . . . 165
27.9 Products of sums . . . 165

28 Partitions and Disjoint Sets 165
28.1 Set of Disjoint Sets . . . 165
28.1.1 Family of Disjoint Sets . . . 166
28.2 Construct Disjoint Sequences . . . 168
28.3 Partitions . . . 169
28.4 Constructions of partitions . . . 169
28.5 Finiteness of partitions . . . 170
28.6 Equivalence of partitions and equivalence classes . . . 170
28.7 Refinement of partitions . . . 171
28.8 The coarsest common refinement of a set of partitions . . . 172

29 Type of finite sets defined as a subtype of sets 173
29.1 Definition of the type . . . 173
29.2 Basic operations and type class instantiations . . . 173
29.3 Other operations . . . 176

29.4 Transferred lemmas from Set.thy . . . 179
29.5 Additional lemmas . . . 194
29.5.1 *ffUnion* . . . 194
29.5.2 *fbind* . . . 195
29.5.3 *fsingleton* . . . 195
29.5.4 *femepty* . . . 195
29.5.5 *fset* . . . 195
29.5.6 *ffilter* . . . 196
29.5.7 *fset-of-list* . . . 196
29.5.8 *finsert* . . . 196
29.5.9 *fimage* . . . 196
29.5.10 bounded quantification . . . 197
29.5.11 *fcard* . . . 197
29.5.12 *sorted-list-of-fset* . . . 199
29.5.13 *ffold* . . . 199
29.5.14 (|⊂|) . . . 200
29.5.15 Group operations . . . 200
29.5.16 Semilattice operations . . . 201
29.6 Choice in fsets . . . 203
29.7 Induction and Cases rules for fsets . . . 203
29.8 Lemmas depending on induction . . . 204
29.9 Setup for Lifting/Transfer . . . 204
29.9.1 Relator and predicator properties . . . 204
29.9.2 Transfer rules for the Transfer package . . . 204
29.10 BNF setup . . . 206
29.11 Size setup . . . 207
29.12 Advanced relator customization . . . 207
29.12.1 Countability . . . 208
29.13 Quickcheck setup . . . 208
29.14 Code Generation Setup . . . 210

30 Type of finite maps defined as a subtype of maps 210
30.1 Auxiliary constants and lemmas over *map* . . . 210
30.2 Abstract characterisation . . . 212
30.3 Operations . . . 212
30.4 BNF setup . . . 226
30.5 *size* setup . . . 229
30.6 Additional operations . . . 230
30.7 Additional properties . . . 231
30.8 Lifting/transfer setup . . . 231
30.9 View as datatype . . . 232
30.10 Code setup . . . 232
30.11 Instances . . . 234
30.12 Tests . . . 234

31 Disjoint FSets **235**

32 Lists with elements distinct as canonical example for datatype invariants **236**
32.1 The type of distinct lists . . . 236
32.2 Executable version obeying invariant . . . 238
32.3 Induction principle and case distinction . . . 239
32.4 Functorial structure . . . 239
32.5 Quickcheck generators . . . 239
32.6 BNF instance . . . 240

33 Type of dual ordered lattices **241**
33.1 Pointwise ordering . . . 242
33.2 Binary infimum and supremum . . . 243
33.3 Top and bottom elements . . . 244
33.4 Complement . . . 245
33.5 Complete lattice operations . . . 246

34 Equipollence and Other Relations Connected with Cardinality **247**
34.1 Eqpoll . . . 248
34.2 The strict relation . . . 250
34.3 Mapping by an injection . . . 251
34.4 Inserting elements into sets . . . 252
34.5 Binary sums and unions . . . 252
34.6 Binary Cartesian products . . . 253
34.7 General Unions . . . 254
34.8 General Cartesian products (Pi) . . . 254
34.9 Misc other resultd . . . 255

35 Continuity and iterations **260**
35.1 Continuity for complete lattices . . . 261
35.1.1 Least fixed points in countable complete lattices . . . 263

36 Extended natural numbers (i.e. with infinity) **264**
36.1 Type definition . . . 264
36.2 Constructors and numbers . . . 265
36.3 Addition . . . 267
36.4 Multiplication . . . 267
36.5 Numerals . . . 268
36.6 Subtraction . . . 268
36.7 Ordering . . . 269
36.8 Cancellation simprocs . . . 272
36.9 Well-ordering . . . 273

36.10Complete Lattice . . . 273
36.11Traditional theorem names . . . 274

37 Liminf and Limsup on conditionally complete lattices 274
37.0.1 *Liminf* and *Limsup* . . . 276
37.1 More Limits . . . 280

38 Extended real number line 281
38.1 Definition and basic properties . . . 283
38.1.1 Addition . . . 285
38.1.2 Linear order on *ereal* . . . 287
38.1.3 Multiplication . . . 293
38.1.4 Power . . . 299
38.1.5 Subtraction . . . 299
38.1.6 Division . . . 303
38.2 Complete lattice . . . 307
38.3 Extended real intervals . . . 309
38.4 Topological space . . . 311
38.5 Relation to *enat* . . . 317
38.6 Limits on *ereal* . . . 318
38.6.1 Convergent sequences . . . 320
38.6.2 Sums . . . 324
38.6.3 Continuity . . . 330
38.6.4 liminf and limsup . . . 332
38.6.5 Tests for code generator . . . 335

39 Indicator Function 335

40 The type of non-negative extended real numbers 339
40.1 Defining the extended non-negative reals . . . 341
40.2 Cancellation simprocs . . . 344
40.3 Order with top . . . 345
40.4 Arithmetic . . . 347
40.5 Coercion from *real* to *ennreal* . . . 351
40.6 Coercion from *ennreal* to *real* . . . 355
40.7 Coercion from *enat* to *ennreal* . . . 356
40.8 Topology on *ennreal* . . . 358
40.9 Approximation lemmas . . . 364
40.10*ennreal* theorems . . . 366

41 Logarithm of Natural Numbers 369
41.1 Preliminaries . . . 369
41.2 Floorlog . . . 369
41.3 . . . 371

41.4 Bitlen . . . 373

42 Various algebraic structures combined with a lattice 374
42.1 Positive Part, Negative Part, Absolute Value . . . 375

43 Floating-Point Numbers 379
43.1 Real operations preserving the representation as floating point number . . . 379
43.2 Arithmetic operations on floating point numbers . . . 381
43.3 Quickcheck . . . 383
43.4 Represent floats as unique mantissa and exponent . . . 384
43.5 Compute arithmetic operations . . . 386
43.6 Lemmas for types *real*, *nat*, *int* . . . 387
43.7 Rounding Real Numbers . . . 387
43.8 Rounding Floats . . . 389
43.9 Truncating Real Numbers . . . 390
43.10 Truncating Floats . . . 392
43.11 Approximation of positive rationals . . . 394
43.12 Division . . . 396
43.13 Approximate Addition . . . 397
43.14 Approximate Multiplication . . . 399
43.15 Approximate Power . . . 400
43.16 Lemmas needed by Approximate . . . 402

44 Pointwise instantiation of functions to algebra type classes 406

45 Pointwise instantiation of functions to division 410
45.1 Syntactic with division . . . 410

46 Functions as a real vector spaces 411

47 Lexicographic order on functions 412

48 The *going-to* filter 413

49 Big sum and product over function bodies 415
49.1 Abstract product . . . 415
49.2 Concrete sum . . . 417
49.3 Concrete product . . . 418

50 Infinite Type Class 419

51 Algebraic operations on sets 420

52 Interval Type **426**
52.1 Membership . . . 430
52.2 Quickcheck . . . 437

53 Approximate Operations on Intervals of Floating Point Numbers **439**
53.1 Intervals with Floating Point Bounds . . . 440
53.2 intros for *real-interval* . . . 441
53.3 bounds for lists . . . 442
53.4 constants for code generation . . . 445

54 Immutable Arrays with Code Generation **445**
54.1 Fundamental operations . . . 445
54.2 Generic code equations . . . 446
54.3 Auxiliary operations for code generation . . . 447
54.4 Code Generation for SML . . . 448
54.5 Code Generation for Haskell . . . 448

55 Definition of Landau symbols **449**
55.1 Definition of Landau symbols . . . 449
55.2 Landau symbols and limits . . . 463
55.3 Flatness of real functions . . . 469
55.4 Asymptotic Equivalence . . . 469

56 Values extended by a bottom element **476**
56.1 Values extended by a top element . . . 478
56.2 Values extended by a top and a bottom element . . . 481

57 Infinite Streams **483**
57.1 prepend list to stream . . . 484
57.2 set of streams with elements in some fixed set . . . 484
57.3 nth, take, drop for streams . . . 485
57.4 unary predicates lifted to streams . . . 488
57.5 recurring stream out of a list . . . 488
57.6 iterated application of a function . . . 489
57.7 stream repeating a single element . . . 490
57.8 stream of natural numbers . . . 490
57.9 flatten a stream of lists . . . 490
57.10merge a stream of streams . . . 491
57.11product of two streams . . . 491
57.12interleave two streams . . . 491
57.13zip . . . 492
57.14zip via function . . . 492

58 List prefixes, suffixes, and homeomorphic embedding 493
58.1 Prefix order on lists . . . 493
58.2 Basic properties of prefixes . . . 494
58.3 Prefixes . . . 497
58.4 Longest Common Prefix . . . 498
58.5 Parallel lists . . . 499
58.6 Suffix order on lists . . . 500
58.7 Suffixes . . . 504
58.8 Homeomorphic embedding on lists . . . 505
58.9 Subsequences (special case of homeomorphic embedding) . . . 507
58.10 Appending elements . . . 508
58.11 Relation to standard list operations . . . 509
58.12 Contiguous sublists . . . 509
58.12.1 *sublist* . . . 509
58.12.2 *sublists* . . . 511
58.13 Parametricity . . . 512

59 Linear Temporal Logic on Streams 513

60 Preliminaries 514

61 Linear temporal logic 514

62 Weak vs. strong until (contributed by Michael Foster, University of Sheffield) 524

63 Lists as vectors 525
63.1 + and − . . . 525
63.2 Inner product . . . 527

64 Definitions of Least Upper Bounds and Greatest Lower Bounds 527
64.1 Rules for the Relations ∗<= and <=∗ . . . 528
64.2 Rules about the Operators *leastP*, *ub* and *lub* . . . 528
64.3 Rules about the Operators *greatestP*, *isLb* and *isGlb* . . . 530

65 An abstract view on maps for code generation. 532
65.1 Parametricity transfer rules . . . 532
65.2 Type definition and primitive operations . . . 534
65.3 Functorial structure . . . 535
65.4 Derived operations . . . 535
65.5 Properties . . . 537
65.5.1 *entries*, *ordered-entries*, and *fold* . . . 545
65.6 Code generator setup . . . 548

66 Monad notation for arbitrary types 548

67 Less common functions on lists **549**

68 (Finite) Multisets **556**
68.1 The type of multisets . . . 556
68.2 Representing multisets . . . 557
68.3 Basic operations . . . 559
68.3.1 Conversion to set and membership . . . 559
68.3.2 Union . . . 561
68.3.3 Difference . . . 561
68.3.4 Min and Max . . . 563
68.3.5 Equality of multisets . . . 564
68.3.6 Pointwise ordering induced by count . . . 565
68.3.7 Intersection and bounded union . . . 569
68.3.8 Additional intersection facts . . . 570
68.3.9 Additional bounded union facts . . . 571
68.4 Replicate and repeat operations . . . 572
68.4.1 Simprocs . . . 574
68.4.2 Conditionally complete lattice . . . 575
68.4.3 Filter (with comprehension syntax) . . . 576
68.4.4 Size . . . 578
68.5 Induction and case splits . . . 580
68.5.1 Strong induction and subset induction for multisets . 582
68.6 Least and greatest elements . . . 582
68.7 The fold combinator . . . 582
68.8 Image . . . 583
68.9 Further conversions . . . 586
68.10 More properties of the replicate, repeat, and image operations 591
68.11 Big operators . . . 594
68.12 Multiset as order-ignorant lists . . . 601
68.13 The multiset order . . . 603
68.13.1 Well-foundedness . . . 604
68.13.2 Closure-free presentation . . . 605
68.13.3 Monotonicity . . . 605
68.13.4 The multiset extension is cancellative for multiset union 606
68.13.5 Strict partial-order properties . . . 607
68.13.6 Strict total-order properties . . . 608
68.14 Quasi-executable version of the multiset extension . . . 608
68.14.1 Monotonicity of multiset union . . . 609
68.14.2 Termination proofs with multiset orders . . . 610
68.15 Legacy theorem bindings . . . 611
68.16 Naive implementation using lists . . . 612
68.17 BNF setup . . . 615
68.18 Size setup . . . 617
68.19 Lemmas about Size . . . 617

68.20The set of multisets of a given size 618

69 More Theorems about the Multiset Order 620
69.1 Alternative Characterizations 620
69.1.1 The Dershowitz–Manna Ordering 620
69.1.2 The Huet–Oppen Ordering 620
69.1.3 Monotonicity 621
69.1.4 Properties of Orders 621
69.1.5 Simplifications 625
69.2 Simprocs 626
69.3 Additional facts and instantiations 626

70 Fixed Length Lists 628

71 Non-negative, non-positive integers and reals 630
71.1 Non-positive integers 630
71.2 Non-negative reals 632
71.3 Non-positive reals 633

72 Numeral Syntax for Types 636
72.1 Numeral Types 636
72.2 *num1* 636
72.3 Locales for for modular arithmetic subtypes 638
72.4 Ring class instances 640
72.5 Order instances 641
72.6 Code setup and type classes for code generation 641
72.7 Syntax 644
72.8 Examples 644

73 ω-words 644
73.1 Type declaration and elementary operations 645
73.2 Subsequence, Prefix, and Suffix 645
73.3 Prepending 648
73.4 The limit set of an ω-word 648
73.5 Index sequences and piecewise definitions 651

74 Combinator syntax for generic, open state monads (single-threaded monads) 653
74.1 Motivation 653
74.2 State transformations and combinators 653
74.3 Monad laws 654
74.4 Do-syntax 654

75 Canonical order on option type 655

76 Futures and parallel lists for code generated towards Isabelle/ML 659
76.1 Futures . . . 659
76.2 Parallel lists . . . 660

77 Input syntax for pattern aliases (or "as-patterns" in Haskell) 660
77.1 Definition . . . 661
77.2 Usage . . . 662

78 Periodic Functions 662

79 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view 665
79.1 Preliminary: auxiliary operations for *almost everywhere zero* . 665
79.2 Type definition . . . 668
79.3 Additive structure . . . 670
79.4 Multiplicative structure . . . 671
79.5 Single-point mappings . . . 673
79.6 Integral domains . . . 675
79.7 Mapping order . . . 675
79.8 Fundamental mapping notions . . . 676
79.9 Degree . . . 678
79.10 Inductive structure . . . 678
79.11 Quasi-functorial structure . . . 679
79.12 Canonical dense representation of $nat \Rightarrow_0 'a$ . . . 680
79.13 Canonical sparse representation of $'a \Rightarrow_0 'b$ . . . 681
79.14 Size estimation . . . 683
79.15 Further mapping operations and properties . . . 684
79.16 Free Abelian Groups Over a Type . . . 684

80 Exponentiation by Squaring 688

81 Preorders with explicit equivalence relation 689

82 Additive group operations on product types 690
82.1 Operations . . . 690
82.2 Class instances . . . 691

83 Roots of real quadratics 692

84 Pretty syntax for Quotient operations 694

85 Quotient infrastructure for the set type 694
85.1 Contravariant set map (vimage) and set relator, rules for the Quotient package . . . 695

86 Quotient infrastructure for the product type **696**
86.1 Rules for the Quotient package 696

87 Quotient infrastructure for the option type **698**
87.1 Rules for the Quotient package 698

88 Quotient infrastructure for the list type **699**
88.1 Rules for the Quotient package 700

89 Quotient infrastructure for the sum type **703**
89.1 Rules for the Quotient package 703

90 Quotient types **704**
90.1 Equivalence relations and quotient types 704
90.2 Equality on quotients 706
90.3 Picking representing elements 706

91 Ramsey's Theorem **706**
91.1 Preliminary definitions 707
91.1.1 The n-element subsets of a set A 707
91.1.2 Further properties, involving equipollence 709
91.1.3 Partition predicates 710
91.2 Finite versions of Ramsey's theorem 710
91.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers 711
91.2.2 Trivial cases 711
91.2.3 Ramsey's theorem with TWO colours and arbitrary exponents (hypergraph version) 711
91.2.4 Full Ramsey's theorem with multiple colours and arbitrary exponents 712
91.2.5 Simple graph version 712
91.3 Preliminaries for the infinitary version 712
91.3.1 "Axiom" of Dependent Choice 712
91.3.2 Partition functions 713
91.4 Ramsey's Theorem: Infinitary Version 713
91.5 Disjunctive Well-Foundedness 714

92 Modulo and congruence on the reals **715**

93 Generic reflection and reification **719**

94 Assigning lengths to types by type classes **720**

95 Saturated arithmetic **722**
95.1 The type of saturated naturals . . . 723
95.2 Enumeration . . . 726

96 Set Idioms **727**
96.1 Idioms for being a suitable union/intersection of something . 727
96.2 The "Relative to" operator . . . 731

97 Signed division: negative results rounded towards zero rather than minus infinity. **735**

98 State monad **738**

99 Comparators on linear quasi-orders **742**
99.1 Basic properties . . . 742
99.2 Fundamental comparator combinators . . . 747
99.3 Direct implementations for linear orders on selected types . . 747

100 Stably sorted lists **748**

101 Alternative sorting algorithms **751**
101.1 Quicksort . . . 751
101.2 Mergesort . . . 752
101.3 Lexicographic products . . . 754

102 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming **754**

103 Time functions for various standard library operations **756**
103.1 List . . . 756

104 A table-based implementation of the reflexive transitive closure **759**

105 Binary Tree **760**
105.1 *map-tree* . . . 762
105.2 *size* . . . 762
105.3 *set-tree* . . . 763
105.4 *subtrees* . . . 763
105.5 *height* and *min-height* . . . 763
105.6 *complete* . . . 764
105.7 *acomplete* . . . 765
105.8 *wbalanced* . . . 765
105.9 *ipl* . . . 765

105.10 List of entries . . . 765
105.11 Binary Search Tree . . . 766
105.12 *heap* . . . 767
105.13 *mirror* . . . 767

106 Multiset of Elements of Binary Tree 767

107 Unordered pairs 769

108 A type of finite bit strings 773
108.1 Preliminaries . . . 773
108.2 Fundamentals . . . 773
108.2.1 Type definition . . . 773
108.2.2 Basic arithmetic . . . 773
108.2.3 Basic tool setup . . . 775
108.2.4 Basic code generation setup . . . 775
108.2.5 Basic conversions . . . 776
108.3 Elementary case distinctions . . . 783
108.3.1 Basic ordering . . . 783
108.4 Enumeration . . . 785
108.5 Bit-wise operations . . . 786
108.6 Conversions including casts . . . 790
108.6.1 Generic unsigned conversion . . . 790
108.6.2 Generic signed conversion . . . 793
108.6.3 More . . . 794
108.7 Arithmetic operations . . . 798
108.8 Ordering . . . 800
108.9 Bit-wise operations . . . 802
108.10 More shift operations . . . 804
108.11 Single-bit operations . . . 805
108.12 Rotation . . . 805
108.13 Split and cat operations . . . 806
108.14 More on conversions . . . 807
108.15 Testing bits . . . 810
108.16 Word Arithmetic . . . 814
108.17 Transferring goals from words to ints . . . 817
108.18 Order on fixed-length words . . . 819
108.19 Conditions for the addition (etc) of two words to overflow . . . 821
108.20 Some proof tool support . . . 823
108.21 More on overflows and monotonicity . . . 823
108.22 Arithmetic type class instantiations . . . 827
108.23 Word and nat . . . 827
108.24 Cardinality, finiteness of set of words . . . 831
108.25 Bitwise Operations on Words . . . 832

108.25.1 shift functions in terms of lists of bools 836
108.25.2 Mask 839
108.25.3 Slices 841
108.25.4 Revcast 842
108.26 Split and cat 842
108.26.1 Split and slice 843
108.27 Rotation 843
108.27.1 Word rotation commutes with bit-wise operations . . 845
108.28 Maximum machine word 845
108.29 Recursion combinator for words 848
108.30 Some more naive computations rules 849
108.31 Executable intervals 851
108.32 Tool support 851

109 The Field of Integers mod 2 851

110 Pointwise order on product types 855
110.1 Pointwise ordering 855
110.2 Binary infimum and supremum 856
110.3 Top and bottom elements 857
110.4 Complete lattice operations 858
110.5 Complete distributive lattices 860
110.6 Bekic's Theorem 860

111 Finite Lattices 860
111.1 Finite Complete Lattices 860
111.2 Finite Distributive Lattices 863
111.3 Linear Orders 863
111.4 Finite Linear Orders 864

112 Lexicographic order on lists 865

113 Lexicographic order on lists 866

114 Prefix order on lists as order class instance 868

115 Lexicographic order on product types 869

116 Subsequence Ordering 870
116.1 Definitions and basic lemmas 871

117 Records based on BNF/datatype machinery 872

118 Implementation of mappings with Association Lists 874

119 Avoidance of pattern matching on natural numbers 878
119.1 Case analysis . . . 879
119.2 Preprocessors . . . 879
119.3 Candidates which need special treatment . . . 879

120 Implementation of natural numbers as binary numerals 879
120.1 Representation . . . 880
120.2 Basic arithmetic . . . 880
120.3 Conversions . . . 882

121 Code generation of prolog programs 882

122 Setup for Numerals 882

123 Implementation of integer numbers by target-language integers 882

124 Implementation of natural numbers by target-language integers 890
124.1 Implementation for *nat* . . . 890

125 Implementation of natural and integer numbers by target-language integers 895

126 Preprocessor setup for floats implemented by target language numerals 895

127 Abstract type of association lists with unique keys 896
127.1 Preliminaries . . . 896
127.2 Type (*'key*, *'value*) *alist* . . . 896
127.3 Primitive operations . . . 897
127.4 Abstract operation properties . . . 898
127.5 Further operations . . . 898
127.5.1 Equality . . . 898
127.5.2 Size . . . 898
127.6 Quickcheck generators . . . 899

128 alist is a BNF 900

129 Multisets partially implemented by association lists 901

130 Implementation of Red-Black Trees 906
130.1 Datatype of RB trees . . . 906
130.2 Tree properties . . . 906
130.2.1 Content of a tree . . . 906
130.2.2 Search tree properties . . . 907

130.2.3 Tree lookup . . . 908
130.2.4 Red-black properties . . . 909
130.3 Insertion . . . 910
130.4 Deletion . . . 914
130.5 Modifying existing entries . . . 919
130.6 Mapping all entries . . . 920
130.7 Folding over entries . . . 920
130.8 Bulkloading a tree . . . 921
130.9 Building a RBT from a sorted list . . . 921
130.10 Union and intersection of sorted associative lists . . . 927
130.11 Code generator setup . . . 944

131 Abstract type of RBT trees 946
131.1 Type definition . . . 946
131.2 Primitive operations . . . 946
131.3 Derived operations . . . 947
131.4 Abstract lookup properties . . . 948
131.5 Quickcheck generators . . . 951
131.6 Hide implementation details . . . 951

132 Implementation of mappings with Red-Black Trees 951
132.1 Data type and invariant . . . 951
132.2 Operations . . . 952
132.3 Invariant preservation . . . 952
132.4 Map Semantics . . . 953

133 Implementation of sets using RBT trees 953

134 Definition of code datatype constructors 953

135 Lemmas 953
135.1 Auxiliary lemmas . . . 953
135.2 fold and filter . . . 954
135.3 foldi and Ball . . . 954
135.4 foldi and Bex . . . 954
135.5 folding over non empty trees and selecting the minimal and maximal element . . . 955
135.5.1 concrete . . . 955
135.5.2 abstract . . . 957

136 Code equations 959

137 Introduction 964

138 Termination 965

139 Partial Functions **966**

140 Higher-Order Functions **966**
140.1 Limitations 967

141 Predefined Functions **967**

142 Locales **968**

143 Fine Points **969**

144 Common constants **970**

145 Pairs **971**

146 Filters **971**

147 Bounded quantifiers **971**

148 Operations on Predicates **971**

149 Setup for Numerals **971**

150 Arithmetic operations **971**
150.1 Arithmetic on naturals and integers 971
150.2 Inductive definitions for ordering on naturals 972

151 Alternative list definitions **972**
151.1 Alternative rules for *length* 972
151.2 Alternative rules for *list-all2* 973
151.3 Alternative rules for membership in lists 973

152 Setup for String.literal **973**

153 Simplification rules for optimisation **973**

154 A Prototype of Quickcheck based on the Predicate Compiler **974**

155 TFL: recursive function definitions **974**
155.1 Lemmas for TFL 974
155.2 Rule setup 975

156 Program extraction from proofs involving datatypes and inductive predicates **976**

157 Refute **976**

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*)
⟨*proof*⟩

corollary *update-conv*: *map-of* (*update k v al*) *k'* = ((*map-of al*)(*k*↦*v*)) *k'*
⟨*proof*⟩

lemma *dom-update*: *fst* ' *set* (*update k v al*) = {*k*} ∪ *fst* ' *set al*
⟨*proof*⟩

lemma *update-keys*:
map fst (*update k v al*) =
(*if k* ∈ *set* (*map fst al*) *then map fst al else map fst al* @ [*k*])
⟨*proof*⟩

lemma *distinct-update*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*update k v al*))
⟨*proof*⟩

lemma *update-filter*:
a ≠ *k* ⟹ *update k v* [*q*←*ps*. *fst q* ≠ *a*] = [*q*←*update k v ps*. *fst q* ≠ *a*]
⟨*proof*⟩

lemma *update-triv*: *map-of al k* = *Some v* ⟹ *update k v al* = *al*
⟨*proof*⟩

lemma *update-nonempty* [*simp*]: *update k v al* ≠ []
⟨*proof*⟩

lemma *update-eqD*: *update k v al = update k v' al' ⟹ v = v'*
⟨*proof*⟩

lemma *update-last* [*simp*]: *update k v (update k v' al) = update k v al*
⟨*proof*⟩

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))
⟨*proof*⟩

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
⟨*proof*⟩

lemma *image-update* [*simp*]: *x ∉ A ⟹ map-of (update x y al) ‘ A = map-of al ‘ A*
⟨*proof*⟩ **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)
⟨*proof*⟩

lemma *updates-key-simp* [*simp*]:
updates (k # ks) vs ps =
(case vs of [] ⇒ ps | v # vs ⇒ updates ks vs (update k v ps))
⟨*proof*⟩

lemma *updates-conv'*: *map-of (updates ks vs al) = (map-of al)(ks[↦]vs)*
⟨*proof*⟩

lemma *updates-conv*: *map-of (updates ks vs al) k = ((map-of al)(ks[↦]vs)) k*
⟨*proof*⟩

lemma *distinct-updates*:
assumes *distinct (map fst al)*
shows *distinct (map fst (updates ks vs al))*
⟨*proof*⟩

lemma *updates-append1*[*simp*]: *size ks < size vs ⟹*
updates (ks@[k]) vs al = update k (vs!size ks) (updates ks vs al)
⟨*proof*⟩

lemma *updates-list-update-drop*[*simp*]:
size ks ≤ *i* ⟹ *i* < *size vs* ⟹
updates ks (*vs*[*i*:=*v*]) *al* = *updates ks vs al*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *updates-apply-notin* [*simp*]:
k ∉ *set ks* ⟹ *map-of* (*updates ks vs al*) *k* = *map-of al k*
⟨*proof*⟩

lemma *updates-append-drop* [*simp*]:
size xs = *size ys* ⟹ *updates* (*xs* @ *zs*) *ys al* = *updates xs ys al*
⟨*proof*⟩

lemma *updates-append2-drop* [*simp*]:
size xs = *size ys* ⟹ *updates xs* (*ys* @ *zs*) *al* = *updates xs ys al*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *delete-conv'*: *map-of* (*delete k al*) = (*map-of al*)(*k* := *None*)
⟨*proof*⟩

corollary *delete-conv*: *map-of* (*delete k al*) *k'* = ((*map-of al*)(*k* := *None*)) *k'*
⟨*proof*⟩

lemma *delete-keys*: *map fst* (*delete k al*) = *removeAll k* (*map fst al*)
⟨*proof*⟩

lemma *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
⟨*proof*⟩

lemma *delete-id* [*simp*]: $k \notin fst\ `\ set\ al \Longrightarrow delete\ k\ al = al$
⟨*proof*⟩

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
⟨*proof*⟩

lemma *map-of-delete* [*simp*]: $k' \neq k \Longrightarrow map\text{-}of\ (delete\ k\ al)\ k' = map\text{-}of\ al\ k'$
⟨*proof*⟩

lemma *delete-notin-dom*: $k \notin fst\ `\ set\ (delete\ k\ al)$
⟨*proof*⟩

lemma *dom-delete-subset*: $fst\ `\ set\ (delete\ k\ al) \subseteq fst\ `\ set\ al$
⟨*proof*⟩

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

lemma *delete-update*: $k \neq l \Longrightarrow delete\ l\ (update\ k\ v\ al) = update\ k\ v\ (delete\ l\ al)$
⟨*proof*⟩

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
⟨*proof*⟩

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
⟨*proof*⟩

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* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))) *k'*
⟨*proof*⟩

lemma *map-of-update-with-aux*:
map-of (*update-with-aux v k f ps*) =
(*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))
⟨*proof*⟩

lemma *dom-update-with-aux*: *fst* ‘ *set* (*update-with-aux v k f ps*) = {*k*} ∪ *fst* ‘ *set ps*
⟨*proof*⟩

lemma *distinct-update-with-aux* [*simp*]:
distinct (*map fst* (*update-with-aux v k f ps*)) = *distinct* (*map fst ps*)
⟨*proof*⟩

lemma *set-update-with-aux*:
distinct (*map fst xs*) ⟹
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*))})
⟨*proof*⟩

lemma *set-delete-aux*: *distinct* (*map fst xs*) ⟹ *set* (*delete-aux k xs*) = *set xs* − {*k*} × *UNIV*
⟨*proof*⟩

lemma *dom-delete-aux*: *distinct* (*map fst ps*) ⟹ *fst* ‘ *set* (*delete-aux k ps*) = *fst* ‘ *set ps* − {*k*}
⟨*proof*⟩

lemma *distinct-delete-aux* [*simp*]: *distinct* (*map fst ps*) ⟹ *distinct* (*map fst* (*delete-aux k ps*))
⟨*proof*⟩

lemma *map-of-delete-aux'*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) = (*map-of xs*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-delete-aux*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) *k'* = ((*map-of xs*)(*k* := *None*)) *k'*
⟨*proof*⟩

lemma *delete-aux-eq-Nil-conv*: *delete-aux k ts* = [] ⟷ *ts* = [] ∨ (∃ *v*. *ts* = [(*k*, *v*)])

⟨*proof*⟩

1.4 *restrict*

qualified definition *restrict* :: *'key set* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *restrict-eq*: *restrict A* = *filter* (λ(*k*, *v*). *k* ∈ *A*)

lemma *restr-simps* [*simp*]:
restrict A [] = []
restrict A (*p*#*ps*) = (*if fst p* ∈ *A then p* # *restrict A ps else restrict A ps*)
⟨*proof*⟩

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
⟨*proof*⟩

corollary *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
⟨*proof*⟩

lemma *distinct-restr*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*restrict A al*))
⟨*proof*⟩

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
⟨*proof*⟩

lemma *restr-in* [*simp*]: *x* ∈ *A* ⟹ *map-of* (*restrict A al*) *x* = *map-of al x*
⟨*proof*⟩

lemma *restr-out* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*restrict A al*) *x* = *None*
⟨*proof*⟩

lemma *dom-restr* [*simp*]: *fst* ‘ *set* (*restrict A al*) = *fst* ‘ *set al* ∩ *A*
⟨*proof*⟩

lemma *restr-upd-same* [*simp*]: *restrict* (−{*x*}) (*update x y al*) = *restrict* (−{*x*}) *al*
⟨*proof*⟩

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *restr-delete* [*simp*]:
delete x (*restrict D al*) = (*if x* ∈ *D then restrict* (*D* − {*x*}) *al else restrict D al*)
⟨*proof*⟩

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
⟨*proof*⟩

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

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*))
⟨*proof*⟩

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
⟨*proof*⟩

1.5 *clearjunk*

qualified function *clearjunk* :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where
clearjunk [] = []
| *clearjunk* (*p*#*ps*) = *p* # *clearjunk* (*delete* (*fst p*) *ps*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *map-of-clearjunk*: *map-of* (*clearjunk al*) = *map-of al*
⟨*proof*⟩

lemma *clearjunk-keys-set*: *set* (*map fst* (*clearjunk al*)) = *set* (*map fst al*)
⟨*proof*⟩

lemma *dom-clearjunk*: *fst* ‘ *set* (*clearjunk al*) = *fst* ‘ *set al*
⟨*proof*⟩

lemma *distinct-clearjunk* [*simp*]: *distinct* (*map fst* (*clearjunk al*))
⟨*proof*⟩

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
⟨*proof*⟩

lemma *ran-map-of*: *ran* (*map-of al*) = *snd* ‘ *set* (*clearjunk al*)
⟨*proof*⟩

lemma *graph-map-of*: *Map.graph* (*map-of al*) = *set* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-updates*: *clearjunk* (*updates ks vs al*) = *updates ks vs* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-delete*: *clearjunk* (*delete x al*) = *delete x* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-restrict*: *clearjunk* (*restrict A al*) = *restrict A* (*clearjunk al*)
⟨*proof*⟩

lemma *distinct-clearjunk-id* [*simp*]: *distinct* (*map fst al*) ⟹ *clearjunk al* = *al*
⟨*proof*⟩

lemma *clearjunk-idem*: *clearjunk* (*clearjunk al*) = *clearjunk al*
⟨*proof*⟩

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
⟨*proof*⟩

lemma *delete-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *delete k* (*map f ps*) = *map f* (*delete k ps*)
⟨*proof*⟩

lemma *clearjunk-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *clearjunk* (*map f ps*) = *map f* (*clearjunk ps*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *length-map-ran*[*simp*]: *length* (*map-ran f al*) = *length al*
⟨*proof*⟩

lemma *map-fst-map-ran*[*simp*]: *map fst* (*map-ran f al*) = *map fst al*
⟨*proof*⟩

lemma *dom-map-ran*: *fst* ‘ *set* (*map-ran f al*) = *fst* ‘ *set al*
⟨*proof*⟩

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *map-option* (*f k*) (*map-of al k*)
⟨*proof*⟩

lemma *distinct-map-ran*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*map-ran f al*))
⟨*proof*⟩

lemma *map-ran-filter*: *map-ran f* [*p*←*ps*. *fst p* ≠ *a*] = [*p*←*map-ran f ps*. *fst p* ≠ *a*]
⟨*proof*⟩

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *merge-updates*: *merge qs ps* = *updates* (*rev* (*map fst ps*)) (*rev* (*map snd ps*)) *qs*
⟨*proof*⟩

lemma *dom-merge*: *fst* ‘ *set* (*merge xs ys*) = *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*
⟨*proof*⟩

lemma *distinct-merge*: *distinct* (*map fst xs*) ⟹ *distinct* (*map fst* (*merge xs ys*))
⟨*proof*⟩

lemma *clearjunk-merge*: *clearjunk* (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
⟨*proof*⟩

lemma *merge-conv′*: *map-of* (*merge xs ys*) = *map-of xs* ++ *map-of ys*

⟨*proof*⟩

corollary *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
⟨*proof*⟩

lemma *merge-empty*: *map-of* (*merge* [] *ys*) = *map-of ys*
⟨*proof*⟩

lemma *merge-assoc* [*simp*]: *map-of* (*merge m1* (*merge m2 m3*)) = *map-of* (*merge* (*merge m1 m2*) *m3*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *merge-None* [*iff*]: (*map-of* (*merge m n*) *k* = *None*) = (*map-of n k* = *None* ∧ *map-of m k* = *None*)
⟨*proof*⟩

lemma *merge-upd* [*simp*]: *map-of* (*merge m* (*update k v n*)) = *map-of* (*update k v* (*merge m n*))
⟨*proof*⟩

lemma *merge-updatess* [*simp*]:
map-of (*merge m* (*updates xs ys n*)) = *map-of* (*updates xs ys* (*merge m n*))
⟨*proof*⟩

lemma *merge-append*: *map-of* (*xs* @ *ys*) = *map-of* (*merge ys xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩
termination

⟨*proof*⟩

lemma *compose-first-None* [*simp*]: *map-of xs k* = *None* ⟹ *map-of* (*compose xs ys*) *k* = *None*
⟨*proof*⟩

lemma *compose-conv*: *map-of* (*compose xs ys*) *k* = (*map-of ys* ∘$_m$ *map-of xs*) *k*
⟨*proof*⟩

lemma *compose-conv'*: *map-of* (*compose xs ys*) = (*map-of ys* ∘$_m$ *map-of xs*)
⟨*proof*⟩

lemma *compose-first-Some* [*simp*]: *map-of xs k* = *Some v* ⟹ *map-of* (*compose xs ys*) *k* = *map-of ys v*
⟨*proof*⟩

lemma *dom-compose*: *fst* ‘ *set* (*compose xs ys*) ⊆ *fst* ‘ *set xs*
⟨*proof*⟩

lemma *distinct-compose*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*compose xs ys*))
⟨*proof*⟩

lemma *compose-delete-twist*: *compose* (*delete k xs*) *ys* = *delete k* (*compose xs ys*)
⟨*proof*⟩

lemma *compose-clearjunk*: *compose xs* (*clearjunk ys*) = *compose xs ys*
⟨*proof*⟩

lemma *clearjunk-compose*: *clearjunk* (*compose xs ys*) = *compose* (*clearjunk xs*) *ys*
⟨*proof*⟩

lemma *compose-empty* [*simp*]: *compose xs* [] = []
⟨*proof*⟩

lemma *compose-Some-iff*:
(*map-of* (*compose xs ys*) *k* = *Some v*) ⟷
(∃ *k'*. *map-of xs k* = *Some k'* ∧ *map-of ys k'* = *Some v*)
⟨*proof*⟩

lemma *map-comp-None-iff*:
map-of (*compose xs ys*) *k* = *None* ⟷
(*map-of xs k* = *None* ∨ (∃ *k'*. *map-of xs k* = *Some k'* ∧ *map-of ys k'* = *None*))
⟨*proof*⟩

1.9 *map-entry*

qualified fun *map-entry* :: *'key* ⇒ (*'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
map-entry k f [] = []
| *map-entry k f* (*p* # *ps*) =
(*if fst p* = *k then* (*k*, *f* (*snd p*)) # *ps else p* # *map-entry k f ps*)

lemma *map-of-map-entry*:
map-of (*map-entry k f xs*) =
(*map-of xs*)(*k* := *case map-of xs k of None* ⇒ *None* | *Some v'* ⇒ *Some* (*f v'*))
⟨*proof*⟩

lemma *dom-map-entry*: *fst* ‘ *set* (*map-entry k f xs*) = *fst* ‘ *set xs*
⟨*proof*⟩

lemma *distinct-map-entry*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*map-entry k f xs*))
⟨*proof*⟩

1.10 *map-default*

fun *map-default* :: *'key* ⇒ *'val* ⇒ (*'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
map-default k v f [] = [(*k*, *v*)]
| *map-default k v f* (*p* # *ps*) =
(*if fst p* = *k then* (*k*, *f* (*snd p*)) # *ps else p* # *map-default k v f ps*)

lemma *map-of-map-default*:
map-of (*map-default k v f xs*) =
(*map-of xs*)(*k* := *case map-of xs k of None* ⇒ *Some v* | *Some v'* ⇒ *Some* (*f v'*))
⟨*proof*⟩

lemma *dom-map-default*: *fst* ‘ *set* (*map-default k v f xs*) = *insert k* (*fst* ‘ *set xs*)
⟨*proof*⟩

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

end

end

2 Axiomatic Declaration of Bounded Natural Functors

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

⟨*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*: $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$
⟨*proof*⟩

lemma *inject-refine*: $g\ (f\ x) = x \Longrightarrow g\ (f\ y) = y \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *convol-apply*: *BNF-Def.convol* $f\ g\ x = (f\ x,\ g\ x)$
⟨*proof*⟩

lemma *Grp-UNIV-id*: *BNF-Def.Grp UNIV id* $= (=)$
⟨*proof*⟩

lemma *sum-comp-cases*:
assumes $f \circ Inl = g \circ Inl$ **and** $f \circ Inr = g \circ Inr$
shows $f = g$
⟨*proof*⟩

lemma *case-sum-Inl-Inr-L*: *case-sum* $(f \circ Inl)\ (f \circ Inr) = f$
⟨*proof*⟩

lemma *eq-o-InrI*: $[\![g \circ Inl = h;\ case\text{-}sum\ h\ f = g]\!] \Longrightarrow f = g \circ Inr$
⟨*proof*⟩

lemma *id-bnf-o*: *BNF-Composition.id-bnf* $\circ\ f = f$

⟨*proof*⟩

lemma *o-id-bnf*: $f \circ$ *BNF-Composition.id-bnf* $= f$
⟨*proof*⟩

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*) $\longleftrightarrow P \wedge Q$
⟨*proof*⟩

lemma *if-distrib-fun*: (*if* c *then* f *else* g) $x =$ (*if* c *then* $f\ x$ *else* $g\ x$)
⟨*proof*⟩

3.1 Coinduction

lemma *eq-comp-compI*: $a \circ b = f \circ x \Longrightarrow x \circ c = id \Longrightarrow f = a \circ (b \circ c)$
⟨*proof*⟩

lemma *self-bounded-weaken-left*: $(a :: 'a :: semilattice\text{-}inf) \leq inf\ a\ b \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *self-bounded-weaken-right*: $(a :: 'a :: semilattice\text{-}inf) \leq inf\ b\ a \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *symp-iff*: $symp\ R \longleftrightarrow R = R^{-1-1}$
⟨*proof*⟩

lemma *equivp-inf*: $[\![equivp\ R; equivp\ S]\!] \Longrightarrow equivp\ (inf\ R\ S)$
⟨*proof*⟩

lemma *vimage2p-rel-prod*:
$(\lambda x\ y.\ rel\text{-}prod\ R\ S\ (BNF\text{-}Def.convol\ f1\ g1\ x)\ (BNF\text{-}Def.convol\ f2\ g2\ y)) =$
$(inf\ (BNF\text{-}Def.vimage2p\ f1\ f2\ R)\ (BNF\text{-}Def.vimage2p\ g1\ g2\ S))$
⟨*proof*⟩

lemma *predicate2I-obj*: $(\forall x\ y.\ P\ x\ y \longrightarrow Q\ x\ y) \Longrightarrow P \leq Q$
⟨*proof*⟩

lemma *predicate2D-obj*: $P \leq Q \Longrightarrow P\ x\ y \longrightarrow Q\ x\ y$
⟨*proof*⟩

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*: $\bigwedge R\ x\ y.\ [\![(\mathit{rel\text{-}fun}\ (\mathit{rel}\ R)\ R)\ \mathit{eval}\ \mathit{eval};\ \mathit{rel}\ (\mathit{inf}\ R\ (\mathit{retr}\ R))\ x\ y]\!] \Longrightarrow$
$\mathit{retr}\ R\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)$
begin

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

lemma *cong-retr*: $\mathit{cong}\ R \Longrightarrow \mathit{cong}\ (\mathit{inf}\ R\ (\mathit{retr}\ R))$
⟨*proof*⟩

lemma *cong-equivp*: $\mathit{cong}\ R \Longrightarrow \mathit{equivp}\ R$
⟨*proof*⟩

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

lemma *gen-cong-reflp*[*intro*, *simp*]: $x = y \Longrightarrow \mathit{gen\text{-}cong}\ R\ x\ y$
⟨*proof*⟩

lemma *gen-cong-symp*[*intro*]: $\mathit{gen\text{-}cong}\ R\ x\ y \Longrightarrow \mathit{gen\text{-}cong}\ R\ y\ x$
⟨*proof*⟩

lemma *gen-cong-transp*[*intro*]: $\mathit{gen\text{-}cong}\ R\ x\ y \Longrightarrow \mathit{gen\text{-}cong}\ R\ y\ z \Longrightarrow \mathit{gen\text{-}cong}\ R\ x\ z$
⟨*proof*⟩

lemma *equivp-gen-cong*: $\mathit{equivp}\ (\mathit{gen\text{-}cong}\ R)$
⟨*proof*⟩

lemma *leq-gen-cong*: $R \leq \mathit{gen\text{-}cong}\ R$
⟨*proof*⟩

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

lemma *gen-cong-minimal*: $[\![R \leq R';\ \mathit{cong}\ R']\!] \Longrightarrow \mathit{gen\text{-}cong}\ R \leq R'$
⟨*proof*⟩

lemma *congdd-base-gen-congdd-base-aux*:
$\mathit{rel}\ (\mathit{gen\text{-}cong}\ R)\ x\ y \Longrightarrow R \leq R' \Longrightarrow \mathit{cong}\ R' \Longrightarrow R'\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)$
⟨*proof*⟩

lemma *cong-gen-cong*: $\mathit{cong}\ (\mathit{gen\text{-}cong}\ R)$
⟨*proof*⟩

lemma *gen-cong-eval-rel-fun*:
$(\mathit{rel\text{-}fun}\ (\mathit{rel}\ (\mathit{gen\text{-}cong}\ R))\ (\mathit{gen\text{-}cong}\ R))\ \mathit{eval}\ \mathit{eval}$
⟨*proof*⟩

lemma *gen-cong-eval*:
rel (*gen-cong R*) *x y* ⟹ *gen-cong R* (*eval x*) (*eval y*)
⟨*proof*⟩

lemma *gen-cong-idem*: *gen-cong* (*gen-cong R*) = *gen-cong R*
⟨*proof*⟩

lemma *gen-cong-rho*:
ϱ = *eval* ∘ *f* ⟹ *rel* (*gen-cong R*) (*f x*) (*f y*) ⟹ *gen-cong R* (ϱ *x*) (ϱ *y*)
⟨*proof*⟩
lemma *coinduction*:
assumes *coind*: ∀ *R*. *R* ≤ *retr R* ⟶ *R* ≤ (=)
assumes *cih*: *R* ≤ *retr* (*gen-cong R*)
shows *R* ≤ (=)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⟨*proof*⟩
interpretation *step*: *cong rel′ eval′ retr* ⟨*proof*⟩

lemma *gen-cong-emb*: *base.gen-cong R* ≤ *step.gen-cong R*
⟨*proof*⟩

end

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

⟨*ML*⟩

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* $(\exists k.\ \neg\ b\ ((c\ ^{\frown\frown}\ k)\ s))$
then Some $((c\ ^{\frown\frown}\ (LEAST\ k.\ \neg\ b\ ((c\ ^{\frown\frown}\ k)\ s)))\ s)$
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*⟩

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

lemma *while-option-stop*: *while-option b c s* = *Some t* $\Longrightarrow \neg\ b\ t$
⟨*proof*⟩

theorem *while-option-rule*:
assumes *step*: $\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s)$
and *result*: *while-option b c s* = *Some t*
and *init*: *P s*
shows *P t*
⟨*proof*⟩

lemma *funpow-commute*:
$[\![\forall k' < k.\ f\ (c\ ((c^{\frown\frown}k')\ s)) = c'\ (f\ ((c^{\frown\frown}k')\ s))]\!] \Longrightarrow f\ ((c^{\frown\frown}k)\ s) = (c'^{\frown\frown}k)\ (f\ s)$
⟨*proof*⟩

lemma *while-option-commute-invariant*:
assumes *Invariant*: $\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s)$
assumes *TestCommute*: $\bigwedge s.\ P\ s \Longrightarrow b\ s = b'\ (f\ s)$
assumes *BodyCommute*: $\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow f\ (c\ s) = c'\ (f\ s)$
assumes *Initial*: *P s*
shows *map-option f* (*while-option b c s*) = *while-option b' c'* (*f s*)
⟨*proof*⟩

lemma *while-option-commute*:
assumes $\bigwedge s.\ b\ s = b'\ (f\ s)\ \bigwedge s.\ [\![b\ s]\!] \Longrightarrow f\ (c\ s) = c'\ (f\ s)$
shows *map-option f* (*while-option b c s*) = *while-option b' c'* (*f s*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *def-while-unfold*:
assumes *fdef*: *f* == *while test do*
shows *f x* = (*if test x then f*(*do x*) *else x*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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*⟩

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*⟩

lemma *lfp-while*:
assumes *mono* f **and** $\bigwedge X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite* C
shows *lfp* $f =$ *while* $(\lambda A.\ f\ A \neq A)\ f\ \{\}$
⟨*proof*⟩

lemma *wf-finite-less*:
assumes *finite* $(C :: 'a::order\ set)$
shows *wf* $\{(x,\ y).\ \{x,\ y\} \subseteq C \wedge x < y\}$
⟨*proof*⟩

lemma *wf-finite-greater*:
assumes *finite* $(C :: 'a::order\ set)$
shows *wf* $\{(x,\ y).\ \{x,\ y\} \subseteq C \wedge y < x\}$
⟨*proof*⟩

lemma *while-option-finite-increasing-Some*:
fixes $f :: 'a::order \Rightarrow 'a$
assumes *mono* f **and** *finite* $(UNIV :: 'a\ set)$ **and** $s \leq f\ s$
shows $\exists P.$ *while-option* $(\lambda A.\ f\ A \neq A)\ f\ s =$ *Some* P
⟨*proof*⟩

lemma *lfp-the-while-option-lattice*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: $'a$ *set*)
shows *lfp f* = *the* (*while-option* ($\lambda A.$ *f A* $\neq$ *A*) *f bot*)
⟨*proof*⟩

lemma *lfp-while-lattice*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: $'a$ *set*)
shows *lfp f* = *while* ($\lambda A.$ *f A* $\neq$ *A*) *f bot*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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* (λ(*x*,*n*). *b x*) (λ(*x*,*n*). (*f x*, *n* + *T-b x* + *T-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*))
⟨*proof*⟩

lemma *T-while-option-if-T-while-option2*:

T-while-option2 b f tb tf (s,n0) = Some (t,n) ⟹ while-option b f s = Some t
⟨*proof*⟩

lemma *T-while-option2-if-T-while-option*:
while-option b f s = Some t ⟹ ∃ n. T-while-option2 b f tb tf (s,n0) = Some (t,n)
⟨*proof*⟩

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 ⇒ 's option*) ⇒ *'s ⇒ 's option* **where**
while-Some f = while-option (λs. f s ≠ None) (the o f)

lemma *while-Some-rec*[*code*]:
while-Some f x = (case f x of None ⇒ Some x | Some y ⇒ while-Some f y)
⟨*proof*⟩

A frequent special case: saturation of a set.

definition *while-saturate* :: (*'a set ⇒ 'a set*) ⇒ *'a set ⇒ 'a set option* **where**
while-saturate f = while-option (λM. ¬ f M ⊆ M) (λM. M ∪ f M)

lemma *while-option-cong*: *(⋀s. b s ⟹ c s = c' s) ⟹ while-option b c s = while-option b c' s*
⟨*proof*⟩

lemma *while-saturate-code*[*code*]: *while-saturate f M =*
while-Some (λM. let M' = f M in if M' ⊆ M then None else Some (M ∪ M')) M
⟨*proof*⟩

lemma *while-saturate-saturated*: *while-saturate F S0 = Some S ⟹ F S ⊆ S*
⟨*proof*⟩

Termination:

lemma *while-option-sat-finite-subset-Some*: **fixes** *C* :: *'a set*
assumes *mono f* **and** *⋀X. X ⊆ C ⟹ f X ⊆ C* **and** *finite C* **and** *M ⊆ C*
shows *∃ S. while-option (λM. ¬ f M ⊆ M) (λM. M ∪ f M) M = Some S*
⟨*proof*⟩

corollary *while-saturate-finite-subset-Some*:
assumes *mono f* **and** *⋀X. X ⊆ C ⟹ f X ⊆ C* **and** *finite C* **and** *M ⊆ C*
shows *∃ S. while-saturate f M = Some S*
⟨*proof*⟩

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

theorem *while-saturate-rule*:
assumes *⋀S. P S ⟹ P (S ∪ F S)*

and *while-saturate F S = Some T*
and *P S*
shows *P T*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *while-saturate-incr*: *while-saturate F S0 = Some S* $\Longrightarrow$ *S0* $\subseteq$ *S*
⟨*proof*⟩

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*⟩

corollary *while-saturate-prefix*:
⟦ *mono f*; *while-saturate f M = Some S*; $M \subseteq P$; $f\ P \subseteq P$ ⟧ $\Longrightarrow$ $S \subseteq P$
⟨*proof*⟩

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 $p :: 'a \Rightarrow bool$
and $f :: 'a \Rightarrow 'a\ list$
and $x :: 'a$
begin

qualified fun *rtrancl-while-test* :: $'a\ list \times 'a\ set \Rightarrow bool$
where *rtrancl-while-test* $(ws,\text{-}) = (ws \neq [] \wedge p(hd\ ws))$

qualified fun *rtrancl-while-step* :: $'a\ list \times 'a\ set \Rightarrow 'a\ list \times 'a\ set$
where *rtrancl-while-step* $(ws,\ Z) =$
$(let\ x = hd\ ws;\ new = remdups\ (filter\ (\lambda y.\ y \notin Z)\ (f\ x))$
$in\ (new\ @\ tl\ ws,\ set\ new \cup Z))$

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*⟩

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*⟩

lemma *rtrancl-while-finite-Some*:
assumes *finite* ({(*x*, *y*). *y* ∈ *set* (*f x*)}∗ `` {*x*}) (**is** *finite ?Cl*)
shows ∃ *y*. *rtrancl-while* = *Some y*
⟨*proof*⟩

end

end

5 The Bourbaki-Witt tower construction for transfinite iteration

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

lemma *ChainsI* [*intro?*]:
(⋀*a b*. ⟦ *a* ∈ *Y*; *b* ∈ *Y* ⟧ ⟹ (*a*, *b*) ∈ *r* ∨ (*b*, *a*) ∈ *r*) ⟹ *Y* ∈ *Chains r*
⟨*proof*⟩

lemma *in-Chains-subset*: ⟦ *M* ∈ *Chains r*; *M'* ⊆ *M* ⟧ ⟹ *M'* ∈ *Chains r*
⟨*proof*⟩

lemma *in-ChainsD*: ⟦ *M* ∈ *Chains r*; *x* ∈ *M*; *y* ∈ *M* ⟧ ⟹ (*x*, *y*) ∈ *r* ∨ (*y*, *x*) ∈ *r*
⟨*proof*⟩

lemma *Chains-FieldD*: ⟦ *M* ∈ *Chains r*; *x* ∈ *M* ⟧ ⟹ *x* ∈ *Field r*

⟨*proof*⟩

lemma *in-Chains-conv-chain*: $M \in Chains\ r \longleftrightarrow Complete\text{-}Partial\text{-}Order.chain\ (\lambda x\ y.\ (x,\ y) \in r)\ M$
⟨*proof*⟩

lemma *partial-order-on-trans*:
$[\![\ partial\text{-}order\text{-}on\ A\ r;\ (x,\ y) \in r;\ (y,\ z) \in r\ ]\!] \Longrightarrow (x,\ z) \in r$
⟨*proof*⟩

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*: $[\![\ (x,\ y) \in leq;\ (y,\ z) \in leq\ ]\!] \Longrightarrow (x,\ z) \in leq$
⟨*proof*⟩

lemma *leq-refl*: $x \in Field\ leq \Longrightarrow (x,\ x) \in leq$
⟨*proof*⟩

lemma *leq-antisym*: $[\![\ (x,\ y) \in leq;\ (y,\ x) \in leq\ ]\!] \Longrightarrow x = y$
⟨*proof*⟩

inductive-set *iterates-above* :: $'a \Rightarrow 'a\ set$
for *a*
where
base: $a \in iterates\text{-}above\ a$
| *step*: $x \in iterates\text{-}above\ a \Longrightarrow f\ x \in iterates\text{-}above\ a$
| *Sup*: $[\![\ M \in Chains\ leq;\ M \neq \{\};\ \bigwedge x.\ x \in M \Longrightarrow x \in iterates\text{-}above\ a\ ]\!] \Longrightarrow lub\ M \in iterates\text{-}above\ a$

definition *fixp-above* :: $'a \Rightarrow 'a$
where $fixp\text{-}above\ a = (if\ a \in Field\ leq\ then\ lub\ (iterates\text{-}above\ a)\ else\ a)$

lemma *fixp-above-outside*: $a \notin Field\ leq \Longrightarrow fixp\text{-}above\ a = a$
⟨*proof*⟩

lemma *fixp-above-inside*: $a \in Field\ leq \Longrightarrow fixp\text{-}above\ a = lub\ (iterates\text{-}above\ a)$
⟨*proof*⟩

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*: ⟦ $x \in$ *iterates-above* a; $a \in$ *Field leq* ⟧ ⟹ $(x, f\,x) \in$ *leq*
⟨*proof*⟩

lemma *iterates-above-Field*: ⟦ $x \in$ *iterates-above* a; $a \in$ *Field leq* ⟧ ⟹ $x \in$ *Field leq*
⟨*proof*⟩

lemma *iterates-above-ge*:
assumes *y*: $y \in$ *iterates-above* a
and *a*: $a \in$ *Field leq*
shows $(a, y) \in$ *leq*
⟨*proof*⟩

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-above* a
shows *lub* $M \in$ *iterates-above* a
⟨*proof*⟩

lemma *iterates-above-successor*:
assumes *y*: $y \in$ *iterates-above* a
and *a*: $a \in$ *Field leq*
shows $y = a \vee y \in$ *iterates-above* $(f\,a)$
⟨*proof*⟩

lemma *iterates-above-Sup-aux*:
assumes *M*: $M \in$ *Chains leq* $M \neq \{\}$
and *M'*: $M' \in$ *Chains leq* $M' \neq \{\}$
and *comp*: $\bigwedge x.\ x \in M \Longrightarrow x \in$ *iterates-above* (*lub* M') $\vee$ *lub* $M' \in$ *iterates-above* x
shows (*lub* M, *lub* M') $\in$ *leq* $\vee$ *lub* $M \in$ *iterates-above* (*lub* M')
⟨*proof*⟩

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
⟨*proof*⟩

lemma *chain-iterates-above*:

assumes *a*: $a \in$ *Field leq*
shows *iterates-above a* $\in$ *Chains leq* (**is** *?C* $\in$ -)
⟨*proof*⟩

lemma *fixp-iterates-above*: *fixp-above a* $\in$ *iterates-above a*
⟨*proof*⟩

lemma *fixp-above-Field*: $a \in$ *Field leq* $\Longrightarrow$ *fixp-above a* $\in$ *Field leq*
⟨*proof*⟩

lemma *fixp-above-unfold*:
assumes *a*: $a \in$ *Field leq*
shows *fixp-above a* = *f* (*fixp-above a*) (**is** *?a* = *f ?a*)
⟨*proof*⟩

end

lemma *fixp-above-induct* [*case-names adm base step*]:
assumes *adm*: *ccpo.admissible lub* ($\lambda x\ y.\ (x,\ y) \in$ *leq*) *P*
and *base*: *P a*
and *step*: $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$
shows *P* (*fixp-above a*)
⟨*proof*⟩

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 $M \in$ *Chains leq*
and $M \neq \{\}$
and *finite M*
shows *lub M* $\in$ *M*
⟨*proof*⟩

lemma *fun-pow-iterates-above*: (*f* ^^ *k*) *a* $\in$ *iterates-above a*
⟨*proof*⟩

lemma *chfin-iterates-above-fun-pow*:
assumes $x \in$ *iterates-above a*
assumes $\forall M \in$ *Chains leq*. *finite M*
shows $\exists j.\ x = (f\ \widehat{\ }\widehat{\ }\ j)\ a$
⟨*proof*⟩

lemma *Chain-finite-iterates-above-fun-pow-iff*:

assumes $\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M$
shows $x \in \mathit{iterates\text{-}above}\ a \longleftrightarrow (\exists j.\ x = (f \frown\frown j)\ a)$
⟨*proof*⟩

lemma *fixp-above-Kleene-iter-ex*:
assumes $(\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M)$
obtains k **where** $\mathit{fixp\text{-}above}\ a = (f \frown\frown k)\ a$
⟨*proof*⟩

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

lemma *funpow-Field-leq*: $(f \frown\frown k)\ a \in \mathit{Field}\ \mathit{leq}$
⟨*proof*⟩

lemma *funpow-prefix*: $j < k \Longrightarrow ((f \frown\frown j)\ a, (f \frown\frown k)\ a) \in \mathit{leq}$
⟨*proof*⟩

lemma *funpow-suffix*: $(f \frown\frown \mathit{Suc}\ k)\ a = (f \frown\frown k)\ a \Longrightarrow ((f \frown\frown (j + k))\ a, (f \frown\frown k)\ a) \in \mathit{leq}$
⟨*proof*⟩

lemma *funpow-stability*: $(f \frown\frown \mathit{Suc}\ k)\ a = (f \frown\frown k)\ a \Longrightarrow ((f \frown\frown j)\ a, (f \frown\frown k)\ a) \in \mathit{leq}$
⟨*proof*⟩

lemma *funpow-in-Chains*: $\{(f \frown\frown k)\ a\ |k.\ \mathit{True}\} \in \mathit{Chains}\ \mathit{leq}$
⟨*proof*⟩

lemma *fixp-above-Kleene-iter*:
assumes $\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M$ — convenient but surely not necessary
assumes $(f \frown\frown \mathit{Suc}\ k)\ a = (f \frown\frown k)\ a$
shows $\mathit{fixp\text{-}above}\ a = (f \frown\frown k)\ a$
⟨*proof*⟩

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

lemma *Chain-finite-wf*: $\mathit{wf}\ \{(f\ ((f \frown\frown k)\ a), (f \frown\frown k)\ a)\ |k.\ f\ ((f \frown\frown k)\ a) \neq (f \frown\frown k)\ a\}$
⟨*proof*⟩

lemma *while-option-finite-increasing*: $\exists P.\ \mathit{while\text{-}option}\ (\lambda A.\ f\ A \neq A)\ f\ a = \mathit{Some}\ P$
⟨*proof*⟩

lemma *fixp-above-the-while-option*: $\mathit{fixp\text{-}above}\ a = \mathit{the}\ (\mathit{while\text{-}option}\ (\lambda A.\ f\ A \neq A)\ f\ a)$
⟨*proof*⟩

lemma *fixp-above-conv-while*: $\mathit{fixp\text{-}above}\ a = \mathit{while}\ (\lambda A.\ f\ A \neq A)\ f\ a$

⟨*proof*⟩

end

end

end

lemma *bourbaki-witt-fixpoint-complete-latticeI*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes $\bigwedge x.\ x \leq f\ x$
shows *bourbaki-witt-fixpoint Sup* $\{(x,\ y).\ x \leq y\}$ f
⟨*proof*⟩

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*
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma *mult-cdiv-cmod-eq*:

‹$l * (k\ cdiv\ l) + k\ cmod\ l = k$›
⟨*proof*⟩

lemma *cmod-cdiv-mult-eq*:
‹$k\ cmod\ l + k\ cdiv\ l * l = k$›
⟨*proof*⟩

lemma *cmod-mult-cdiv-eq*:
‹$k\ cmod\ l + l * (k\ cdiv\ l) = k$›
⟨*proof*⟩

lemma *minus-cdiv-mult-eq-cmod*:
‹$k - k\ cdiv\ l * l = k\ cmod\ l$›
⟨*proof*⟩

lemma *minus-mult-cdiv-eq-cmod*:
‹$k - l * (k\ cdiv\ l) = k\ cmod\ l$›
⟨*proof*⟩

lemma *minus-cmod-eq-cdiv-mult*:
‹$k - k\ cmod\ l = k\ cdiv\ l * l$›
⟨*proof*⟩

lemma *minus-cmod-eq-mult-cdiv*:
‹$k - k\ cmod\ l = l * (k\ cdiv\ l)$›
⟨*proof*⟩

lemma *cdiv-0-eq* [*simp*]:
‹$k\ cdiv\ 0 = 0$›
⟨*proof*⟩

lemma *cmod-0-eq* [*simp*]:
‹$k\ cmod\ 0 = k$›
⟨*proof*⟩

lemma *cdiv-1-eq* [*simp*]:
‹$k\ cdiv\ 1 = k$›
⟨*proof*⟩

lemma *cmod-1-eq* [*simp*]:
‹$k\ cmod\ 1 = 0$›
⟨*proof*⟩

lemma *zero-cdiv-eq* [*simp*]:
‹$0\ cdiv\ k = 0$›
⟨*proof*⟩

lemma *zero-cmod-eq* [*simp*]:
‹$0\ cmod\ k = 0$›

⟨*proof*⟩

lemma *cdiv-minus-eq*:
‹*k cdiv* − *l* = − (*k cdiv l*)›
⟨*proof*⟩

lemma *cmod-minus-eq* [*simp*]:
‹*k cmod* − *l* = *k cmod l*›
⟨*proof*⟩

lemma *cdiv-abs-eq*:
‹*k cdiv* |*l*| = *sgn l* ∗ (*k cdiv l*)›
⟨*proof*⟩

lemma *cmod-abs-eq* [*simp*]:
‹*k cmod* |*l*| = *k cmod l*›
⟨*proof*⟩

lemma *nonzero-mult-cdiv-cancel-right*:
‹*k* ∗ *l cdiv l* = *k*› **if** ‹*l* ≠ *0*›
⟨*proof*⟩

lemma *cdiv-self-eq* [*simp*]:
‹*k cdiv k* = *1*› **if** ‹*k* ≠ *0*›
⟨*proof*⟩

lemma *cmod-self-eq* [*simp*]:
‹*k cmod k* = *0*›
⟨*proof*⟩

lemma *cmod-less-divisor*:
‹*k cmod l* < |*l*| − |*l*| *div 2*› **if** ‹*l* ≠ *0*›
⟨*proof*⟩

lemma *cmod-less-equal-divisor*:
‹*k cmod l* ≤ |*l*| *div 2*› **if** ‹*l* ≠ *0*›
⟨*proof*⟩

lemma *divisor-less-equal-cmod'*:
‹|*l*| *div 2* − |*l*| ≤ *k cmod l*› **if** ‹*l* ≠ *0*›
⟨*proof*⟩

lemma *divisor-less-equal-cmod*:
‹− (|*l*| *div 2*) ≤ *k cmod l*› **if** ‹*l* ≠ *0*›
⟨*proof*⟩

lemma *abs-cmod-less-equal*:
‹|*k cmod l*| ≤ |*l*| *div 2*› **if** ‹*l* ≠ *0*›
⟨*proof*⟩

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
⟨*proof*⟩

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*]›
⟨*proof*⟩

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*]›
⟨*proof*⟩

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

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

instance
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *phantom-comp-of-phantom* [*simp*]: *phantom* ∘ *of-phantom* = *id*
and *of-phantom-comp-phantom* [*simp*]: *of-phantom* ∘ *phantom* = *id*
⟨*proof*⟩

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

⟨*ML*⟩

lemma *of-phantom-inject* [*simp*]:
of-phantom x = *of-phantom y* ⟷ *x* = *y*
⟨*proof*⟩

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*)

⟨*ML*⟩

lemma *card-prod* [*simp*]: *CARD*('*a* × '*b*) = *CARD*('*a*) ∗ *CARD*('*b*)

⟨*proof*⟩

lemma *card-UNIV-sum*: $CARD('a + 'b) = (\mathit{if}\ CARD('a) \neq 0 \wedge CARD('b) \neq 0\ \mathit{then}\ CARD('a) + CARD('b)\ \mathit{else}\ 0)$
⟨*proof*⟩

lemma *card-sum* [*simp*]: $CARD('a + 'b) = CARD('a::\mathit{finite}) + CARD('b::\mathit{finite})$
⟨*proof*⟩

lemma *card-UNIV-option*: $CARD('a\ option) = (\mathit{if}\ CARD('a) = 0\ \mathit{then}\ 0\ \mathit{else}\ CARD('a) + 1)$
⟨*proof*⟩

lemma *card-option* [*simp*]: $CARD('a\ option) = Suc\ CARD('a::\mathit{finite})$
⟨*proof*⟩

lemma *card-UNIV-set*: $CARD('a\ set) = (\mathit{if}\ CARD('a) = 0\ \mathit{then}\ 0\ \mathit{else}\ 2 \mathbin{^\frown} CARD('a))$
⟨*proof*⟩

lemma *card-set* [*simp*]: $CARD('a\ set) = 2 \mathbin{^\frown} CARD('a::\mathit{finite})$
⟨*proof*⟩

lemma *card-nat* [*simp*]: $CARD(nat) = 0$
⟨*proof*⟩

lemma *card-fun*: $CARD('a \Rightarrow 'b) = (\mathit{if}\ CARD('a) \neq 0 \wedge CARD('b) \neq 0 \vee CARD('b) = 1\ \mathit{then}\ CARD('b) \mathbin{^\frown} CARD('a)\ \mathit{else}\ 0)$
⟨*proof*⟩

corollary *finite-UNIV-fun*:
$\mathit{finite}\ (UNIV :: ('a \Rightarrow 'b)\ set) \longleftrightarrow$
$\mathit{finite}\ (UNIV :: 'a\ set) \wedge \mathit{finite}\ (UNIV :: 'b\ set) \vee CARD('b) = 1$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *card-literal*: $CARD(String.literal) = 0$
⟨*proof*⟩

9.2 Classes with at least 1 and 2

Class finite already captures "at least 1"

lemma *zero-less-card-finite* [*simp*]: $0 < CARD('a::\mathit{finite})$
⟨*proof*⟩

lemma *one-le-card-finite* [*simp*]: $Suc\ 0 \leq CARD('a::\mathit{finite})$
⟨*proof*⟩

class *CARD-1* =

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

subclass *finite*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *one-less-int-card*: *1* < *int CARD*('*a*::*card2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⟨*proof*⟩
end

instantiation *int* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*int*) *False*
definition *card-UNIV* = *Phantom*(*int*) *0*
instance ⟨*proof*⟩
end

instantiation *natural* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*natural*) *False*
definition *card-UNIV* = *Phantom*(*natural*) *0*
instance
⟨*proof*⟩
end

instantiation *integer* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*integer*) *False*
definition *card-UNIV* = *Phantom*(*integer*) *0*
instance
⟨*proof*⟩
end

instantiation *list* :: (*type*) *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a list*) *False*
definition *card-UNIV* = *Phantom*('*a list*) *0*
instance ⟨*proof*⟩
end

instantiation *unit* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*unit*) *True*
definition *card-UNIV* = *Phantom*(*unit*) *1*
instance ⟨*proof*⟩
end

instantiation *bool* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*bool*) *True*
definition *card-UNIV* = *Phantom*(*bool*) *2*
instance ⟨*proof*⟩
end

instantiation *char* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*char*) *True*
definition *card-UNIV* = *Phantom*(*char*) *256*
instance ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
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
⟨*proof*⟩
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 ⟨*proof*⟩
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
⟨*proof*⟩
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 ⟨*proof*⟩
end

instantiation *option* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a option*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance ⟨*proof*⟩
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 ⟨*proof*⟩
end

instantiation *String.literal* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*String.literal*) *False*
definition *card-UNIV* = *Phantom*(*String.literal*) *0*
instance
⟨*proof*⟩
end

instantiation *set* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a set*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance ⟨*proof*⟩
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 ⟨*proof*⟩
end

lemma *UNIV-finite-1*: *UNIV* = *set* [*finite-1*.a_1]
⟨*proof*⟩

lemma *UNIV-finite-2*: *UNIV* = *set* [*finite-2*.a_1, *finite-2*.a_2]
⟨*proof*⟩

lemma *UNIV-finite-3*: *UNIV* = *set* [*finite-3*.a_1, *finite-3*.a_2, *finite-3*.a_3]
⟨*proof*⟩

lemma *UNIV-finite-4*: *UNIV* = *set* [*finite-4*.a_1, *finite-4*.a_2, *finite-4*.a_3, *finite-4*.a_4]
⟨*proof*⟩

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]
⟨*proof*⟩

instantiation *Enum.finite-1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-1*) *1*
instance
⟨*proof*⟩
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
⟨*proof*⟩
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
⟨*proof*⟩
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
⟨*proof*⟩
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
⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *card-UNIV'-code* [*code*]:
card-UNIV' = *card-UNIV*
⟨*proof*⟩

end

lemma *card-Compl*:
finite A ⟹ *card* (− *A*) = *card* (*UNIV* :: '*a set*) − *card* (*A* :: '*a set*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩ **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*)
⟨*proof*⟩ **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*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

notepad begin — test code setup
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *missing-pattern-match-code* [*code*, *code-unfold*]:
missing-pattern-match = *Code.abort*
⟨*proof*⟩

⟨*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* ⟨*proof*⟩
setup-lifting *type-definition-lazy*
lift-definition *delay* :: (*unit* ⇒ *'a*) ⇒ *'a lazy* **is** λ*f*. *f* () ⟨*proof*⟩
lift-definition *force* :: *'a lazy* ⇒ *'a* **is** λ*x*. *x* ⟨*proof*⟩

code-datatype *delay*
lemma *force-delay* [*code*]: *force* (*delay f*) = *f* () ⟨*proof*⟩
lemma *delay-force*: *delay* (λ-. *force s*) = *s* ⟨*proof*⟩

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*) (λ*T U*. *typerep.Typerep* (*STR ''fun''*) [*T*, *U*]) (λ*T*. *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*
⟨*proof*⟩

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* ⟨*proof*⟩
⟨*ML*⟩

12.2 Implementation

⟨*ML*⟩

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*)
⟨*proof*⟩

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*: $x = (y :: xml\text{-}tree)$
⟨*proof*⟩

context begin
⟨*ML*⟩

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*
⟨*proof*⟩

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*)]
⟨*proof*⟩

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*)]
⟨*proof*⟩

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*⟩

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* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
  where *combine-per P R* = $(\lambda x\ y.\ P\ x \wedge P\ y) \sqcap R$

lemma *combine-per-simp* [*simp*]:
  combine-per P R x y $\longleftrightarrow$ *P x* $\wedge$ *P y* $\wedge$ *x* $\approx$ *y* **for** *R* (**infixl** ‹$\approx$› *50*)
  ⟨*proof*⟩

lemma *combine-per-top* [*simp*]: *combine-per* $\top$ *R* = *R*
  ⟨*proof*⟩

lemma *combine-per-eq* [*simp*]: *combine-per P HOL.eq* = *HOL.eq* $\sqcap$ $(\lambda x\ y.\ P\ x)$
  ⟨*proof*⟩

lemma *symp-combine-per*: *symp R* $\Longrightarrow$ *symp* (*combine-per P R*)
  ⟨*proof*⟩

lemma *transp-combine-per*: *transp R* $\Longrightarrow$ *transp* (*combine-per P R*)
  ⟨*proof*⟩

lemma *combine-perI*: *P x* $\Longrightarrow$ *P y* $\Longrightarrow$ *x* $\approx$ *y* $\Longrightarrow$ *combine-per P R x y* **for** *R*
(**infixl** ‹$\approx$› *50*)
  ⟨*proof*⟩

lemma *symp-combine-per-symp*: *symp R* $\Longrightarrow$ *symp* (*combine-per P R*)
  ⟨*proof*⟩

lemma *transp-combine-per-transp*: *transp R* $\Longrightarrow$ *transp* (*combine-per P R*)
  ⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *linorder-chain* [*simp*, *intro*!]:
fixes *Y* :: - :: *linorder set*
shows *Complete-Partial-Order.chain* (≤) *Y*
⟨*proof*⟩

lemma *fun-lub-apply*: ⋀*Sup*. *fun-lub Sup Y x* = *Sup* ((λ*f*. *f x*) ‘ *Y*)
⟨*proof*⟩

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* {} = (λ-. *lub* {})
⟨*proof*⟩

lemma *chain-fun-ordD*:
assumes *Complete-Partial-Order.chain* (*fun-ord le*) *Y*
shows *Complete-Partial-Order.chain le* ((λ*f*. *f x*) ‘ *Y*)
⟨*proof*⟩

lemma *chain-Diff*:
Complete-Partial-Order.chain ord A
⟹ *Complete-Partial-Order.chain ord* (*A* − *B*)
⟨*proof*⟩

lemma *chain-rel-prodD1*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain orda* (*fst* ‘ *Y*)
⟨*proof*⟩

lemma *chain-rel-prodD2*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain ordb* (*snd* ‘ *Y*)
⟨*proof*⟩

context *ccpo* **begin**

lemma *ccpo-fun*: *class.ccpo* (*fun-lub Sup*) (*fun-ord* (≤)) (*mk-less* (*fun-ord* (≤)))
⟨*proof*⟩

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain* (≤) *Y* ⟹ *Sup Y* ≤ *x* ⟷ (∀ *y*∈*Y*. *y* ≤ *x*)
⟨*proof*⟩

lemma *Sup-minus-bot*:

assumes *chain*: *Complete-Partial-Order.chain* $(\leq)$ A
shows $\bigsqcup(A - \{\bigsqcup\{\}\}) = \bigsqcup A$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *mono-lub*:
fixes *le-b* (**infix** ‹$\sqsubseteq$› *60*)
assumes *chain*: *Complete-Partial-Order.chain* (*fun-ord* $(\leq)$) Y
and *mono*: $\bigwedge f.\ f \in Y \Longrightarrow$ *monotone le-b* $(\leq)$ f
shows *monotone* $(\sqsubseteq)$ $(\leq)$ (*fun-lub Sup* Y)
⟨*proof*⟩

context
fixes *le-b* (**infix** ‹$\sqsubseteq$› *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*⟩

lemma *diag-Sup*: $\bigsqcup((\lambda x.\ \bigsqcup(f\ x\ `\ Y))\ `\ Y) = \bigsqcup((\lambda x.\ f\ x\ x)\ `\ Y)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

lemma *Sup-image-mono-le*:
fixes *le-b* (**infix** ‹$\sqsubseteq$› *60*) **and** *Sup-b* (‹$\bigvee$›)
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* $(f\ `\ Y) \leq f\ (\bigvee Y)$
⟨*proof*⟩

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 $\bigsqcup((\lambda x.\ \bigsqcup(x\ `\ Y))\ `\ Z) = \bigsqcup((\lambda x.\ \bigsqcup((\lambda f.\ f\ x)\ `\ Z))\ `\ Y)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *fixp-mono*:
assumes *fg*: *fun-ord* $(\leq)$ f g

and *f*: *monotone* (≤) (≤) *f*
and *g*: *monotone* (≤) (≤) *g*
shows *ccpo-class.fixp f* ≤ *ccpo-class.fixp g*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

end

end

lemma *monotone2monotone*:
assumes *2*: ⋀*x*. *monotone ordb ordc* (λ*y*. *f x y*)
and *t*: *monotone orda ordb* (λ*x*. *t x*)
and *1*: ⋀*y*. *monotone orda ordc* (λ*x*. *f x y*)
and *trans*: *transp ordc*
shows *monotone orda ordc* (λ*x*. *f x* (*t x*))
⟨*proof*⟩

15.1 Continuity

definition *cont* :: (*'a set* ⇒ *'a*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b set* ⇒ *'b*) ⇒ (*'b* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool*
where
cont luba orda lubb ordb f ⟷
(∀ *Y*. *Complete-Partial-Order.chain orda Y* ⟶ *Y* ≠ {} ⟶ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*))

definition *mcont* :: (*'a set* ⇒ *'a*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b set* ⇒ *'b*) ⇒ (*'b* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ *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*⟩

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 $(\bigwedge f.\ \mathit{monotone}\ R\ S\ f \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{monotone}\ R\ S\ g \Longrightarrow \mathit{thesis}$
⟨*proof*⟩

end

declare *if-mono*[*simp*]

lemma *monotone-id'* [*cont-intro*]: $\mathit{monotone}\ \mathit{ord}\ \mathit{ord}\ (\lambda x.\ x)$
⟨*proof*⟩

lemma *monotone-applyI*:
$\mathit{monotone}\ \mathit{orda}\ \mathit{ordb}\ F \Longrightarrow \mathit{monotone}\ (\mathit{fun\text{-}ord}\ \mathit{orda})\ \mathit{ordb}\ (\lambda f.\ F\ (f\ x))$
⟨*proof*⟩

lemma *monotone-if-fun* [*partial-function-mono*]:
$[\![\ \mathit{monotone}\ (\mathit{fun\text{-}ord}\ \mathit{orda})\ (\mathit{fun\text{-}ord}\ \mathit{ordb})\ F;\ \mathit{monotone}\ (\mathit{fun\text{-}ord}\ \mathit{orda})\ (\mathit{fun\text{-}ord}\ \mathit{ordb})\ G\ ]\!]$
$\Longrightarrow \mathit{monotone}\ (\mathit{fun\text{-}ord}\ \mathit{orda})\ (\mathit{fun\text{-}ord}\ \mathit{ordb})\ (\lambda f\ n.\ \mathit{if}\ c\ n\ \mathit{then}\ F\ f\ n\ \mathit{else}\ G\ f\ n)$
⟨*proof*⟩

lemma *monotone-fun-apply-fun* [*partial-function-mono*]:
$\mathit{monotone}\ (\mathit{fun\text{-}ord}\ (\mathit{fun\text{-}ord}\ \mathit{ord}))\ (\mathit{fun\text{-}ord}\ \mathit{ord})\ (\lambda f\ n.\ f\ t\ (g\ n))$
⟨*proof*⟩

lemma *monotone-fun-ord-apply*:
$\mathit{monotone}\ \mathit{orda}\ (\mathit{fun\text{-}ord}\ \mathit{ordb})\ f \longleftrightarrow (\forall x.\ \mathit{monotone}\ \mathit{orda}\ \mathit{ordb}\ (\lambda y.\ f\ y\ x))$
⟨*proof*⟩

context *preorder* **begin**

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

lemma *monotone-const* [*simp*, *cont-intro*]: $\mathit{monotone}\ \mathit{ord}\ (\leq)\ (\lambda\text{-}.\ c)$
⟨*proof*⟩

end

lemma *transp-le* [*cont-intro*, *simp*]:
class.preorder ord (*mk-less ord*) ⟹ *transp ord*
⟨*proof*⟩

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

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

lemma *transp-le* [*cont-intro*, *simp*]: *transp leq*
⟨*proof*⟩

lemma *preorder* [*cont-intro*, *simp*]: *class.preorder leq* (*mk-less leq*)
⟨*proof*⟩

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

end

lemma *contI* [*intro?*]:
(⋀*Y*. ⟦ *Complete-Partial-Order.chain orda Y*; *Y* ≠ {} ⟧ ⟹ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*))
⟹ *cont luba orda lubb ordb f*
⟨*proof*⟩

lemma *contD*:
⟦ *cont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; *Y* ≠ {} ⟧
⟹ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*)
⟨*proof*⟩

lemma *cont-id* [*simp*, *cont-intro*]: ⋀*Sup*. *cont Sup ord Sup ord id*
⟨*proof*⟩

lemma *cont-id′* [*simp*, *cont-intro*]: ⋀*Sup*. *cont Sup ord Sup ord* (λ*x*. *x*)
⟨*proof*⟩

lemma *cont-applyI* [*cont-intro*]:
assumes *cont*: *cont luba orda lubb ordb g*
shows *cont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* (λ*f*. *g* (*f x*))
⟨*proof*⟩

lemma *call-cont*: *cont* (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λ*f*. *f t*)
⟨*proof*⟩

lemma *cont-if* [*cont-intro*]:
⟦ *cont luba orda lubb ordb f*; *cont luba orda lubb ordb g* ⟧
⟹ *cont luba orda lubb ordb* (λ*x*. *if c then f x else g x*)
⟨*proof*⟩

lemma *mcontI* [*intro?*]:
⟦ *monotone orda ordb f*; *cont luba orda lubb ordb f* ⟧ ⟹ *mcont luba orda lubb ordb f*
⟨*proof*⟩

lemma *mcont-mono*: *mcont luba orda lubb ordb f* ⟹ *monotone orda ordb f*
⟨*proof*⟩

lemma *mcont-cont* [*simp*]: *mcont luba orda lubb ordb f* ⟹ *cont luba orda lubb ordb f*
⟨*proof*⟩

lemma *mcont-monoD*:
⟦ *mcont luba orda lubb ordb f*; *orda x y* ⟧ ⟹ *ordb* (*f x*) (*f y*)
⟨*proof*⟩

lemma *mcont-contD*:
⟦ *mcont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; *Y* ≠ {} ⟧
⟹ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*)
⟨*proof*⟩

lemma *mcont-call* [*cont-intro*, *simp*]:
mcont (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λ*f*. *f t*)
⟨*proof*⟩

lemma *mcont-id′* [*cont-intro*, *simp*]: *mcont lub ord lub ord* (λ*x*. *x*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

context *ccpo* **begin**

lemma *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup* (≤) (λ*x*. *c*)
⟨*proof*⟩

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda Sup (≤) (λ*x*. *c*)
⟨*proof*⟩

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*⟩

lemma *mcont2mcont'*:
⟦ ⋀*x*. *mcont lub' ord' Sup* (≤) (λ*y*. *f x y*);
⋀*y*. *mcont lub ord Sup* (≤) (λ*x*. *f x y*);
mcont lub ord lub' ord' (λ*y*. *t y*) ⟧
⟹ *mcont lub ord Sup* (≤) (λ*x*. *f x* (*t x*))
⟨*proof*⟩

lemma *mcont2mcont*:
⟦*mcont lub' ord' Sup* (≤) (λ*x*. *f x*); *mcont lub ord lub' ord'* (λ*x*. *t x*)⟧
⟹ *mcont lub ord Sup* (≤) (λ*x*. *f* (*t x*))
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *iterates-mcont*:

assumes *f*: $f \in$ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord* ($\leq$)) *F*
and *mono*: $\bigwedge f$. *mcont lub* ($\sqsubseteq$) *Sup* ($\leq$) $f \Longrightarrow$ *mcont lub* ($\sqsubseteq$) *Sup* ($\leq$) (*F f*)
shows *mcont lub* ($\sqsubseteq$) *Sup* ($\leq$) *f*
⟨*proof*⟩

lemma *fixp-preserves-mcont*:
assumes *mono*: $\bigwedge x$. *monotone* (*fun-ord* ($\leq$)) ($\leq$) (λf. *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*)
⟨*proof*⟩

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* ($\leq$)) ($\leq$) (λf. *U* (*F* (*C f*)) *x*)
and *eq*: *f* $\equiv$ *C* (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* ($\leq$)) (λf. *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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

end

lemmas *fixp-preserves-mono1* = *fixp-preserves-mono-uc*[*of* λx. *x* - λx. *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* λf. *case-prod* (*case-prod f*) - λf. *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mono4* =
fixp-preserves-mono-uc[*of* λf. *case-prod* (*case-prod* (*case-prod f*)) - λf. *curry* (*curry* (*curry f*)), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]

lemmas *fixp-preserves-mcont1* = *fixp-preserves-mcont-uc*[*of* λx. *x* - λx. *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*)
⟨*proof*⟩

lemma (**in** *ccpo*) *mcont-if-bot*:
fixes *bot* **and** *lub* (‹$\bigsqcup$›) **and** *ord* (**infix** ‹$\sqsubseteq$› *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; $Y \neq \{\}$; $\bigwedge x.\ x \in Y \Longrightarrow \neg\ x \le bound\ ]\!] \Longrightarrow f\ (\bigsqcup Y) = \bigsqcup (f\ `\ Y)$
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*⟩

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

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda lub leq ($\lambda x.$ *c*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* $(\lambda x.\ P\ (f\ x))$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *admissible-imp* [*cont-intro*]:
(*Q* ⟹ *ccpo.admissible lub ord* (λ*x*. *P x*))
⟹ *ccpo.admissible lub ord* (λ*x*. *Q* ⟶ *P x*)
⟨*proof*⟩

lemma *admissible-not-mem'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-not-mem*: *ccpo.admissible Union* (⊆) (λ*A*. *x* ∉ *A*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible Sup* (≤) (λ*x*. ¬ (≤) *x y*)
⟨*proof*⟩

end

lemma (**in** *preorder*) *preorder* [*cont-intro*, *simp*]: *class.preorder* (≤) (*mk-less* (≤))
⟨*proof*⟩

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

⟨*ML*⟩

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*

⟨*ML*⟩

context *ccpo* **begin**

lemma *compactI*:
assumes *ccpo.admissible Sup* (≤) (λ*y*. ¬ *x* ≤ *y*)
shows *ccpo.compact Sup* (≤) *x*
⟨*proof*⟩

lemma *compact-bot*:
assumes $x = Sup\ \{\}$
shows *ccpo.compact Sup* $(\leq)$ x
⟨*proof*⟩

end

lemma *admissible-compact-neq'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-compact-neq*: *ccpo.compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ k \neq x)$
⟨*proof*⟩

lemma *admissible-neq-compact'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-neq-compact*: *ccpo.compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ x \neq k)$
⟨*proof*⟩

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* $(\leq)$ P
and *mono*: *monotone* $(\leq)$ $(\leq)$ f
and *bot*: $P\ (\bigsqcup \{\})$
and *step*: $\bigwedge x.\ [\![\ x \leq$ *ccpo-class.fixp* $f;\ P\ x\ ]\!] \Longrightarrow P\ (f\ x)$
shows P (*ccpo-class.fixp* f)
⟨*proof*⟩

end

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

lemma *fixp-strong-induct-uc*:
fixes $F :: 'c \Rightarrow 'c$
and $U :: 'c \Rightarrow 'b \Rightarrow 'a$
and $C :: ('b \Rightarrow 'a) \Rightarrow 'c$
and $P :: ('b \Rightarrow 'a) \Rightarrow bool$
assumes *mono*: $\bigwedge x.$ *mono-body* $(\lambda f.\ U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C$ (*fixp-fun* $(\lambda f.\ U\ (F\ (C\ f))))$
and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
and *adm*: *ccpo.admissible lub-fun le-fun P*
and *bot*: $P\ (\lambda\text{-}.$ *lub* $\{\})$
and *step*: $\bigwedge f'.\ [\![\ P\ (U\ f');$ *le-fun* $(U\ f')\ (U\ f)\ ]\!] \Longrightarrow P\ (U\ (F\ f'))$

shows $P\ (U\ f)$
⟨*proof*⟩

end

15.3 (=) as order

definition *lub-singleton* :: $('a\ set \Rightarrow 'a) \Rightarrow bool$
where *lub-singleton lub* $\longleftrightarrow (\forall\, a.\ lub\ \{a\} = a)$

definition *the-Sup* :: $'a\ set \Rightarrow 'a$
where *the-Sup* $A = (THE\ a.\ a \in A)$

lemma *lub-singleton-the-Sup* [*cont-intro*, *simp*]: *lub-singleton the-Sup*
⟨*proof*⟩

lemma (**in** *ccpo*) *lub-singleton*: *lub-singleton Sup*
⟨*proof*⟩

lemma (**in** *partial-function-definitions*) *lub-singleton* [*cont-intro*, *simp*]: *lub-singleton lub*
⟨*proof*⟩

lemma *preorder-eq* [*cont-intro*, *simp*]:
class.preorder (=) (*mk-less* (=))
⟨*proof*⟩

lemma *monotone-eqI* [*cont-intro*]:
assumes *class.preorder ord* (*mk-less ord*)
shows *monotone* (=) *ord f*
⟨*proof*⟩

lemma *cont-eqI* [*cont-intro*]:
fixes $f :: 'a \Rightarrow 'b$
assumes *lub-singleton lub*
shows *cont the-Sup* (=) *lub ord f*
⟨*proof*⟩

lemma *mcont-eqI* [*cont-intro*, *simp*]:
⟦ *class.preorder ord* (*mk-less ord*); *lub-singleton lub* ⟧
⟹ *mcont the-Sup* (=) *lub ord f*
⟨*proof*⟩

15.4 ccpo for products

definition *prod-lub* :: $('a\ set \Rightarrow 'a) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('a \times 'b)\ set \Rightarrow 'a \times 'b$
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*)

⟨*proof*⟩

lemma *prod-lub-empty* [*simp*]: *prod-lub luba lubb* {} = (*luba* {}, *lubb* {})
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*: $\bigwedge y.$ *cont luba orda Sup* $(\leq)$ $(\lambda x.\ f\ (x,\ y))$
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* $(\leq)$ *f*
⟨*proof*⟩

lemma *cont-case-prodI*:
assumes *monotone* (*rel-prod orda ordb*) $(\leq)$ (*case-prod f*)
and $\bigwedge x.$ *cont lubb ordb Sup* $(\leq)$ $(\lambda y.\ f\ x\ y)$
and $\bigwedge y.$ *cont luba orda Sup* $(\leq)$ $(\lambda x.\ f\ x\ y)$
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* $(\leq)$ (*case-prod f*)
⟨*proof*⟩

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* ⟧
$\Longrightarrow$ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup* $(\leq)$ (*case-prod f*) $\longleftrightarrow$
$(\forall x.$ *cont lubb ordb Sup* $(\leq)$ $(\lambda y.\ f\ x\ y)) \wedge (\forall y.$ *cont luba orda Sup* $(\leq)$ $(\lambda x.\ f\ x\ y))$
⟨*proof*⟩

end

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

lemma *mono2mono2*:
assumes *f*: *monotone* (*rel-prod ordb ordc*) *leq* $(\lambda(x,\ y).\ f\ x\ y)$
and *t*: *monotone orda ordb* $(\lambda x.\ t\ x)$
and t': *monotone orda ordc* $(\lambda x.\ t'\ x)$
shows *monotone orda leq* $(\lambda x.\ f\ (t\ x)\ (t'\ x))$
⟨*proof*⟩

lemma *cont-case-prodI* [*cont-intro*]:
⟦ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
$\bigwedge x.$ *cont lubb ordb lub leq* $(\lambda y.\ f\ x\ y)$;
$\bigwedge y.$ *cont luba orda lub leq* $(\lambda x.\ f\ x\ y)$;
class.preorder orda (*mk-less orda*);
class.preorder ordb (*mk-less ordb*) ⟧
$\Longrightarrow$ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*)
⟨*proof*⟩

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* ⟧
$\Longrightarrow$ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) $\longleftrightarrow$

$(\forall x.\ cont\ lubb\ ordb\ lub\ leq\ (\lambda y.\ f\ x\ y)) \wedge (\forall y.\ cont\ luba\ orda\ lub\ leq\ (\lambda x.\ f\ x\ y))$
⟨*proof*⟩

lemma *mcont-case-prod-iff* [*simp*]:
⟦ *class.preorder orda* (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
$\Longrightarrow$ *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) $\longleftrightarrow$
$(\forall x.\ mcont\ lubb\ ordb\ lub\ leq\ (\lambda y.\ f\ x\ y)) \wedge (\forall y.\ mcont\ luba\ orda\ lub\ leq\ (\lambda x.\ f\ x\ y))$
⟨*proof*⟩

end

lemma *mono2mono-case-prod* [*cont-intro*]:
assumes $\bigwedge x\ y.$ *monotone orda ordb* $(\lambda f.\ pair\ f\ x\ y)$
shows *monotone orda ordb* $(\lambda f.\ case\text{-}prod\ (pair\ f)\ x)$
⟨*proof*⟩

15.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: *class.ccpo Sup* $(\leq)$ $(<)$
⟨*proof*⟩

lemma *complete-lattice-ccpo′*: *class.ccpo Sup* $(\leq)$ (*mk-less* $(\leq)$)
⟨*proof*⟩

lemma *complete-lattice-partial-function-definitions*:
partial-function-definitions $(\leq)$ *Sup*
⟨*proof*⟩

lemma *complete-lattice-partial-function-definitions-dual*:
partial-function-definitions $(\geq)$ *Inf*
⟨*proof*⟩

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* $(\leq)$ $(\lambda x.\ f\ x \sqcap g\ x)$
⟨*proof*⟩

lemma *mcont-const* [*simp*]: *mcont lub ord Sup* $(\leq)$ $(\lambda\text{-}.\ c)$
⟨*proof*⟩

lemma *mono2mono-sup*:
assumes *f*: *monotone ord* (≤) (λ*x*. *f x*)
and *g*: *monotone ord* (≤) (λ*x*. *g x*)
shows *monotone ord* (≤) (λ*x*. *f x* ⊔ *g x*)
⟨*proof*⟩

lemma *Sup-image-sup*:
assumes $Y \neq \{\}$
shows $\bigsqcup((\sqcup)\ x\ `\ Y) = x \sqcup \bigsqcup Y$
⟨*proof*⟩

lemma *mcont-sup1*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *x* ⊔ *y*)
⟨*proof*⟩

lemma *mcont-sup2*: *mcont Sup* (≤) *Sup* (≤) (λ*x*. *x* ⊔ *y*)
⟨*proof*⟩

lemma *mcont2mcont-sup* [*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*)
⟨*proof*⟩

end

lemmas [*cont-intro*] = *admissible-leI*[*OF complete-lattice-ccpo'*]

context *complete-distrib-lattice* **begin**

lemma *mcont-inf1*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *x* ⊓ *y*)
⟨*proof*⟩

lemma *mcont-inf2*: *mcont Sup* (≤) *Sup* (≤) (λ*x*. *x* ⊓ *y*)
⟨*proof*⟩

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*)
⟨*proof*⟩

end

interpretation *lfp*: *partial-function-definitions* (≤) :: - :: *complete-lattice* ⇒ - *Sup*
⟨*proof*⟩

⟨*ML*⟩

interpretation *gfp*: *partial-function-definitions* (≥) :: - :: *complete-lattice* ⇒ - *Inf*

⟨*proof*⟩

⟨*ML*⟩

lemma *insert-mono* [*partial-function-mono*]:
monotone (*fun-ord* (⊆)) (⊆) *A* ⟹ *monotone* (*fun-ord* (⊆)) (⊆) (λ*y*. *insert x* (*A y*))
⟨*proof*⟩

lemma *mono2mono-insert* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-insert*: *monotone* (⊆) (⊆) (*insert x*)
⟨*proof*⟩

lemma *mcont2mcont-insert*[*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-insert*: *mcont Union* (⊆) *Union* (⊆) (*insert x*)
⟨*proof*⟩

lemma *mono2mono-image* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-image*: *monotone* (⊆) (⊆) ((') *f*)
⟨*proof*⟩

lemma *cont-image*: *cont Union* (⊆) *Union* (⊆) ((') *f*)
⟨*proof*⟩

lemma *mcont2mcont-image* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-image*: *mcont Union* (⊆) *Union* (⊆) ((') *f*)
⟨*proof*⟩

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro*, *simp*]:
monotone ord (⊆) *f* ⟹ *monotone ord* (≤) (λ*x*. ⨆*f x*)
⟨*proof*⟩

lemma *cont-Sup*:
assumes *cont lub ord Union* (⊆) *f*
shows *cont lub ord Sup* (≤) (λ*x*. ⨆*f x*)
⟨*proof*⟩

lemma *mcont-Sup*: *mcont lub ord Union* (⊆) *f* ⟹ *mcont lub ord Sup* (≤) (λ*x*. ⨆*f x*)
⟨*proof*⟩

lemma *monotone-SUP*:
⟦ *monotone ord* (⊆) *f*; ⋀*y*. *monotone ord* (≤) (λ*x*. *g x y*) ⟧ ⟹ *monotone ord* (≤) (λ*x*. ⨆*y*∈*f x*. *g x y*)
⟨*proof*⟩

lemma *monotone-SUP2*:

$(\bigwedge y.\ y \in A \Longrightarrow \mathit{monotone}\ \mathit{ord}\ (\leq)\ (\lambda x.\ g\ x\ y)) \Longrightarrow \mathit{monotone}\ \mathit{ord}\ (\leq)\ (\lambda x.\ \bigsqcup y{\in}A.\ g\ x\ y)$
⟨*proof*⟩

lemma *cont-SUP*:
assumes *f*: *mcont lub ord Union* (⊆) *f*
and *g*: $\bigwedge y.$ *mcont lub ord Sup* (≤) (λ*x*. *g x y*)
shows *cont lub ord Sup* (≤) (λ*x*. $\bigsqcup y{\in}f\ x.$ *g x y*)
⟨*proof*⟩

lemma *mcont-SUP* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Union* (⊆) *f*; $\bigwedge y.$ *mcont lub ord Sup* (≤) (λ*x*. *g x y*) ⟧
⟹ *mcont lub ord Sup* (≤) (λ*x*. $\bigsqcup y{\in}f\ x.$ *g x y*)
⟨*proof*⟩

end

lemma *admissible-Ball* [*cont-intro*, *simp*]:
⟦ $\bigwedge x.$ *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*. $\forall x{\in}f\ A.$ *P A x*)
⟨*proof*⟩

lemma *admissible-Bex'*[*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-Bex*: *ccpo.admissible Union* (⊆) (λ*A*. $\exists x{\in}A.$ *P x*)
⟨*proof*⟩

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* ⟨*proof*⟩
interpretation *b*: *ccpo lubb ordb mk-less ordb* ⟨*proof*⟩

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*⟩

interpretation *ab*: *ccpo prod-lub luba lubb rel-prod orda ordb mk-less* (*rel-prod orda ordb*)

⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

end

lemma *parallel-fixp-induct-uc*:
assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: ⋀*x*. *monotone* (*fun-ord orda*) *orda* (λ*f*. *U1* (*F* (*C1 f*)) *x*)
and *G*: ⋀*y*. *monotone* (*fun-ord ordb*) *ordb* (λ*g*. *U2* (*G* (*C2 g*)) *y*)
and *eq1*: *f* ≡ *C1* (*ccpo.fixp* (*fun-lub luba*) (*fun-ord orda*) (λ*f*. *U1* (*F* (*C1 f*))))
and *eq2*: *g* ≡ *C2* (*ccpo.fixp* (*fun-lub lubb*) (*fun-ord ordb*) (λ*g*. *U2* (*G* (*C2 g*))))
and *inverse*: ⋀*f*. *U1* (*C1 f*) = *f*
and *inverse2*: ⋀*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*)) (λ*x*. *P* (*fst x*) (*snd x*))
and *bot*: *P* (λ-. *luba* {}) (λ-. *lubb* {})
and *step*: ⋀*f g*. *P* (*U1 f*) (*U2 g*) ⟹ *P* (*U1* (*F f*)) (*U2* (*G g*))
shows *P* (*U1 f*) (*U2 g*)
⟨*proof*⟩

lemmas *parallel-fixp-induct-1-1* = *parallel-fixp-induct-uc*[
of - - - - λ*x*. *x* - λ*x*. *x* λ*x*. *x* - λ*x*. *x*,
OF - - - - - - *refl refl*]

lemmas *parallel-fixp-induct-2-2* = *parallel-fixp-induct-uc*[
of - - - - *case-prod* - *curry case-prod* - *curry*,
where *P*=λ*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*
⟨*proof*⟩

lemma *mcont-fst*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *luba orda fst*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *monotone-snd*: *monotone* (*rel-prod orda ordb*) *ordb snd*
⟨*proof*⟩

lemma *mcont-snd*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordb snd*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *monotone-Pair*:
⟦ *monotone ord orda f*; *monotone ord ordb g* ⟧
⟹ *monotone ord* (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *compact-flat-lub* [*cont-intro*]: *ccpo.compact* (*flat-lub x*) (*flat-ord x*) *y*
⟨*proof*⟩

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* ===> (=)) (=) (=)
⟨*proof*⟩

lemma *bi-total-Rel-match* [*parametricity-preprocess*]:
bi-total A = *Rel-match* ((*A* ===> (=)) ===> (=)) *All All*
⟨*proof*⟩

lemma *is-equality-Rel*: *is-equality A* ⟹ *Transfer.Rel A t t*
⟨*proof*⟩

lemma *Rel-Rel-match*: *Transfer.Rel R x y* ⟹ *Rel-match R x y*
⟨*proof*⟩

lemma *Rel-match-Rel*: *Rel-match R x y* ⟹ *Transfer.Rel R x y*
⟨*proof*⟩

lemma *Rel-Rel-match-eq*: *Transfer.Rel R x y* = *Rel-match R x y*
⟨*proof*⟩

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*)
⟨*proof*⟩

end

⟨*ML*⟩

end
theory *Confluence* **imports**
Main
begin

16 Confluence

definition *semiconfluentp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
semiconfluentp $r \longleftrightarrow r^{-1-1}\ OO\ r^{**} \leq r^{**}\ OO\ r^{-1-1**}$

definition *confluentp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
confluentp $r \longleftrightarrow r^{-1-1**}\ OO\ r^{**} \leq r^{**}\ OO\ r^{-1-1**}$

definition *strong-confluentp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
strong-confluentp $r \longleftrightarrow r^{-1-1}\ OO\ r \leq r^{**}\ OO\ (r^{-1-1})^{==}$

lemma *semiconfluentpI* [*intro?*]:
semiconfluentp r **if** $\bigwedge x\ y\ z.\ [\![\ r\ x\ y;\ r^{**}\ x\ z\ ]\!] \Longrightarrow \exists u.\ r^{**}\ y\ u \wedge r^{**}\ z\ u$
⟨*proof*⟩

lemma *semiconfluentpD*: $\exists u.\ r^{**}\ y\ u \wedge r^{**}\ z\ u$ **if** *semiconfluentp* $r\ r\ x\ y\ r^{**}\ x\ z$
⟨*proof*⟩

lemma *confluentpI*:
confluentp r **if** $\bigwedge x\ y\ z.\ [\![\ r^{**}\ x\ y;\ r^{**}\ x\ z\ ]\!] \Longrightarrow \exists u.\ r^{**}\ y\ u \wedge r^{**}\ z\ u$
⟨*proof*⟩

lemma *confluentpD*: $\exists u.\ r^{**}\ y\ u \wedge r^{**}\ z\ u$ **if** *confluentp* $r\ r^{**}\ x\ y\ r^{**}\ x\ z$
⟨*proof*⟩

lemma *strong-confluentpI* [*intro?*]:
strong-confluentp r **if** $\bigwedge x\ y\ z.\ [\![\ r\ x\ y;\ r\ x\ z\ ]\!] \Longrightarrow \exists u.\ r^{**}\ y\ u \wedge r^{==}\ z\ u$
⟨*proof*⟩

lemma *strong-confluentpD*: $\exists u.\ r^{**}\ y\ u \wedge r^{==}\ z\ u$ **if** *strong-confluentp* $r\ r\ x\ y\ r\ x\ z$
⟨*proof*⟩

lemma *semiconfluentp-imp-confluentp*: *confluentp* r **if** r: *semiconfluentp* r
⟨*proof*⟩

lemma *confluentp-imp-semiconfluentp*: *semiconfluentp* r **if** *confluentp* r
⟨*proof*⟩

lemma *confluentp-eq-semiconfluentp*: *confluentp* $r \longleftrightarrow$ *semiconfluentp* r
⟨*proof*⟩

lemma *confluentp-conv-strong-confluentp-rtranclp*:
confluentp $r \longleftrightarrow$ *strong-confluentp* (r^{**})
⟨*proof*⟩

lemma *strong-confluentp-into-semiconfluentp*:
semiconfluentp r **if** r: *strong-confluentp* r
⟨*proof*⟩

lemma *strong-confluentp-imp-confluentp*: *confluentp r* **if** *strong-confluentp r*
⟨*proof*⟩

lemma *semiconfluentp-equivclp*: *equivclp r* = r^{**} *OO* r^{-1-1**} **if** *r*: *semiconfluentp r*
⟨*proof*⟩

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 ∀ *A* ∈ $\mathcal{A}$. *finite A*
shows ∃ $\mathcal{B}$⊆$\mathcal{A}$. *finite* $\mathcal{B}$ ∧ (⋂ $\mathcal{B}$) = (⋂ $\mathcal{A}$)
⟨*proof*⟩

locale *wide-intersection-finite* =
fixes *E* :: *'Fa* ⇒ *'Fa* ⇒ *bool*
and *mapFa* :: (*'a* ⇒ *'a*) ⇒ *'Fa* ⇒ *'Fa*
and *setFa* :: *'Fa* ⇒ *'a set*
assumes *equiv*: *equivp E*
and *map-E*: *E x y* ⟹ *E* (*mapFa f x*) (*mapFa f y*)
and *map-id*: *mapFa id x* = *x*
and *map-cong*: ∀ *a*∈*setFa x*. *f a* = *g a* ⟹ *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* ⊆ *Y* **and** *z*: *setFa z* ⊆ *Z* **and** *a*: *a* ∈ *Y a* ∈ *Z*
shows ∃ *x*. *E x y* ∧ *setFa x* ⊆ *Y* ∧ *setFa x* ⊆ *Z*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

end

Subdistributivity for quotients via confluence

lemma *rtranclp-transp-reflp*: $R^{**} = R$ **if** *transp R reflp R*
⟨*proof*⟩

lemma *rtranclp-equivp*: $R^{**} = R$ **if** *equivp R*
⟨*proof*⟩

locale *confluent-quotient* =
fixes *Rb* :: $'Fb \Rightarrow 'Fb \Rightarrow bool$
and *Ea* :: $'Fa \Rightarrow 'Fa \Rightarrow bool$
and *Eb* :: $'Fb \Rightarrow 'Fb \Rightarrow bool$
and *Ec* :: $'Fc \Rightarrow 'Fc \Rightarrow bool$
and *Eab* :: $'Fab \Rightarrow 'Fab \Rightarrow bool$
and *Ebc* :: $'Fbc \Rightarrow 'Fbc \Rightarrow bool$
and *π-Faba* :: $'Fab \Rightarrow 'Fa$
and *π-Fabb* :: $'Fab \Rightarrow 'Fb$
and *π-Fbcb* :: $'Fbc \Rightarrow 'Fb$
and *π-Fbcc* :: $'Fbc \Rightarrow 'Fc$
and *rel-Fab* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'Fa \Rightarrow 'Fb \Rightarrow bool$
and *rel-Fbc* :: $('b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'Fb \Rightarrow 'Fc \Rightarrow bool$
and *rel-Fac* :: $('a \Rightarrow 'c \Rightarrow bool) \Rightarrow 'Fa \Rightarrow 'Fc \Rightarrow bool$
and *set-Fab* :: $'Fab \Rightarrow ('a \times 'b)\ set$
and *set-Fbc* :: $'Fbc \Rightarrow ('b \times 'c)\ set$
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*: $Rb^{**}\ (\pi\text{-}Fabb\ x)\ y \Longrightarrow \exists z.\ Eab\ x\ z \wedge y = \pi\text{-}Fabb\ z \wedge set\text{-}Fab\ z \subseteq set\text{-}Fab\ x$
⟨*proof*⟩

lemma *retract-bc*: Rb^{**} (π-*Fbcb x*) *y* $\Longrightarrow$ $\exists$ *z*. *Ebc x z* $\wedge$ *y* = π-*Fbcb z* $\wedge$ *set-Fbc z* $\subseteq$ *set-Fbc x*
⟨*proof*⟩

lemma *subdistributivity*: *rel-Fab A OO Eb OO rel-Fbc B* $\leq$ *Ea OO rel-Fac* (*A OO B*) *OO Ec*
⟨*proof*⟩

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* => *'b* + *nat*, *x* ::*'a* + *nat*) $\wedge$ *f k* = *Inr 0*}

typedef (*'a*, *'b*) *node* = *Node* :: ((*nat* => *'b* + *nat*) * (*'a* + *nat*)) *set*
morphisms *Rep-Node Abs-Node*
⟨*proof*⟩

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* => (*'b* + *nat*)] => (*nat* => (*'b* + *nat*))

where *Push* == (%*b h*. *case-nat b h*)

definition *Push-Node* :: [(*'b* + *nat*), (*'a*, *'b*) *node*] => (*'a*, *'b*) *node*
where *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

definition *Atom* :: (*'a* + *nat*) => (*'a*, *'b*) *dtree*
where *Atom* == (%*x*. {*Abs-Node*((%*k*. *Inr 0*, *x*))})
definition *Scons* :: [(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*
where *Scons M N* == (*Push-Node* (*Inr 1*) ‘ *M*) *Un* (*Push-Node* (*Inr* (*Suc 1*)) ‘ *N*)

definition *Leaf* :: *'a => ('a, 'b) dtree*
  where *Leaf == Atom ∘ Inl*
definition *Numb* :: *nat => ('a, 'b) dtree*
  where *Numb == Atom ∘ Inr*

definition *In0* :: *('a, 'b) dtree => ('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 == ⋃{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
⟨*proof*⟩

lemma *Push-inject1*: $Push\ i\ f = Push\ j\ g$ ==> $i=j$
⟨*proof*⟩

lemma *Push-inject2*: $Push\ i\ f = Push\ j\ g$ ==> $f=g$
⟨*proof*⟩

lemma *Push-inject*:
[| $Push\ i\ f = Push\ j\ g$; [| $i=j$; $f=g$ |] ==> P |] ==> P
⟨*proof*⟩

lemma *Push-neq-K0*: $Push\ (Inr\ (Suc\ k))\ f = (\%z.\ Inr\ 0)$ ==> P
⟨*proof*⟩

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*]

lemma *Node-K0-I*: $(\lambda k.\ Inr\ 0,\ a) \in Node$
⟨*proof*⟩

lemma *Node-Push-I*: $p \in Node \Longrightarrow apfst\ (Push\ i)\ p \in Node$
⟨*proof*⟩

17.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: $Scons\ M\ N \neq Atom(a)$
⟨*proof*⟩

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*]

lemma *inj-Atom*: $inj(Atom)$
⟨*proof*⟩
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*]

lemma *Atom-Atom-eq* [*iff*]: $(Atom(a)=Atom(b)) = (a=b)$
⟨*proof*⟩

lemma *inj-Leaf*: *inj*(*Leaf*)
⟨*proof*⟩

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*]

lemma *inj-Numb*: *inj*(*Numb*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M′ N′* ==> *M*<=*M′*
⟨*proof*⟩

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M′ N′* ==> *N*<=*N′*
⟨*proof*⟩

lemma *Scons-inject1*: *Scons M N* = *Scons M′ N′* ==> *M*=*M′*
⟨*proof*⟩

lemma *Scons-inject2*: *Scons M N* = *Scons M′ N′* ==> *N*=*N′*
⟨*proof*⟩

lemma *Scons-inject*:
[| *Scons M N* = *Scons M′ N′*; [| *M*=*M′*; *N*=*N′* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Scons-Scons-eq* [*iff*]: $(Scons\ M\ N = Scons\ M'\ N') = (M=M' \wedge N=N')$
⟨*proof*⟩

lemma *Scons-not-Leaf* [*iff*]: $Scons\ M\ N \neq Leaf(a)$
⟨*proof*⟩

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr 0*, *x*)) = *0*
⟨*proof*⟩

lemma *ndepth-Push-Node-aux*:
case-nat (*Inr* (*Suc i*)) *f k* = *Inr 0* ⟶ *Suc*(*LEAST x*. *f x* = *Inr 0*) ≤ *k*
⟨*proof*⟩

lemma *ndepth-Push-Node*:
ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
⟨*proof*⟩

lemma *ntrunc-0* [*simp*]: *ntrunc 0 M* = {}
⟨*proof*⟩

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
⟨*proof*⟩

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
⟨*proof*⟩

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
⟨*proof*⟩

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)

⟨*proof*⟩

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

17.3 Set Constructions

lemma *uprodI* [*intro*!]: ⟦*M*∈*A*; *N*∈*B*⟧ ⟹ *Scons M N* ∈ *uprod A B*
⟨*proof*⟩

lemma *uprodE* [*elim*!]:
⟦*c* ∈ *uprod A B*;
⋀*x y*. ⟦*x* ∈ *A*; *y* ∈ *B*; *c* = *Scons x y*⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

lemma *uprodE2*: ⟦*Scons M N* ∈ *uprod A B*; ⟦*M* ∈ *A*; *N* ∈ *B*⟧ ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

lemma *usum-In0I* [*intro*]: *M* ∈ *A* ⟹ *In0*(*M*) ∈ *usum A B*
⟨*proof*⟩

lemma *usum-In1I* [*intro*]: *N* ∈ *B* ⟹ *In1*(*N*) ∈ *usum A B*
⟨*proof*⟩

lemma *usumE* [*elim*!]:
⟦*u* ∈ *usum A B*;
⋀*x*. ⟦*x* ∈ *A*; *u*=*In0*(*x*)⟧ ⟹ *P*;

$$\bigwedge y.\ [\![y \in B;\ u{=}In1(y)]\!] \Longrightarrow P$$
$$[\![ ]\!] \Longrightarrow P$$
⟨*proof*⟩

lemma *In0-not-In1* [*iff*]: $In0(M) \neq In1(N)$
⟨*proof*⟩

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*]

lemma *In0-inject*: $In0(M) = In0(N) \Longrightarrow M{=}N$
⟨*proof*⟩

lemma *In1-inject*: $In1(M) = In1(N) \Longrightarrow M{=}N$
⟨*proof*⟩

lemma *In0-eq* [*iff*]: $(In0\ M = In0\ N) = (M{=}N)$
⟨*proof*⟩

lemma *In1-eq* [*iff*]: $(In1\ M = In1\ N) = (M{=}N)$
⟨*proof*⟩

lemma *inj-In0*: *inj In0*
⟨*proof*⟩

lemma *inj-In1*: *inj In1*
⟨*proof*⟩

lemma *Lim-inject*: $Lim\ f = Lim\ g \Longrightarrow f = g$
⟨*proof*⟩

lemma *ntrunc-subsetI*: $ntrunc\ k\ M <= M$
⟨*proof*⟩

lemma *ntrunc-subsetD*: $(!!k.\ ntrunc\ k\ M <= N) \Longrightarrow M<=N$
⟨*proof*⟩

lemma *ntrunc-equality*: $(!!k.\ ntrunc\ k\ M = ntrunc\ k\ N) \Longrightarrow M{=}N$
⟨*proof*⟩

lemma *ntrunc-o-equality*:
[| !!k. (ntrunc(k) ∘ h1) = (ntrunc(k) ∘ h2) |] ==> h1=h2
⟨*proof*⟩

lemma *uprod-mono*: [| A<=A'; B<=B' |] ==> uprod A B <= uprod A' B'
⟨*proof*⟩

lemma *usum-mono*: [| A<=A'; B<=B' |] ==> usum A B <= usum A' B'
⟨*proof*⟩

lemma *Scons-mono*: [| M<=M'; N<=N' |] ==> Scons M N <= Scons M' N'
⟨*proof*⟩

lemma *In0-mono*: M<=N ==> In0(M) <= In0(N)
⟨*proof*⟩

lemma *In1-mono*: M<=N ==> In1(M) <= In1(N)
⟨*proof*⟩

lemma *Split* [*simp*]: Split c (Scons M N) = c M N
⟨*proof*⟩

lemma *Case-In0* [*simp*]: Case c d (In0 M) = c(M)
⟨*proof*⟩

lemma *Case-In1* [*simp*]: Case c d (In1 N) = d(N)
⟨*proof*⟩

lemma *ntrunc-UN1*: ntrunc k (UN x. f(x)) = (UN x. ntrunc k (f x))
⟨*proof*⟩

lemma *Scons-UN1-x*: Scons (UN x. f x) M = (UN x. Scons (f x) M)
⟨*proof*⟩

lemma *Scons-UN1-y*: Scons M (UN x. f x) = (UN x. Scons M (f x))
⟨*proof*⟩

lemma *In0-UN1*: In0(UN x. f(x)) = (UN x. In0(f(x)))
⟨*proof*⟩

lemma *In1-UN1*: $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
⟨*proof*⟩

lemma *dprodI* [*intro*!]:
$[\![(M,M') \in r;\ (N,N') \in s]\!] \Longrightarrow (Scons\ M\ N,\ Scons\ M'\ N') \in dprod\ r\ s$
⟨*proof*⟩

lemma *dprodE* [*elim*!]:
$[\![c \in dprod\ r\ s;$
$\bigwedge x\ y\ x'\ y'.\ [\![(x,x') \in r;\ (y,y') \in s;$
$c = (Scons\ x\ y,\ Scons\ x'\ y')]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *dsum-In0I* [*intro*]: $(M,M') \in r \Longrightarrow (In0(M),\ In0(M')) \in dsum\ r\ s$
⟨*proof*⟩

lemma *dsum-In1I* [*intro*]: $(N,N') \in s \Longrightarrow (In1(N),\ In1(N')) \in dsum\ r\ s$
⟨*proof*⟩

lemma *dsumE* [*elim*!]:
$[\![w \in dsum\ r\ s;$
$\bigwedge x\ x'.\ [\![\ (x,x') \in r;\ \ w = (In0(x),\ In0(x'))\ ]\!] \Longrightarrow P;$
$\bigwedge y\ y'.\ [\![\ (y,y') \in s;\ \ w = (In1(y),\ In1(y'))\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *dprod-mono*: [| $r<=r'$; $s<=s'$ |] ==> $dprod\ r\ s <= dprod\ r'\ s'$
⟨*proof*⟩

lemma *dsum-mono*: [| $r<=r'$; $s<=s'$ |] ==> $dsum\ r\ s <= dsum\ r'\ s'$
⟨*proof*⟩

lemma *dprod-Sigma*: $(dprod\ (A \times B)\ (C \times D)) <= (uprod\ A\ C) \times (uprod\ B\ D)$
⟨*proof*⟩

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)))*
⟨*proof*⟩

lemma *dsum-Sigma*: *(dsum (A × B) (C × D)) <= (usum A C) × (usum B D)*
⟨*proof*⟩

lemmas *dsum-subset-Sigma = subset-trans [OF dsum-mono dsum-Sigma]*

lemma *Domain-dprod* [*simp*]: *Domain (dprod r s) = uprod (Domain r) (Domain s)*
⟨*proof*⟩

lemma *Domain-dsum* [*simp*]: *Domain (dsum r s) = usum (Domain r) (Domain s)*
⟨*proof*⟩

hides popular names

hide-type (**open**) *node item*
hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

⟨*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*
⟨*proof*⟩

lemma *triangle-Suc* [*simp*]: *triangle* (*Suc n*) = *triangle n* + *Suc n*
⟨*proof*⟩

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* + *m*
⟨*proof*⟩

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode n*) = *n*
⟨*proof*⟩

lemma *prod-decode-triangle-add*: *prod-decode* (*triangle k* + *m*) = *prod-decode-aux k m*
⟨*proof*⟩

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode x*) = *x*
⟨*proof*⟩

lemma *inj-prod-encode*: *inj-on prod-encode A*
⟨*proof*⟩

lemma *inj-prod-decode*: *inj-on prod-decode A*
⟨*proof*⟩

lemma *surj-prod-encode*: *surj prod-encode*
⟨*proof*⟩

lemma *surj-prod-decode*: *surj prod-decode*
⟨*proof*⟩

lemma *bij-prod-encode*: *bij prod-encode*
⟨*proof*⟩

lemma *bij-prod-decode*: *bij prod-decode*
⟨*proof*⟩

lemma *prod-encode-eq* [*simp*]: *prod-encode x = prod-encode y ⟷ x = y*
⟨*proof*⟩

lemma *prod-decode-eq* [*simp*]: *prod-decode x = prod-decode y ⟷ x = y*
⟨*proof*⟩

Ordering properties

lemma *le-prod-encode-1*: *a ≤ prod-encode (a, b)*
⟨*proof*⟩

lemma *le-prod-encode-2*: *b ≤ prod-encode (a, b)*
⟨*proof*⟩

lemma *prod-encode-mono1*:
assumes *a' ≤ a*
shows *prod-encode (a', b) ≤ prod-encode (a, b)*
⟨*proof*⟩

lemma *prod-encode-mono2*:
assumes *b' ≤ b*
shows *prod-encode (a, b') ≤ prod-encode (a, b)*
⟨*proof*⟩

lemma *prod-encode-mono*:
assumes *a' ≤ a b' ≤ b*
shows *prod-encode (a', b') ≤ prod-encode (a, b)*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sum-decode-inverse* [*simp*]: *sum-encode (sum-decode n) = n*
⟨*proof*⟩

lemma *inj-sum-encode*: *inj-on sum-encode A*
⟨*proof*⟩

lemma *inj-sum-decode*: *inj-on sum-decode A*
⟨*proof*⟩

lemma *surj-sum-encode*: *surj sum-encode*

⟨*proof*⟩

lemma *surj-sum-decode*: *surj sum-decode*
⟨*proof*⟩

lemma *bij-sum-encode*: *bij sum-encode*
⟨*proof*⟩

lemma *bij-sum-decode*: *bij sum-decode*
⟨*proof*⟩

lemma *sum-encode-eq*: *sum-encode x = sum-encode y ⟷ x = y*
⟨*proof*⟩

lemma *sum-decode-eq*: *sum-decode x = sum-decode y ⟷ x = y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode n*) = *n*
⟨*proof*⟩

lemma *inj-int-encode*: *inj-on int-encode A*
⟨*proof*⟩

lemma *inj-int-decode*: *inj-on int-decode A*
⟨*proof*⟩

lemma *surj-int-encode*: *surj int-encode*
⟨*proof*⟩

lemma *surj-int-decode*: *surj int-decode*
⟨*proof*⟩

lemma *bij-int-encode*: *bij int-encode*
⟨*proof*⟩

lemma *bij-int-decode*: *bij int-decode*

⟨*proof*⟩

lemma *int-encode-eq*: *int-encode x* = *int-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *int-decode-eq*: *int-decode x* = *int-decode y* ⟷ *x* = *y*
⟨*proof*⟩

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*)
⟨*proof*⟩

termination *list-decode*
⟨*proof*⟩

lemma *list-encode-inverse* [*simp*]: *list-decode* (*list-encode x*) = *x*
⟨*proof*⟩

lemma *list-decode-inverse* [*simp*]: *list-encode* (*list-decode n*) = *n*
⟨*proof*⟩

lemma *inj-list-encode*: *inj-on list-encode A*
⟨*proof*⟩

lemma *inj-list-decode*: *inj-on list-decode A*
⟨*proof*⟩

lemma *surj-list-encode*: *surj list-encode*
⟨*proof*⟩

lemma *surj-list-decode*: *surj list-decode*
⟨*proof*⟩

lemma *bij-list-encode*: *bij list-encode*
⟨*proof*⟩

lemma *bij-list-decode*: *bij list-decode*
⟨*proof*⟩

lemma *list-encode-eq*: *list-encode x* = *list-encode y* ⟷ *x* = *y*

⟨*proof*⟩

lemma *list-decode-eq*: *list-decode x* = *list-decode y* ⟷ *x* = *y*
⟨*proof*⟩

18.5 Finite sets of naturals

18.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite* (*Suc* −‘ *F*) ⟷ *finite F*
⟨*proof*⟩

lemma *vimage-Suc-insert-0*: *Suc* −‘ *insert 0 A* = *Suc* −‘ *A*
⟨*proof*⟩

lemma *vimage-Suc-insert-Suc*: *Suc* −‘ *insert* (*Suc n*) *A* = *insert n* (*Suc* −‘ *A*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *set-encode-inf*: ¬ *finite A* ⟹ *set-encode A* = *0*
⟨*proof*⟩

lemma *set-encode-insert* [*simp*]: *finite A* ⟹ *n* ∉ *A* ⟹ *set-encode* (*insert n A*)
= 2^*n* + *set-encode A*
⟨*proof*⟩

lemma *even-set-encode-iff*: *finite A* ⟹ *even* (*set-encode A*) ⟷ *0* ∉ *A*
⟨*proof*⟩

lemma *set-encode-vimage-Suc*: *set-encode* (*Suc* −‘ *A*) = *set-encode A div 2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-decode-Suc* [*simp*]: *Suc n* ∈ *set-decode x* ⟷ *n* ∈ *set-decode* (*x div 2*)
⟨*proof*⟩

lemma *set-decode-zero* [*simp*]: *set-decode 0* = {}
⟨*proof*⟩

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* −‘ *set-decode x*
⟨*proof*⟩

lemma *set-decode-plus-power-2*:
n ∉ *set-decode z* ⟹ *set-decode* (*2* ^ *n* + *z*) = *insert n* (*set-decode z*)
⟨*proof*⟩

lemma *finite-set-decode* [*simp*]: *finite* (*set-decode n*)
⟨*proof*⟩

18.5.4 Proof of isomorphism

lemma *set-decode-inverse* [*simp*]: *set-encode* (*set-decode n*) = *n*
⟨*proof*⟩

lemma *set-encode-inverse* [*simp*]: *finite A* ⟹ *set-decode* (*set-encode A*) = *A*
⟨*proof*⟩

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
⟨*proof*⟩

lemma *set-encode-eq*: *finite A* ⟹ *finite B* ⟹ *set-encode A* = *set-encode B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *subset-decode-imp-le*:
assumes *set-decode m* ⊆ *set-decode n*
shows *m* ≤ *n*
⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *inj-on-to-nat*[*simp*, *intro*]: *inj-on to-nat S*
  ⟨*proof*⟩

lemma *surj-from-nat* [*simp*]: *surj from-nat*
  ⟨*proof*⟩

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* $\longleftrightarrow$ *x* = *y*
  ⟨*proof*⟩

lemma *from-nat-to-nat* [*simp*]:
  from-nat (*to-nat x*) = *x*
  ⟨*proof*⟩

19.3 Finite types are countable

subclass (**in** *finite*) *countable*
⟨*proof*⟩

19.4 Automatically proving countability of old-style datatypes

context

begin

qualified inductive *finite-item* :: *'a Old-Datatype.item* ⇒ *bool* **where**
  undefined: *finite-item undefined*
| *In0*: *finite-item x* ⟹ *finite-item* (*Old-Datatype.In0 x*)
| *In1*: *finite-item x* ⟹ *finite-item* (*Old-Datatype.In1 x*)
| *Leaf*: *finite-item* (*Old-Datatype.Leaf a*)
| *Scons*: ⟦*finite-item x*; *finite-item y*⟧ ⟹ *finite-item* (*Old-Datatype.Scons x y*)

qualified function *nth-item* :: *nat* ⇒ (*'a*::*countable*) *Old-Datatype.item*
where
  nth-item 0 = *undefined*
| *nth-item* (*Suc n*) =
  (*case sum-decode n of*
    Inl i ⇒
    (*case sum-decode i of*
      Inl j ⇒ *Old-Datatype.In0* (*nth-item j*)
    | *Inr j* ⇒ *Old-Datatype.In1* (*nth-item j*))
  | *Inr i* ⇒
    (*case sum-decode i of*
      Inl j ⇒ *Old-Datatype.Leaf* (*from-nat j*)
    | *Inr j* ⇒
      (*case prod-decode j of*
        (*a*, *b*) ⇒ *Old-Datatype.Scons* (*nth-item a*) (*nth-item b*))))
⟨*proof*⟩

lemma *le-sum-encode-Inl*: *x* ≤ *y* ⟹ *x* ≤ *sum-encode* (*Inl y*)
  ⟨*proof*⟩

lemma *le-sum-encode-Inr*: *x* ≤ *y* ⟹ *x* ≤ *sum-encode* (*Inr y*)
  ⟨*proof*⟩ **termination**
⟨*proof*⟩

lemma *nth-item-covers*: *finite-item x* ⟹ ∃ *n*. *nth-item n* = *x*
⟨*proof*⟩

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*⟩

⟨*ML*⟩

end

19.5 Automatically proving countability of datatypes

⟨*ML*⟩

19.6 More Countable types

Naturals

instance *nat* :: *countable*
⟨*proof*⟩

Pairs

instance *prod* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Sums

instance *sum* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Integers

instance *int* :: *countable*
⟨*proof*⟩

Options

instance *option* :: (*countable*) *countable*
⟨*proof*⟩

Lists

instance *list* :: (*countable*) *countable*
⟨*proof*⟩

String literals

instance *String.literal* :: *countable*
⟨*proof*⟩

Functions

instance *fun* :: (*finite*, *countable*) *countable*
⟨*proof*⟩

Typereps

instance *typerep* :: *countable*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *Rats-eq-range-nat-to-rat-surj*: $\mathbb{Q} = range\ nat\text{-}to\text{-}rat\text{-}surj$
⟨*proof*⟩

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
$\mathbb{Q} = range\ (of\text{-}rat \circ nat\text{-}to\text{-}rat\text{-}surj)$
⟨*proof*⟩

lemma *surj-of-rat-nat-to-rat-surj*:
$r \in \mathbb{Q} \Longrightarrow \exists\, n.\ r = of\text{-}rat\ (nat\text{-}to\text{-}rat\text{-}surj\ n)$
⟨*proof*⟩

end

instance *rat* :: *countable*
⟨*proof*⟩

theorem *rat-denum*: $\exists\, f :: nat \Rightarrow rat.\ surj\ f$
⟨*proof*⟩

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*: $infinite\ S \longleftrightarrow (\forall\, m.\ \exists\, n{\geq}m.\ n \in S)$
for S :: *nat set*
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*: $infinite\ S \longleftrightarrow (\forall\, m.\ \exists\, n{>}m.\ n \in S)$
for S :: *nat set*
⟨*proof*⟩

lemma *finite-nat-iff-bounded*: $finite\ S \longleftrightarrow (\exists\, k.\ S \subseteq \{..{<}k\})$
for S :: *nat set*
⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*: $finite\ S \longleftrightarrow (\exists\, k.\ S \subseteq \{..\ k\})$
for S :: *nat set*
⟨*proof*⟩

lemma *finite-nat-bounded*: *finite* $S \Longrightarrow \exists\, k.\ S \subseteq \{..<k\}$
for S :: *nat set*
⟨*proof*⟩

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*: $\forall\, m{>}k.\ \exists\, n{>}m.\ n \in S \Longrightarrow$ *infinite* (S::*nat set*)
⟨*proof*⟩

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) $\Longrightarrow R$
⟨*proof*⟩

lemma *range-inj-infinite*:
fixes f :: *nat* $\Rightarrow$ $'a$
assumes *inj f*
shows *infinite* (*range f*)
⟨*proof*⟩

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*⟩

proposition *infinite-int-iff-unbounded-le*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n.\ |n| \geq m \wedge n \in S)$
for S :: *int set*
⟨*proof*⟩

proposition *infinite-int-iff-unbounded*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n.\ |n| > m \wedge n \in S)$
for S :: *int set*
⟨*proof*⟩

proposition *finite-int-iff-bounded*: *finite* $S \longleftrightarrow (\exists\, k.\ abs\ `\ S \subseteq \{..<k\})$
for S :: *int set*
⟨*proof*⟩

proposition *finite-int-iff-bounded-le*: *finite* $S \longleftrightarrow (\exists\, k.\ abs\ `\ S \subseteq \{..\ k\})$
for S :: *int set*
⟨*proof*⟩

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*⟩

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*]: ¬ (*INFM x. P x*) ⟷ (*MOST x.* ¬ *P x*)
⟨*proof*⟩

lemma *not-MOST* [*simp*]: ¬ (*MOST x. P x*) ⟷ (*INFM x.* ¬ *P x*)
⟨*proof*⟩

lemma *INFM-const* [*simp*]: (*INFM x::'a. P*) ⟷ *P* ∧ *infinite* (*UNIV::'a set*)
⟨*proof*⟩

lemma *MOST-const* [*simp*]: (*MOST x::'a. P*) ⟷ *P* ∨ *finite* (*UNIV::'a set*)
⟨*proof*⟩

lemma *INFM-imp-distrib*: (*INFM x. P x* ⟶ *Q x*) ⟷ ((*MOST x. P x*) ⟶ (*INFM x. Q x*))
⟨*proof*⟩

lemma *MOST-imp-iff*: *MOST x. P x* ⟹ (*MOST x. P x* ⟶ *Q x*) ⟷ (*MOST x. Q x*)
⟨*proof*⟩

lemma *INFM-conjI*: *INFM x. P x* ⟹ *MOST x. Q x* ⟹ *INFM x. P x* ∧ *Q x*
⟨*proof*⟩

Properties of quantifiers with injective functions.

lemma *INFM-inj*: *INFM x. P* (*f x*) ⟹ *inj f* ⟹ *INFM x. P x*
⟨*proof*⟩

lemma *MOST-inj*: *MOST x. P x* ⟹ *inj f* ⟹ *MOST x. P* (*f x*)
⟨*proof*⟩

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
¬ (*INFM x. x = a*)
¬ (*INFM x. a = x*)
⟨*proof*⟩

lemma *MOST-neq* [*simp*]:
MOST x. x ≠ *a*
MOST x. a ≠ *x*
⟨*proof*⟩

lemma *INFM-neq* [*simp*]:

$(INFM\ x{::}'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
$(INFM\ x{::}'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-eq* [*simp*]:
$(MOST\ x{::}'a.\ x = a) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
$(MOST\ x{::}'a.\ a = x) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-eq-imp*:
$MOST\ x.\ x = a \longrightarrow P\ x$
$MOST\ x.\ a = x \longrightarrow P\ x$
⟨*proof*⟩

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_\infty n.\ P\ n) \longleftrightarrow (\exists\, m.\ \forall\, n{>}m.\ P\ n)$
for $P :: nat \Rightarrow bool$
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ n) \longleftrightarrow (\exists\, m.\ \forall\, n{\geq}m.\ P\ n)$
for $P :: nat \Rightarrow bool$
⟨*proof*⟩

lemma *INFM-nat*: $(\exists_\infty n.\ P\ n) \longleftrightarrow (\forall\, m.\ \exists\, n{>}m.\ P\ n)$
for $P :: nat \Rightarrow bool$
⟨*proof*⟩

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ n) \longleftrightarrow (\forall\, m.\ \exists\, n{\geq}m.\ P\ n)$
for $P :: nat \Rightarrow bool$
⟨*proof*⟩

lemma *MOST-INFM*: $infinite\ (UNIV{::}'a\ set) \Longrightarrow MOST\ x{::}'a.\ P\ x \Longrightarrow INFM\ x{::}'a.\ P\ x$
⟨*proof*⟩

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
⟨*proof*⟩

lemma *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
⟨*proof*⟩

lemma *MOST-ge-nat*: $MOST\ n{::}nat.\ m \leq n$
⟨*proof*⟩
lemma *Inf-many-def*: $Inf\text{-}many\ P \longleftrightarrow infinite\ \{x.\ P\ x\}$ ⟨*proof*⟩
lemma *Alm-all-def*: $Alm\text{-}all\ P \longleftrightarrow \neg\ (INFM\ x.\ \neg\ P\ x)$ ⟨*proof*⟩
lemma *INFM-iff-infinite*: $(INFM\ x.\ P\ x) \longleftrightarrow infinite\ \{x.\ P\ x\}$ ⟨*proof*⟩
lemma *MOST-iff-cofinite*: $(MOST\ x.\ P\ x) \longleftrightarrow finite\ \{x.\ \neg\ P\ x\}$ ⟨*proof*⟩
lemma *INFM-EX*: $(\exists_\infty x.\ P\ x) \Longrightarrow (\exists\, x.\ P\ x)$ ⟨*proof*⟩

lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ ⟨*proof*⟩
lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-conjI*: *MOST* $x.\ P\ x \Longrightarrow$ *MOST* $x.\ Q\ x \Longrightarrow$ *MOST* $x.\ P\ x \wedge Q\ x$ ⟨*proof*⟩
lemma *INFM-finite-Bex-distrib*: *finite* $A \Longrightarrow$ (*INFM* $y.\ \exists x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists x{\in}A.$ *INFM* $y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *MOST-finite-Ball-distrib*: *finite* $A \Longrightarrow$ (*MOST* $y.\ \forall x{\in}A.\ P\ x\ y) \longleftrightarrow (\forall x{\in}A.$ *MOST* $y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *INFM-E*: *INFM* $x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow$ *thesis*$) \Longrightarrow$ *thesis* ⟨*proof*⟩
lemma *MOST-I*: $(\bigwedge x.\ P\ x) \Longrightarrow$ *MOST* $x.\ P\ x$ ⟨*proof*⟩
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* $S\ 0 = (LEAST\ n.\ n \in S)$
  | *enumerate-Suc*: *enumerate* $S\ (Suc\ n) =$ *enumerate* $(S - \{LEAST\ n.\ n \in S\})\ n$

lemma *enumerate-Suc′*: *enumerate* $S\ (Suc\ n) =$ *enumerate* $(S - \{$*enumerate* $S\ 0\})\ n$
  ⟨*proof*⟩

lemma *enumerate-in-set*: *infinite* $S \Longrightarrow$ *enumerate* $S\ n \in S$
⟨*proof*⟩

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite* $S \Longrightarrow$ *enumerate* $S\ n <$ *enumerate* $S\ (Suc\ n)$
⟨*proof*⟩

lemma *enumerate-mono*: $m < n \Longrightarrow$ *infinite* $S \Longrightarrow$ *enumerate* $S\ m <$ *enumerate* $S\ n$
  ⟨*proof*⟩

lemma *enumerate-mono-iff* [*simp*]:
  infinite $S \Longrightarrow$ *enumerate* $S\ m <$ *enumerate* $S\ n \longleftrightarrow m < n$
  ⟨*proof*⟩

lemma *enumerate-mono-le-iff* [*simp*]:
infinite S $\Longrightarrow$ *enumerate S m* $\leq$ *enumerate S n* $\longleftrightarrow$ $m \leq n$
⟨*proof*⟩

lemma *le-enumerate*:
assumes *S*: *infinite S*
shows $n \leq$ *enumerate S n*
⟨*proof*⟩

lemma *infinite-enumerate*:
assumes *fS*: *infinite S*
shows $\exists r{::}nat \Rightarrow nat.$ *strict-mono r* $\wedge$ $(\forall n.$ *r n* $\in S)$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *enumerate-Ex*:
fixes *S* :: *nat set*
assumes *S*: *infinite S*
and *s*: *s* $\in$ *S*
shows $\exists n.$ *enumerate S n* = *s*
⟨*proof*⟩

lemma *inj-enumerate*:
fixes *S* :: *'a*::*wellorder set*
assumes *S*: *infinite S*
shows *inj* (*enumerate S*)
⟨*proof*⟩

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*⟩

lemma
fixes *S* :: *nat set*
assumes *S*: *infinite S*
shows *range-enumerate*: *range* (*enumerate S*) = *S*
and *strict-mono-enumerate*: *strict-mono* (*enumerate S*)
⟨*proof*⟩

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 $A = \{\}$
⟨*proof*⟩

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*⟩

20.5 Properties of *enumerate* on finite sets

lemma *finite-enumerate-in-set*: $[\![\mathit{finite}\ S;\ n < \mathit{card}\ S]\!] \Longrightarrow \mathit{enumerate}\ S\ n \in S$
⟨*proof*⟩

lemma *finite-enumerate-step*: $[\![\mathit{finite}\ S;\ \mathit{Suc}\ n < \mathit{card}\ S]\!] \Longrightarrow \mathit{enumerate}\ S\ n < \mathit{enumerate}\ S\ (\mathit{Suc}\ n)$
⟨*proof*⟩

lemma *finite-enumerate-mono*: $[\![m < n;\ \mathit{finite}\ S;\ n < \mathit{card}\ S]\!] \Longrightarrow \mathit{enumerate}\ S\ m < \mathit{enumerate}\ S\ n$
⟨*proof*⟩

lemma *finite-enumerate-mono-iff* [*simp*]:
$[\![\mathit{finite}\ S;\ m < \mathit{card}\ S;\ n < \mathit{card}\ S]\!] \Longrightarrow \mathit{enumerate}\ S\ m < \mathit{enumerate}\ S\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *finite-le-enumerate*:
assumes *finite S n* $<$ *card S*
shows $n \leq \mathit{enumerate}\ S\ n$
⟨*proof*⟩

lemma *finite-enumerate*:
assumes *fS*: *finite S*
shows $\exists\, r{::}\mathit{nat}{\Rightarrow}\mathit{nat}.\ \mathit{strict\text{-}mono\text{-}on}\ \{..{<}\mathit{card}\ S\}\ r \wedge (\forall\, n{<}\mathit{card}\ S.\ r\ n \in S)$
⟨*proof*⟩

lemma *finite-enumerate-Suc''*:
fixes *S* :: *'a*::*wellorder set*
assumes *finite S Suc n* $<$ *card S*
shows *enumerate S* (*Suc n*) = (*LEAST s. s* $\in$ *S* $\wedge$ *enumerate S n* $<$ *s*)
⟨*proof*⟩

lemma *finite-enumerate-initial-segment*:
fixes *S* :: *'a*::*wellorder set*
assumes *finite S* **and** *n*: *n* $<$ *card* (*S* $\cap$ {..$<$*s*})

shows *enumerate* $(S \cap \{..<s\})\ n = enumerate\ S\ n$
⟨*proof*⟩

lemma *finite-enumerate-Ex*:
fixes $S :: 'a::wellorder\ set$
assumes S: *finite* S
and s: $s \in S$
shows $\exists n<card\ S.\ enumerate\ S\ n = s$
⟨*proof*⟩

lemma *finite-enum-subset*:
assumes $\bigwedge i.\ i < card\ X \Longrightarrow enumerate\ X\ i = enumerate\ Y\ i$ **and** *finite X finite Y card* $X \leq card\ Y$
shows $X \subseteq Y$
⟨*proof*⟩

lemma *finite-enum-ext*:
assumes $\bigwedge i.\ i < card\ X \Longrightarrow enumerate\ X\ i = enumerate\ Y\ i$ **and** *finite X finite Y card* $X = card\ Y$
shows $X = Y$
⟨*proof*⟩

lemma *finite-bij-enumerate*:
fixes $S :: 'a::wellorder\ set$
assumes S: *finite* S
shows *bij-betw* $(enumerate\ S)\ \{..<card\ S\}\ S$
⟨*proof*⟩

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*⟩

end

21 Countable sets

theory *Countable-Set*
imports *Countable Infinite-Set*
begin

21.1 Predicate for countable sets

definition *countable* :: $'a\ set \Rightarrow bool$ **where**
$countable\ S \longleftrightarrow (\exists f::'a \Rightarrow nat.\ inj\text{-}on\ f\ S)$

lemma *countable-as-injective-image-subset*: $countable\ S \longleftrightarrow (\exists f.\ \exists K::nat\ set.\ S = f\ `\ K \wedge inj\text{-}on\ f\ K)$

⟨*proof*⟩

lemma *countableE*:
assumes *S*: *countable S* **obtains** *f* :: *'a* ⇒ *nat* **where** *inj-on f S*
⟨*proof*⟩

lemma *countableI*: *inj-on* (*f*::*'a* ⇒ *nat*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableI'*: *inj-on* (*f*::*'a* ⇒ *'b*::*countable*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableE-bij*:
assumes *S*: *countable S* **obtains** *f* :: *nat* ⇒ *'a* **and** *C* :: *nat set* **where** *bij-betw f C S*
⟨*proof*⟩

lemma *countableI-bij*: *bij-betw f* (*C*::*nat set*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countable-finite*: *finite S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableI-bij1*: *bij-betw f A B* ⟹ *countable A* ⟹ *countable B*
⟨*proof*⟩

lemma *countableI-bij2*: *bij-betw f B A* ⟹ *countable A* ⟹ *countable B*
⟨*proof*⟩

lemma *countable-iff-bij*[*simp*]: *bij-betw f A B* ⟹ *countable A* ⟷ *countable B*
⟨*proof*⟩

lemma *countable-subset*: *A* ⊆ *B* ⟹ *countable B* ⟹ *countable A*
⟨*proof*⟩

lemma *countableI-type*[*intro*, *simp*]: *countable* (*A*:: *'a* :: *countable set*)
⟨*proof*⟩

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*⟩

lemma *countable-infiniteE'*:
assumes *countable A infinite A*
obtains *g* **where** *bij-betw g* (*UNIV* :: *nat set*) *A*
⟨*proof*⟩

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*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *to-nat-on-infinite*: *countable S* ⟹ *infinite S* ⟹ *bij-betw* (*to-nat-on S*) *S UNIV*
⟨*proof*⟩

lemma *bij-betw-from-nat-into-finite*: *finite S* ⟹ *bij-betw* (*from-nat-into S*) {..< *card S*} *S*
⟨*proof*⟩

lemma *bij-betw-from-nat-into*: *countable S* ⟹ *infinite S* ⟹ *bij-betw* (*from-nat-into S*) *UNIV S*
⟨*proof*⟩

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*⟩

end

lemma *countable-as-injective-image*:
assumes *countable A infinite A*
obtains *f* :: *nat* ⇒ *'a* **where** *A* = *range f inj f*
⟨*proof*⟩

lemma *inj-on-to-nat-on*[*intro*]: *countable A* ⟹ *inj-on* (*to-nat-on A*) *A*
⟨*proof*⟩

lemma *to-nat-on-inj*[*simp*]:
countable A $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ *b* $\in$ *A* $\Longrightarrow$ *to-nat-on A a* = *to-nat-on A b* $\longleftrightarrow$ *a* = *b*
⟨*proof*⟩

lemma *from-nat-into-to-nat-on*[*simp*]: *countable A* $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ *from-nat-into A* (*to-nat-on A a*) = *a*
⟨*proof*⟩

lemma *subset-range-from-nat-into*: *countable A* $\Longrightarrow$ *A* $\subseteq$ *range* (*from-nat-into A*)
⟨*proof*⟩

lemma *from-nat-into*: *A* $\neq$ {} $\Longrightarrow$ *from-nat-into A n* $\in$ *A*
⟨*proof*⟩

lemma *range-from-nat-into-subset*: *A* $\neq$ {} $\Longrightarrow$ *range* (*from-nat-into A*) $\subseteq$ *A*
⟨*proof*⟩

lemma *range-from-nat-into*[*simp*]: *A* $\neq$ {} $\Longrightarrow$ *countable A* $\Longrightarrow$ *range* (*from-nat-into A*) = *A*
⟨*proof*⟩

lemma *image-to-nat-on*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow$ *to-nat-on A* ' *A* = *UNIV*
⟨*proof*⟩

lemma *to-nat-on-surj*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow$ $\exists$ *a*$\in$*A*. *to-nat-on A a* = *n*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *from-nat-into-surj*: *countable A* $\Longrightarrow$ $a \in A \Longrightarrow \exists n.$ *from-nat-into A n* = *a*
⟨*proof*⟩

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* $\longleftrightarrow$ $A = B$
⟨*proof*⟩

lemma *inj-on-from-nat-into*: *inj-on from-nat-into* $(\{A.\ A \neq \{\} \wedge$ *countable A*$\})$
⟨*proof*⟩

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* $(SIGMA\ i : I.\ A\ i)$
⟨*proof*⟩

lemma *countable-image*[*intro*, *simp*]:
assumes *countable A*
shows *countable* $(f`A)$
⟨*proof*⟩

lemma *countable-image-inj-on*: *countable* $(f\ `\ A) \Longrightarrow$ *inj-on f A* $\Longrightarrow$ *countable A*
⟨*proof*⟩

lemma *countable-image-inj-Int-vimage*:
$[\![$*inj-on f S*; *countable A*$]\!] \Longrightarrow$ *countable* $(S \cap f\ -`\ A)$
⟨*proof*⟩

lemma *countable-image-inj-gen*:
$[\![$*inj-on f S*; *countable A*$]\!] \Longrightarrow$ *countable* $\{x \in S.\ f\ x \in A\}$
⟨*proof*⟩

lemma *countable-image-inj-eq*:
inj-on f S $\Longrightarrow$ *countable*$(f\ `\ S) \longleftrightarrow$ *countable S*
⟨*proof*⟩

lemma *countable-image-inj*:
$[\![$*countable A*; *inj f*$]\!] \Longrightarrow$ *countable* $\{x.\ f\ x \in A\}$
⟨*proof*⟩

lemma *countable-UN*[*intro*, *simp*]:
fixes I :: *'i set* **and** A :: *'i => 'a set*
assumes I: *countable I*
assumes A: $\bigwedge i.\ i \in I \Longrightarrow$ *countable* $(A\ i)$
shows *countable* $(\bigcup i \in I.\ A\ i)$

⟨*proof*⟩

lemma *countable-Un*[*intro*]: *countable A* ⟹ *countable B* ⟹ *countable* (*A* ∪ *B*)
⟨*proof*⟩

lemma *countable-Un-iff*[*simp*]: *countable* (*A* ∪ *B*) ⟷ *countable A* ∧ *countable B*
⟨*proof*⟩

lemma *countable-Plus*[*intro*, *simp*]:
countable A ⟹ *countable B* ⟹ *countable* (*A* <+> *B*)
⟨*proof*⟩

lemma *countable-empty*[*intro*, *simp*]: *countable* {}
⟨*proof*⟩

lemma *countable-insert*[*intro*, *simp*]: *countable A* ⟹ *countable* (*insert a A*)
⟨*proof*⟩

lemma *countable-Int1*[*intro*, *simp*]: *countable A* ⟹ *countable* (*A* ∩ *B*)
⟨*proof*⟩

lemma *countable-Int2*[*intro*, *simp*]: *countable B* ⟹ *countable* (*A* ∩ *B*)
⟨*proof*⟩

lemma *countable-INT*[*intro*, *simp*]: *i* ∈ *I* ⟹ *countable* (*A i*) ⟹ *countable* (⋂ *i*∈*I*. *A i*)
⟨*proof*⟩

lemma *countable-Diff*[*intro*, *simp*]: *countable A* ⟹ *countable* (*A* − *B*)
⟨*proof*⟩

lemma *countable-insert-eq* [*simp*]: *countable* (*insert x A*) = *countable A*
⟨*proof*⟩

lemma *countable-vimage*: *B* ⊆ *range f* ⟹ *countable* (*f* −‘ *B*) ⟹ *countable B*
⟨*proof*⟩

lemma *surj-countable-vimage*: *surj f* ⟹ *countable* (*f* −‘ *B*) ⟹ *countable B*
⟨*proof*⟩

lemma *countable-Collect*[*simp*]: *countable A* ⟹ *countable* {*a* ∈ *A*. *φ a*}
⟨*proof*⟩

lemma *countable-Image*:
assumes ⋀*y*. *y* ∈ *Y* ⟹ *countable* (*X* ‘‘ {*y*})
assumes *countable Y*
shows *countable* (*X* ‘‘ *Y*)
⟨*proof*⟩

lemma *countable-relpow*:
fixes $X :: 'a\ rel$
assumes *Image-X*: $\bigwedge Y.\ countable\ Y \Longrightarrow countable\ (X\ ``\ Y)$
assumes Y: $countable\ Y$
shows $countable\ ((X\ ^{\frown\frown}\ i)\ ``\ Y)$
⟨*proof*⟩

lemma *countable-funpow*:
fixes $f :: 'a\ set \Rightarrow 'a\ set$
assumes $\bigwedge A.\ countable\ A \Longrightarrow countable\ (f\ A)$
and $countable\ A$
shows $countable\ ((f\ ^{\frown\frown}\ n)\ A)$
⟨*proof*⟩

lemma *countable-rtrancl*:
$(\bigwedge Y.\ countable\ Y \Longrightarrow countable\ (X\ ``\ Y)) \Longrightarrow countable\ Y \Longrightarrow countable\ (X^*\ ``\ Y)$
⟨*proof*⟩

lemma *countable-lists*[*intro*, *simp*]:
assumes A: $countable\ A$ **shows** $countable\ (lists\ A)$
⟨*proof*⟩

lemma *Collect-finite-eq-lists*: $Collect\ finite = set\ `\ lists\ UNIV$
⟨*proof*⟩

lemma *countable-Collect-finite*: $countable\ (Collect\ (finite::'a::countable\ set \Rightarrow bool))$
⟨*proof*⟩

lemma *countable-int*: $countable\ \mathbb{Z}$
⟨*proof*⟩

lemma *countable-rat*: $countable\ \mathbb{Q}$
⟨*proof*⟩

lemma *Collect-finite-subset-eq-lists*: $\{A.\ finite\ A \wedge A \subseteq T\} = set\ `\ lists\ T$
⟨*proof*⟩

lemma *countable-Collect-finite-subset*:
$countable\ T \Longrightarrow countable\ \{A.\ finite\ A \wedge A \subseteq T\}$
⟨*proof*⟩

lemma *countable-Fpow*: $countable\ S \Longrightarrow countable\ (Fpow\ S)$
⟨*proof*⟩

lemma *countable-set-option* [*simp*]: $countable\ (set\text{-}option\ x)$
⟨*proof*⟩

21.4 Misc lemmas

lemma *countable-subset-image*:
countable $B \wedge B \subseteq (f \mathbin{`} A) \longleftrightarrow (\exists A'.\ \mathit{countable}\ A' \wedge A' \subseteq A \wedge (B = f \mathbin{`} A'))$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *ex-subset-image-inj*:
$(\exists T.\ T \subseteq f \mathbin{`} S \wedge P\ T) \longleftrightarrow (\exists T.\ T \subseteq S \wedge \mathit{inj\text{-}on}\ f\ T \wedge P\ (f \mathbin{`} T))$
⟨*proof*⟩

lemma *all-subset-image-inj*:
$(\forall T.\ T \subseteq f \mathbin{`} S \longrightarrow P\ T) \longleftrightarrow (\forall T.\ T \subseteq S \wedge \mathit{inj\text{-}on}\ f\ T \longrightarrow P(f \mathbin{`} T))$
⟨*proof*⟩

lemma *ex-countable-subset-image-inj*:
$(\exists T.\ \mathit{countable}\ T \wedge T \subseteq f \mathbin{`} S \wedge P\ T) \longleftrightarrow$
$(\exists T.\ \mathit{countable}\ T \wedge T \subseteq S \wedge \mathit{inj\text{-}on}\ f\ T \wedge P\ (f \mathbin{`} T))$
⟨*proof*⟩

lemma *all-countable-subset-image-inj*:
$(\forall T.\ \mathit{countable}\ T \wedge T \subseteq f \mathbin{`} S \longrightarrow P\ T) \longleftrightarrow (\forall T.\ \mathit{countable}\ T \wedge T \subseteq S \wedge \mathit{inj\text{-}on}\ f\ T \longrightarrow P(f \mathbin{`} T))$
⟨*proof*⟩

lemma *ex-countable-subset-image*:
$(\exists T.\ \mathit{countable}\ T \wedge T \subseteq f \mathbin{`} S \wedge P\ T) \longleftrightarrow (\exists T.\ \mathit{countable}\ T \wedge T \subseteq S \wedge P\ (f \mathbin{`} T))$
⟨*proof*⟩

lemma *all-countable-subset-image*:
$(\forall T.\ \mathit{countable}\ T \wedge T \subseteq f \mathbin{`} S \longrightarrow P\ T) \longleftrightarrow (\forall T.\ \mathit{countable}\ T \wedge T \subseteq S \longrightarrow P(f \mathbin{`} T))$
⟨*proof*⟩

lemma *countable-image-eq*:
$\mathit{countable}(f \mathbin{`} S) \longleftrightarrow (\exists T.\ \mathit{countable}\ T \wedge T \subseteq S \wedge f \mathbin{`} S = f \mathbin{`} T)$
⟨*proof*⟩

lemma *countable-image-eq-inj*:
$\mathit{countable}(f \mathbin{`} S) \longleftrightarrow (\exists T.\ \mathit{countable}\ T \wedge T \subseteq S \wedge f \mathbin{`} S = f \mathbin{`} T \wedge \mathit{inj\text{-}on}\ f\ T)$
⟨*proof*⟩

lemma *infinite-countable-subset'*:
assumes *X*: *infinite X* **shows** $\exists C \subseteq X.\ \mathit{countable}\ C \wedge \mathit{infinite}\ C$
⟨*proof*⟩

lemma *countable-all*:
assumes *S*: *countable S*
shows $(\forall s \in S.\ P\ s) \longleftrightarrow (\forall n{::}\mathit{nat}.\ \mathit{from\text{-}nat\text{-}into}\ S\ n \in S \longrightarrow P\ (\mathit{from\text{-}nat\text{-}into}$

S n))
⟨*proof*⟩

lemma *finite-sequence-to-countable-set*:
assumes *countable X*
obtains *F* **where** ⋀*i*. *F i* ⊆ *X* ⋀*i*. *F i* ⊆ *F* (*Suc i*) ⋀*i*. *finite* (*F i*) (⋃ *i*. *F i*) = *X*
⟨*proof*⟩

lemma *transfer-countable*[*transfer-rule*]:
bi-unique R ⟹ *rel-fun* (*rel-set R*) (=) *countable countable*
⟨*proof*⟩

21.5 Uncountable

abbreviation *uncountable* **where**
uncountable A ≡ ¬ *countable A*

lemma *uncountable-def*: *uncountable A* ⟷ *A* ≠ {} ∧ ¬ (∃*f*::(*nat* ⇒ *'a*). *range f* = *A*)
⟨*proof*⟩

lemma *uncountable-bij-betw*: *bij-betw f A B* ⟹ *uncountable B* ⟹ *uncountable A*
⟨*proof*⟩

lemma *uncountable-infinite*: *uncountable A* ⟹ *infinite A*
⟨*proof*⟩

lemma *uncountable-minus-countable*:
uncountable A ⟹ *countable B* ⟹ *uncountable* (*A* − *B*)
⟨*proof*⟩

lemma *countable-Diff-eq* [*simp*]: *countable* (*A* − {*x*}) = *countable A*
⟨*proof*⟩

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* ⇒ *'a set*
where *pairwise* (λ*i j*. *disjnt* (*C i*) (*C j*)) *UNIV* (⋃ *i*. *C i*) ⊆ *A* ⋀*i*. *infinite* (*C i*)
⟨*proof*⟩

end

22 Countable Complete Lattices

theory *Countable-Complete-Lattices*
imports *Main Countable-Set*
begin

lemma *UNIV-nat-eq*: $UNIV = insert\ 0\ (range\ Suc)$
⟨*proof*⟩

class *countable-complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +
assumes *ccInf-lower*: $countable\ A \Longrightarrow x \in A \Longrightarrow Inf\ A \leq x$
assumes *ccInf-greatest*: $countable\ A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow z \leq x) \Longrightarrow z \leq Inf\ A$
assumes *ccSup-upper*: $countable\ A \Longrightarrow x \in A \Longrightarrow x \leq Sup\ A$
assumes *ccSup-least*: $countable\ A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow x \leq z) \Longrightarrow Sup\ A \leq z$
assumes *ccInf-empty* [*simp*]: $Inf\ \{\} = top$
assumes *ccSup-empty* [*simp*]: $Sup\ \{\} = bot$
begin

subclass *bounded-lattice*
⟨*proof*⟩

lemma *ccINF-lower*: $countable\ A \Longrightarrow i \in A \Longrightarrow (INF\ i \in A.\ f\ i) \leq f\ i$
⟨*proof*⟩

lemma *ccINF-greatest*: $countable\ A \Longrightarrow (\bigwedge i.\ i \in A \Longrightarrow u \leq f\ i) \Longrightarrow u \leq (INF\ i \in A.\ f\ i)$
⟨*proof*⟩

lemma *ccSUP-upper*: $countable\ A \Longrightarrow i \in A \Longrightarrow f\ i \leq (SUP\ i \in A.\ f\ i)$
⟨*proof*⟩

lemma *ccSUP-least*: $countable\ A \Longrightarrow (\bigwedge i.\ i \in A \Longrightarrow f\ i \leq u) \Longrightarrow (SUP\ i \in A.\ f\ i) \leq u$
⟨*proof*⟩

lemma *ccInf-lower2*: $countable\ A \Longrightarrow u \in A \Longrightarrow u \leq v \Longrightarrow Inf\ A \leq v$
⟨*proof*⟩

lemma *ccINF-lower2*: $countable\ A \Longrightarrow i \in A \Longrightarrow f\ i \leq u \Longrightarrow (INF\ i \in A.\ f\ i) \leq u$
⟨*proof*⟩

lemma *ccSup-upper2*: $countable\ A \Longrightarrow u \in A \Longrightarrow v \leq u \Longrightarrow v \leq Sup\ A$
⟨*proof*⟩

lemma *ccSUP-upper2*: $countable\ A \Longrightarrow i \in A \Longrightarrow u \leq f\ i \Longrightarrow u \leq (SUP\ i \in A.\ f\ i)$
⟨*proof*⟩

lemma *le-ccInf-iff*: *countable A* $\Longrightarrow$ $b \leq Inf\ A \longleftrightarrow (\forall\, a{\in}A.\ b \leq a)$
⟨*proof*⟩

lemma *le-ccINF-iff*: *countable A* $\Longrightarrow$ $u \leq (INF\ i \in A.\ f\ i) \longleftrightarrow (\forall\, i{\in}A.\ u \leq f\ i)$
⟨*proof*⟩

lemma *ccSup-le-iff*: *countable A* $\Longrightarrow$ $Sup\ A \leq b \longleftrightarrow (\forall\, a{\in}A.\ a \leq b)$
⟨*proof*⟩

lemma *ccSUP-le-iff*: *countable A* $\Longrightarrow$ $(SUP\ i \in A.\ f\ i) \leq u \longleftrightarrow (\forall\, i{\in}A.\ f\ i \leq u)$
⟨*proof*⟩

lemma *ccInf-insert* [*simp*]: *countable A* $\Longrightarrow$ $Inf\ (insert\ a\ A) = inf\ a\ (Inf\ A)$
⟨*proof*⟩

lemma *ccINF-insert* [*simp*]: *countable A* $\Longrightarrow$ $(INF\ x{\in}insert\ a\ A.\ f\ x) = inf\ (f\ a)\ (Inf\ (f\ `\ A))$
⟨*proof*⟩

lemma *ccSup-insert* [*simp*]: *countable A* $\Longrightarrow$ $Sup\ (insert\ a\ A) = sup\ a\ (Sup\ A)$
⟨*proof*⟩

lemma *ccSUP-insert* [*simp*]: *countable A* $\Longrightarrow$ $(SUP\ x{\in}insert\ a\ A.\ f\ x) = sup\ (f\ a)\ (Sup\ (f\ `\ A))$
⟨*proof*⟩

lemma *ccINF-empty* [*simp*]: $(INF\ x{\in}\{\}.\ f\ x) = top$
⟨*proof*⟩

lemma *ccSUP-empty* [*simp*]: $(SUP\ x{\in}\{\}.\ f\ x) = bot$
⟨*proof*⟩

lemma *ccInf-superset-mono*: *countable A* $\Longrightarrow$ $B \subseteq A \Longrightarrow Inf\ A \leq Inf\ B$
⟨*proof*⟩

lemma *ccSup-subset-mono*: *countable B* $\Longrightarrow$ $A \subseteq B \Longrightarrow Sup\ A \leq Sup\ B$
⟨*proof*⟩

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*⟩

lemma *ccINF-mono*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ $(\bigwedge m.\ m \in B \Longrightarrow \exists\, n{\in}A.\ f\ n \leq g\ m) \Longrightarrow (INF\ n{\in}A.\ f\ n) \leq (INF\ n{\in}B.\ g\ n)$
⟨*proof*⟩

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*⟩

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* $n{\in}B.\ g\ n$)
⟨*proof*⟩

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* $x{\in}B.\ g\ x$)
⟨*proof*⟩

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* $x{\in}B.\ g\ x$)
⟨*proof*⟩

lemma *less-eq-ccInf-inter*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *sup* (*Inf A*) (*Inf B*) $\leq$ *Inf* $(A \cap B)$
⟨*proof*⟩

lemma *ccSup-inter-less-eq*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Sup* $(A \cap B) \leq$ *inf* (*Sup A*) (*Sup B*)
⟨*proof*⟩

lemma *ccInf-union-distrib*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Inf* $(A \cup B) =$ *inf* (*Inf A*) (*Inf B*)
⟨*proof*⟩

lemma *ccINF-union*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ (*INF* $i{\in}A \cup B.\ M\ i) =$ *inf* (*INF* $i{\in}A.\ M\ i$) (*INF* $i{\in}B.\ M\ i$)
⟨*proof*⟩

lemma *ccSup-union-distrib*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Sup* $(A \cup B) =$ *sup* (*Sup A*) (*Sup B*)
⟨*proof*⟩

lemma *ccSUP-union*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ (*SUP* $i{\in}A \cup B.\ M\ i) =$ *sup* (*SUP* $i{\in}A.\ M\ i$) (*SUP* $i{\in}B.\ M\ i$)
⟨*proof*⟩

lemma *ccINF-inf-distrib*: *countable A* $\Longrightarrow$ *inf* (*INF* $a{\in}A.\ f\ a$) (*INF* $a{\in}A.\ g\ a) =$

(*INF a∈A. inf* (*f a*) (*g a*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *ccINF-const* [*simp*]: *A* ≠ {} ⟹ (*INF i* ∈ *A. f*) = *f*
⟨*proof*⟩

lemma *ccSUP-const* [*simp*]: *A* ≠ {} ⟹ (*SUP i* ∈ *A. f*) = *f*
⟨*proof*⟩

lemma *ccINF-top* [*simp*]: (*INF x∈A. top*) = *top*
⟨*proof*⟩

lemma *ccSUP-bot* [*simp*]: (*SUP x∈A. bot*) = *bot*
⟨*proof*⟩

lemma *ccINF-commute*: *countable A* ⟹ *countable B* ⟹ (*INF i∈A. INF j∈B. f i j*) = (*INF j∈B. INF i∈A. f i j*)
⟨*proof*⟩

lemma *ccSUP-commute*: *countable A* ⟹ *countable B* ⟹ (*SUP i∈A. SUP j∈B. f i j*) = (*SUP j∈B. SUP i∈A. f i j*)
⟨*proof*⟩

end

context
fixes *a* :: *'a*::{*countable-complete-lattice*, *linorder*}
begin

lemma *less-ccSup-iff*: *countable S* ⟹ *a* < *Sup S* ⟷ (∃ *x*∈*S. a* < *x*)
⟨*proof*⟩

lemma *less-ccSUP-iff*: *countable A* ⟹ *a* < (*SUP i∈A. f i*) ⟷ (∃ *x*∈*A. a* < *f x*)
⟨*proof*⟩

lemma *ccInf-less-iff*: *countable S* ⟹ *Inf S* < *a* ⟷ (∃ *x*∈*S. x* < *a*)
⟨*proof*⟩

lemma *ccINF-less-iff*: *countable A* ⟹ (*INF i∈A. f i*) < *a* ⟷ (∃ *x*∈*A. f x* < *a*)
⟨*proof*⟩

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
assumes *sup-ccInf*: *countable B* ⟹ *sup a* (*Inf B*) = (*INF b∈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*))
⟨*proof*⟩

lemma *inf-ccSUP*:
countable B $\Longrightarrow$ *inf a* (*SUP b*$\in$*B. f b*) = (*SUP b*$\in$*B. inf a* (*f b*))
⟨*proof*⟩

subclass *distrib-lattice*
⟨*proof*⟩

lemma *ccInf-sup*:
countable B $\Longrightarrow$ *sup* (*Inf B*) *a* = (*INF b*$\in$*B. sup b a*)
⟨*proof*⟩

lemma *ccSup-inf*:
countable B $\Longrightarrow$ *inf* (*Sup B*) *a* = (*SUP b*$\in$*B. inf b a*)
⟨*proof*⟩

lemma *ccINF-sup*:
countable B $\Longrightarrow$ *sup* (*INF b*$\in$*B. f b*) *a* = (*INF b*$\in$*B. sup* (*f b*) *a*)
⟨*proof*⟩

lemma *ccSUP-inf*:
countable B $\Longrightarrow$ *inf* (*SUP b*$\in$*B. f b*) *a* = (*SUP b*$\in$*B. inf* (*f b*) *a*)
⟨*proof*⟩

lemma *ccINF-sup-distrib2*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *sup* (*INF a*$\in$*A. f a*) (*INF b*$\in$*B. g b*) = (*INF a*$\in$*A. INF b*$\in$*B. sup* (*f a*) (*g b*))
⟨*proof*⟩

lemma *ccSUP-inf-distrib2*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *inf* (*SUP a*$\in$*A. f a*) (*SUP b*$\in$*B. g b*) = (*SUP a*$\in$*A. SUP b*$\in$*B. inf* (*f a*) (*g b*))
⟨*proof*⟩

context
fixes *f* :: *'a* $\Rightarrow$ *'b*::*countable-complete-lattice*
assumes *mono f*
begin

lemma *mono-ccInf*:
countable A $\Longrightarrow$ *f* (*Inf A*) $\leq$ (*INF x*$\in$*A. f x*)
⟨*proof*⟩

lemma *mono-ccSup*:
countable A $\Longrightarrow$ $(SUP\ x{\in}A.\ f\ x) \leq f\ (Sup\ A)$
⟨*proof*⟩

lemma *mono-ccINF*:
countable I $\Longrightarrow$ $f\ (INF\ i \in I.\ A\ i) \leq (INF\ x \in I.\ f\ (A\ x))$
⟨*proof*⟩

lemma *mono-ccSUP*:
countable I $\Longrightarrow$ $(SUP\ x \in I.\ f\ (A\ x)) \leq f\ (SUP\ i \in I.\ A\ i)$
⟨*proof*⟩

end

end

22.0.1 Instances of countable complete lattices

instance *fun* :: (*type*, *countable-complete-lattice*) *countable-complete-lattice*
⟨*proof*⟩

subclass (**in** *complete-lattice*) *countable-complete-lattice*
⟨*proof*⟩

subclass (**in** *complete-distrib-lattice*) *countable-complete-distrib-lattice*
⟨*proof*⟩

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* $\longleftrightarrow$ $|A| \leq o\ |UNIV{::}nat\ set|$
⟨*proof*⟩

lemma *countable-card-le-natLeq*: *countable A* $\longleftrightarrow$ $|A| \leq o\ natLeq$
⟨*proof*⟩

lemma *countable-or-card-of*:
assumes *countable A*

shows (*finite A* ∧ |*A*| <*o* |*UNIV*::*nat set*|) ∨
(*infinite A* ∧ |*A*| =*o* |*UNIV*::*nat set*|)
⟨*proof*⟩

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*|
⟨*proof*⟩

lemma *countable-or*:
countable A ⟹ (∃ *f*::'*a*⇒*nat*. *finite A* ∧ *inj-on f A*) ∨ (∃ *f*::'*a*⇒*nat*. *infinite A* ∧ *bij-betw f A UNIV*)
⟨*proof*⟩

lemma *countable-cases*[*elim*]:
assumes *countable A*
obtains (*Fin*) *f* :: '*a*⇒*nat* **where** *finite A inj-on f A*
| (*Inf*) *f* :: '*a*⇒*nat* **where** *infinite A bij-betw f A UNIV*
⟨*proof*⟩

lemma *countable-ordLeq*:
assumes |*A*| ≤*o* |*B*| **and** *countable B*
shows *countable A*
⟨*proof*⟩

lemma *countable-ordLess*:
assumes *AB*: |*A*| <*o* |*B*| **and** *B*: *countable B*
shows *countable A*
⟨*proof*⟩

end

23.2 The type of countable sets

typedef '*a cset* = {*A* :: '*a set*. *countable A*} **morphisms** *rcset acset*
⟨*proof*⟩

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* ⟨*proof*⟩

lift-definition *less-eq-cset* :: *'a cset* ⇒ *'a cset* ⇒ *bool*
is *subset-eq* **parametric** *subset-transfer* ⟨*proof*⟩

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*) ===> (=)) (⊂) (<)
⟨*proof*⟩

lift-definition *sup-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *union* **parametric** *union-transfer* ⟨*proof*⟩

lift-definition *inf-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *inter* **parametric** *inter-transfer* ⟨*proof*⟩

lift-definition *minus-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *minus* **parametric** *Diff-transfer* ⟨*proof*⟩

instance ⟨*proof*⟩

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*
⟨*proof*⟩
lift-definition *cinsert* :: *'a* ⇒ *'a cset* ⇒ *'a cset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
⟨*proof*⟩
abbreviation *csingle* :: *'a* ⇒ *'a cset* **where** *csingle x* ≡ *cinsert x cempty*
lift-definition *cimage* :: (*'a* ⇒ *'b*) ⇒ *'a cset* ⇒ *'b cset* **is** (‘) **parametric** *image-transfer*
⟨*proof*⟩
lift-definition *cBall* :: *'a cset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Ball* **parametric** *Ball-transfer*
⟨*proof*⟩
lift-definition *cBex* :: *'a cset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Bex* **parametric** *Bex-transfer*
⟨*proof*⟩
lift-definition *cUnion* :: *'a cset cset* ⇒ *'a cset* **is** *Union* **parametric** *Union-transfer*

⟨*proof*⟩
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*)
⟨*proof*⟩

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* ⟨*proof*⟩

23.3.2 *cinsert*

lemma *countable-insert-iff*: *countable* (*insert x A*) $\longleftrightarrow$ *countable A*
⟨*proof*⟩

lemma *set-cinsert*:
assumes *cin x A*
obtains *B* **where** *A* = *cinsert x B* **and** ¬ *cin x B*
⟨*proof*⟩

lemma *mk-disjoint-cinsert*: *cin a A* $\Longrightarrow$ ∃ *B*. *A* = *cinsert a B* ∧ ¬ *cin a B*
⟨*proof*⟩

23.3.3 *cimage*

lemma *subset-cimage-iff*: *csubset-eq B* (*cimage f A*) $\longleftrightarrow$ (∃ *AA*. *csubset-eq AA A* ∧ *B* = *cimage f AA*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⟶ *Q x*) = (*P* ⟶ *cBall A Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P x* ⟶ *Q*) = (*cBex A P* ⟶ *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*)
⟨*proof*⟩

lemma *atomize-cBall*:

$(\bigwedge x.\ cin\ x\ A \Longrightarrow P\ x) == Trueprop\ (cBall\ A\ (\lambda x.\ P\ x))$
⟨*proof*⟩

23.3.5 *cUnion*

lemma *cUNION-cimage*: $cUNION\ (cimage\ f\ A)\ g = cUNION\ A\ (g \circ f)$
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *rel-cset-alt-def*:
$rel\text{-}cset\ R\ a\ b \longleftrightarrow$
$(\forall t \in rcset\ a.\ \exists u \in rcset\ b.\ R\ t\ u) \wedge$
$(\forall t \in rcset\ b.\ \exists u \in rcset\ a.\ R\ u\ t)$
⟨*proof*⟩

lemma *rel-cset-iff*:
$rel\text{-}cset\ R\ a\ b \longleftrightarrow$
$(\forall t.\ cin\ t\ a \longrightarrow (\exists u.\ cin\ u\ b \wedge R\ t\ u)) \wedge$
$(\forall t.\ cin\ t\ b \longrightarrow (\exists u.\ cin\ u\ a \wedge R\ u\ t))$
⟨*proof*⟩

lemma *rel-cset-cUNION*:
$[\![\ rel\text{-}cset\ Q\ A\ B;\ rel\text{-}fun\ Q\ (rel\text{-}cset\ R)\ f\ g\ ]\!]$
$\Longrightarrow rel\text{-}cset\ R\ (cUnion\ (cimage\ f\ A))\ (cUnion\ (cimage\ g\ B))$
⟨*proof*⟩

lemma *rel-cset-csingle-iff* [*simp*]: $rel\text{-}cset\ R\ (csingle\ x)\ (csingle\ y) \longleftrightarrow R\ x\ y$
⟨*proof*⟩

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*]:
$(A ===> rel\text{-}cset\ A ===> rel\text{-}cset\ A)\ cinsert\ cinsert$
⟨*proof*⟩

lemma *cUn-parametric* [*transfer-rule*]:
$(rel\text{-}cset\ A ===> rel\text{-}cset\ A ===> rel\text{-}cset\ A)\ cUn\ cUn$
⟨*proof*⟩

lemma *cUnion-parametric* [*transfer-rule*]:
(*rel-cset* (*rel-cset A*) ===> *rel-cset A*) *cUnion cUnion*
⟨*proof*⟩

lemma *cimage-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-cset A* ===> *rel-cset B*) *cimage cimage*
⟨*proof*⟩

lemma *cBall-parametric* [*transfer-rule*]:
(*rel-cset A* ===> (*A* ===> (=)) ===> (=)) *cBall cBall*
⟨*proof*⟩

lemma *cBex-parametric* [*transfer-rule*]:
(*rel-cset A* ===> (*A* ===> (=)) ===> (=)) *cBex cBex*
⟨*proof*⟩

lemma *rel-cset-parametric* [*transfer-rule*]:
((*A* ===> *B* ===> (=)) ===> *rel-cset A* ===> *rel-cset B* ===> (=)) *rel-cset rel-cset*
⟨*proof*⟩

Rules requiring bi-unique, bi-total or right-total relations

lemma *cin-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*A* ===> *rel-cset A* ===> (=)) *cin cin*
⟨*proof*⟩

lemma *cInt-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cInt cInt*
⟨*proof*⟩

lemma *cDiff-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cDiff cDiff*
⟨*proof*⟩

lemma *csubset-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> (=)) *csubset-eq csubset-eq*
⟨*proof*⟩

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*⟩

lemma *card-of-countable-sets-Func*:
$|\{X.\ X \subseteq A \wedge countable\ X \wedge X \neq \{\}\}| \leq o\ |A|\ ^{c}\ natLeq$
⟨*proof*⟩

lemma *ordLeq-countable-subsets*:
$|A| \leq o\ |\{X.\ X \subseteq A \wedge countable\ X\}|$
⟨*proof*⟩

end

lemma *finite-countable-subset*:
$finite\ \{X.\ X \subseteq A \wedge countable\ X\} \longleftrightarrow finite\ A$
⟨*proof*⟩

lemma *rcset-to-rcset*: $countable\ A \Longrightarrow rcset\ (the\text{-}inv\ rcset\ A) = A$
including *cset.lifting*
⟨*proof*⟩

lemma *Collect-Int-Times*: $\{(x,\ y).\ R\ x\ y\} \cap A \times B = \{(x,\ y).\ R\ x\ y \wedge x \in A \wedge y \in B\}$
⟨*proof*⟩

lemma *rel-cset-aux*:
$(\forall t \in rcset\ a.\ \exists u \in rcset\ b.\ R\ t\ u) \wedge (\forall t \in rcset\ b.\ \exists u \in rcset\ a.\ R\ u\ t) \longleftrightarrow$
$((BNF\text{-}Def.Grp\ \{x.\ rcset\ x \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (cimage\ fst))^{-1-1}\ OO$
$BNF\text{-}Def.Grp\ \{x.\ rcset\ x \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (cimage\ snd))\ a\ b$ (**is** *?L = ?R*)
⟨*proof*⟩ **including** *cset.lifting*
⟨*proof*⟩

context
includes *cardinal-syntax*
begin

bnf *'a cset*
map: *cimage*
sets: *rcset*
bd: *card-suc natLeq*
wits: *cempty*
rel: *rel-cset*
⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩

end

end

24 Debugging facilities for code generated towards Isabelle/ML

theory *Debug*
imports *Main*
begin

context
begin

qualified definition *trace* :: *String.literal* ⇒ *unit* **where**
[*simp*]: *trace s* = ()

qualified definition *tracing* :: *String.literal* ⇒ *'a* ⇒ *'a* **where**
[*simp*]: *tracing s* = *id*

lemma [*code*]:
tracing s = (*let u* = *trace s in id*)
⟨*proof*⟩ **definition** *flush* :: *'a* ⇒ *unit* **where**
[*simp*]: *flush x* = ()

qualified definition *flushing* :: *'a* ⇒ *'b* ⇒ *'b* **where**
[*simp*]: *flushing x* = *id*

lemma [*code*, *code-unfold*]:
flushing x = (*let u* = *flush x in id*)
⟨*proof*⟩ **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*)
  ⟨*proof*⟩

lemma *reduce-holds*:
  strict-mono s ⟹ *P n* (*s* ∘ *reduce s n*)
  ⟨*proof*⟩

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*⟩

lemma *seqseq-holds*:
  P n (*seqseq* (*Suc n*))
⟨*proof*⟩

definition *diagseq* :: *nat* ⇒ *nat* **where** *diagseq i* = *seqseq i i*

lemma *diagseq-mono*: *diagseq n* < *diagseq* (*Suc n*)
⟨*proof*⟩

lemma *subseq-diagseq*: *strict-mono diagseq*
  ⟨*proof*⟩

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*⟩

lemma *ex-subseq-reduce-index*: *seqseq* (*n* + *k*) = *seqseq n* ∘ *fold-reduce n k*
  ⟨*proof*⟩

lemma *seqseq-fold-reduce*: *seqseq n* = *fold-reduce 0 n*

⟨*proof*⟩

lemma *diagseq-fold-reduce*: *diagseq n* = *fold-reduce 0 n n*
⟨*proof*⟩

lemma *fold-reduce-add*: *fold-reduce 0* (*m* + *n*) = *fold-reduce 0 m* ∘ *fold-reduce m n*
⟨*proof*⟩

lemma *diagseq-add*: *diagseq* (*k* + *n*) = (*seqseq k* ∘ (*fold-reduce k n*)) (*k* + *n*)
⟨*proof*⟩

lemma *diagseq-sub*:
assumes *m* ≤ *n* **shows** *diagseq n* = (*seqseq m* ∘ (*fold-reduce m* (*n* − *m*))) *n*
⟨*proof*⟩

lemma *subseq-diagonal-rest*: *strict-mono* (λ*x*. *fold-reduce k x* (*k* + *x*))
⟨*proof*⟩

lemma *diagseq-seqseq*: *diagseq* ∘ ((+) *k*) = (*seqseq k* ∘ (λ*x*. *fold-reduce k x* (*k* + *x*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *floor-log-zero* [*simp*]: *floor-log 0 = 0*
⟨*proof*⟩

lemma *floor-log-one* [*simp*]: *floor-log 1 = 0*
⟨*proof*⟩

lemma *floor-log-Suc-zero* [*simp*]: *floor-log (Suc 0) = 0*
⟨*proof*⟩

lemma *floor-log-rec*: $n \geq 2 \Longrightarrow$ *floor-log n = Suc (floor-log (n div 2))*
⟨*proof*⟩

lemma *floor-log-twice* [*simp*]: $n \neq 0 \Longrightarrow$ *floor-log (2 * n) = Suc (floor-log n)*
⟨*proof*⟩

lemma *floor-log-half* [*simp*]: *floor-log (n div 2) = floor-log n − 1*
⟨*proof*⟩

lemma *floor-log-power* [*simp*]: *floor-log (2 ^ n) = n*
⟨*proof*⟩

lemma *floor-log-mono*: *mono floor-log*
⟨*proof*⟩

lemma *floor-log-exp2-le*:
assumes *n > 0*
shows *2 ^ floor-log n ≤ n*
⟨*proof*⟩

lemma *floor-log-exp2-gt*: *2 * 2 ^ floor-log n > n*
⟨*proof*⟩

lemma *floor-log-exp2-ge*: *2 * 2 ^ floor-log n ≥ n*
⟨*proof*⟩

lemma *floor-log-le-iff*: $m \leq n \Longrightarrow$ *floor-log m ≤ floor-log n*
⟨*proof*⟩

lemma *floor-log-eqI*:
assumes *n > 0 2^k ≤ n n < 2 * 2^k*
shows *floor-log n = k*
⟨*proof*⟩

lemma *floor-log-altdef*: *floor-log n = (if n = 0 then 0 else nat ⌊log 2 (real-of-nat n)⌋)*
⟨*proof*⟩

26.2 Discrete square root

definition *floor-sqrt* :: *nat* ⇒ *nat*
  where *floor-sqrt n* = *Max* {*m*. $m^2 \leq n$}

lemma *floor-sqrt-aux*:
  fixes *n* :: *nat*
  shows *finite* {*m*. $m^2 \leq n$} **and** {*m*. $m^2 \leq n$} ≠ {}
⟨*proof*⟩

lemma *floor-sqrt-unique*:
  assumes *m*^2 ≤ *n* *n* < (*Suc m*)^2
  shows *floor-sqrt n* = *m*
⟨*proof*⟩

lemma *floor-sqrt-inverse-power2* [*simp*]: *floor-sqrt* (n^2) = *n*
⟨*proof*⟩

lemma *floor-sqrt-zero* [*simp*]: *floor-sqrt 0* = *0*
  ⟨*proof*⟩

lemma *floor-sqrt-one* [*simp*]: *floor-sqrt 1* = *1*
  ⟨*proof*⟩

lemma *floor-sqrt-Suc-0* [*simp*]:
  ‹*floor-sqrt* (*Suc 0*) = *1*›
  ⟨*proof*⟩

lemma *mono-floor-sqrt*: *mono floor-sqrt*
⟨*proof*⟩

lemma *mono-floor-sqrt'*: *m* ≤ *n* ⟹ *floor-sqrt m* ≤ *floor-sqrt n*
  ⟨*proof*⟩

lemma *floor-sqrt-greater-zero-iff* [*simp*]: *floor-sqrt n* > *0* ⟷ *n* > *0*
⟨*proof*⟩

lemma *floor-sqrt-power2-le* [*simp*]: (*floor-sqrt n*)^2 ≤ *n*
⟨*proof*⟩

lemma *floor-sqrt-le*: *floor-sqrt n* ≤ *n*
  ⟨*proof*⟩

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
  ⟨*proof*⟩

lemma *le-floor-sqrt-iff*: *x* ≤ *floor-sqrt y* ⟷ *x*^2 ≤ *y*
⟨*proof*⟩

lemma *le-floor-sqrtI*: $x\hat{}2 \leq y \Longrightarrow x \leq$ *floor-sqrt* y
⟨*proof*⟩

lemma *floor-sqrt-le-iff*:
‹*floor-sqrt* $y \leq x \longleftrightarrow (\forall z.\ z^2 \leq y \longrightarrow z \leq x)$›
⟨*proof*⟩

lemma *floor-sqrt-leI*:
$(\bigwedge z.\ z\hat{}2 \leq y \Longrightarrow z \leq x) \Longrightarrow$ *floor-sqrt* $y \leq x$
⟨*proof*⟩

lemma *floor-sqrt-less-eq-half*:
‹*floor-sqrt* $n \leq$ *Suc* n *div* 2›
⟨*proof*⟩

lemma *floor-sqrt-Suc*:
floor-sqrt (*Suc* n) = (*if* $\exists m.$ *Suc* $n = m\hat{}2$ *then Suc* (*floor-sqrt* n) *else floor-sqrt* n)
⟨*proof*⟩

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* $n \in \{m..<m + q\}$ *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›
⟨*proof*⟩

lemma *floor-sqrt-between-singleton*:
‹*floor-sqrt-between* m (*Suc* 0) n =
(*if* $m^2 \leq n \wedge n <$ (*Suc* m)^2 *then* m *else* 0)›
⟨*proof*⟩

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** ‹$q > 0$›
⟨*proof*⟩

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*⟩

lemma [*code*]:
‹*floor-sqrt n* = *floor-sqrt-between 0* (*Suc* (*Suc n div 2*)) *n*›
⟨*proof*⟩

end

27 Pi and Function Sets

theory *FuncSet*
imports *Main*
abbrevs $PiE = Pi_E$
and $PIE = \Pi_E$
begin

definition $Pi :: 'a\ set \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b)\ set$
where $Pi\ A\ B = \{f.\ \forall x.\ x \in A \longrightarrow f\ x \in B\ x\}$

definition $extensional :: 'a\ set \Rightarrow ('a \Rightarrow 'b)\ set$
where $extensional\ A = \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = undefined\}$

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
-Pi :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ (*'a* ⇒ *'b*) *set*
(‹(‹*indent=3 notation=‹binder* Π∈››Π -∈-./ -)› *10*)
-lam :: *pttrn* ⇒ *'a set* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*)
(‹(‹*indent=3 notation=‹binder* λ∈››λ-∈-./ -)› [*0*, *0*, *3*] *3*)
syntax-consts
-Pi ⇌ *Pi* **and**
-lam ⇌ *restrict*
translations
Π *x*∈*A*. *B* ⇌ *CONST Pi A* (λ*x*. *B*)
λ*x*∈*A*. *f* ⇌ *CONST restrict* (λ*x*. *f*) *A*

definition *compose* :: *'a set* ⇒ (*'b* ⇒ *'c*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'c*)
where *compose A g f* = (λ*x*∈*A*. *g* (*f x*))

27.1 Basic Properties of *Pi*

lemma *Pi-I*[*intro*!]: (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B x*) ⟹ *f* ∈ *Pi A B*
⟨*proof*⟩

lemma *Pi-I'*[*simp*]: (⋀*x*. *x* ∈ *A* ⟶ *f x* ∈ *B x*) ⟹ *f* ∈ *Pi A B*
⟨*proof*⟩

lemma *funcsetI*: (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B*) ⟹ *f* ∈ *A* → *B*
⟨*proof*⟩

lemma *Pi-mem*: *f* ∈ *Pi A B* ⟹ *x* ∈ *A* ⟹ *f x* ∈ *B x*
⟨*proof*⟩

lemma *Pi-iff*: *f* ∈ *Pi I X* ⟷ (∀ *i*∈*I*. *f i* ∈ *X i*)
⟨*proof*⟩

lemma *PiE* [*elim*]: *f* ∈ *Pi A B* ⟹ (*f x* ∈ *B x* ⟹ *Q*) ⟹ (*x* ∉ *A* ⟹ *Q*) ⟹ *Q*
⟨*proof*⟩

lemma *Pi-cong*: (⋀*w*. *w* ∈ *A* ⟹ *f w* = *g w*) ⟹ *f* ∈ *Pi A B* ⟷ *g* ∈ *Pi A B*
⟨*proof*⟩

lemma *funcset-id* [*simp*]: (λ*x*. *x*) ∈ *A* → *A*
⟨*proof*⟩

lemma *funcset-mem*: *f* ∈ *A* → *B* ⟹ *x* ∈ *A* ⟹ *f x* ∈ *B*
⟨*proof*⟩

lemma *funcset-image*: $f \in A \to B \Longrightarrow f \mathbin{`} A \subseteq B$
⟨*proof*⟩

lemma *image-subset-iff-funcset*: $F \mathbin{`} A \subseteq B \longleftrightarrow F \in A \to B$
⟨*proof*⟩

lemma *funcset-to-empty-iff*: $A \to \{\} = (\textit{if } A{=}\{\} \textit{ then } \textit{UNIV} \textit{ else } \{\})$
⟨*proof*⟩

lemma *Pi-eq-empty*[*simp*]: $(\Pi\ x \in A.\ B\ x) = \{\} \longleftrightarrow (\exists x{\in}A.\ B\ x = \{\})$
⟨*proof*⟩

lemma *Pi-empty* [*simp*]: $\textit{Pi}\ \{\}\ B = \textit{UNIV}$
⟨*proof*⟩

lemma *Pi-Int*: $\textit{Pi}\ I\ E \cap \textit{Pi}\ I\ F = (\Pi\ i{\in}I.\ E\ i \cap F\ i)$
⟨*proof*⟩

lemma *Pi-UN*:
fixes $A :: \textit{nat} \Rightarrow 'i \Rightarrow 'a\ \textit{set}$
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*⟩

lemma *Pi-UNIV* [*simp*]: $A \to \textit{UNIV} = \textit{UNIV}$
⟨*proof*⟩

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow \textit{Pi}\ A\ B \subseteq \textit{Pi}\ A\ C$
⟨*proof*⟩

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \Longrightarrow \textit{Pi}\ A\ B \subseteq \textit{Pi}\ A'\ B$
⟨*proof*⟩

lemma *prod-final*:
assumes *1*: $\textit{fst} \circ f \in \textit{Pi}\ A\ B$
and *2*: $\textit{snd} \circ f \in \textit{Pi}\ A\ C$
shows $f \in (\Pi\ z \in A.\ B\ z \times C\ z)$
⟨*proof*⟩

lemma *Pi-split-domain*[*simp*]: $x \in \textit{Pi}\ (I \cup J)\ X \longleftrightarrow x \in \textit{Pi}\ I\ X \wedge x \in \textit{Pi}\ J\ X$
⟨*proof*⟩

lemma *Pi-split-insert-domain*[*simp*]: $x \in \textit{Pi}\ (\textit{insert}\ i\ I)\ X \longleftrightarrow x \in \textit{Pi}\ I\ X \wedge x\ i \in X\ i$
⟨*proof*⟩

lemma *Pi-cancel-fupd-range*[*simp*]: $i \notin I \Longrightarrow x \in Pi\ I\ (B(i := b)) \longleftrightarrow x \in Pi\ I\ B$
⟨*proof*⟩

lemma *Pi-cancel-fupd*[*simp*]: $i \notin I \Longrightarrow x(i := a) \in Pi\ I\ B \longleftrightarrow x \in Pi\ I\ B$
⟨*proof*⟩

lemma *Pi-fupd-iff*: $i \in I \Longrightarrow f \in Pi\ I\ (B(i := A)) \longleftrightarrow f \in Pi\ (I - \{i\})\ B \wedge f\ i \in A$
⟨*proof*⟩

lemma *fst-Pi*: $fst \in A \times B \to A$ **and** *snd-Pi*: $snd \in A \times B \to B$
⟨*proof*⟩

27.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \to B \Longrightarrow g \in B \to C \Longrightarrow compose\ A\ g\ f \in A \to C$
⟨*proof*⟩

lemma *compose-assoc*:
assumes $f \in A \to B$
shows $compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
⟨*proof*⟩

lemma *compose-eq*: $x \in A \Longrightarrow compose\ A\ g\ f\ x = g\ (f\ x)$
⟨*proof*⟩

lemma *surj-compose*: $f\ `\ A = B \Longrightarrow g\ `\ B = C \Longrightarrow compose\ A\ g\ f\ `\ A = C$
⟨*proof*⟩

27.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \Longrightarrow (\bigwedge i.\ i \in J =simp=> f\ i = g\ i) \Longrightarrow restrict\ f\ I = restrict\ g\ J$
⟨*proof*⟩

lemma *restrictI*[*intro!*]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
⟨*proof*⟩

lemma *restrict-apply*[*simp*]: $(\lambda y{\in}A.\ f\ y)\ x = (if\ x \in A\ then\ f\ x\ else\ undefined)$
⟨*proof*⟩

lemma *restrict-apply'*: $x \in A \Longrightarrow (\lambda y{\in}A.\ f\ y)\ x = f\ x$
⟨*proof*⟩

lemma *restrict-ext*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
⟨*proof*⟩

lemma *restrict-UNIV*: *restrict f UNIV* = *f*
⟨*proof*⟩

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* (*restrict f A*) *A* ⟷ *inj-on f A*
⟨*proof*⟩

lemma *inj-on-restrict-iff*: *A* ⊆ *B* ⟹ *inj-on* (*restrict f B*) *A* ⟷ *inj-on f A*
⟨*proof*⟩

lemma *Id-compose*: *f* ∈ *A* → *B* ⟹ *f* ∈ *extensional A* ⟹ *compose A* (λ*y*∈*B*. *y*) *f* = *f*
⟨*proof*⟩

lemma *compose-Id*: *g* ∈ *A* → *B* ⟹ *g* ∈ *extensional A* ⟹ *compose A g* (λ*x*∈*A*. *x*) = *g*
⟨*proof*⟩

lemma *image-restrict-eq* [*simp*]: (*restrict f A*) ‘ *A* = *f* ‘ *A*
⟨*proof*⟩

lemma *restrict-restrict*[*simp*]: *restrict* (*restrict f A*) *B* = *restrict f* (*A* ∩ *B*)
⟨*proof*⟩

lemma *restrict-fupd*[*simp*]: *i* ∉ *I* ⟹ *restrict* (*f* (*i* := *x*)) *I* = *restrict f I*
⟨*proof*⟩

lemma *restrict-upd*[*simp*]: *i* ∉ *I* ⟹ (*restrict f I*)(*i* := *y*) = *restrict* (*f*(*i* := *y*)) (*insert i I*)
⟨*proof*⟩

lemma *restrict-Pi-cancel*: *restrict x I* ∈ *Pi I A* ⟷ *x* ∈ *Pi I A*
⟨*proof*⟩

lemma *sum-restrict′* [*simp*]: *sum′* (λ*i*∈*I*. *g i*) *I* = *sum′* (λ*i*. *g i*) *I*
⟨*proof*⟩

lemma *prod-restrict′* [*simp*]: *prod′* (λ*i*∈*I*. *g i*) *I* = *prod′* (λ*i*. *g i*) *I*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *bij-betw-imp-funcset*: *bij-betw f A B* ⟹ *f* ∈ *A* → *B*
⟨*proof*⟩

lemma *inj-on-compose*: *bij-betw f A B* ⟹ *inj-on g B* ⟹ *inj-on* (*compose A g f*) *A*
⟨*proof*⟩

lemma *bij-betw-compose*: *bij-betw f A B* ⟹ *bij-betw g B C* ⟹ *bij-betw* (*compose A g f*) *A C*
⟨*proof*⟩

lemma *bij-betw-restrict-eq* [*simp*]: *bij-betw* (*restrict f A*) *A B* = *bij-betw f A B*
⟨*proof*⟩

27.5 Extensionality

lemma *extensional-empty*[*simp*]: *extensional* {} = {λ*x*. *undefined*}
⟨*proof*⟩

lemma *extensional-arb*: *f* ∈ *extensional A* ⟹ *x* ∉ *A* ⟹ *f x* = *undefined*
⟨*proof*⟩

lemma *restrict-extensional* [*simp*]: *restrict f A* ∈ *extensional A*
⟨*proof*⟩

lemma *compose-extensional* [*simp*]: *compose A f g* ∈ *extensional A*
⟨*proof*⟩

lemma *extensionalityI*:
assumes *f* ∈ *extensional A*
and *g* ∈ *extensional A*
and ⋀*x*. *x* ∈ *A* ⟹ *f x* = *g x*
shows *f* = *g*
⟨*proof*⟩

lemma *extensional-restrict*: *f* ∈ *extensional A* ⟹ *restrict f A* = *f*
⟨*proof*⟩

lemma *extensional-subset*: *f* ∈ *extensional A* ⟹ *A* ⊆ *B* ⟹ *f* ∈ *extensional B*
⟨*proof*⟩

lemma *inv-into-funcset*: *f* ' *A* = *B* ⟹ (λ*x*∈*B*. *inv-into A f x*) ∈ *B* → *A*
⟨*proof*⟩

lemma *compose-inv-into-id*: *bij-betw f A B* ⟹ *compose A* (λ*y*∈*B*. *inv-into A f y*) *f* = (λ*x*∈*A*. *x*)
⟨*proof*⟩

lemma *compose-id-inv-into*: f ' $A = B \Longrightarrow$ *compose* B f $(\lambda y{\in}B.$ *inv-into* A f $y)$ $= (\lambda x{\in}B.\ x)$
⟨*proof*⟩

lemma *extensional-insert*[*intro*, *simp*]:
assumes $a \in$ *extensional* (*insert* i I)
shows $a(i := b) \in$ *extensional* (*insert* i I)
⟨*proof*⟩

lemma *extensional-Int*[*simp*]: *extensional* $I \cap$ *extensional* $I' =$ *extensional* $(I \cap I')$
⟨*proof*⟩

lemma *extensional-UNIV*[*simp*]: *extensional UNIV* $=$ *UNIV*
⟨*proof*⟩

lemma *restrict-extensional-sub*[*intro*]: $A \subseteq B \Longrightarrow$ *restrict* f $A \in$ *extensional* B
⟨*proof*⟩

lemma *extensional-insert-undefined*[*intro*, *simp*]:
$a \in$ *extensional* (*insert* i I) $\Longrightarrow$ $a(i :=$ *undefined*$) \in$ *extensional* I
⟨*proof*⟩

lemma *extensional-insert-cancel*[*intro*, *simp*]:
$a \in$ *extensional* $I \Longrightarrow$ $a \in$ *extensional* (*insert* i I)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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* $\Pi_E{\in}$››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*
⟨*proof*⟩

lemma *PiE-empty-domain*[*simp*]: $Pi_E\ \{\}\ T = \{\lambda x.\ undefined\}$
⟨*proof*⟩

lemma *PiE-UNIV-domain*: $Pi_E\ UNIV\ T = Pi\ UNIV\ T$
⟨*proof*⟩

lemma *PiE-empty-range*[*simp*]: $i \in I \Longrightarrow F\ i = \{\} \Longrightarrow (\Pi_E\ i{\in}I.\ F\ i) = \{\}$
⟨*proof*⟩

lemma *PiE-eq-empty-iff*: $Pi_E\ I\ F = \{\} \longleftrightarrow (\exists\, i{\in}I.\ F\ i = \{\})$
⟨*proof*⟩

lemma *PiE-arb*: $f \in Pi_E\ S\ T \Longrightarrow x \notin S \Longrightarrow f\ x = undefined$
⟨*proof*⟩

lemma *PiE-mem*: $f \in Pi_E\ S\ T \Longrightarrow x \in S \Longrightarrow f\ x \in T\ x$
⟨*proof*⟩

lemma *PiE-fun-upd*: $y \in T\ x \Longrightarrow f \in Pi_E\ S\ T \Longrightarrow f(x := y) \in Pi_E\ (insert\ x\ S)\ T$
⟨*proof*⟩

lemma *fun-upd-in-PiE*: $x \notin S \Longrightarrow f \in Pi_E\ (insert\ x\ S)\ T \Longrightarrow f(x := undefined) \in Pi_E\ S\ T$
⟨*proof*⟩

lemma *PiE-insert-eq*: $Pi_E\ (insert\ x\ S)\ T = (\lambda(y,\ g).\ g(x := y))\ `\ (T\ x \times Pi_E\ S\ T)$
⟨*proof*⟩

lemma *PiE-Int*: $Pi_E\ I\ A \cap Pi_E\ I\ B = Pi_E\ I\ (\lambda x.\ A\ x \cap B\ x)$
⟨*proof*⟩

lemma *PiE-cong*: $(\bigwedge i.\ i{\in}I \Longrightarrow A\ i = B\ i) \Longrightarrow Pi_E\ I\ A = Pi_E\ I\ B$
⟨*proof*⟩

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** $f\ x = undefined$
⟨*proof*⟩

lemma *PiE-I*[*intro*!]:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\bigwedge x.\ x \notin A \Longrightarrow f\ x = undefined) \Longrightarrow f \in Pi_E\ A\ B$
⟨*proof*⟩

lemma *PiE-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow Pi_E\ A\ B \subseteq Pi_E\ A\ C$
⟨*proof*⟩

lemma *PiE-iff*: $f \in Pi_E\ I\ X \longleftrightarrow (\forall i{\in}I.\ f\ i \in X\ i) \wedge f \in extensional\ I$
⟨*proof*⟩

lemma *restrict-PiE-iff*: $restrict\ f\ I \in Pi_E\ I\ X \longleftrightarrow (\forall i \in I.\ f\ i \in X\ i)$
⟨*proof*⟩

lemma *ext-funcset-to-sing-iff* [*simp*]: $A \to_E \{a\} = \{\lambda x{\in}A.\ a\}$
⟨*proof*⟩

lemma *PiE-restrict*[*simp*]: $f \in Pi_E\ A\ B \Longrightarrow restrict\ f\ A = f$
⟨*proof*⟩

lemma *restrict-PiE*[*simp*]: $restrict\ f\ I \in Pi_E\ I\ S \longleftrightarrow f \in Pi\ I\ S$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *extensional-funcset-fun-upd-restricts-rangeI*:
$\forall y \in S.\ f\ x \neq f\ y \Longrightarrow f \in (insert\ x\ S) \to_E T \Longrightarrow f(x := undefined) \in S \to_E (T - \{f\ x\})$
⟨*proof*⟩

lemma *extensional-funcset-fun-upd-extends-rangeI*:
assumes $a \in T\ f \in S \to_E (T - \{a\})$
shows $f(x := a) \in insert\ x\ S \to_E\ T$
⟨*proof*⟩

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* ⟷ - ∨ *?rhs*)
⟨*proof*⟩

lemma *PiE-eq*: $PiE\ I\ S = PiE\ I\ T \longleftrightarrow PiE\ I\ S = \{\} \wedge PiE\ I\ T = \{\} \vee (\forall i \in I.\ S\ i = T\ i)$
⟨*proof*⟩

lemma *PiE-UNIV* [*simp*]: $PiE\ UNIV\ (\lambda i.\ UNIV) = UNIV$
⟨*proof*⟩

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*⟩

lemma *PiE-singleton*:
assumes $f \in extensional\ A$
shows $PiE\ A\ (\lambda x.\ \{f\ x\}) = \{f\}$
⟨*proof*⟩

lemma *PiE-eq-singleton*: $(\Pi_E\ i{\in}I.\ S\ i) = \{\lambda i{\in}I.\ f\ i\} \longleftrightarrow (\forall i{\in}I.\ S\ i = \{f\ i\})$
⟨*proof*⟩

lemma *PiE-over-singleton-iff*: $(\Pi_E\ x{\in}\{a\}.\ B\ x) = (\bigcup b \in B\ a.\ \{\lambda x \in \{a\}.\ b\})$
⟨*proof*⟩

lemma *all-PiE-elements*:
$(\forall z \in PiE\ I\ S.\ \forall i \in I.\ P\ i\ (z\ i)) \longleftrightarrow PiE\ I\ S = \{\} \vee (\forall i \in I.\ \forall x \in S\ i.\ P\ i\ x)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *PiE-ext*: $[\![x \in PiE\ k\ s;\ y \in PiE\ k\ s;\ \bigwedge i.\ i \in k \Longrightarrow x\ i = y\ i]\!] \Longrightarrow x = y$
⟨*proof*⟩

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* $(f(x := a))\ S$
⟨*proof*⟩

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*⟩

lemma *extensional-funcset-extend-domain-inj-onI*:
assumes $x \notin S$
shows $\mathit{inj\text{-}on}\ (\lambda(y,\ g).\ g(x := y))\ \{(y,\ g).\ y \in T \wedge g \in S \to_E (T - \{y\}) \wedge \mathit{inj\text{-}on}\ g\ S\}$
⟨*proof*⟩

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*⟩

lemma *function-factors-left*: $(\forall x\ y.\ (g\ x = g\ y) \longrightarrow (f\ x = f\ y)) \longleftrightarrow (\exists h.\ f = h \circ g)$
⟨*proof*⟩

lemma *function-factors-right-gen*: $(\forall x.\ P\ x \longrightarrow (\exists y.\ g\ y = f\ x)) \longleftrightarrow (\exists h.\ \forall x.\ P\ x \longrightarrow f\ x = g(h\ x))$
⟨*proof*⟩

lemma *function-factors-right*: $(\forall x.\ \exists y.\ g\ y = f\ x) \longleftrightarrow (\exists h.\ f = g \circ h)$
⟨*proof*⟩

lemma *restrict-compose-right*: $\mathit{restrict}\ (g \circ \mathit{restrict}\ f\ S)\ S = \mathit{restrict}\ (g \circ f)\ S$
⟨*proof*⟩

lemma *restrict-compose-left*: $f\ `\ S \subseteq T \Longrightarrow \mathit{restrict}\ (\mathit{restrict}\ g\ T \circ f)\ S = \mathit{restrict}\ (g \circ f)\ S$
⟨*proof*⟩

27.7.3 Cardinality

lemma *finite-PiE*: $\mathit{finite}\ S \Longrightarrow (\bigwedge i.\ i \in S \Longrightarrow \mathit{finite}\ (T\ i)) \Longrightarrow \mathit{finite}\ (\Pi_E\ i \in S.\ T\ i)$
⟨*proof*⟩

lemma *inj-combinator*: $x \notin S \Longrightarrow \mathit{inj\text{-}on}\ (\lambda(y,\ g).\ g(x := y))\ (T\ x \times \mathit{Pi}_E\ S\ T)$
⟨*proof*⟩

lemma *card-PiE*: $\mathit{finite}\ S \Longrightarrow \mathit{card}\ (\Pi_E\ i \in S.\ T\ i) = (\prod\ i{\in}S.\ \mathit{card}\ (T\ i))$
⟨*proof*⟩

lemma *card-funcsetE*: *finite* $A \Longrightarrow$ *card* $(A \rightarrow_E B) =$ *card* $B \mathbin{\hat{}}$ *card* A
⟨*proof*⟩

lemma *card-inj-on-subset-funcset*:
assumes *finB*: *finite* B
and *finC*: *finite* C
and AB: $A \subseteq B$
shows *card* $\{f \in B \rightarrow_E C.$ *inj-on* $f\ A\} =$
card $C\hat{}$(*card* $B -$ *card* $A) *$ *prod* $((-)$ (*card* $C))$ $\{0 ..<$ *card* $A\}$
⟨*proof*⟩

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 $f \in A \rightarrow B$ *finite* A *finite* B $B \neq \{\}$
shows $\exists\, y{\in}B.$ *card* $(f -` \{y\} \cap A) *$ *card* $B \geq$ *card* A
⟨*proof*⟩

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 $(\prod x{\in}A. \sum y{\in}B\ x.\ f\ x\ y) = (\sum g{\in}PiE\ A\ B. \prod x{\in}A.\ f\ x\ (g\ x))$
⟨*proof*⟩

end

28 Partitions and Disjoint Sets

theory *Disjoint-Sets*
imports *FuncSet*
begin

lemma *mono-imp-UN-eq-last*: *mono* $A \Longrightarrow (\bigcup i{\leq}n.\ A\ i) = A\ n$
⟨*proof*⟩

28.1 Set of Disjoint Sets

abbreviation *disjoint* :: $'a$ *set set* $\Rightarrow$ *bool* **where** *disjoint* $\equiv$ *pairwise disjnt*

lemma *disjoint-def*: *disjoint* $A \longleftrightarrow (\forall\, a{\in}A.\ \forall\, b{\in}A.\ a \neq b \longrightarrow a \cap b = \{\})$
⟨*proof*⟩

lemma *disjointI*:
$(\bigwedge a\ b.\ a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}) \Longrightarrow$ *disjoint* A
⟨*proof*⟩

lemma *disjointD*:
disjoint $A \Longrightarrow a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}$
⟨*proof*⟩

lemma *disjoint-image*: *inj-on* f $(\bigcup A) \Longrightarrow$ *disjoint* $A \Longrightarrow$ *disjoint* $((`)\ f\ `\ A)$
⟨*proof*⟩

lemma assumes *disjoint* $(A \cup B)$
shows *disjoint-unionD1*: *disjoint* A **and** *disjoint-unionD2*: *disjoint* B
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *psubset-Union-pairwise-disjoint*:
assumes $\mathcal{B}$: *pairwise disjnt* $\mathcal{B}$ **and** $\mathcal{A} \subset \mathcal{B} - \{\{\}\}$
shows $\bigcup\mathcal{A} \subset \bigcup\mathcal{B}$
⟨*proof*⟩

28.1.1 Family of Disjoint Sets

definition *disjoint-family-on* :: $('i \Rightarrow 'a\ set) \Rightarrow 'i\ set \Rightarrow bool$ **where**
disjoint-family-on $A\ S \longleftrightarrow (\forall\, m{\in}S.\ \forall\, n{\in}S.\ m \neq n \longrightarrow A\ m \cap A\ n = \{\})$

abbreviation *disjoint-family* $A \equiv$ *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** $x \in A$ *disjnt* C $(B\ x)$
⟨*proof*⟩

lemma *disjoint-family-onD*:
disjoint-family-on A I $\Longrightarrow$ $i \in I \Longrightarrow j \in I \Longrightarrow i \neq j \Longrightarrow A\ i \cap A\ j = \{\}$
⟨*proof*⟩

lemma *disjoint-family-subset*: *disjoint-family A* $\Longrightarrow$ $(\bigwedge x.\ B\ x \subseteq A\ x) \Longrightarrow$ *disjoint-family B*
⟨*proof*⟩

lemma *disjoint-family-on-insert*:
$i \notin I \Longrightarrow$ *disjoint-family-on A* (*insert i I*) $\longleftrightarrow$ $A\ i \cap (\bigcup i{\in}I.\ A\ i) = \{\} \wedge$ *disjoint-family-on A I*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *disjoint-family-on-mono*:
$A \subseteq B \Longrightarrow$ *disjoint-family-on f B* $\Longrightarrow$ *disjoint-family-on f A*
⟨*proof*⟩

lemma *disjoint-family-Suc*:
$(\bigwedge n.\ A\ n \subseteq A\ (Suc\ n)) \Longrightarrow$ *disjoint-family* $(\lambda i.\ A\ (Suc\ i) - A\ i)$
⟨*proof*⟩

lemma *disjoint-family-on-disjoint-image*:
disjoint-family-on A I $\Longrightarrow$ *disjoint* $(A\ `\ I)$
⟨*proof*⟩

lemma *disjoint-family-on-vimageI*: *disjoint-family-on F I* $\Longrightarrow$ *disjoint-family-on* $(\lambda i.\ f\ -`\ F\ i)\ I$
⟨*proof*⟩

lemma *disjoint-image-disjoint-family-on*:
assumes *d*: *disjoint* $(A\ `\ I)$ **and** *i*: *inj-on A I*
shows *disjoint-family-on A I*
⟨*proof*⟩

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*⟩

lemma *card-UN-disjoint'*:
assumes *disjoint-family-on A I* $\bigwedge i.\ i \in I \Longrightarrow$ *finite* $(A\ i)$ *finite I*
shows *card* $(\bigcup i{\in}I.\ A\ i) = (\sum i{\in}I.\ card\ (A\ i))$

⟨*proof*⟩

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*⟩

lemma *distinct-list-bind*:
assumes *distinct xs* $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow$ *distinct* (*f x*)
disjoint-family-on (*set* ∘ *f*) (*set xs*)
shows *distinct* (*List.bind xs f*)
⟨*proof*⟩

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* ($\bigcup i{\in}I.\ A\ i$) ($\bigcup i{\in}I.\ A'\ i$)
⟨*proof*⟩

lemma *disjoint-union*: *disjoint C* $\Longrightarrow$ *disjoint B* $\Longrightarrow \bigcup C \cap \bigcup B = \{\} \Longrightarrow$ *disjoint* (*C* ∪ *B*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Union-disjoint-sets*:
assumes $\forall\, A{\in}C.$ *finite A* **and** *disjoint C*
shows *F g* ($\bigcup C$) = (*F* ∘ *F*) *g C*
⟨*proof*⟩

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*⟩

28.2 Construct Disjoint Sequences

definition *disjointed* :: (*nat* ⇒ *′a set*) ⇒ *nat* ⇒ *′a set* **where**

disjointed A n = *A n* − (⋃ *i*∈{*0*..<*n*}. *A i*)

lemma *finite-UN-disjointed-eq*: (⋃ *i*∈{*0*..<*n*}. *disjointed A i*) = (⋃ *i*∈{*0*..<*n*}. *A i*)
⟨*proof*⟩

lemma *UN-disjointed-eq*: (⋃ *i*. *disjointed A i*) = (⋃ *i*. *A i*)
⟨*proof*⟩

lemma *less-disjoint-disjointed*: *m* < *n* ⟹ *disjointed A m* ∩ *disjointed A n* = {}
⟨*proof*⟩

lemma *disjoint-family-disjointed*: *disjoint-family* (*disjointed A*)
⟨*proof*⟩

lemma *disjointed-subset*: *disjointed A n* ⊆ *A n*
⟨*proof*⟩

lemma *disjointed-0*[*simp*]: *disjointed A 0* = *A 0*
⟨*proof*⟩

lemma *disjointed-mono*: *mono A* ⟹ *disjointed A* (*Suc n*) = *A* (*Suc n*) − *A n*
⟨*proof*⟩

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-on A P ⟷ ⋃ *P* = *A* ∧ *disjoint P* ∧ {} ∉ *P*

lemma *partition-onI*:
⋃ *P* = *A* ⟹ (⋀*p q*. *p* ∈ *P* ⟹ *q* ∈ *P* ⟹ *p* ≠ *q* ⟹ *disjnt p q*) ⟹ {} ∉ *P*
⟹ *partition-on A P*
⟨*proof*⟩

lemma *partition-onD1*: *partition-on A P* ⟹ *A* = ⋃ *P*
⟨*proof*⟩

lemma *partition-onD2*: *partition-on A P* ⟹ *disjoint P*
⟨*proof*⟩

lemma *partition-onD3*: *partition-on A P* ⟹ {} ∉ *P*
⟨*proof*⟩

28.4 Constructions of partitions

lemma *partition-on-empty*: *partition-on* {} *P* ⟷ *P* = {}
⟨*proof*⟩

lemma *partition-on-space*: $A \neq \{\} \Longrightarrow$ *partition-on* A $\{A\}$
⟨*proof*⟩

lemma *partition-on-singletons*: *partition-on* A $((\lambda x.\ \{x\})\ `\ A)$
⟨*proof*⟩

lemma *partition-on-transform*:
assumes P: *partition-on* A P
assumes *F-UN*: $\bigcup(F\ `\ P) = F\ (\bigcup P)$ **and** *F-disjnt*: $\bigwedge p\ q.\ p \in P \Longrightarrow q \in P \Longrightarrow$ *disjnt* p $q \Longrightarrow$ *disjnt* $(F\ p)$ $(F\ q)$
shows *partition-on* $(F\ A)$ $(F\ `\ P - \{\{\}\})$
⟨*proof*⟩

lemma *partition-on-restrict*: *partition-on* A $P \Longrightarrow$ *partition-on* $(B \cap A)$ $((\cap)\ B\ `\ P - \{\{\}\})$
⟨*proof*⟩

lemma *partition-on-vimage*: *partition-on* A $P \Longrightarrow$ *partition-on* $(f -`\ A)$ $((-`)\ f\ `\ P - \{\{\}\})$
⟨*proof*⟩

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*⟩

lemma *partition-on-insert*:
assumes *disjnt* p $(\bigcup P)$
shows *partition-on* A $(\mathit{insert}\ p\ P) \longleftrightarrow$ *partition-on* $(A-p)$ $P \wedge p \subseteq A \wedge p \neq \{\}$
⟨*proof*⟩

28.5 Finiteness of partitions

lemma *finitely-many-partition-on*:
assumes *finite* A
shows *finite* $\{P.$ *partition-on* A $P\}$
⟨*proof*⟩

lemma *finite-elements*: *finite* $A \Longrightarrow$ *partition-on* A $P \Longrightarrow$ *finite* P
⟨*proof*⟩

lemma *product-partition*:
assumes *partition-on* A P **and** $\bigwedge p.\ p \in P \Longrightarrow$ *finite* p
shows *card* $A = (\sum p{\in}P.$ *card* $p)$
⟨*proof*⟩

28.6 Equivalence of partitions and equivalence classes

lemma *partition-on-quotient*:

assumes *r*: *equiv A r*
shows *partition-on A* (*A* // *r*)
⟨*proof*⟩

lemma *equiv-partition-on*:
assumes *P*: *partition-on A P*
shows *equiv A* {(*x*, *y*). ∃ *p* ∈ *P*. *x* ∈ *p* ∧ *y* ∈ *p*}
⟨*proof*⟩

lemma *partition-on-eq-quotient*:
assumes *P*: *partition-on A P*
shows *A* // {(*x*, *y*). ∃ *p* ∈ *P*. *x* ∈ *p* ∧ *y* ∈ *p*} = *P*
⟨*proof*⟩

lemma *partition-on-alt*: *partition-on A P* ⟷ (∃ *r*. *equiv A r* ∧ *P* = *A* // *r*)
⟨*proof*⟩

lemma (**in** *comm-monoid-set*) *partition*:
assumes *finite X partition-on X A*
shows *F g X* = *F* (λ*B*. *F g B*) *A*
⟨*proof*⟩

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*: ⋀*x*. *x* ∈ *X* ⟹ *f* (*h x*) + *f x* = *0*
assumes *h*: ⋀*x*. *x* ∈ *X* ⟹ *h x* ∈ *X* ⋀*x*. *x* ∈ *X* ⟹ *h* (*h x*) = *x* ⋀*x*. *x* ∈ *X* ⟹ *h x* ≠ *x*
shows (∑ *x*∈*X*. *f x*) = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *refines-asym1*:
assumes *refines A P Q refines A Q P*
shows $P \subseteq Q$
⟨*proof*⟩

lemma *refines-asym*: ⟦*refines A P Q*; *refines A Q P*⟧ ⟹ $P{=}Q$
⟨*proof*⟩

lemma *refines-trans*: ⟦*refines A P Q*; *refines A Q R*⟧ ⟹ *refines A P R*
⟨*proof*⟩

lemma *refines-obtains-subset*:
assumes *refines A P Q q* ∈ *Q*
shows *partition-on q* {*p* ∈ *P*. *p* ⊆ *q*}
⟨*proof*⟩

28.8 The coarsest common refinement of a set of partitions

definition *common-refinement* :: *'a set set set* ⇒ *'a set set*
where *common-refinement* $\mathcal{P} \equiv (\bigcup f \in (\Pi_E\ P{\in}\mathcal{P}.\ P).\ \{\bigcap\ (f\ `\ \mathcal{P})\}) - \{\{\}\}$

With non-extensional function space

lemma *common-refinement*: *common-refinement* $\mathcal{P} = (\bigcup f \in (\Pi\ P{\in}\mathcal{P}.\ P).\ \{\bigcap\ (f\ `\ \mathcal{P})\}) - \{\{\}\}$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *common-refinement-exists*: ⟦*X* ∈ *common-refinement* $\mathcal{P}$; *P* ∈ $\mathcal{P}$⟧ ⟹ ∃ *R*∈*P*. *X* ⊆ *R*
⟨*proof*⟩

lemma *Union-common-refinement*: $\bigcup$ (*common-refinement* $\mathcal{P}$) = ($\bigcap$ *P*∈$\mathcal{P}$. $\bigcup$ *P*)
⟨*proof*⟩

lemma *partition-on-common-refinement*:
assumes *A*: $\bigwedge$*P*. *P* ∈ $\mathcal{P}$ ⟹ *partition-on A P* **and** $\mathcal{P} \neq \{\}$
shows *partition-on A* (*common-refinement* $\mathcal{P}$)
⟨*proof*⟩

lemma *refines-common-refinement*:
assumes $\bigwedge$*P*. *P* ∈ $\mathcal{P}$ ⟹ *partition-on A P P* ∈ $\mathcal{P}$
shows *refines A* (*common-refinement* $\mathcal{P}$) *P*
⟨*proof*⟩

The common refinement is itself refined by any other

lemma *common-refinement-coarsest*:
assumes $\bigwedge$*P*. *P* ∈ $\mathcal{P}$ ⟹ *partition-on A P partition-on A R* $\bigwedge$*P*. *P* ∈ $\mathcal{P}$ ⟹ *refines A R P* $\mathcal{P} \neq \{\}$
shows *refines A R* (*common-refinement* $\mathcal{P}$)

⟨*proof*⟩

lemma *finite-common-refinement*:
assumes *finite* $\mathcal{P}$ $\bigwedge P.\ P \in \mathcal{P} \Longrightarrow$ *finite* P
shows *finite* (*common-refinement* $\mathcal{P}$)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

setup-lifting *type-definition-fset*

29.2 Basic operations and type class instantiations

instantiation *fset* :: (*finite*) *finite*
begin
instance ⟨*proof*⟩
end

instantiation *fset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

lift-definition *bot-fset* :: *'a fset* **is** {} **parametric** *empty-transfer* ⟨*proof*⟩

lift-definition *less-eq-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **is** *subset-eq* **parametric** *subset-transfer*
⟨*proof*⟩

definition *less-fset* :: *'a fset* ⇒ *'a fset* ⇒ *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*) ===> (=)) (⊂) (<)
⟨*proof*⟩

lift-definition *sup-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *union* **parametric** *union-transfer*
⟨*proof*⟩

lift-definition *inf-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *inter* **parametric** *inter-transfer*
⟨*proof*⟩

lift-definition *minus-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *minus* **parametric** *Diff-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

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 ⟨*proof*⟩
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* {})
⟨*proof*⟩

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* {})
⟨*proof*⟩

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* ⟨*proof*⟩

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* {}) ⟨*proof*⟩

lift-definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Sup A*) *then Sup A else* {}
parametric *Sup-fset-transfer* ⟨*proof*⟩

lemma *finite-Sup*: ∃ *z*. *finite z* ∧ (∀ *a*. *a* ∈ *X* ⟶ *a* ≤ *z*) ⟹ *finite* (*Sup X*)
⟨*proof*⟩

lemma *transfer-bdd-below*[*transfer-rule*]: (*rel-set* (*pcr-fset* (=)) ===> (=)) *bdd-below bdd-below*
⟨*proof*⟩

end

instance
⟨*proof*⟩
end

instantiation *fset* :: (*finite*) *complete-lattice*
begin

lift-definition *top-fset* :: *'a fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer UNIV-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

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* ⟨*proof*⟩

instance
  ⟨*proof*⟩
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*
  ⟨*proof*⟩

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*

⟨*ML*⟩

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*

$\forall A|\subseteq|B.\ P \rightharpoonup \forall A.\ A\ |\subseteq|\ B \longrightarrow P$
$\exists A|\subseteq|B.\ P \rightharpoonup \exists A.\ A\ |\subseteq|\ B \wedge P$

context includes *lifting-syntax*
begin

lemma *fmember-transfer0*[*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-unique A*
  shows (*A* ===> *pcr-fset A* ===> (=)) (∈) (|∈|)
  ⟨*proof*⟩

lemma *fBall-transfer0*[*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-unique A*
  shows (*pcr-fset A* ===> (*A* ===> (=)) ===> (=)) (*Ball*) (*fBall*)
  ⟨*proof*⟩

lemma *fBex-transfer0*[*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-unique A*
  shows (*pcr-fset A* ===> (*A* ===> (=)) ===> (=)) (*Bex*) (*fBex*)
  ⟨*proof*⟩

lift-definition *ffilter* :: ('*a* ⇒ *bool*) ⇒ '*a fset* ⇒ '*a fset* **is** *Set.filter*
  parametric *Lifting-Set.filter-transfer* ⟨*proof*⟩

lift-definition *fPow* :: '*a fset* ⇒ '*a fset fset* **is** *Pow* **parametric** *Pow-transfer*
⟨*proof*⟩

lift-definition *fcard* :: '*a fset* ⇒ *nat* **is** *card* **parametric** *card-transfer* ⟨*proof*⟩

lift-definition *fimage* :: ('*a* ⇒ '*b*) ⇒ '*a fset* ⇒ '*b fset* (**infixr** ‹|`|› *90*) **is** *image*
  parametric *image-transfer* ⟨*proof*⟩

lift-definition *fthe-elem* :: '*a fset* ⇒ '*a* **is** *the-elem* ⟨*proof*⟩

lift-definition *fbind* :: '*a fset* ⇒ ('*a* ⇒ '*b fset*) ⇒ '*b fset* **is** *Set.bind* **parametric**
bind-transfer
⟨*proof*⟩

lift-definition *ffUnion* :: '*a fset fset* ⇒ '*a fset* **is** *Union* **parametric** *Union-transfer*
⟨*proof*⟩

lift-definition *ffold* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ '*a fset* ⇒ '*b* **is** *Finite-Set.fold* ⟨*proof*⟩

lift-definition *fset-of-list* :: '*a list* ⇒ '*a fset* **is** *set* ⟨*proof*⟩

lift-definition *sorted-list-of-fset* :: '*a*::*linorder fset* ⇒ '*a list* **is** *sorted-list-of-set*
⟨*proof*⟩

29.4 Transferred lemmas from Set.thy

lemma *fset-eqI*: $(\bigwedge x.\ (x \mid\in\mid A) = (x \mid\in\mid B)) \Longrightarrow A = B$
⟨*proof*⟩

lemma *fset-eq-iff*[*no-atp*]: $(A = B) = (\forall x.\ (x \mid\in\mid A) = (x \mid\in\mid B))$
⟨*proof*⟩

lemma *fBallI*[*no-atp*]: $(\bigwedge x.\ x \mid\in\mid A \Longrightarrow P\ x) \Longrightarrow fBall\ A\ P$
⟨*proof*⟩

lemma *fbspec*[*no-atp*]: $fBall\ A\ P \Longrightarrow x \mid\in\mid A \Longrightarrow P\ x$
⟨*proof*⟩

lemma *fBallE*[*no-atp*]: $fBall\ A\ P \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow (x \mid\notin\mid A \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *fBexI*[*no-atp*]: $P\ x \Longrightarrow x \mid\in\mid A \Longrightarrow fBex\ A\ P$
⟨*proof*⟩

lemma *rev-fBexI*[*no-atp*]: $x \mid\in\mid A \Longrightarrow P\ x \Longrightarrow fBex\ A\ P$
⟨*proof*⟩

lemma *fBexCI*[*no-atp*]: $(fBall\ A\ (\lambda x.\ \neg\ P\ x) \Longrightarrow P\ a) \Longrightarrow a \mid\in\mid A \Longrightarrow fBex\ A\ P$
⟨*proof*⟩

lemma *fBexE*[*no-atp*]: $fBex\ A\ P \Longrightarrow (\bigwedge x.\ x \mid\in\mid A \Longrightarrow P\ x \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *fBall-triv*[*no-atp*]: $fBall\ A\ (\lambda x.\ P) = ((\exists x.\ x \mid\in\mid A) \longrightarrow P)$
⟨*proof*⟩

lemma *fBex-triv*[*no-atp*]: $fBex\ A\ (\lambda x.\ P) = ((\exists x.\ x \mid\in\mid A) \wedge P)$
⟨*proof*⟩

lemma *fBex-triv-one-point1*[*no-atp*]: $fBex\ A\ (\lambda x.\ x = a) = (a \mid\in\mid A)$
⟨*proof*⟩

lemma *fBex-triv-one-point2*[*no-atp*]: $fBex\ A\ ((=)\ a) = (a \mid\in\mid A)$
⟨*proof*⟩

lemma *fBex-one-point1*[*no-atp*]: $fBex\ A\ (\lambda x.\ x = a \wedge P\ x) = (a \mid\in\mid A \wedge P\ a)$
⟨*proof*⟩

lemma *fBex-one-point2*[*no-atp*]: $fBex\ A\ (\lambda x.\ a = x \wedge P\ x) = (a \mid\in\mid A \wedge P\ a)$
⟨*proof*⟩

lemma *fBall-one-point1*[*no-atp*]: $fBall\ A\ (\lambda x.\ x = a \longrightarrow P\ x) = (a \mid\in\mid A \longrightarrow P\ a)$
⟨*proof*⟩

lemma *fBall-one-point2*[*no-atp*]: *fBall A* ($\lambda x.\ a = x \longrightarrow P\ x$) = ($a$ |∈| $A \longrightarrow P\ a$)
⟨*proof*⟩

lemma *fBall-conj-distrib*: *fBall A* ($\lambda x.\ P\ x \wedge Q\ x$) = (*fBall A P* ∧ *fBall A Q*)
⟨*proof*⟩

lemma *fBex-disj-distrib*: *fBex A* ($\lambda x.\ P\ x \vee Q\ x$) = (*fBex A P* ∨ *fBex A Q*)
⟨*proof*⟩

lemma *fBall-cong*[*fundef-cong*]: $A = B \Longrightarrow (\bigwedge x.\ x$ |∈| $B \Longrightarrow P\ x = Q\ x) \Longrightarrow$ *fBall A P* = *fBall B Q*
⟨*proof*⟩

lemma *fBex-cong*[*fundef-cong*]: $A = B \Longrightarrow (\bigwedge x.\ x$ |∈| $B \Longrightarrow P\ x = Q\ x) \Longrightarrow$ *fBex A P* = *fBex B Q*
⟨*proof*⟩

lemma *fsubsetI*[*intro*!]: $(\bigwedge x.\ x$ |∈| $A \Longrightarrow x$ |∈| $B) \Longrightarrow A$ |⊆| B
⟨*proof*⟩

lemma *fsubsetD*[*elim*, *intro?*]: A |⊆| $B \Longrightarrow c$ |∈| $A \Longrightarrow c$ |∈| B
⟨*proof*⟩

lemma *rev-fsubsetD*[*no-atp*,*intro?*]: c |∈| $A \Longrightarrow A$ |⊆| $B \Longrightarrow c$ |∈| B
⟨*proof*⟩

lemma *fsubsetCE*[*no-atp*,*elim*]: A |⊆| $B \Longrightarrow (c$ |∉| $A \Longrightarrow P) \Longrightarrow (c$ |∈| $B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *fsubset-eq*[*no-atp*]: (A |⊆| B) = *fBall A* ($\lambda x.\ x$ |∈| B)
⟨*proof*⟩

lemma *contra-fsubsetD*[*no-atp*]: A |⊆| $B \Longrightarrow c$ |∉| $B \Longrightarrow c$ |∉| A
⟨*proof*⟩

lemma *fsubset-refl*: A |⊆| A
⟨*proof*⟩

lemma *fsubset-trans*: A |⊆| $B \Longrightarrow B$ |⊆| $C \Longrightarrow A$ |⊆| C
⟨*proof*⟩

lemma *fset-rev-mp*: c |∈| $A \Longrightarrow A$ |⊆| $B \Longrightarrow c$ |∈| B
⟨*proof*⟩

lemma *fset-mp*: A |⊆| $B \Longrightarrow c$ |∈| $A \Longrightarrow c$ |∈| B
⟨*proof*⟩

lemma *fsubset-not-fsubset-eq*[*code*]: $(A \mid\subset\mid B) = (A \mid\subseteq\mid B \wedge \neg\ B \mid\subseteq\mid A)$
⟨*proof*⟩

lemma *eq-fmem-trans*: $a = b \Longrightarrow b \mid\in\mid A \Longrightarrow a \mid\in\mid A$
⟨*proof*⟩

lemma *fsubset-antisym*[*intro*!]: $A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid A \Longrightarrow A = B$
⟨*proof*⟩

lemma *fequalityD1*: $A = B \Longrightarrow A \mid\subseteq\mid B$
⟨*proof*⟩

lemma *fequalityD2*: $A = B \Longrightarrow B \mid\subseteq\mid A$
⟨*proof*⟩

lemma *fequalityE*: $A = B \Longrightarrow (A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *fequalityCE*[*elim*]:
$A = B \Longrightarrow (c \mid\in\mid A \Longrightarrow c \mid\in\mid B \Longrightarrow P) \Longrightarrow (c \mid\notin\mid A \Longrightarrow c \mid\notin\mid B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *eqfset-imp-iff*: $A = B \Longrightarrow (x \mid\in\mid A) = (x \mid\in\mid B)$
⟨*proof*⟩

lemma *eqfelem-imp-iff*: $x = y \Longrightarrow (x \mid\in\mid A) = (y \mid\in\mid A)$
⟨*proof*⟩

lemma *fempty-iff*[*simp*]: $(c \mid\in\mid \{||\}) = \mathit{False}$
⟨*proof*⟩

lemma *fempty-fsubsetI*[*iff*]: $\{||\} \mid\subseteq\mid x$
⟨*proof*⟩

lemma *equalsffemptyI*: $(\bigwedge y.\ y \mid\in\mid A \Longrightarrow \mathit{False}) \Longrightarrow A = \{||\}$
⟨*proof*⟩

lemma *equalsffemptyD*: $A = \{||\} \Longrightarrow a \mid\notin\mid A$
⟨*proof*⟩

lemma *fBall-fempty*[*simp*]: $\mathit{fBall}\ \{||\}\ P = \mathit{True}$
⟨*proof*⟩

lemma *fBex-fempty*[*simp*]: $\mathit{fBex}\ \{||\}\ P = \mathit{False}$
⟨*proof*⟩

lemma *fPow-iff*[*iff*]: $(A \mid\in\mid \mathit{fPow}\ B) = (A \mid\subseteq\mid B)$
⟨*proof*⟩

lemma *fPowI*: $A \mid\subseteq\mid B \Longrightarrow A \mid\in\mid fPow\ B$
⟨*proof*⟩

lemma *fPowD*: $A \mid\in\mid fPow\ B \Longrightarrow A \mid\subseteq\mid B$
⟨*proof*⟩

lemma *fPow-bottom*: $\{||\} \mid\in\mid fPow\ B$
⟨*proof*⟩

lemma *fPow-top*: $A \mid\in\mid fPow\ A$
⟨*proof*⟩

lemma *fPow-not-fempty*: $fPow\ A \neq \{||\}$
⟨*proof*⟩

lemma *finter-iff*[*simp*]: $(c \mid\in\mid A \mid\cap\mid B) = (c \mid\in\mid A \wedge c \mid\in\mid B)$
⟨*proof*⟩

lemma *finterI*[*intro*!]: $c \mid\in\mid A \Longrightarrow c \mid\in\mid B \Longrightarrow c \mid\in\mid A \mid\cap\mid B$
⟨*proof*⟩

lemma *finterD1*: $c \mid\in\mid A \mid\cap\mid B \Longrightarrow c \mid\in\mid A$
⟨*proof*⟩

lemma *finterD2*: $c \mid\in\mid A \mid\cap\mid B \Longrightarrow c \mid\in\mid B$
⟨*proof*⟩

lemma *finterE*[*elim*!]: $c \mid\in\mid A \mid\cap\mid B \Longrightarrow (c \mid\in\mid A \Longrightarrow c \mid\in\mid B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *funion-iff*[*simp*]: $(c \mid\in\mid A \mid\cup\mid B) = (c \mid\in\mid A \vee c \mid\in\mid B)$
⟨*proof*⟩

lemma *funionI1*[*elim?*]: $c \mid\in\mid A \Longrightarrow c \mid\in\mid A \mid\cup\mid B$
⟨*proof*⟩

lemma *funionI2*[*elim?*]: $c \mid\in\mid B \Longrightarrow c \mid\in\mid A \mid\cup\mid B$
⟨*proof*⟩

lemma *funionCI*[*intro*!]: $(c \mid\notin\mid B \Longrightarrow c \mid\in\mid A) \Longrightarrow c \mid\in\mid A \mid\cup\mid B$
⟨*proof*⟩

lemma *funionE*[*elim*!]: $c \mid\in\mid A \mid\cup\mid B \Longrightarrow (c \mid\in\mid A \Longrightarrow P) \Longrightarrow (c \mid\in\mid B \Longrightarrow P)$
$\Longrightarrow P$
⟨*proof*⟩

lemma *fminus-iff*[*simp*]: $(c \mid\in\mid A \mid-\mid B) = (c \mid\in\mid A \wedge c \mid\notin\mid B)$
⟨*proof*⟩

lemma *fminusI*[*intro*!]: $c \mid\in\mid A \Longrightarrow c \mid\notin\mid B \Longrightarrow c \mid\in\mid A \mid-\mid B$
⟨*proof*⟩

lemma *fminusD1*: $c \mid\in\mid A \mid-\mid B \Longrightarrow c \mid\in\mid A$
⟨*proof*⟩

lemma *fminusD2*: $c \mid\in\mid A \mid-\mid B \Longrightarrow c \mid\in\mid B \Longrightarrow P$
⟨*proof*⟩

lemma *fminusE*[*elim*!]: $c \mid\in\mid A \mid-\mid B \Longrightarrow (c \mid\in\mid A \Longrightarrow c \mid\notin\mid B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *finsert-iff*[*simp*]: $(a \mid\in\mid \mathit{finsert}\ b\ A) = (a = b \vee a \mid\in\mid A)$
⟨*proof*⟩

lemma *finsertI1*: $a \mid\in\mid \mathit{finsert}\ a\ B$
⟨*proof*⟩

lemma *finsertI2*: $a \mid\in\mid B \Longrightarrow a \mid\in\mid \mathit{finsert}\ b\ B$
⟨*proof*⟩

lemma *finsertE*[*elim*!]: $a \mid\in\mid \mathit{finsert}\ b\ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a \mid\in\mid A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *finsertCI*[*intro*!]: $(a \mid\notin\mid B \Longrightarrow a = b) \Longrightarrow a \mid\in\mid \mathit{finsert}\ b\ B$
⟨*proof*⟩

lemma *fsubset-finsert-iff*:
$(A \mid\subseteq\mid \mathit{finsert}\ x\ B) = (\mathit{if}\ x \mid\in\mid A\ \mathit{then}\ A \mid-\mid \{|x|\} \mid\subseteq\mid B\ \mathit{else}\ A \mid\subseteq\mid B)$
⟨*proof*⟩

lemma *finsert-ident*: $x \mid\notin\mid A \Longrightarrow x \mid\notin\mid B \Longrightarrow (\mathit{finsert}\ x\ A = \mathit{finsert}\ x\ B) = (A = B)$
⟨*proof*⟩

lemma *fsingletonI*[*intro*!,*no-atp*]: $a \mid\in\mid \{|a|\}$
⟨*proof*⟩

lemma *fsingletonD*[*dest*!,*no-atp*]: $b \mid\in\mid \{|a|\} \Longrightarrow b = a$
⟨*proof*⟩

lemma *fsingleton-iff*: $(b \mid\in\mid \{|a|\}) = (b = a)$
⟨*proof*⟩

lemma *fsingleton-inject*[*dest*!]: $\{|a|\} = \{|b|\} \Longrightarrow a = b$
⟨*proof*⟩

lemma *fsingleton-finsert-inj-eq*[*iff*,*no-atp*]: $(\{|b|\} = \mathit{finsert}\ a\ A) = (a = b \wedge A\ |\subseteq|\ \{|b|\})$
⟨*proof*⟩

lemma *fsingleton-finsert-inj-eq'*[*iff*,*no-atp*]: $(\mathit{finsert}\ a\ A = \{|b|\}) = (a = b \wedge A\ |\subseteq|\ \{|b|\})$
⟨*proof*⟩

lemma *fsubset-fsingletonD*: $A\ |\subseteq|\ \{|x|\} \Longrightarrow A = \{||\} \vee A = \{|x|\}$
⟨*proof*⟩

lemma *fminus-single-finsert*: $A\ |-|\ \{|x|\}\ |\subseteq|\ B \Longrightarrow A\ |\subseteq|\ \mathit{finsert}\ x\ B$
⟨*proof*⟩

lemma *fdoubleton-eq-iff*: $(\{|a,\ b|\} = \{|c,\ d|\}) = (a = c \wedge b = d \vee a = d \wedge b = c)$
⟨*proof*⟩

lemma *funion-fsingleton-iff*:
$(A\ |\cup|\ B = \{|x|\}) = (A = \{||\} \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{||\} \vee A = \{|x|\} \wedge B = \{|x|\})$
⟨*proof*⟩

lemma *fsingleton-funion-iff*:
$(\{|x|\} = A\ |\cup|\ B) = (A = \{||\} \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{||\} \vee A = \{|x|\} \wedge B = \{|x|\})$
⟨*proof*⟩

lemma *fimage-eqI*[*simp*, *intro*]: $b = f\ x \Longrightarrow x\ |\in|\ A \Longrightarrow b\ |\in|\ f\ |`|\ A$
⟨*proof*⟩

lemma *fimageI*: $x\ |\in|\ A \Longrightarrow f\ x\ |\in|\ f\ |`|\ A$
⟨*proof*⟩

lemma *rev-fimage-eqI*: $x\ |\in|\ A \Longrightarrow b = f\ x \Longrightarrow b\ |\in|\ f\ |`|\ A$
⟨*proof*⟩

lemma *fimageE*[*elim*!]: $b\ |\in|\ f\ |`|\ A \Longrightarrow (\bigwedge x.\ b = f\ x \Longrightarrow x\ |\in|\ A \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$
⟨*proof*⟩

lemma *Compr-fimage-eq*: $\{x.\ x\ |\in|\ f\ |`|\ A \wedge P\ x\} = f\ `\ \{x.\ x\ |\in|\ A \wedge P\ (f\ x)\}$
⟨*proof*⟩

lemma *fimage-funion*: $f\ |`|\ (A\ |\cup|\ B) = f\ |`|\ A\ |\cup|\ f\ |`|\ B$
⟨*proof*⟩

lemma *fimage-iff*: $(z\ |\in|\ f\ |`|\ A) = \mathit{fBex}\ A\ (\lambda x.\ z = f\ x)$
⟨*proof*⟩

lemma *fimage-fsubset-iff*[*no-atp*]: $(f \mid`\mid A \mid\subseteq\mid B) = fBall\ A\ (\lambda x.\ f\ x \mid\in\mid B)$
⟨*proof*⟩

lemma *fimage-fsubsetI*: $(\bigwedge x.\ x \mid\in\mid A \Longrightarrow f\ x \mid\in\mid B) \Longrightarrow f \mid`\mid A \mid\subseteq\mid B$
⟨*proof*⟩

lemma *fimage-ident*[*simp*]: $(\lambda x.\ x) \mid`\mid Y = Y$
⟨*proof*⟩

lemma *if-split-fmem1*: $((if\ Q\ then\ x\ else\ y) \mid\in\mid b) = ((Q \longrightarrow x \mid\in\mid b) \wedge (\neg\ Q \longrightarrow y \mid\in\mid b))$
⟨*proof*⟩

lemma *if-split-fmem2*: $(a \mid\in\mid (if\ Q\ then\ x\ else\ y)) = ((Q \longrightarrow a \mid\in\mid x) \wedge (\neg\ Q \longrightarrow a \mid\in\mid y))$
⟨*proof*⟩

lemma *pfsubsetI*[*intro*!,*no-atp*]: $A \mid\subseteq\mid B \Longrightarrow A \neq B \Longrightarrow A \mid\subset\mid B$
⟨*proof*⟩

lemma *pfsubsetE*[*elim*!,*no-atp*]: $A \mid\subset\mid B \Longrightarrow (A \mid\subseteq\mid B \Longrightarrow \neg\ B \mid\subseteq\mid A \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *pfsubset-finsert-iff*:
$(A \mid\subset\mid finsert\ x\ B) =$
$(if\ x \mid\in\mid B\ then\ A \mid\subset\mid B\ else\ if\ x \mid\in\mid A\ then\ A \mid-\mid \{|x|\} \mid\subset\mid B\ else\ A \mid\subseteq\mid B)$
⟨*proof*⟩

lemma *pfsubset-eq*: $(A \mid\subset\mid B) = (A \mid\subseteq\mid B \wedge A \neq B)$
⟨*proof*⟩

lemma *pfsubset-imp-fsubset*: $A \mid\subset\mid B \Longrightarrow A \mid\subseteq\mid B$
⟨*proof*⟩

lemma *pfsubset-trans*: $A \mid\subset\mid B \Longrightarrow B \mid\subset\mid C \Longrightarrow A \mid\subset\mid C$
⟨*proof*⟩

lemma *pfsubsetD*: $A \mid\subset\mid B \Longrightarrow c \mid\in\mid A \Longrightarrow c \mid\in\mid B$
⟨*proof*⟩

lemma *pfsubset-fsubset-trans*: $A \mid\subset\mid B \Longrightarrow B \mid\subseteq\mid C \Longrightarrow A \mid\subset\mid C$
⟨*proof*⟩

lemma *fsubset-pfsubset-trans*: $A \mid\subseteq\mid B \Longrightarrow B \mid\subset\mid C \Longrightarrow A \mid\subset\mid C$
⟨*proof*⟩

lemma *pfsubset-imp-ex-fmem*: $A \mid\subset\mid B \Longrightarrow \exists\, b.\ b \mid\in\mid B \mid-\mid A$

⟨*proof*⟩

lemma *fimage-fPow-mono*: $f\ |`|\ A\ |\subseteq|\ B \Longrightarrow (|`|)\ f\ |`|\ fPow\ A\ |\subseteq|\ fPow\ B$
⟨*proof*⟩

lemma *fimage-fPow-surj*: $f\ |`|\ A = B \Longrightarrow (|`|)\ f\ |`|\ fPow\ A = fPow\ B$
⟨*proof*⟩

lemma *fsubset-finsertI*: $B\ |\subseteq|\ finsert\ a\ B$
⟨*proof*⟩

lemma *fsubset-finsertI2*: $A\ |\subseteq|\ B \Longrightarrow A\ |\subseteq|\ finsert\ b\ B$
⟨*proof*⟩

lemma *fsubset-finsert*: $x\ |\notin|\ A \Longrightarrow (A\ |\subseteq|\ finsert\ x\ B) = (A\ |\subseteq|\ B)$
⟨*proof*⟩

lemma *funion-upper1*: $A\ |\subseteq|\ A\ |\cup|\ B$
⟨*proof*⟩

lemma *funion-upper2*: $B\ |\subseteq|\ A\ |\cup|\ B$
⟨*proof*⟩

lemma *funion-least*: $A\ |\subseteq|\ C \Longrightarrow B\ |\subseteq|\ C \Longrightarrow A\ |\cup|\ B\ |\subseteq|\ C$
⟨*proof*⟩

lemma *finter-lower1*: $A\ |\cap|\ B\ |\subseteq|\ A$
⟨*proof*⟩

lemma *finter-lower2*: $A\ |\cap|\ B\ |\subseteq|\ B$
⟨*proof*⟩

lemma *finter-greatest*: $C\ |\subseteq|\ A \Longrightarrow C\ |\subseteq|\ B \Longrightarrow C\ |\subseteq|\ A\ |\cap|\ B$
⟨*proof*⟩

lemma *fminus-fsubset*: $A\ |-|\ B\ |\subseteq|\ A$
⟨*proof*⟩

lemma *fminus-fsubset-conv*: $(A\ |-|\ B\ |\subseteq|\ C) = (A\ |\subseteq|\ B\ |\cup|\ C)$
⟨*proof*⟩

lemma *fsubset-fempty*[*simp*]: $(A\ |\subseteq|\ \{||\}) = (A = \{||\})$
⟨*proof*⟩

lemma *not-pfsubset-fempty*[*iff*]: $\neg\ A\ |\subset|\ \{||\}$
⟨*proof*⟩

lemma *finsert-is-funion*: $finsert\ a\ A = \{|a|\}\ |\cup|\ A$
⟨*proof*⟩

lemma *finsert-not-fempty*[*simp*]: *finsert a A* ≠ {|||}
⟨*proof*⟩

lemma *fempty-not-finsert*: {|||} ≠ *finsert a A*
⟨*proof*⟩

lemma *finsert-absorb*: *a* |∈| *A* ⟹ *finsert a A* = *A*
⟨*proof*⟩

lemma *finsert-absorb2*[*simp*]: *finsert x* (*finsert x A*) = *finsert x A*
⟨*proof*⟩

lemma *finsert-commute*: *finsert x* (*finsert y A*) = *finsert y* (*finsert x A*)
⟨*proof*⟩

lemma *finsert-fsubset*[*simp*]: (*finsert x A* |⊆| *B*) = (*x* |∈| *B* ∧ *A* |⊆| *B*)
⟨*proof*⟩

lemma *finsert-inter-finsert*[*simp*]: *finsert a A* |∩| *finsert a B* = *finsert a* (*A* |∩| *B*)
⟨*proof*⟩

lemma *finsert-disjoint*[*simp*,*no-atp*]:
(*finsert a A* |∩| *B* = {|||}) = (*a* |∉| *B* ∧ *A* |∩| *B* = {|||})
({|||} = *finsert a A* |∩| *B*) = (*a* |∉| *B* ∧ {|||} = *A* |∩| *B*)
⟨*proof*⟩

lemma *disjoint-finsert*[*simp*,*no-atp*]:
(*B* |∩| *finsert a A* = {|||}) = (*a* |∉| *B* ∧ *B* |∩| *A* = {|||})
({|||} = *A* |∩| *finsert b B*) = (*b* |∉| *A* ∧ {|||} = *A* |∩| *B*)
⟨*proof*⟩

lemma *fimage-fempty*[*simp*]: *f* |‘| {|||} = {|||}
⟨*proof*⟩

lemma *fimage-finsert*[*simp*]: *f* |‘| *finsert a B* = *finsert* (*f a*) (*f* |‘| *B*)
⟨*proof*⟩

lemma *fimage-constant*: *x* |∈| *A* ⟹ (λ*x*. *c*) |‘| *A* = {|*c*|}
⟨*proof*⟩

lemma *fimage-constant-conv*: (λ*x*. *c*) |‘| *A* = (*if A* = {|||} *then* {|||} *else* {|*c*|})
⟨*proof*⟩

lemma *fimage-fimage*: *f* |‘| *g* |‘| *A* = (λ*x*. *f* (*g x*)) |‘| *A*
⟨*proof*⟩

lemma *finsert-fimage*[*simp*]: *x* |∈| *A* ⟹ *finsert* (*f x*) (*f* |‘| *A*) = *f* |‘| *A*
⟨*proof*⟩

lemma *fimage-is-fempty*[*iff*]: $(f \mid`\mid A = \{||\}) = (A = \{||\})$
⟨*proof*⟩

lemma *fempty-is-fimage*[*iff*]: $(\{||\} = f \mid`\mid A) = (A = \{||\})$
⟨*proof*⟩

lemma *fimage-cong*: $M = N \Longrightarrow (\bigwedge x.\ x \mid\in\mid N \Longrightarrow f\,x = g\,x) \Longrightarrow f \mid`\mid M = g \mid`\mid N$
⟨*proof*⟩

lemma *fimage-finter-fsubset*: $f \mid`\mid (A \mid\cap\mid B) \mid\subseteq\mid f \mid`\mid A \mid\cap\mid f \mid`\mid B$
⟨*proof*⟩

lemma *fimage-fminus-fsubset*: $f \mid`\mid A \mid-\mid f \mid`\mid B \mid\subseteq\mid f \mid`\mid (A \mid-\mid B)$
⟨*proof*⟩

lemma *finter-absorb*: $A \mid\cap\mid A = A$
⟨*proof*⟩

lemma *finter-left-absorb*: $A \mid\cap\mid (A \mid\cap\mid B) = A \mid\cap\mid B$
⟨*proof*⟩

lemma *finter-commute*: $A \mid\cap\mid B = B \mid\cap\mid A$
⟨*proof*⟩

lemma *finter-left-commute*: $A \mid\cap\mid (B \mid\cap\mid C) = B \mid\cap\mid (A \mid\cap\mid C)$
⟨*proof*⟩

lemma *finter-assoc*: $A \mid\cap\mid B \mid\cap\mid C = A \mid\cap\mid (B \mid\cap\mid C)$
⟨*proof*⟩

lemma *finter-ac*:
$A \mid\cap\mid B \mid\cap\mid C = A \mid\cap\mid (B \mid\cap\mid C)$
$A \mid\cap\mid (A \mid\cap\mid B) = A \mid\cap\mid B$
$A \mid\cap\mid B = B \mid\cap\mid A$
$A \mid\cap\mid (B \mid\cap\mid C) = B \mid\cap\mid (A \mid\cap\mid C)$
⟨*proof*⟩

lemma *finter-absorb1*: $B \mid\subseteq\mid A \Longrightarrow A \mid\cap\mid B = B$
⟨*proof*⟩

lemma *finter-absorb2*: $A \mid\subseteq\mid B \Longrightarrow A \mid\cap\mid B = A$
⟨*proof*⟩

lemma *finter-fempty-left*: $\{||\} \mid\cap\mid B = \{||\}$
⟨*proof*⟩

lemma *finter-fempty-right*: $A \mid\cap\mid \{||\} = \{||\}$

⟨*proof*⟩

lemma *disjoint-iff-fnot-equal*: $(A \mid\cap\mid B = \{||\}) = \mathit{fBall}\ A\ (\lambda x.\ \mathit{fBall}\ B\ ((\neq)\ x))$
⟨*proof*⟩

lemma *finter-funion-distrib*: $A \mid\cap\mid (B \mid\cup\mid C) = A \mid\cap\mid B \mid\cup\mid (A \mid\cap\mid C)$
⟨*proof*⟩

lemma *finter-funion-distrib2*: $B \mid\cup\mid C \mid\cap\mid A = B \mid\cap\mid A \mid\cup\mid (C \mid\cap\mid A)$
⟨*proof*⟩

lemma *finter-fsubset-iff*[*no-atp*, *simp*]: $(C \mid\subseteq\mid A \mid\cap\mid B) = (C \mid\subseteq\mid A \wedge C \mid\subseteq\mid B)$
⟨*proof*⟩

lemma *funion-absorb*: $A \mid\cup\mid A = A$
⟨*proof*⟩

lemma *funion-left-absorb*: $A \mid\cup\mid (A \mid\cup\mid B) = A \mid\cup\mid B$
⟨*proof*⟩

lemma *funion-commute*: $A \mid\cup\mid B = B \mid\cup\mid A$
⟨*proof*⟩

lemma *funion-left-commute*: $A \mid\cup\mid (B \mid\cup\mid C) = B \mid\cup\mid (A \mid\cup\mid C)$
⟨*proof*⟩

lemma *funion-assoc*: $A \mid\cup\mid B \mid\cup\mid C = A \mid\cup\mid (B \mid\cup\mid C)$
⟨*proof*⟩

lemma *funion-ac*:
$A \mid\cup\mid B \mid\cup\mid C = A \mid\cup\mid (B \mid\cup\mid C)$
$A \mid\cup\mid (A \mid\cup\mid B) = A \mid\cup\mid B$
$A \mid\cup\mid B = B \mid\cup\mid A$
$A \mid\cup\mid (B \mid\cup\mid C) = B \mid\cup\mid (A \mid\cup\mid C)$
⟨*proof*⟩

lemma *funion-absorb1*: $A \mid\subseteq\mid B \Longrightarrow A \mid\cup\mid B = B$
⟨*proof*⟩

lemma *funion-absorb2*: $B \mid\subseteq\mid A \Longrightarrow A \mid\cup\mid B = A$
⟨*proof*⟩

lemma *funion-fempty-left*: $\{||\} \mid\cup\mid B = B$
⟨*proof*⟩

lemma *funion-fempty-right*: $A \mid\cup\mid \{||\} = A$
⟨*proof*⟩

lemma *funion-finsert-left*[*simp*]: $\mathit{finsert}\ a\ B \mid\cup\mid C = \mathit{finsert}\ a\ (B \mid\cup\mid C)$

⟨*proof*⟩

lemma *funion-finsert-right*[*simp*]: $A \mid\cup\mid \mathit{finsert}\ a\ B = \mathit{finsert}\ a\ (A \mid\cup\mid B)$
⟨*proof*⟩

lemma *finter-finsert-left*: $\mathit{finsert}\ a\ B \mid\cap\mid C = (\mathit{if}\ a \mid\in\mid C\ \mathit{then}\ \mathit{finsert}\ a\ (B \mid\cap\mid C)\ \mathit{else}\ B \mid\cap\mid C)$
⟨*proof*⟩

lemma *finter-finsert-left-ifffempty*[*simp*]: $a \mid\notin\mid C \Longrightarrow \mathit{finsert}\ a\ B \mid\cap\mid C = B \mid\cap\mid C$
⟨*proof*⟩

lemma *finter-finsert-left-if1*[*simp*]: $a \mid\in\mid C \Longrightarrow \mathit{finsert}\ a\ B \mid\cap\mid C = \mathit{finsert}\ a\ (B \mid\cap\mid C)$
⟨*proof*⟩

lemma *finter-finsert-right*:
$A \mid\cap\mid \mathit{finsert}\ a\ B = (\mathit{if}\ a \mid\in\mid A\ \mathit{then}\ \mathit{finsert}\ a\ (A \mid\cap\mid B)\ \mathit{else}\ A \mid\cap\mid B)$
⟨*proof*⟩

lemma *finter-finsert-right-ifffempty*[*simp*]: $a \mid\notin\mid A \Longrightarrow A \mid\cap\mid \mathit{finsert}\ a\ B = A \mid\cap\mid B$
⟨*proof*⟩

lemma *finter-finsert-right-if1*[*simp*]: $a \mid\in\mid A \Longrightarrow A \mid\cap\mid \mathit{finsert}\ a\ B = \mathit{finsert}\ a\ (A \mid\cap\mid B)$
⟨*proof*⟩

lemma *funion-finter-distrib*: $A \mid\cup\mid (B \mid\cap\mid C) = A \mid\cup\mid B \mid\cap\mid (A \mid\cup\mid C)$
⟨*proof*⟩

lemma *funion-finter-distrib2*: $B \mid\cap\mid C \mid\cup\mid A = B \mid\cup\mid A \mid\cap\mid (C \mid\cup\mid A)$
⟨*proof*⟩

lemma *funion-finter-crazy*:
$A \mid\cap\mid B \mid\cup\mid (B \mid\cap\mid C) \mid\cup\mid (C \mid\cap\mid A) = A \mid\cup\mid B \mid\cap\mid (B \mid\cup\mid C) \mid\cap\mid (C \mid\cup\mid A)$
⟨*proof*⟩

lemma *fsubset-funion-eq*: $(A \mid\subseteq\mid B) = (A \mid\cup\mid B = B)$
⟨*proof*⟩

lemma *funion-fempty*[*iff*]: $(A \mid\cup\mid B = \{||\}) = (A = \{||\} \wedge B = \{||\})$
⟨*proof*⟩

lemma *funion-fsubset-iff*[*no-atp*, *simp*]: $(A \mid\cup\mid B \mid\subseteq\mid C) = (A \mid\subseteq\mid C \wedge B \mid\subseteq\mid C)$
⟨*proof*⟩

lemma *funion-fminus-finter*: $A \mid-\mid B \mid\cup\mid (A \mid\cap\mid B) = A$
⟨*proof*⟩

lemma *ffunion-empty*[*simp*]: *ffUnion* {||} = {||}
⟨*proof*⟩

lemma *ffunion-mono*: *A* |⊆| *B* ⟹ *ffUnion A* |⊆| *ffUnion B*
⟨*proof*⟩

lemma *ffunion-insert*[*simp*]: *ffUnion* (*finsert a B*) = *a* |∪| *ffUnion B*
⟨*proof*⟩

lemma *fminus-finter2*: *A* |∩| *C* |−| (*B* |∩| *C*) = *A* |∩| *C* |−| *B*
⟨*proof*⟩

lemma *funion-finter-assoc-eq*: (*A* |∩| *B* |∪| *C* = *A* |∩| (*B* |∪| *C*)) = (*C* |⊆| *A*)
⟨*proof*⟩

lemma *fBall-funion*: *fBall* (*A* |∪| *B*) *P* = (*fBall A P* ∧ *fBall B P*)
⟨*proof*⟩

lemma *fBex-funion*: *fBex* (*A* |∪| *B*) *P* = (*fBex A P* ∨ *fBex B P*)
⟨*proof*⟩

lemma *fminus-eq-fempty-iff*[*simp*,*no-atp*]: (*A* |−| *B* = {||}) = (*A* |⊆| *B*)
⟨*proof*⟩

lemma *fminus-cancel*[*simp*]: *A* |−| *A* = {||}
⟨*proof*⟩

lemma *fminus-idemp*[*simp*]: *A* |−| *B* |−| *B* = *A* |−| *B*
⟨*proof*⟩

lemma *fminus-triv*: *A* |∩| *B* = {||} ⟹ *A* |−| *B* = *A*
⟨*proof*⟩

lemma *fempty-fminus*[*simp*]: {||} |−| *A* = {||}
⟨*proof*⟩

lemma *fminus-fempty*[*simp*]: *A* |−| {||} = *A*
⟨*proof*⟩

lemma *fminus-finsertffempty*[*simp*,*no-atp*]: *x* |∉| *A* ⟹ *A* |−| *finsert x B* = *A* |−| *B*
⟨*proof*⟩

lemma *fminus-finsert*: *A* |−| *finsert a B* = *A* |−| *B* |−| {|*a*|}
⟨*proof*⟩

lemma *fminus-finsert2*: *A* |−| *finsert a B* = *A* |−| {|*a*|} |−| *B*
⟨*proof*⟩

lemma *finsert-fminus-if*: *finsert x A |−| B = (if x |∈| B then A |−| B else finsert x (A |−| B))*
⟨*proof*⟩

lemma *finsert-fminus1*[*simp*]: *x |∈| B ⟹ finsert x A |−| B = A |−| B*
⟨*proof*⟩

lemma *finsert-fminus-single*[*simp*]: *finsert a (A |−| {|a|}) = finsert a A*
⟨*proof*⟩

lemma *finsert-fminus*: *a |∈| A ⟹ finsert a (A |−| {|a|}) = A*
⟨*proof*⟩

lemma *fminus-finsert-absorb*: *x |∉| A ⟹ finsert x A |−| {|x|} = A*
⟨*proof*⟩

lemma *fminus-disjoint*[*simp*]: *A |∩| (B |−| A) = {||}*
⟨*proof*⟩

lemma *fminus-partition*: *A |⊆| B ⟹ A |∪| (B |−| A) = B*
⟨*proof*⟩

lemma *double-fminus*: *A |⊆| B ⟹ B |⊆| C ⟹ B |−| (C |−| A) = A*
⟨*proof*⟩

lemma *funion-fminus-cancel*[*simp*]: *A |∪| (B |−| A) = A |∪| B*
⟨*proof*⟩

lemma *funion-fminus-cancel2*[*simp*]: *B |−| A |∪| A = B |∪| A*
⟨*proof*⟩

lemma *fminus-funion*: *A |−| (B |∪| C) = A |−| B |∩| (A |−| C)*
⟨*proof*⟩

lemma *fminus-finter*: *A |−| (B |∩| C) = A |−| B |∪| (A |−| C)*
⟨*proof*⟩

lemma *funion-fminus*: *A |∪| B |−| C = A |−| C |∪| (B |−| C)*
⟨*proof*⟩

lemma *finter-fminus*: *A |∩| B |−| C = A |∩| (B |−| C)*
⟨*proof*⟩

lemma *fminus-finter-distrib*: *C |∩| (A |−| B) = C |∩| A |−| (C |∩| B)*
⟨*proof*⟩

lemma *fminus-finter-distrib2*: *A |−| B |∩| C = A |∩| C |−| (B |∩| C)*
⟨*proof*⟩

lemma *fUNIV-bool*[*no-atp*]: $fUNIV = \{|False,\ True|\}$
⟨*proof*⟩

lemma *fPow-fempty*[*simp*]: $fPow\ \{||\} = \{|\{||\}|\}$
⟨*proof*⟩

lemma *fPow-finsert*: $fPow\ (finsert\ a\ A) = fPow\ A\ |\cup|\ finsert\ a\ |`|\ fPow\ A$
⟨*proof*⟩

lemma *funion-fPow-fsubset*: $fPow\ A\ |\cup|\ fPow\ B\ |\subseteq|\ fPow\ (A\ |\cup|\ B)$
⟨*proof*⟩

lemma *fPow-finter-eq*[*simp*]: $fPow\ (A\ |\cap|\ B) = fPow\ A\ |\cap|\ fPow\ B$
⟨*proof*⟩

lemma *fset-eq-fsubset*: $(A = B) = (A\ |\subseteq|\ B \wedge B\ |\subseteq|\ A)$
⟨*proof*⟩

lemma *fsubset-iff*[*no-atp*]: $(A\ |\subseteq|\ B) = (\forall t.\ t\ |\in|\ A \longrightarrow t\ |\in|\ B)$
⟨*proof*⟩

lemma *fsubset-iff-pfsubset-eq*: $(A\ |\subseteq|\ B) = (A\ |\subset|\ B \vee A = B)$
⟨*proof*⟩

lemma *all-not-fin-conv*[*simp*]: $(\forall x.\ x\ |\notin|\ A) = (A = \{||\})$
⟨*proof*⟩

lemma *ex-fin-conv*: $(\exists x.\ x\ |\in|\ A) = (A \neq \{||\})$
⟨*proof*⟩

lemma *fimage-mono*: $A\ |\subseteq|\ B \Longrightarrow f\ |`|\ A\ |\subseteq|\ f\ |`|\ B$
⟨*proof*⟩

lemma *fPow-mono*: $A\ |\subseteq|\ B \Longrightarrow fPow\ A\ |\subseteq|\ fPow\ B$
⟨*proof*⟩

lemma *finsert-mono*: $C\ |\subseteq|\ D \Longrightarrow finsert\ a\ C\ |\subseteq|\ finsert\ a\ D$
⟨*proof*⟩

lemma *funion-mono*: $A\ |\subseteq|\ C \Longrightarrow B\ |\subseteq|\ D \Longrightarrow A\ |\cup|\ B\ |\subseteq|\ C\ |\cup|\ D$
⟨*proof*⟩

lemma *finter-mono*: $A\ |\subseteq|\ C \Longrightarrow B\ |\subseteq|\ D \Longrightarrow A\ |\cap|\ B\ |\subseteq|\ C\ |\cap|\ D$
⟨*proof*⟩

lemma *fminus-mono*: $A\ |\subseteq|\ C \Longrightarrow D\ |\subseteq|\ B \Longrightarrow A\ |-|\ B\ |\subseteq|\ C\ |-|\ D$
⟨*proof*⟩

lemma *fin-mono*: $A \mid\subseteq\mid B \Longrightarrow x \mid\in\mid A \longrightarrow x \mid\in\mid B$
⟨*proof*⟩

lemma *fthe-felem-eq*[*simp*]: *fthe-elem* $\{|x|\} = x$
⟨*proof*⟩

lemma *fLeast-mono*:
mono f $\Longrightarrow$ *fBex S* $(\lambda x.$ *fBall S* $((\leq)\ x)) \Longrightarrow$ $(LEAST\ y.\ y \mid\in\mid f \mid`\mid S) = f\ (LEAST\ x.\ x \mid\in\mid S)$
⟨*proof*⟩

lemma *fbind-fbind*: *fbind* $(fbind\ A\ B)\ C = fbind\ A\ (\lambda x.\ fbind\ (B\ x)\ C)$
⟨*proof*⟩

lemma *fempty-fbind*[*simp*]: *fbind* $\{||\}\ f = \{||\}$
⟨*proof*⟩

lemma *nonfempty-fbind-const*: $A \neq \{||\} \Longrightarrow fbind\ A\ (\lambda\text{-}.\ B) = B$
⟨*proof*⟩

lemma *fbind-const*: *fbind* $A\ (\lambda\text{-}.\ B) = (if\ A = \{||\}\ then\ \{||\}\ else\ B)$
⟨*proof*⟩

lemma *ffmember-filter*[*simp*]: $(x \mid\in\mid ffilter\ P\ A) = (x \mid\in\mid A \wedge P\ x)$
⟨*proof*⟩

lemma *fequalityI*: $A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid A \Longrightarrow A = B$
⟨*proof*⟩

lemma *fset-of-list-simps*[*simp*]:
fset-of-list $[] = \{||\}$
fset-of-list $(x21\ \#\ x22) =$ *finsert x21* $(fset\text{-}of\text{-}list\ x22)$
⟨*proof*⟩

lemma *fset-of-list-append*[*simp*]: *fset-of-list* $(xs\ @\ ys) =$ *fset-of-list xs* $\mid\cup\mid$ *fset-of-list ys*
⟨*proof*⟩

lemma *fset-of-list-rev*[*simp*]: *fset-of-list* $(rev\ xs) =$ *fset-of-list xs*
⟨*proof*⟩

lemma *fset-of-list-map*[*simp*]: *fset-of-list* $(map\ f\ xs) = f \mid`\mid$ *fset-of-list xs*
⟨*proof*⟩

29.5 Additional lemmas

29.5.1 *ffUnion*

lemma *ffUnion-funion-distrib*[*simp*]: *ffUnion* $(A \mid\cup\mid B) =$ *ffUnion A* $\mid\cup\mid$ *ffUnion B*

⟨*proof*⟩

29.5.2 *fbind*

lemma *fbind-cong*[*fundef-cong*]: $A = B \Longrightarrow (\bigwedge x.\ x \mid\in\mid B \Longrightarrow f\ x = g\ x) \Longrightarrow$ *fbind* $A\ f =$ *fbind* $B\ g$
⟨*proof*⟩

29.5.3 *fsingleton*

lemma *fsingletonE*: $b \mid\in\mid \{|a|\} \Longrightarrow (b = a \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$
⟨*proof*⟩

29.5.4 *femepty*

lemma *fempty-ffilter*[*simp*]: *ffilter* (λ-. *False*) $A = \{||\}$
⟨*proof*⟩

lemma *femptyE* [*elim*!]: $a \mid\in\mid \{||\} \Longrightarrow P$
⟨*proof*⟩

29.5.5 *fset*

lemma *fset-simps*[*simp*]:
fset $\{||\} = \{\}$
fset (*finsert* $x\ X$) = *insert* x (*fset* X)
⟨*proof*⟩

lemma *finite-fset* [*simp*]:
shows *finite* (*fset* S)
⟨*proof*⟩

lemmas *fset-cong* = *fset-inject*

lemma *filter-fset* [*simp*]:
shows *fset* (*ffilter* $P\ xs$) = *Collect* $P \cap$ *fset* xs
⟨*proof*⟩

lemma *inter-fset*[*simp*]: *fset* $(A \mid\cap\mid B) =$ *fset* $A \cap$ *fset* B
⟨*proof*⟩

lemma *union-fset*[*simp*]: *fset* $(A \mid\cup\mid B) =$ *fset* $A \cup$ *fset* B
⟨*proof*⟩

lemma *minus-fset*[*simp*]: *fset* $(A \mid-\mid B) =$ *fset* $A -$ *fset* B
⟨*proof*⟩

29.5.6 *ffilter*

lemma *subset-ffilter*:
ffilter P A $|\subseteq|$ *ffilter Q A* = $(\forall\ x.\ x\ |\in|\ A \longrightarrow P\ x \longrightarrow Q\ x)$
⟨*proof*⟩

lemma *eq-ffilter*:
(*ffilter P A* = *ffilter Q A*) = $(\forall x.\ x\ |\in|\ A \longrightarrow P\ x = Q\ x)$
⟨*proof*⟩

lemma *pfsubset-ffilter*:
$(\bigwedge x.\ x\ |\in|\ A \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow (x\ |\in|\ A \wedge \neg\ P\ x \wedge Q\ x) \Longrightarrow$
ffilter P A $|\subset|$ *ffilter Q A*
⟨*proof*⟩

29.5.7 *fset-of-list*

lemma *fset-of-list-filter*[*simp*]:
fset-of-list (*filter P xs*) = *ffilter P* (*fset-of-list xs*)
⟨*proof*⟩

lemma *fset-of-list-subset*[*intro*]:
set xs $\subseteq$ *set ys* $\Longrightarrow$ *fset-of-list xs* $|\subseteq|$ *fset-of-list ys*
⟨*proof*⟩

lemma *fset-of-list-elem*: $(x\ |\in|$ *fset-of-list xs*$) \longleftrightarrow (x \in$ *set xs*$)$
⟨*proof*⟩

29.5.8 *finsert*

lemma *set-finsert*:
assumes $x\ |\in|\ A$
obtains B **where** $A =$ *finsert x B* **and** $x\ |\notin|\ B$
⟨*proof*⟩

lemma *mk-disjoint-finsert*: $a\ |\in|\ A \Longrightarrow \exists\, B.\ A =$ *finsert a B* $\wedge\ a\ |\notin|\ B$
⟨*proof*⟩

lemma *finsert-eq-iff*:
assumes $a\ |\notin|\ A$ **and** $b\ |\notin|\ B$
shows (*finsert a A = finsert b B*) =
(*if a = b then A = B else* $\exists\, C.\ A =$ *finsert b C* $\wedge\ b\ |\notin|\ C \wedge B =$ *finsert a C* $\wedge$
$a\ |\notin|\ C$)
⟨*proof*⟩

29.5.9 *fimage*

lemma *subset-fimage-iff*: $(B\ |\subseteq|\ f|`|A) = (\exists\ AA.\ AA\ |\subseteq|\ A \wedge B = f|`|AA)$
⟨*proof*⟩

lemma *fimage-strict-mono*:
assumes *inj-on f* (*fset B*) **and** A $|\subset|$ B
shows f $|`|$ $A$ $|\subset|$ $f$ $|`|$ B
— TODO: Configure transfer framework to lift ⟦*inj-on ?f ?B*; *?A* ⊂ *?B*⟧ ⟹ *?f* ‘ *?A* ⊂ *?f* ‘ *?B*.
⟨*proof*⟩

29.5.10 bounded quantification

lemma *bex-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *fBex A* (λ*x*. *P x* ∧ *Q*) = (*fBex A P* ∧ *Q*)
⋀*A P Q*. *fBex A* (λ*x*. *P* ∧ *Q x*) = (*P* ∧ *fBex A Q*)
⋀*P*. *fBex* {||} *P* = *False*
⋀*a B P*. *fBex* (*finsert a B*) *P* = (*P a* ∨ *fBex B P*)
⋀*A P f*. *fBex* (*f* $|`|$ *A*) *P* = *fBex A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *fBex A P*) = *fBall A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *ball-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *fBall A* (λ*x*. *P x* ∨ *Q*) = (*fBall A P* ∨ *Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P* ∨ *Q x*) = (*P* ∨ *fBall A Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P* ⟶ *Q x*) = (*P* ⟶ *fBall A Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P x* ⟶ *Q*) = (*fBex A P* ⟶ *Q*)
⋀*P*. *fBall* {||} *P* = *True*
⋀*a B P*. *fBall* (*finsert a B*) *P* = (*P a* ∧ *fBall B P*)
⋀*A P f*. *fBall* (*f* $|`|$ *A*) *P* = *fBall A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *fBall A P*) = *fBex A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *atomize-fBall*:
(⋀*x*. *x* $|\in|$ *A* ==> *P x*) == *Trueprop* (*fBall A* (λ*x*. *P x*))
⟨*proof*⟩

lemma *fBall-mono*[*mono*]: *P* ≤ *Q* ⟹ *fBall S P* ≤ *fBall S Q*
⟨*proof*⟩

lemma *fBex-mono*[*mono*]: *P* ≤ *Q* ⟹ *fBex S P* ≤ *fBex S Q*
⟨*proof*⟩

end

29.5.11 *fcard*

lemma *fcard-fempty*:
fcard {||} = *0*
⟨*proof*⟩

lemma *fcard-finsert-disjoint*:
x $|\notin|$ *A* ⟹ *fcard* (*finsert x A*) = *Suc* (*fcard A*)
⟨*proof*⟩

lemma *fcard-finsert-if*:
fcard (*finsert x A*) = (*if x* |∈| *A then fcard A else Suc* (*fcard A*))
⟨*proof*⟩

lemma *fcard-0-eq* [*simp*, *no-atp*]:
fcard A = *0* ⟷ *A* = {||}
⟨*proof*⟩

lemma *fcard-Suc-fminus1*:
x |∈| *A* ⟹ *Suc* (*fcard* (*A* |−| {|*x*|})) = *fcard A*
⟨*proof*⟩

lemma *fcard-fminus-fsingleton*:
x |∈| *A* ⟹ *fcard* (*A* |−| {|*x*|}) = *fcard A* − *1*
⟨*proof*⟩

lemma *fcard-fminus-fsingleton-if*:
fcard (*A* |−| {|*x*|}) = (*if x* |∈| *A then fcard A* − *1 else fcard A*)
⟨*proof*⟩

lemma *fcard-fminus-finsert*[*simp*]:
assumes *a* |∈| *A* **and** *a* |∉| *B*
shows *fcard* (*A* |−| *finsert a B*) = *fcard* (*A* |−| *B*) − *1*
⟨*proof*⟩

lemma *fcard-finsert*: *fcard* (*finsert x A*) = *Suc* (*fcard* (*A* |−| {|*x*|}))
⟨*proof*⟩

lemma *fcard-finsert-le*: *fcard A* ≤ *fcard* (*finsert x A*)
⟨*proof*⟩

lemma *fcard-mono*:
A |⊆| *B* ⟹ *fcard A* ≤ *fcard B*
⟨*proof*⟩

lemma *fcard-seteq*: *A* |⊆| *B* ⟹ *fcard B* ≤ *fcard A* ⟹ *A* = *B*
⟨*proof*⟩

lemma *pfsubset-fcard-mono*: *A* |⊂| *B* ⟹ *fcard A* < *fcard B*
⟨*proof*⟩

lemma *fcard-funion-finter*:
fcard A + *fcard B* = *fcard* (*A* |∪| *B*) + *fcard* (*A* |∩| *B*)
⟨*proof*⟩

lemma *fcard-funion-disjoint*:
A |∩| *B* = {||} ⟹ *fcard* (*A* |∪| *B*) = *fcard A* + *fcard B*
⟨*proof*⟩

lemma *fcard-funion-fsubset*:
B |⊆| *A* ⟹ *fcard* (*A* |−| *B*) = *fcard A* − *fcard B*
⟨*proof*⟩

lemma *diff-fcard-le-fcard-fminus*:
fcard A − *fcard B* ≤ *fcard*(*A* |−| *B*)
⟨*proof*⟩

lemma *fcard-fminus1-less*: *x* |∈| *A* ⟹ *fcard* (*A* |−| {|*x*|}) < *fcard A*
⟨*proof*⟩

lemma *fcard-fminus2-less*:
x |∈| *A* ⟹ *y* |∈| *A* ⟹ *fcard* (*A* |−| {|*x*|} |−| {|*y*|}) < *fcard A*
⟨*proof*⟩

lemma *fcard-fminus1-le*: *fcard* (*A* |−| {|*x*|}) ≤ *fcard A*
⟨*proof*⟩

lemma *fcard-pfsubset*: *A* |⊆| *B* ⟹ *fcard A* < *fcard B* ⟹ *A* < *B*
⟨*proof*⟩

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*
⟨*proof*⟩

29.5.13 *ffold*

context *comp-fun-commute*
begin
lemma *ffold-empty*[*simp*]: *ffold f z* {||} = *z*
⟨*proof*⟩

lemma *ffold-finsert* [*simp*]:
assumes *x* |∉| *A*
shows *ffold f z* (*finsert x A*) = *f x* (*ffold f z A*)
⟨*proof*⟩

lemma *ffold-fun-left-comm*:
f x (*ffold f z A*) = *ffold f* (*f x z*) *A*
⟨*proof*⟩

lemma *ffold-finsert2*:
x |∉| *A* ⟹ *ffold f z* (*finsert x A*) = *ffold f* (*f x z*) *A*
⟨*proof*⟩

lemma *ffold-rec*:

assumes x $|\in|$ A
shows *ffold* f z A = f x (*ffold* f z (A $|-|$ {$|x|$}))
⟨*proof*⟩

lemma *ffold-finsert-fremove*:
ffold f z (*finsert* x A) = f x (*ffold* f z (A $|-|$ {$|x|$}))
⟨*proof*⟩
end

lemma *ffold-fimage*:
assumes *inj-on* g (*fset* A)
shows *ffold* f z (g $|`|$ A) = *ffold* (f ∘ g) z A
⟨*proof*⟩

lemma *ffold-cong*:
assumes *comp-fun-commute* f *comp-fun-commute* g
⋀x. x $|\in|$ A ⟹ f x = g x
and s = t **and** A = B
shows *ffold* f s A = *ffold* g t B
⟨*proof*⟩

context *comp-fun-idem*
begin

lemma *ffold-finsert-idem*:
ffold f z (*finsert* x A) = f x (*ffold* f z A)
⟨*proof*⟩

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
⟨*proof*⟩

end

29.5.14 ($|\subset|$)

lemma *wfP-pfsubset*: *wfP* ($|\subset|$)
⟨*proof*⟩

29.5.15 Group operations

locale *comm-monoid-fset* = *comm-monoid*
begin

sublocale *set*: *comm-monoid-set* ⟨*proof*⟩

lift-definition F :: ('b ⇒ 'a) ⇒ 'b *fset* ⇒ 'a **is** *set.F* ⟨*proof*⟩

lemma *cong*[*fundef-cong*]: $A = B \Longrightarrow (\bigwedge x.\ x \mid\in\mid B \Longrightarrow g\ x = h\ x) \Longrightarrow F\ g\ A = F\ h\ B$
⟨*proof*⟩

lemma *cong-simp*[*cong*]:
$[\![A = B;\ \bigwedge x.\ x \mid\in\mid B =simp=> g\ x = h\ x]\!] \Longrightarrow F\ g\ A = F\ h\ B$
⟨*proof*⟩

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*⟩

end

29.5.16 Semilattice operations

locale *semilattice-fset = semilattice*
begin

sublocale *set*: *semilattice-set* ⟨*proof*⟩

lift-definition $F :: 'a\ fset \Rightarrow 'a$ **is** *set.F* ⟨*proof*⟩

lemma *eq-fold*: $F\ (finsert\ x\ A) = ffold\ f\ x\ A$
⟨*proof*⟩

lemma *singleton* [*simp*]: $F\ \{|x|\} = x$
⟨*proof*⟩

lemma *insert-not-elem*: $x \mid\notin\mid A \Longrightarrow A \neq \{||\} \Longrightarrow F\ (finsert\ x\ A) = x * F\ A$
⟨*proof*⟩

lemma *in-idem*: $x \mid\in\mid A \Longrightarrow x * F\ A = F\ A$
⟨*proof*⟩

lemma *insert* [*simp*]: $A \neq \{||\} \Longrightarrow F\ (finsert\ x\ A) = x * F\ A$
⟨*proof*⟩

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*⟩

sublocale *fMax*: *semilattice-order-fset max greater-eq greater*
  rewrites *semilattice-set.F max = Max*
  defines *fMax = fMax.F*
⟨*proof*⟩

end

lemma *mono-fMax-commute*: *mono f* ⟹ *A* ≠ {||} ⟹ *f* (*fMax A*) = *fMax* (*f* |‘| *A*)
  ⟨*proof*⟩

lemma *mono-fMin-commute*: *mono f* ⟹ *A* ≠ {||} ⟹ *f* (*fMin A*) = *fMin* (*f* |‘| *A*)
  ⟨*proof*⟩

lemma *fMax-in*[*simp*]: *A* ≠ {||} ⟹ *fMax A* |∈| *A*
  ⟨*proof*⟩

lemma *fMin-in*[*simp*]: *A* ≠ {||} ⟹ *fMin A* |∈| *A*
  ⟨*proof*⟩

lemma *fMax-ge*[*simp*]: *x* |∈| *A* ⟹ *x* ≤ *fMax A*
  ⟨*proof*⟩

lemma *fMin-le*[*simp*]: *x* |∈| *A* ⟹ *fMin A* ≤ *x*
  ⟨*proof*⟩

lemma *fMax-eqI*: (⋀*y*. *y* |∈| *A* ⟹ *y* ≤ *x*) ⟹ *x* |∈| *A* ⟹ *fMax A* = *x*
  ⟨*proof*⟩

lemma *fMin-eqI*: (⋀*y*. *y* |∈| *A* ⟹ *x* ≤ *y*) ⟹ *x* |∈| *A* ⟹ *fMin A* = *x*
  ⟨*proof*⟩

lemma *fMax-finsert*[*simp*]: *fMax* (*finsert x A*) = (*if A* = {||} *then x else max x* (*fMax A*))
  ⟨*proof*⟩

lemma *fMin-finsert*[*simp*]: *fMin* (*finsert x A*) = (*if A* = {||} *then x else min x* (*fMin A*))
  ⟨*proof*⟩

context *linorder* **begin**

lemma *fset-linorder-max-induct*[*case-names fempty finsert*]:
assumes $P\ \{||\}$
and $\bigwedge x\ S.\ [\![\forall\, y.\ y\ |\in|\ S \longrightarrow y < x;\ P\ S]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ S)$
shows $P\ S$
⟨*proof*⟩

lemma *fset-linorder-min-induct*[*case-names fempty finsert*]:
assumes $P\ \{||\}$
and $\bigwedge x\ S.\ [\![\forall\, y.\ y\ |\in|\ S \longrightarrow y > x;\ P\ S]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ S)$
shows $P\ S$
⟨*proof*⟩

end

29.6 Choice in fsets

lemma *fset-choice*:
assumes $\forall\, x.\ x\ |\in|\ A \longrightarrow (\exists\, y.\ P\ x\ y)$
shows $\exists f.\ \forall\, x.\ x\ |\in|\ A \longrightarrow P\ x\ (f\ x)$
⟨*proof*⟩

29.7 Induction and Cases rules for fsets

lemma *fset-exhaust* [*case-names empty insert*, *cases type*: *fset*]:
assumes *fempty-case*: $S = \{||\} \Longrightarrow P$
and *finsert-case*: $\bigwedge x\ S'.\ S = \mathit{finsert}\ x\ S' \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *fset-induct* [*case-names empty insert*]:
assumes *fempty-case*: $P\ \{||\}$
and *finsert-case*: $\bigwedge x\ S.\ P\ S \Longrightarrow P\ (\mathit{finsert}\ x\ S)$
shows $P\ S$
⟨*proof*⟩

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*⟩

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*⟩

lemma *fset-strong-cases*:
obtains $xs = \{||\}$
| $ys\ x$ **where** $x\ |\notin|\ ys$ **and** $xs = finsert\ x\ ys$
⟨*proof*⟩

lemma *fset-induct2*:
$P\ \{||\}\ \{||\} \Longrightarrow$
$(\bigwedge x\ xs.\ x\ |\notin|\ xs \Longrightarrow P\ (finsert\ x\ xs)\ \{||\}) \Longrightarrow$
$(\bigwedge y\ ys.\ y\ |\notin|\ ys \Longrightarrow P\ \{||\}\ (finsert\ y\ ys)) \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ [\![P\ xs\ ys;\ x\ |\notin|\ xs;\ y\ |\notin|\ ys]\!] \Longrightarrow P\ (finsert\ x\ xs)\ (finsert\ y\ ys)) \Longrightarrow$
$P\ xsa\ ysa$
⟨*proof*⟩

29.8 Lemmas depending on induction

lemma *ffUnion-fsubset-iff*: $ffUnion\ A\ |\subseteq|\ B \longleftrightarrow fBall\ A\ (\lambda x.\ x\ |\subseteq|\ B)$
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *rel-fset-alt-def*: $rel\text{-}fset\ R = (\lambda A\ B.\ (\forall x.\exists y.\ x|\in|A \longrightarrow y|\in|B \wedge R\ x\ y)$
$\wedge\ (\forall y.\ \exists x.\ y|\in|B \longrightarrow x|\in|A \wedge R\ x\ y))$
⟨*proof*⟩

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*⟩

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 $\{||\}\ \{||\}$
⟨*proof*⟩

lemma *finsert-transfer* [*transfer-rule*]:
$(A ===> rel\text{-}fset\ A ===> rel\text{-}fset\ A)\ finsert\ finsert$
⟨*proof*⟩

lemma *funion-transfer* [*transfer-rule*]:

(*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *funion funion*
⟨*proof*⟩

lemma *ffUnion-transfer* [*transfer-rule*]:
(*rel-fset* (*rel-fset A*) ===> *rel-fset A*) *ffUnion ffUnion*
⟨*proof*⟩

lemma *fimage-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-fset A* ===> *rel-fset B*) *fimage fimage*
⟨*proof*⟩

lemma *fBall-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> (=)) ===> (=)) *fBall fBall*
⟨*proof*⟩

lemma *fBex-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> (=)) ===> (=)) *fBex fBex*
⟨*proof*⟩

lemma *fPow-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset* (*rel-fset A*)) *fPow fPow*
⟨*proof*⟩

lemma *rel-fset-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> (=)) ===> *rel-fset A* ===> *rel-fset B* ===> (=))
rel-fset rel-fset
⟨*proof*⟩

lemma *bind-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *rel-fset B*) ===> *rel-fset B*) *fbind fbind*
⟨*proof*⟩

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *rel-fset A* ===> (=)) (|∈|) (|∈|)
⟨*proof*⟩

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *finter finter*
⟨*proof*⟩

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) (|−|) (|−|)
⟨*proof*⟩

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> (=)) (|⊆|) (|⊆|)
⟨*proof*⟩

lemma *fSup-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Sup Sup*
⟨*proof*⟩

lemma *fInf-transfer* [*transfer-rule*]:
assumes *bi-unique A* **and** *bi-total A*
shows (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Inf Inf*
⟨*proof*⟩

lemma *ffilter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows ((*A* ===> (=)) ===> *rel-fset A* ===> *rel-fset A*) *ffilter ffilter*
⟨*proof*⟩

lemma *card-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-fset A* ===> (=)) *fcard fcard*
⟨*proof*⟩

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 ⟷ (∀ *t* ∈ *fset a*. ∃ *u* ∈ *fset b*. *R t u*) ∧ (∀ *t* ∈ *fset b*. ∃ *u* ∈ *fset a*. *R u t*)
⟨*proof*⟩

lemma *fset-to-fset*: *finite A* ⟹ *fset* (*the-inv fset A*) = *A*
⟨*proof*⟩

lemma *rel-fset-aux*:
(∀ *t* ∈ *fset a*. ∃ *u* ∈ *fset b*. *R t u*) ∧ (∀ *u* ∈ *fset b*. ∃ *t* ∈ *fset a*. *R t u*) ⟷
((*BNF-Def.Grp* {*a*. *fset a* ⊆ {(*a*, *b*). *R a b*}} (*fimage fst*))$^{-1-1}$ *OO*
BNF-Def.Grp {*a*. *fset a* ⊆ {(*a*, *b*). *R a b*}} (*fimage snd*)) *a b* (**is** *?L* = *?R*)
⟨*proof*⟩

bnf *'a fset*
  map: *fimage*
  sets: *fset*
  bd: *natLeq*
  wits: {||}
  rel: *rel-fset*
⟨*proof*⟩

lemma *rel-fset-fset*: *rel-set* χ (*fset A1*) (*fset A2*) = *rel-fset* χ *A1 A2*
  ⟨*proof*⟩

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** λf. *sum* (*Suc* ∘ *f*) ⟨*proof*⟩
end

instantiation *fset* :: (*type*) *size*
begin
definition *size-fset* **where**
  size-fset-overloaded-def: *size-fset* = *FSet.size-fset* (λ-. *0*)
instance ⟨*proof*⟩
end

lemma *size-fset-simps*[*simp*]: *size-fset f X* = ($\sum$ *x* ∈ *fset X*. *Suc* (*f x*))
  ⟨*proof*⟩

lemma *size-fset-overloaded-simps*[*simp*]: *size X* = ($\sum$ *x* ∈ *fset X*. *Suc 0*)
  ⟨*proof*⟩

lemma *fset-size-o-map*: *inj f* ⟹ *size-fset g* ∘ *fimage f* = *size-fset* (*g* ∘ *f*)
  ⟨*proof*⟩

⟨*ML*⟩

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*) $\wedge$ *rel-set* φ (*Inr* $-$‘ *A1*) (*Inr* $-$‘ *A2*)
(**is** *?L* $\longleftrightarrow$ *?Rl* $\wedge$ *?Rr*)
⟨*proof*⟩

29.12.1 Countability

lemma *exists-fset-of-list*: $\exists$ *xs*. *fset-of-list xs* = *S*
including *fset.lifting*
⟨*proof*⟩

lemma *fset-of-list-surj*[*simp*, *intro*]: *surj fset-of-list*
⟨*proof*⟩

instance *fset* :: (*countable*) *countable*
⟨*proof*⟩

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 ⟨*proof*⟩

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 (λA. f A orelse Quickcheck-Exhaustive.full-exhaustive (λx. if fst x |∈| fst A then None else f (valtermify-finsert x A)) (i − 1)) (i − 1)))

instance ⟨*proof*⟩

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*⟩

definition *random-fset i = random-aux-fset i i*

instance ⟨*proof*⟩

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., $(x \mid\in\mid f \mid`\mid \mathit{fset\text{-}of\text{-}list}\ xs) = (x \in f\ `\ \mathit{set}\ xs)$.

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* :: $('b \Rightarrow 'c \Rightarrow \mathit{bool}) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'c) \Rightarrow \mathit{bool}$ **where**
rel-map $f \equiv (=)$ ===> *rel-option* f

lemma *ran-transfer*[*transfer-rule*]: (*rel-map* A ===> *rel-set* A) *ran ran*
⟨*proof*⟩

lemma *ran-alt-def*: *ran* $m = (\mathit{the} \circ m)\ `\ \mathit{dom}\ m$
⟨*proof*⟩

parametric-constant *dom-transfer*[*transfer-rule*]: *dom-def*

definition *map-upd* :: $'a \Rightarrow 'b \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-upd $k\ v\ m = m(k \mapsto v)$

parametric-constant *map-upd-transfer*[*transfer-rule*]: *map-upd-def*

definition *map-filter* :: $('a \Rightarrow bool) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-filter P m = $(\lambda x.$ *if P x then m x else None*$)$

parametric-constant *map-filter-transfer*[*transfer-rule*]: *map-filter-def*

lemma *map-filter-map-of*[*simp*]: *map-filter P* (*map-of m*) = *map-of* [$(k,$ -$) \leftarrow$ *m*. *P k*]
⟨*proof*⟩

lemma *map-filter-finite*[*intro*]:
assumes *finite* (*dom m*)
shows *finite* (*dom* (*map-filter P m*))
⟨*proof*⟩

definition *map-drop* :: $'a \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-drop a = *map-filter* $(\lambda a'.\ a' \neq a)$

parametric-constant *map-drop-transfer*[*transfer-rule*]: *map-drop-def*

definition *map-drop-set* :: *'a set* $\Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-drop-set A = *map-filter* $(\lambda a.\ a \notin A)$

parametric-constant *map-drop-set-transfer*[*transfer-rule*]: *map-drop-set-def*

definition *map-restrict-set* :: *'a set* $\Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-restrict-set A = *map-filter* $(\lambda a.\ a \in A)$

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*
⟨*proof*⟩

definition *map-pred* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$ **where**
map-pred P m $\longleftrightarrow$ $(\forall x.$ *case m x of None* $\Rightarrow$ *True* | *Some y* $\Rightarrow$ *P x y*$)$

parametric-constant *map-pred-transfer*[*transfer-rule*]: *map-pred-def*

definition *rel-map-on-set* :: *'a set* $\Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'c)$ $\Rightarrow$ *bool* **where**
rel-map-on-set S P = *eq-onp* $(\lambda x.\ x \in S)$ ===> *rel-option P*

lemma *dom-comp*: *dom* $(m \circ_m n) \subseteq$ *dom n*
⟨*proof*⟩

lemma *dom-comp-finite*: *finite* (*dom n*) $\Longrightarrow$ *finite* (*dom* (*map-comp m n*))
⟨*proof*⟩

parametric-constant *map-comp-transfer*[*transfer-rule*]: *map-comp-def*

end

30.2 Abstract characterisation

typedef $('a, 'b)$ *fmap* $= \{m.$ *finite* $(dom\ m)\} :: ('a \rightharpoonup 'b)$ *set*
  morphisms *fmlookup Abs-fmap*
⟨*proof*⟩

setup-lifting *type-definition-fmap*

lemma *dom-fmlookup-finite*[*intro*, *simp*]: *finite* $(dom\ (fmlookup\ m))$
⟨*proof*⟩

lemma *fmap-ext*:
  assumes $\bigwedge x.\ fmlookup\ m\ x = fmlookup\ n\ x$
  shows $m = n$
⟨*proof*⟩

30.3 Operations

context
  includes *fset.lifting*
begin

lift-definition *fmran* :: $('a, 'b)$ *fmap* $\Rightarrow$ $'b$ *fset*
  is *ran*
  parametric *ran-transfer*
⟨*proof*⟩

lemma *fmlookup-ran-iff*: $y \mid\in\mid fmran\ m \longleftrightarrow (\exists x.\ fmlookup\ m\ x = Some\ y)$
⟨*proof*⟩

lemma *fmranI*: $fmlookup\ m\ x = Some\ y \Longrightarrow y \mid\in\mid fmran\ m$ ⟨*proof*⟩

lemma *fmranE*[*elim*]:
  assumes $y \mid\in\mid fmran\ m$
  obtains x **where** $fmlookup\ m\ x = Some\ y$
⟨*proof*⟩

lift-definition *fmdom* :: $('a, 'b)$ *fmap* $\Rightarrow$ $'a$ *fset*
  is *dom*
  parametric *dom-transfer*
⟨*proof*⟩

lemma *fmlookup-dom-iff*: $x \mid\in\mid fmdom\ m \longleftrightarrow (\exists a.\ fmlookup\ m\ x = Some\ a)$
⟨*proof*⟩

lemma *fmdom-notI*: $fmlookup\ m\ x = None \Longrightarrow x \mid\notin\mid fmdom\ m$ ⟨*proof*⟩
lemma *fmdomI*: $fmlookup\ m\ x = Some\ y \Longrightarrow x \mid\in\mid fmdom\ m$ ⟨*proof*⟩

lemma *fmdom-notD*[*dest*]: $x \mid\notin\mid fmdom\ m \Longrightarrow fmlookup\ m\ x = None$ ⟨*proof*⟩

lemma *fmdomE*[*elim*]:
  assumes $x \mid\in\mid fmdom\ m$
  obtains y **where** $fmlookup\ m\ x = Some\ y$
⟨*proof*⟩

lift-definition $fmdom' :: ('a, 'b)\ fmap \Rightarrow 'a\ set$
  is *dom*
  parametric *dom-transfer*
⟨*proof*⟩

lemma *fmlookup-dom'-iff*: $x \in fmdom'\ m \longleftrightarrow (\exists\, a.\ fmlookup\ m\ x = Some\ a)$
⟨*proof*⟩

lemma *fmdom'-notI*: $fmlookup\ m\ x = None \Longrightarrow x \notin fmdom'\ m$ ⟨*proof*⟩
lemma *fmdom'I*: $fmlookup\ m\ x = Some\ y \Longrightarrow x \in fmdom'\ m$ ⟨*proof*⟩
lemma *fmdom'-notD*[*dest*]: $x \notin fmdom'\ m \Longrightarrow fmlookup\ m\ x = None$ ⟨*proof*⟩

lemma *fmdom'E*[*elim*]:
  assumes $x \in fmdom'\ m$
  obtains $x\ y$ **where** $fmlookup\ m\ x = Some\ y$
⟨*proof*⟩

lemma *fmdom'-alt-def*: $fmdom'\ m = fset\ (fmdom\ m)$
⟨*proof*⟩

lemma *finite-fmdom'*[*simp*]: $finite\ (fmdom'\ m)$
⟨*proof*⟩

lemma *dom-fmlookup*[*simp*]: $dom\ (fmlookup\ m) = fmdom'\ m$
⟨*proof*⟩

lift-definition $fmempty :: ('a, 'b)\ fmap$
  is *Map.empty*
⟨*proof*⟩

lemma *fmempty-lookup*[*simp*]: $fmlookup\ fmempty\ x = None$
⟨*proof*⟩

lemma *fmdom-empty*[*simp*]: $fmdom\ fmempty = \{||\}$ ⟨*proof*⟩
lemma *fmdom'-empty*[*simp*]: $fmdom'\ fmempty = \{\}$ ⟨*proof*⟩
lemma *fmran-empty*[*simp*]: $fmran\ fmempty = fempty$ ⟨*proof*⟩

lift-definition $fmupd :: 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ fmap \Rightarrow ('a, 'b)\ fmap$
  is *map-upd*
  parametric *map-upd-transfer*
⟨*proof*⟩

lemma *fmupd-lookup*[*simp*]: *fmlookup* (*fmupd a b m*) *a'* = (*if a* = *a' then Some b else fmlookup m a'*)
⟨*proof*⟩

lemma *fmdom-fmupd*[*simp*]: *fmdom* (*fmupd a b m*) = *finsert a* (*fmdom m*) ⟨*proof*⟩
lemma *fmdom'-fmupd*[*simp*]: *fmdom'* (*fmupd a b m*) = *insert a* (*fmdom' m*) ⟨*proof*⟩

lemma *fmupd-reorder-neq*:
assumes *a* ≠ *b*
shows *fmupd a x* (*fmupd b y m*) = *fmupd b y* (*fmupd a x m*)
⟨*proof*⟩

lemma *fmupd-idem*[*simp*]: *fmupd a x* (*fmupd a y m*) = *fmupd a x m*
⟨*proof*⟩

lift-definition *fmfilter* :: ('*a* ⇒ *bool*) ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap*
is *map-filter*
parametric *map-filter-transfer*
⟨*proof*⟩

lemma *fmdom-filter*[*simp*]: *fmdom* (*fmfilter P m*) = *ffilter P* (*fmdom m*)
⟨*proof*⟩

lemma *fmdom'-filter*[*simp*]: *fmdom'* (*fmfilter P m*) = *Set.filter P* (*fmdom' m*)
⟨*proof*⟩

lemma *fmlookup-filter*[*simp*]: *fmlookup* (*fmfilter P m*) *x* = (*if P x then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmfilter-empty*[*simp*]: *fmfilter P fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmfilter-true*[*simp*]:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ *P x*
shows *fmfilter P m* = *m*
⟨*proof*⟩

lemma *fmfilter-false*[*simp*]:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ ¬ *P x*
shows *fmfilter P m* = *fmempty*
⟨*proof*⟩

lemma *fmfilter-comp*[*simp*]: *fmfilter P* (*fmfilter Q m*) = *fmfilter* (λ*x*. *P x* ∧ *Q x*) *m*
⟨*proof*⟩

lemma *fmfilter-comm*: *fmfilter P* (*fmfilter Q m*) = *fmfilter Q* (*fmfilter P m*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *fmfilter-upd*[*simp*]:
fmfilter P (*fmupd x y m*) = (*if P x then fmupd x y* (*fmfilter P m*) *else fmfilter P m*)
⟨*proof*⟩

lift-definition *fmdrop* :: *'a* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
is *map-drop*
parametric *map-drop-transfer*
⟨*proof*⟩

lemma *fmdrop-lookup*[*simp*]: *fmlookup* (*fmdrop a m*) *a* = *None*
⟨*proof*⟩

lift-definition *fmdrop-set* :: *'a set* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
is *map-drop-set*
parametric *map-drop-set-transfer*
⟨*proof*⟩

lift-definition *fmdrop-fset* :: *'a fset* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
is *map-drop-set*
parametric *map-drop-set-transfer*
⟨*proof*⟩

lift-definition *fmrestrict-set* :: *'a set* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
is *map-restrict-set*
parametric *map-restrict-set-transfer*
⟨*proof*⟩

lift-definition *fmrestrict-fset* :: *'a fset* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
is *map-restrict-set*
parametric *map-restrict-set-transfer*
⟨*proof*⟩

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 (λa. a |∈| B)
⟨*proof*⟩

lemma *fmdom-drop*[*simp*]: *fmdom (fmdrop a m) = fmdom m − {|a|}* ⟨*proof*⟩
lemma *fmdom′-drop*[*simp*]: *fmdom′ (fmdrop a m) = fmdom′ m − {a}* ⟨*proof*⟩
lemma *fmdom′-drop-set*[*simp*]: *fmdom′ (fmdrop-set A m) = fmdom′ m − A* ⟨*proof*⟩
lemma *fmdom-drop-fset*[*simp*]: *fmdom (fmdrop-fset A m) = fmdom m − A* ⟨*proof*⟩
lemma *fmdom′-restrict-set*: *fmdom′ (fmrestrict-set A m) ⊆ A* ⟨*proof*⟩
lemma *fmdom-restrict-fset*: *fmdom (fmrestrict-fset A m) |⊆| A* ⟨*proof*⟩

lemma *fmdrop-fmupd*: *fmdrop x (fmupd y z m) = (if x = y then fmdrop x m else fmupd y z (fmdrop x m))*
⟨*proof*⟩

lemma *fmdrop-idle*: *x |∉| fmdom B ⟹ fmdrop x B = B*
⟨*proof*⟩

lemma *fmdrop-idle′*: *x ∉ fmdom′ B ⟹ fmdrop x B = B*
⟨*proof*⟩

lemma *fmdrop-fmupd-same*: *fmdrop x (fmupd x y m) = fmdrop x m*
⟨*proof*⟩

lemma *fmdom′-restrict-set-precise*: *fmdom′ (fmrestrict-set A m) = fmdom′ m ∩ A*
⟨*proof*⟩

lemma *fmdom′-restrict-fset-precise*: *fmdom (fmrestrict-fset A m) = fmdom m |∩| A*
⟨*proof*⟩

lemma *fmdom′-drop-fset*[*simp*]: *fmdom′ (fmdrop-fset A m) = fmdom′ m − fset A*
⟨*proof*⟩

lemma *fmdom′-restrict-fset*: *fmdom′ (fmrestrict-fset A m) ⊆ fset A*
⟨*proof*⟩

lemma *fmlookup-drop*[*simp*]:
fmlookup (fmdrop a m) x = (if x ≠ a then fmlookup m x else None)
⟨*proof*⟩

lemma *fmlookup-drop-set*[*simp*]:
fmlookup (fmdrop-set A m) x = (if x ∉ A then fmlookup m x else None)
⟨*proof*⟩

lemma *fmlookup-drop-fset*[*simp*]:
fmlookup (fmdrop-fset A m) x = (if x |∉| A then fmlookup m x else None)
⟨*proof*⟩

lemma *fmlookup-restrict-set*[*simp*]:
fmlookup (*fmrestrict-set A m*) *x* = (*if x* ∈ *A then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmlookup-restrict-fset*[*simp*]:
fmlookup (*fmrestrict-fset A m*) *x* = (*if x* |∈| *A then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmrestrict-set-dom*[*simp*]: *fmrestrict-set* (*fmdom*′ *m*) *m* = *m*
⟨*proof*⟩

lemma *fmrestrict-fset-dom*[*simp*]: *fmrestrict-fset* (*fmdom m*) *m* = *m*
⟨*proof*⟩

lemma *fmdrop-empty*[*simp*]: *fmdrop a fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-set-empty*[*simp*]: *fmdrop-set A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-fset-empty*[*simp*]: *fmdrop-fset A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-fset-fmdom*[*simp*]: *fmdrop-fset* (*fmdom A*) *A* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-set-fmdom*[*simp*]: *fmdrop-set* (*fmdom*′ *A*) *A* = *fmempty*
⟨*proof*⟩

lemma *fmrestrict-set-empty*[*simp*]: *fmrestrict-set A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmrestrict-fset-empty*[*simp*]: *fmrestrict-fset A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-set-null*[*simp*]: *fmdrop-set* {} *m* = *m*
⟨*proof*⟩

lemma *fmdrop-fset-null*[*simp*]: *fmdrop-fset* {||} *m* = *m*
⟨*proof*⟩

lemma *fmdrop-set-single*[*simp*]: *fmdrop-set* {*a*} *m* = *fmdrop a m*
⟨*proof*⟩

lemma *fmdrop-fset-single*[*simp*]: *fmdrop-fset* {|*a*|} *m* = *fmdrop a m*
⟨*proof*⟩

lemma *fmrestrict-set-null*[*simp*]: *fmrestrict-set* {} *m* = *fmempty*
⟨*proof*⟩

lemma *fmrestrict-fset-null*[*simp*]: *fmrestrict-fset* {||} *m* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-comm*: *fmdrop a* (*fmdrop b m*) = *fmdrop b* (*fmdrop a m*)
⟨*proof*⟩

lemma *fmdrop-set-insert*[*simp*]: *fmdrop-set* (*insert x S*) *m* = *fmdrop x* (*fmdrop-set S m*)
⟨*proof*⟩

lemma *fmdrop-fset-insert*[*simp*]: *fmdrop-fset* (*finsert x S*) *m* = *fmdrop x* (*fmdrop-fset S m*)
⟨*proof*⟩

lemma *fmrestrict-set-twice*[*simp*]: *fmrestrict-set S* (*fmrestrict-set T m*) = *fmrestrict-set* (*S* ∩ *T*) *m*
⟨*proof*⟩

lemma *fmrestrict-fset-twice*[*simp*]: *fmrestrict-fset S* (*fmrestrict-fset T m*) = *fmrestrict-fset* (*S* |∩| *T*) *m*
⟨*proof*⟩

lemma *fmrestrict-set-drop*[*simp*]: *fmrestrict-set S* (*fmdrop b m*) = *fmrestrict-set* (*S* − {*b*}) *m*
⟨*proof*⟩

lemma *fmrestrict-fset-drop*[*simp*]: *fmrestrict-fset S* (*fmdrop b m*) = *fmrestrict-fset* (*S* − {| *b* |}) *m*
⟨*proof*⟩

lemma *fmdrop-fmrestrict-set*[*simp*]: *fmdrop b* (*fmrestrict-set S m*) = *fmrestrict-set* (*S* − {*b*}) *m*
⟨*proof*⟩

lemma *fmdrop-fmrestrict-fset*[*simp*]: *fmdrop b* (*fmrestrict-fset S m*) = *fmrestrict-fset* (*S* − {| *b* |}) *m*
⟨*proof*⟩

lemma *fmdrop-idem*[*simp*]: *fmdrop a* (*fmdrop a m*) = *fmdrop a m*
⟨*proof*⟩

lemma *fmdrop-set-twice*[*simp*]: *fmdrop-set S* (*fmdrop-set T m*) = *fmdrop-set* (*S* ∪ *T*) *m*
⟨*proof*⟩

lemma *fmdrop-fset-twice*[*simp*]: *fmdrop-fset S* (*fmdrop-fset T m*) = *fmdrop-fset* (*S* |∪| *T*) *m*
⟨*proof*⟩

lemma *fmdrop-set-fmdrop*[*simp*]: *fmdrop-set S* (*fmdrop b m*) = *fmdrop-set* (*insert b S*) *m*
⟨*proof*⟩

lemma *fmdrop-fset-fmdrop*[*simp*]: *fmdrop-fset S* (*fmdrop b m*) = *fmdrop-fset* (*finsert b S*) *m*
⟨*proof*⟩

lift-definition *fmadd* :: ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* (**infixl** ‹+$_f$› *100*)
is *map-add*
parametric *map-add-transfer*
⟨*proof*⟩

lemma *fmlookup-add*[*simp*]:
fmlookup (*m* +$_f$ *n*) *x* = (*if x* |∈| *fmdom n then fmlookup n x else fmlookup m x*)
⟨*proof*⟩

lemma *fmdom-add*[*simp*]: *fmdom* (*m* +$_f$ *n*) = *fmdom m* |∪| *fmdom n* ⟨*proof*⟩
lemma *fmdom'-add*[*simp*]: *fmdom'* (*m* +$_f$ *n*) = *fmdom' m* ∪ *fmdom' n* ⟨*proof*⟩

lemma *fmadd-drop-left-dom*: *fmdrop-fset* (*fmdom n*) *m* +$_f$ *n* = *m* +$_f$ *n*
⟨*proof*⟩

lemma *fmadd-restrict-right-dom*: *fmrestrict-fset* (*fmdom n*) (*m* +$_f$ *n*) = *n*
⟨*proof*⟩

lemma *fmfilter-add-distrib*[*simp*]: *fmfilter P* (*m* +$_f$ *n*) = *fmfilter P m* +$_f$ *fmfilter P n*
⟨*proof*⟩

lemma *fmdrop-add-distrib*[*simp*]: *fmdrop a* (*m* +$_f$ *n*) = *fmdrop a m* +$_f$ *fmdrop a n*
⟨*proof*⟩

lemma *fmdrop-set-add-distrib*[*simp*]: *fmdrop-set A* (*m* +$_f$ *n*) = *fmdrop-set A m* +$_f$ *fmdrop-set A n*
⟨*proof*⟩

lemma *fmdrop-fset-add-distrib*[*simp*]: *fmdrop-fset A* (*m* +$_f$ *n*) = *fmdrop-fset A m* +$_f$ *fmdrop-fset A n*
⟨*proof*⟩

lemma *fmrestrict-set-add-distrib*[*simp*]:
fmrestrict-set A (*m* +$_f$ *n*) = *fmrestrict-set A m* +$_f$ *fmrestrict-set A n*
⟨*proof*⟩

lemma *fmrestrict-fset-add-distrib*[*simp*]:
fmrestrict-fset A (*m* $++_f$ *n*) = *fmrestrict-fset A m* $++_f$ *fmrestrict-fset A n*
⟨*proof*⟩

lemma *fmadd-empty*[*simp*]: *fmempty* $++_f$ *m* = *m m* $++_f$ *fmempty* = *m*
⟨*proof*⟩

lemma *fmadd-idempotent*[*simp*]: *m* $++_f$ *m* = *m*
⟨*proof*⟩

lemma *fmadd-assoc*[*simp*]: *m* $++_f$ (*n* $++_f$ *p*) = *m* $++_f$ *n* $++_f$ *p*
⟨*proof*⟩

lemma *fmadd-fmupd*[*simp*]: *m* $++_f$ *fmupd a b n* = *fmupd a b* (*m* $++_f$ *n*)
⟨*proof*⟩

lift-definition *fmpred* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*b*) *fmap* ⇒ *bool*
is *map-pred*
parametric *map-pred-transfer*
⟨*proof*⟩

lemma *fmpredI*[*intro*]:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ *P x y*
shows *fmpred P m*
⟨*proof*⟩

lemma *fmpredD*[*dest*]: *fmpred P m* ⟹ *fmlookup m x* = *Some y* ⟹ *P x y*
⟨*proof*⟩

lemma *fmpred-iff*: *fmpred P m* ⟷ (∀ *x y*. *fmlookup m x* = *Some y* ⟶ *P x y*)
⟨*proof*⟩

lemma *fmpred-alt-def*: *fmpred P m* ⟷ *fBall* (*fmdom m*) (λ*x*. *P x* (*the* (*fmlookup m x*)))
⟨*proof*⟩

lemma *fmpred-mono-strong*:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ *P x y* ⟹ *Q x y*
shows *fmpred P m* ⟹ *fmpred Q m*
⟨*proof*⟩

lemma *fmpred-mono*[*mono*]: *P* ≤ *Q* ⟹ *fmpred P* ≤ *fmpred Q*
⟨*proof*⟩

lemma *fmpred-empty*[*intro*!, *simp*]: *fmpred P fmempty*
⟨*proof*⟩

lemma *fmpred-upd*[*intro*]: *fmpred P m* ⟹ *P x y* ⟹ *fmpred P* (*fmupd x y m*)
⟨*proof*⟩

lemma *fmpred-updD*[*dest*]: *fmpred P* (*fmupd x y m*) ⟹ *P x y*
⟨*proof*⟩

lemma *fmpred-add*[*intro*]: *fmpred P m* ⟹ *fmpred P n* ⟹ *fmpred P* (*m* ++$_f$ *n*)
⟨*proof*⟩

lemma *fmpred-filter*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmfilter Q m*)
⟨*proof*⟩

lemma *fmpred-drop*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmdrop a m*)
⟨*proof*⟩

lemma *fmpred-drop-set*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmdrop-set A m*)
⟨*proof*⟩

lemma *fmpred-drop-fset*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmdrop-fset A m*)
⟨*proof*⟩

lemma *fmpred-restrict-set*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmrestrict-set A m*)
⟨*proof*⟩

lemma *fmpred-restrict-fset*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmrestrict-fset A m*)
⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *fmsubset* :: ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* ⇒ *bool* (**infix** ‹⊆$_f$› *50*)
is *map-le*
⟨*proof*⟩

lemma *fmsubset-refl*[*simp*]: *m* ⊆$_f$ *m*
⟨*proof*⟩

lemma *fmsubset-trans*[*trans*]: *m*$_1$ ⊆$_f$ *m*$_2$ ⟹ *m*$_2$ ⊆$_f$ *m*$_3$ ⟹ *m*$_1$ ⊆$_f$ *m*$_3$
⟨*proof*⟩

lemma *fmsubset-antisym*: *m*$_1$ ⊆$_f$ *m*$_2$ ⟹ *m*$_2$ ⊆$_f$ *m*$_1$ ⟹ *m*$_1$ = *m*$_2$
⟨*proof*⟩

lemma *fmsubset-alt-def*: *m* ⊆$_f$ *n* ⟷ *fmpred* (λ*k v*. *fmlookup n k* = *Some v*) *m*
⟨*proof*⟩

lemma *fmsubset-pred*: *fmpred P m* ⟹ *n* ⊆$_f$ *m* ⟹ *fmpred P n*
⟨*proof*⟩

lemma *fmsubset-filter-mono*: $m \subseteq_f n \Longrightarrow$ *fmfilter* P $m \subseteq_f$ *fmfilter* P n
⟨*proof*⟩

lemma *fmsubset-drop-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop* a $m \subseteq_f$ *fmdrop* a n
⟨*proof*⟩

lemma *fmsubset-drop-set-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop-set* A $m \subseteq_f$ *fmdrop-set* A n
⟨*proof*⟩

lemma *fmsubset-drop-fset-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop-fset* A $m \subseteq_f$ *fmdrop-fset* A n
⟨*proof*⟩

lemma *fmsubset-restrict-set-mono*: $m \subseteq_f n \Longrightarrow$ *fmrestrict-set* A $m \subseteq_f$ *fmrestrict-set* A n
⟨*proof*⟩

lemma *fmsubset-restrict-fset-mono*: $m \subseteq_f n \Longrightarrow$ *fmrestrict-fset* A $m \subseteq_f$ *fmrestrict-fset* A n
⟨*proof*⟩

lemma *fmfilter-subset*[*simp*]: *fmfilter* P $m \subseteq_f m$
⟨*proof*⟩

lemma *fmsubset-drop*[*simp*]: *fmdrop* a $m \subseteq_f m$
⟨*proof*⟩

lemma *fmsubset-drop-set*[*simp*]: *fmdrop-set* S $m \subseteq_f m$
⟨*proof*⟩

lemma *fmsubset-drop-fset*[*simp*]: *fmdrop-fset* S $m \subseteq_f m$
⟨*proof*⟩

lemma *fmsubset-restrict-set*[*simp*]: *fmrestrict-set* S $m \subseteq_f m$
⟨*proof*⟩

lemma *fmsubset-restrict-fset*[*simp*]: *fmrestrict-fset* S $m \subseteq_f m$
⟨*proof*⟩

lift-definition *fset-of-fmap* :: $('a, 'b)$ *fmap* $\Rightarrow$ $('a \times 'b)$ *fset* **is** *Map.graph*
⟨*proof*⟩

lemma *fset-of-fmap-inj*[*intro*, *simp*]: *inj fset-of-fmap*
⟨*proof*⟩

lemma *fset-of-fmap-iff*[*simp*]: $(a, b) \mid\in\mid$ *fset-of-fmap* $m \longleftrightarrow$ *fmlookup* m a $=$ *Some* b
⟨*proof*⟩

lemma *fset-of-fmap-iff'*: $(a, b) \in fset\ (fset\text{-}of\text{-}fmap\ m) \longleftrightarrow fmlookup\ m\ a = Some\ b$
⟨*proof*⟩

lift-definition *fmap-of-list* :: $('a \times 'b)\ list \Rightarrow ('a, 'b)\ fmap$
is *map-of*
parametric *map-of-transfer*
⟨*proof*⟩

lemma *fmap-of-list-simps*[*simp*]:
$fmap\text{-}of\text{-}list\ [] = fmempty$
$fmap\text{-}of\text{-}list\ ((k, v)\ \#\ kvs) = fmupd\ k\ v\ (fmap\text{-}of\text{-}list\ kvs)$
⟨*proof*⟩

lemma *fmap-of-list-app*[*simp*]: $fmap\text{-}of\text{-}list\ (xs\ @\ ys) = fmap\text{-}of\text{-}list\ ys\ ++_f\ fmap\text{-}of\text{-}list\ xs$
⟨*proof*⟩

lemma *fmupd-alt-def*: $fmupd\ k\ v\ m = m\ ++_f\ fmap\text{-}of\text{-}list\ [(k, v)]$
⟨*proof*⟩

lemma *fmpred-of-list*[*intro*]:
assumes $\bigwedge k\ v.\ (k, v) \in set\ xs \Longrightarrow P\ k\ v$
shows $fmpred\ P\ (fmap\text{-}of\text{-}list\ xs)$
⟨*proof*⟩

lemma *fmap-of-list-SomeD*: $fmlookup\ (fmap\text{-}of\text{-}list\ xs)\ k = Some\ v \Longrightarrow (k, v) \in set\ xs$
⟨*proof*⟩

lemma *fmdom-fmap-of-list*[*simp*]: $fmdom\ (fmap\text{-}of\text{-}list\ xs) = fset\text{-}of\text{-}list\ (map\ fst\ xs)$
⟨*proof*⟩

lift-definition *fmrel-on-fset* :: $'a\ fset \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('a, 'b)\ fmap \Rightarrow ('a, 'c)\ fmap \Rightarrow bool$
is *rel-map-on-set*
⟨*proof*⟩

lemma *fmrel-on-fset-alt-def*: $fmrel\text{-}on\text{-}fset\ S\ P\ m\ n \longleftrightarrow fBall\ S\ (\lambda x.\ rel\text{-}option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x))$
⟨*proof*⟩

lemma *fmrel-on-fsetI*[*intro*]:
assumes $\bigwedge x.\ x\ |\in|\ S \Longrightarrow rel\text{-}option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$
shows $fmrel\text{-}on\text{-}fset\ S\ P\ m\ n$
⟨*proof*⟩

lemma *fmrel-on-fset-mono*[*mono*]: $R \leq Q \Longrightarrow$ *fmrel-on-fset* S $R \leq$ *fmrel-on-fset* S Q
⟨*proof*⟩

lemma *fmrel-on-fsetD*: x |∈| $S \Longrightarrow$ *fmrel-on-fset* S P m $n \Longrightarrow$ *rel-option* P (*fmlookup* m x) (*fmlookup* n x)
⟨*proof*⟩

lemma *fmrel-on-fsubset*: *fmrel-on-fset* S R m $n \Longrightarrow$ T |⊆| $S \Longrightarrow$ *fmrel-on-fset* T R m n
⟨*proof*⟩

lemma *fmrel-on-fset-unionI*:
fmrel-on-fset A R m $n \Longrightarrow$ *fmrel-on-fset* B R m $n \Longrightarrow$ *fmrel-on-fset* (A |∪| B) R m n
⟨*proof*⟩

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)
⟨*proof*⟩

lift-definition *fmimage* :: ('a, 'b) *fmap* ⇒ 'a *fset* ⇒ 'b *fset* **is** λm S. {b|a b. m a = *Some* b ∧ a ∈ S}
⟨*proof*⟩

lemma *fmimage-alt-def*: *fmimage* m S = *fmran* (*fmrestrict-fset* S m)
⟨*proof*⟩

lemma *fmimage-empty*[*simp*]: *fmimage* m *fempty* = *fempty*
⟨*proof*⟩

lemma *fmimage-subset-ran*[*simp*]: *fmimage* m S |⊆| *fmran* m
⟨*proof*⟩

lemma *fmimage-dom*[*simp*]: *fmimage* m (*fmdom* m) = *fmran* m
⟨*proof*⟩

lemma *fmimage-inter*: *fmimage* m (A |∩| B) |⊆| *fmimage* m A |∩| *fmimage* m B
⟨*proof*⟩

lemma *fimage-inter-dom*[*simp*]:
fmimage m (*fmdom* m |∩| A) = *fmimage* m A
fmimage m (A |∩| *fmdom* m) = *fmimage* m A
⟨*proof*⟩

lemma *fmimage-union*[*simp*]: *fmimage* m (A |∪| B) = *fmimage* m A |∪| *fmimage* m B
⟨*proof*⟩

lemma *fmimage-Union*[*simp*]: *fmimage m* (*ffUnion A*) = *ffUnion* (*fmimage m* |‘| *A*)
⟨*proof*⟩

lemma *fmimage-filter*[*simp*]: *fmimage* (*fmfilter P m*) *A* = *fmimage m* (*ffilter P A*)
⟨*proof*⟩

lemma *fmimage-drop*[*simp*]: *fmimage* (*fmdrop a m*) *A* = *fmimage m* (*A* − {|*a*|})
⟨*proof*⟩

lemma *fmimage-drop-fset*[*simp*]: *fmimage* (*fmdrop-fset B m*) *A* = *fmimage m* (*A* − *B*)
⟨*proof*⟩

lemma *fmimage-restrict-fset*[*simp*]: *fmimage* (*fmrestrict-fset B m*) *A* = *fmimage m* (*A* |∩| *B*)
⟨*proof*⟩

lemma *fmfilter-ran*[*simp*]: *fmran* (*fmfilter P m*) = *fmimage m* (*ffilter P* (*fmdom m*))
⟨*proof*⟩

lemma *fmran-drop*[*simp*]: *fmran* (*fmdrop a m*) = *fmimage m* (*fmdom m* − {|*a*|})
⟨*proof*⟩

lemma *fmran-drop-fset*[*simp*]: *fmran* (*fmdrop-fset A m*) = *fmimage m* (*fmdom m* − *A*)
⟨*proof*⟩

lemma *fmran-restrict-fset*: *fmran* (*fmrestrict-fset A m*) = *fmimage m* (*fmdom m* |∩| *A*)
⟨*proof*⟩

lemma *fmlookup-image-iff*: *y* |∈| *fmimage m A* ⟷ (∃ *x*. *fmlookup m x* = *Some y* ∧ *x* |∈| *A*)
⟨*proof*⟩

lemma *fmimageI*: *fmlookup m x* = *Some y* ⟹ *x* |∈| *A* ⟹ *y* |∈| *fmimage m A*
⟨*proof*⟩

lemma *fmimageE*[*elim*]:
assumes *y* |∈| *fmimage m A*
obtains *x* **where** *fmlookup m x* = *Some y x* |∈| *A*
⟨*proof*⟩

lift-definition *fmcomp* :: (*'b*, *'c*) *fmap* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'c*) *fmap* (**infixl** ‹∘$_f$› *55*)
is *map-comp*

parametric *map-comp-transfer*
⟨*proof*⟩

lemma *fmlookup-comp*[*simp*]: *fmlookup* (*m* ∘$_f$ *n*) *x* = *Option.bind* (*fmlookup n x*) (*fmlookup m*)
⟨*proof*⟩

end

30.4 BNF setup

lift-bnf ('*a*, *fmran'*: '*b*) *fmap* [*wits*: *Map.empty*]
for *map*: *fmmap*
rel: *fmrel*
⟨*proof*⟩

declare *fmap.pred-mono*[*mono*]

lemma *fmran'-alt-def*: *fmran'* *m* = *fset* (*fmran m*)
including *fset.lifting*
⟨*proof*⟩

lemma *fmlookup-ran'-iff*: *y* ∈ *fmran'* *m* ⟷ (∃ *x*. *fmlookup m x* = *Some y*)
⟨*proof*⟩

lemma *fmran'I*: *fmlookup m x* = *Some y* ⟹ *y* ∈ *fmran'* *m*
⟨*proof*⟩

lemma *fmran'E*[*elim*]:
assumes *y* ∈ *fmran'* *m*
obtains *x* **where** *fmlookup m x* = *Some y*
⟨*proof*⟩

lemma *fmrel-iff*: *fmrel R m n* ⟷ (∀ *x*. *rel-option R* (*fmlookup m x*) (*fmlookup n x*))
⟨*proof*⟩

lemma *fmrelI*[*intro*]:
assumes ⋀*x*. *rel-option R* (*fmlookup m x*) (*fmlookup n x*)
shows *fmrel R m n*
⟨*proof*⟩

lemma *fmrel-upd*[*intro*]: *fmrel P m n* ⟹ *P x y* ⟹ *fmrel P* (*fmupd k x m*) (*fmupd k y n*)
⟨*proof*⟩

lemma *fmrelD*[*dest*]: *fmrel P m n* ⟹ *rel-option P* (*fmlookup m x*) (*fmlookup n x*)
⟨*proof*⟩

lemma *fmrel-addI*[*intro*]:
assumes *fmrel P m n fmrel P a b*
shows *fmrel P* (m $++_f$ a) (n $++_f$ b)
⟨*proof*⟩

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*⟩

lemma *fmrel-filter*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmfilter Q m*) (*fmfilter Q n*)
⟨*proof*⟩

lemma *fmrel-drop*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmdrop a m*) (*fmdrop a n*)
⟨*proof*⟩

lemma *fmrel-drop-set*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmdrop-set A m*) (*fmdrop-set A n*)
⟨*proof*⟩

lemma *fmrel-drop-fset*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmdrop-fset A m*) (*fmdrop-fset A n*)
⟨*proof*⟩

lemma *fmrel-restrict-set*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmrestrict-set A m*) (*fmrestrict-set A n*)
⟨*proof*⟩

lemma *fmrel-restrict-fset*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmrestrict-fset A m*) (*fmrestrict-fset A n*)
⟨*proof*⟩

lemma *fmrel-on-fset-fmrel-restrict*:
fmrel-on-fset S P m n ⟷ *fmrel P* (*fmrestrict-fset S m*) (*fmrestrict-fset S n*)
⟨*proof*⟩

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*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *fmrel-fmdom-eq*:

assumes *fmrel P x y*
shows *fmdom x = fmdom y*
⟨*proof*⟩

lemma *fmrel-fmdom'-eq*: *fmrel P x y ⟹ fmdom' x = fmdom' y*
⟨*proof*⟩

lemma *fmrel-rel-fmran*:
assumes *fmrel P x y*
shows *rel-fset P (fmran x) (fmran y)*
⟨*proof*⟩

lemma *fmrel-rel-fmran'*: *fmrel P x y ⟹ rel-set P (fmran' x) (fmran' y)*
⟨*proof*⟩

lemma *pred-fmap-fmpred*[*simp*]: *pred-fmap P = fmpred (λ-. P)*
⟨*proof*⟩

lemma *pred-fmap-id*[*simp*]: *pred-fmap id (fmmap f m) ⟷ pred-fmap f m*
⟨*proof*⟩

lemma *pred-fmapD*: *pred-fmap P m ⟹ x |∈| fmran m ⟹ P x*
⟨*proof*⟩

lemma *fmlookup-map*[*simp*]: *fmlookup (fmmap f m) x = map-option f (fmlookup m x)*
⟨*proof*⟩

lemma *fmpred-map*[*simp*]: *fmpred P (fmmap f m) ⟷ fmpred (λk v. P k (f v)) m*
⟨*proof*⟩

lemma *fmpred-id*[*simp*]: *fmpred (λ-. id) (fmmap f m) ⟷ fmpred (λ-. f) m*
⟨*proof*⟩

lemma *fmmap-add*[*simp*]: *fmmap f (m ++$_f$ n) = fmmap f m ++$_f$ fmmap f n*
⟨*proof*⟩

lemma *fmmap-empty*[*simp*]: *fmmap f fmempty = fmempty*
⟨*proof*⟩

lemma *fmdom-map*[*simp*]: *fmdom (fmmap f m) = fmdom m*
including *fset.lifting*
⟨*proof*⟩

lemma *fmdom'-map*[*simp*]: *fmdom' (fmmap f m) = fmdom' m*
⟨*proof*⟩

lemma *fmran-fmmap*[*simp*]: *fmran (fmmap f m) = f |‘| fmran m*
including *fset.lifting*

⟨*proof*⟩

lemma *fmran'-fmmap*[*simp*]: *fmran'* (*fmmap f m*) = *f* ‘ *fmran' m*
⟨*proof*⟩

lemma *fmfilter-fmmap*[*simp*]: *fmfilter P* (*fmmap f m*) = *fmmap f* (*fmfilter P m*)
⟨*proof*⟩

lemma *fmdrop-fmmap*[*simp*]: *fmdrop a* (*fmmap f m*) = *fmmap f* (*fmdrop a m*)
⟨*proof*⟩

lemma *fmdrop-set-fmmap*[*simp*]: *fmdrop-set A* (*fmmap f m*) = *fmmap f* (*fmdrop-set A m*)
⟨*proof*⟩

lemma *fmdrop-fset-fmmap*[*simp*]: *fmdrop-fset A* (*fmmap f m*) = *fmmap f* (*fmdrop-fset A m*)
⟨*proof*⟩

lemma *fmrestrict-set-fmmap*[*simp*]: *fmrestrict-set A* (*fmmap f m*) = *fmmap f* (*fmrestrict-set A m*)
⟨*proof*⟩

lemma *fmrestrict-fset-fmmap*[*simp*]: *fmrestrict-fset A* (*fmmap f m*) = *fmmap f* (*fmrestrict-fset A m*)
⟨*proof*⟩

lemma *fmmap-subset*[*intro*]: *m* ⊆$_f$ *n* ⟹ *fmmap f m* ⊆$_f$ *fmmap f n*
⟨*proof*⟩

lemma *fmmap-fset-of-fmap*: *fset-of-fmap* (*fmmap f m*) = (λ(*k*, *v*). (*k*, *f v*)) |‘| *fset-of-fmap m*
including *fset.lifting*
⟨*proof*⟩

lemma *fmmap-fmupd*: *fmmap f* (*fmupd x y m*) = *fmupd x* (*f y*) (*fmmap f m*)
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *size-fmap-overloaded-simps*[*simp*]: *size x* = *size* (*fset-of-fmap x*)
⟨*proof*⟩

lemma *fmap-size-o-map*: *size-fmap f g* ∘ *fmmap h* = *size-fmap f* (*g* ∘ *h*)
⟨*proof*⟩

⟨*ML*⟩

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*)
⟨*proof*⟩

lemma *fmpred-fmmap-keys*[*simp*]: *fmpred P* (*fmmap-keys f m*) = *fmpred* (λ*a b*. *P a* (*f a b*)) *m*
⟨*proof*⟩

lemma *fmdom-fmmap-keys*[*simp*]: *fmdom* (*fmmap-keys f m*) = *fmdom m*
including *fset.lifting*
⟨*proof*⟩

lemma *fmlookup-fmmap-keys*[*simp*]: *fmlookup* (*fmmap-keys f m*) *x* = *map-option* (*f x*) (*fmlookup m x*)
⟨*proof*⟩

lemma *fmfilter-fmmap-keys*[*simp*]: *fmfilter P* (*fmmap-keys f m*) = *fmmap-keys f* (*fmfilter P m*)
⟨*proof*⟩

lemma *fmdrop-fmmap-keys*[*simp*]: *fmdrop a* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop a m*)
⟨*proof*⟩

lemma *fmdrop-set-fmmap-keys*[*simp*]: *fmdrop-set A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop-set A m*)
⟨*proof*⟩

lemma *fmdrop-fset-fmmap-keys*[*simp*]: *fmdrop-fset A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop-fset A m*)
⟨*proof*⟩

lemma *fmrestrict-set-fmmap-keys*[*simp*]: *fmrestrict-set A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmrestrict-set A m*)
⟨*proof*⟩

lemma *fmrestrict-fset-fmmap-keys*[*simp*]: *fmrestrict-fset A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmrestrict-fset A m*)
⟨*proof*⟩

lemma *fmmap-keys-subset*[*intro*]: *m* ⊆$_f$ *n* ⟹ *fmmap-keys f m* ⊆$_f$ *fmmap-keys f n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-of-sorted-list*[*simp*]: *map-of* (*sorted-list-of-fmap m*) = *fmlookup m*
⟨*proof*⟩
including *fset.lifting*
⟨*proof*⟩

30.7 Additional properties

lemma *fmchoice'*:
assumes *finite S* ∀ *x* ∈ *S*. ∃ *y*. *Q x y*
shows ∃ *m*. *fmdom' m* = *S* ∧ *fmpred Q m*
⟨*proof*⟩

30.8 Lifting/transfer setup

context includes *lifting-syntax* **begin**

lemma *fmempty-transfer*[*simp*, *intro*, *transfer-rule*]: *fmrel P fmempty fmempty*
⟨*proof*⟩

lemma *fmadd-transfer*[*transfer-rule*]:
(*fmrel P* ===> *fmrel P* ===> *fmrel P*) *fmadd fmadd*
⟨*proof*⟩

lemma *fmupd-transfer*[*transfer-rule*]:
((=) ===> *P* ===> *fmrel P* ===> *fmrel P*) *fmupd fmupd*
⟨*proof*⟩

end

lemma *Quotient-fmap-bnf*[*quot-map*]:
assumes *Quotient R Abs Rep T*
shows *Quotient* (*fmrel R*) (*fmmap Abs*) (*fmmap Rep*) (*fmrel T*)
⟨*proof*⟩

30.9 View as datatype

lemma *fmap-distinct*[*simp*]:
fmempty ≠ *fmupd k v m*
fmupd k v m ≠ *fmempty*
⟨*proof*⟩

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′*
⟨*proof*⟩ **including** *fmap.lifting* **and** *fset.lifting*
⟨*proof*⟩

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*⟩

30.10 Code setup

instantiation *fmap* :: (*type*, *equal*) *equal* **begin**

definition *equal-fmap* ≡ *fmrel HOL.equal*

instance ⟨*proof*⟩

end

lemma *fBall-alt-def*: *fBall S P* ⟷ (∀ *x*. *x* |∈| *S* ⟶ *P x*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fmempty-of-list*[*code*]: *fmempty* = *fmap-of-list* []
⟨*proof*⟩

lemma *fmran-of-list*[*code*]: *fmran* (*fmap-of-list m*) = *snd* |‘| *fset-of-list* (*AList.clearjunk m*)
⟨*proof*⟩

lemma *fmdom-of-list*[*code*]: *fmdom* (*fmap-of-list m*) = *fst* |‘| *fset-of-list m*
⟨*proof*⟩

lemma *fmfilter-of-list*[*code*]: *fmfilter P* (*fmap-of-list m*) = *fmap-of-list* (*filter* (λ(*k*, -). *P k*) *m*)
⟨*proof*⟩

lemma *fmadd-of-list*[*code*]: *fmap-of-list m* ++$_f$ *fmap-of-list n* = *fmap-of-list* (*AList.merge m n*)
⟨*proof*⟩

lemma *fmmap-of-list*[*code*]: *fmmap f* (*fmap-of-list m*) = *fmap-of-list* (*map* (*apsnd f*) *m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fmimage-of-list*[*code*]:
fmimage (*fmap-of-list m*) *A* = *fset-of-list* (*map snd* (*filter* (λ(*k*, -). *k* |∈| *A*) (*AList.clearjunk m*)))
⟨*proof*⟩

lemma *fmcomp-list*[*code*]:
fmap-of-list m ∘$_f$ *fmap-of-list n* = *fmap-of-list* (*AList.compose n m*)
⟨*proof*⟩

end

30.11 Instances

lemma *exists-map-of*:
assumes *finite* (*dom m*) **shows** ∃ *xs*. *map-of xs* = *m*
⟨*proof*⟩

lemma *exists-fmap-of-list*: ∃ *xs*. *fmap-of-list xs* = *m*
⟨*proof*⟩

lemma *fmap-of-list-surj*[*simp*, *intro*]: *surj fmap-of-list*
⟨*proof*⟩

instance *fmap* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

instance *fmap* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *test2* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b fset*) *fmap* **is** λ*a b*. *fmupd a* {*b*} *fmempty*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *fdisjnt-alt-def*: *fdisjnt M N* ⟷ (*M* |∩| *N* = {||})
⟨*proof*⟩

lemma *fdisjnt-insert*: *x* |∉| *N* ⟹ *fdisjnt M N* ⟹ *fdisjnt* (*finsert x M*) *N*
⟨*proof*⟩

lemma *fdisjnt-subset-right*: *N'* |⊆| *N* ⟹ *fdisjnt M N* ⟹ *fdisjnt M N'*
⟨*proof*⟩

lemma *fdisjnt-subset-left*: *N'* |⊆| *N* ⟹ *fdisjnt N M* ⟹ *fdisjnt N' M*
⟨*proof*⟩

lemma *fdisjnt-union-right*: *fdisjnt M A* ⟹ *fdisjnt M B* ⟹ *fdisjnt M* (*A* |∪| *B*)
⟨*proof*⟩

lemma *fdisjnt-union-left*: *fdisjnt A M* ⟹ *fdisjnt B M* ⟹ *fdisjnt* (*A* |∪| *B*) *M*
⟨*proof*⟩

lemma *fdisjnt-swap*: *fdisjnt M N* ⟹ *fdisjnt N M*
including *fset.lifting* ⟨*proof*⟩

lemma *distinct-append-fset*:
assumes *distinct xs distinct ys fdisjnt* (*fset-of-list xs*) (*fset-of-list ys*)
shows *distinct* (*xs* @ *ys*)
⟨*proof*⟩

lemma *fdisjnt-contrI*:
assumes ⋀*x*. *x* |∈| *M* ⟹ *x* |∈| *N* ⟹ *False*
shows *fdisjnt M N*
⟨*proof*⟩

lemma *fdisjnt-Union-left*: *fdisjnt* (*ffUnion S*) *T* ⟷ *fBall S* (λ*S*. *fdisjnt S T*)
⟨*proof*⟩

lemma *fdisjnt-Union-right*: *fdisjnt T* (*ffUnion S*) ⟷ *fBall S* (λ*S*. *fdisjnt T S*)

⟨*proof*⟩

lemma *fdisjnt-ge-max*: *fBall X* (λ*x*. *x* > *fMax Y*) ⟹ *fdisjnt X Y*
⟨*proof*⟩

end

lemma *fmadd-disjnt*: *fdisjnt* (*fmdom m*) (*fmdom n*) ⟹ *m* ++$_f$ *n* = *n* ++$_f$ *m*
⟨*proof*⟩
including *fset.lifting* **and** *fmap.lifting*
⟨*proof*⟩

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*⟩

context begin

qualified definition *dlist-eq* **where** *dlist-eq* = *BNF-Def.vimage2p remdups remdups* (=)

qualified lemma *equivp-dlist-eq*: *equivp dlist-eq*
  ⟨*proof*⟩ **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*
  ⟨*proof*⟩

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
⟨*proof*⟩

lemma *dlist-eqI*:
list-of-dlist dxs = list-of-dlist dys ⟹ dxs = dys
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *list-of-dlist-Dlist* [*simp*]:
list-of-dlist (*Dlist xs*) = *remdups xs*
⟨*proof*⟩

lemma *remdups-list-of-dlist* [*simp*]:
remdups (*list-of-dlist dxs*) = *list-of-dlist dxs*
⟨*proof*⟩

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
Dlist (*list-of-dlist dxs*) = *dxs*
⟨*proof*⟩

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 = []
⟨*proof*⟩

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.insert x dxs*) = *List.insert x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.remove x dxs*) = *remove1 x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.map f dxs*) = *remdups* (*List.map f* (*list-of-dlist dxs*))
⟨*proof*⟩

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.filter P dxs*) = *List.filter P* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-rotate* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.rotate n dxs*) = *List.rotate n* (*list-of-dlist dxs*)
⟨*proof*⟩

Explicit executable conversion

definition *dlist-of-list* [*simp*]:
dlist-of-list = *Dlist*

lemma [*code abstract*]:
list-of-dlist (*dlist-of-list xs*) = *remdups xs*
⟨*proof*⟩

Equality

instantiation *dlist* :: (*equal*) *equal*
begin

definition *HOL.equal dxs dys* ⟷ *HOL.equal* (*list-of-dlist dxs*) (*list-of-dlist dys*)

instance
⟨*proof*⟩

end

declare *equal-dlist-def* [*code*]

lemma [*code nbe*]: *HOL.equal* (*dxs* :: 'a::*equal dlist*) *dxs* ⟷ *True*
⟨*proof*⟩

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*⟩

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*⟩

32.4 Functorial structure

functor *map*: *map*
⟨*proof*⟩

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*⟩ **lemma** *double-Cons1* [*simp*]: *double xs* (*x* # *xs*) **if** *x* ∈ *set xs*
⟨*proof*⟩ **lemma** *double-Cons-same* [*simp*]: *double xs ys* ⟹ *double* (*x* # *xs*) (*x* # *ys*)
⟨*proof*⟩ **lemma** *doubles-Cons-same*: *double*** *xs ys* ⟹ *double*** (*x* # *xs*) (*x* # *ys*)
⟨*proof*⟩ **lemma** *remdups-into-doubles*: *double*** (*remdups xs*) *xs*
⟨*proof*⟩ **lemma** *dlist-eq-into-doubles*: *Dlist.dlist-eq* ≤ *equivclp double*
⟨*proof*⟩ **lemma** *factor-double-map*: *double* (*map f xs*) *ys* ⟹ ∃ *zs*. *Dlist.dlist-eq xs zs* ∧ *ys* = *map f zs* ∧ *set zs* ⊆ *set xs*
⟨*proof*⟩ **lemma** *dlist-eq-set-eq*: *Dlist.dlist-eq xs ys* ⟹ *set xs* = *set ys*
⟨*proof*⟩ **lemma** *dlist-eq-map-respect*: *Dlist.dlist-eq xs ys* ⟹ *Dlist.dlist-eq* (*map f xs*) (*map f ys*)
⟨*proof*⟩ **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*
⟨*proof*⟩

lifting-update *dlist.lifting*
lifting-forget *dlist.lifting*

end

context begin
interpretation *Quotient-dlist*: *Quotient-dlist* ⟨*proof*⟩

lift-bnf (*plugins del*: *code*) *'a dlist*
⟨*proof*⟩ **lemma** *list-of-dlist-transfer*[*transfer-rule*]:
bi-unique R ⟹ (*rel-fun* (*Quotient-dlist.pcr-dlist R*) (*list-all2 R*)) *remdups list-of-dlist*
⟨*proof*⟩

lemma *list-of-dlist-map-dlist*[*simp*]:
list-of-dlist (*map-dlist f xs*) = *remdups* (*map f* (*list-of-dlist xs*))
⟨*proof*⟩

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* ⟨*proof*⟩

setup-lifting *type-definition-dual*

code-datatype *dual*

lemma *dual-eqI*:
  $x = y$ **if** *undual* x = *undual* y
  ⟨*proof*⟩

lemma *dual-eq-iff*:
  $x = y \longleftrightarrow$ *undual* x = *undual* y
  ⟨*proof*⟩

lemma *eq-dual-iff* [*iff*]:
  dual x = *dual* $y \longleftrightarrow x = y$
  ⟨*proof*⟩

lemma *undual-dual* [*simp*, *code*]:
  undual (*dual* x) = x
  ⟨*proof*⟩

lemma *dual-undual* [*simp*]:
  dual (*undual* x) = x
  ⟨*proof*⟩

lemma *undual-comp-dual* [*simp*]:
  undual ∘ *dual* = *id*
  ⟨*proof*⟩

lemma *dual-comp-undual* [*simp*]:
  dual ∘ *undual* = *id*
  ⟨*proof*⟩

lemma *inj-dual*:
  inj dual
  ⟨*proof*⟩

lemma *inj-undual*:
inj undual
⟨*proof*⟩

lemma *surj-dual*:
surj dual
⟨*proof*⟩

lemma *surj-undual*:
surj undual
⟨*proof*⟩

lemma *bij-dual*:
bij dual
⟨*proof*⟩

lemma *bij-undual*:
bij undual
⟨*proof*⟩

instance *dual* :: (*finite*) *finite*
⟨*proof*⟩

instantiation *dual* :: (*equal*) *equal*
begin

lift-definition *equal-dual* :: *'a dual* ⇒ *'a dual* ⇒ *bool*
is *HOL.equal* ⟨*proof*⟩

instance
⟨*proof*⟩

end

33.1 Pointwise ordering

instantiation *dual* :: (*ord*) *ord*
begin

lift-definition *less-eq-dual* :: *'a dual* ⇒ *'a dual* ⇒ *bool*
is (≥) ⟨*proof*⟩

lift-definition *less-dual* :: *'a dual* ⇒ *'a dual* ⇒ *bool*
is (>) ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *dual-less-eqI*:
$x \leq y$ **if** *undual* $y \leq$ *undual* x
⟨*proof*⟩

lemma *dual-less-eq-iff*:
$x \leq y \longleftrightarrow$ *undual* $y \leq$ *undual* x
⟨*proof*⟩

lemma *less-eq-dual-iff* [*iff*]:
dual $x \leq$ *dual* $y \longleftrightarrow y \leq x$
⟨*proof*⟩

lemma *dual-lessI*:
$x < y$ **if** *undual* $y <$ *undual* x
⟨*proof*⟩

lemma *dual-less-iff*:
$x < y \longleftrightarrow$ *undual* $y <$ *undual* x
⟨*proof*⟩

lemma *less-dual-iff* [*iff*]:
dual $x <$ *dual* $y \longleftrightarrow y < x$
⟨*proof*⟩

instance *dual* :: (*preorder*) *preorder*
⟨*proof*⟩

instance *dual* :: (*order*) *order*
⟨*proof*⟩

33.2 Binary infimum and supremum

instantiation *dual* :: (*sup*) *inf*
begin

lift-definition *inf-dual* :: *'a dual* ⇒ *'a dual* ⇒ *'a dual*
is *sup* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *undual-inf-eq* [*simp*]:
undual (*inf x y*) = *sup* (*undual x*) (*undual y*)
⟨*proof*⟩

lemma *dual-sup-eq* [*simp*]:
dual (*sup x y*) = *inf* (*dual x*) (*dual y*)

⟨*proof*⟩

instantiation *dual* :: (*inf*) *sup*
begin

lift-definition *sup-dual* :: *'a dual* ⇒ *'a dual* ⇒ *'a dual*
is *inf* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *undual-sup-eq* [*simp*]:
undual (*sup x y*) = *inf* (*undual x*) (*undual y*)
⟨*proof*⟩

lemma *dual-inf-eq* [*simp*]:
dual (*inf x y*) = *sup* (*dual x*) (*dual y*)
⟨*proof*⟩

instance *dual* :: (*semilattice-sup*) *semilattice-inf*
⟨*proof*⟩

instance *dual* :: (*semilattice-inf*) *semilattice-sup*
⟨*proof*⟩

instance *dual* :: (*lattice*) *lattice* ⟨*proof*⟩

instance *dual* :: (*distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

33.3 Top and bottom elements

instantiation *dual* :: (*top*) *bot*
begin

lift-definition *bot-dual* :: *'a dual*
is *top* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *undual-bot-eq* [*simp*]:
undual bot = *top*
⟨*proof*⟩

lemma *dual-top-eq* [*simp*]:
dual top = *bot*

⟨*proof*⟩

instantiation *dual* :: (*bot*) *top*
begin

lift-definition *top-dual* :: *'a dual*
is *bot* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *undual-top-eq* [*simp*]:
undual top = *bot*
⟨*proof*⟩

lemma *dual-bot-eq* [*simp*]:
dual bot = *top*
⟨*proof*⟩

instance *dual* :: (*order-top*) *order-bot*
⟨*proof*⟩

instance *dual* :: (*order-bot*) *order-top*
⟨*proof*⟩

instance *dual* :: (*bounded-lattice-top*) *bounded-lattice-bot* ⟨*proof*⟩

instance *dual* :: (*bounded-lattice-bot*) *bounded-lattice-top* ⟨*proof*⟩

instance *dual* :: (*bounded-lattice*) *bounded-lattice* ⟨*proof*⟩

33.4 Complement

instantiation *dual* :: (*uminus*) *uminus*
begin

lift-definition *uminus-dual* :: *'a dual* ⇒ *'a dual*
is *uminus* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *undual-uminus-eq* [*simp*]:
undual (− *x*) = − *undual x*
⟨*proof*⟩

lemma *dual-uminus-eq* [*simp*]:

dual (− *x*) = − *dual x*
⟨*proof*⟩

instantiation *dual* :: (*boolean-algebra*) *boolean-algebra*
begin

lift-definition *minus-dual* :: *'a dual* ⇒ *'a dual* ⇒ *'a dual*
is λ*x y*. − (*y* − *x*) ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *undual-minus-eq* [*simp*]:
undual (*x* − *y*) = − (*undual y* − *undual x*)
⟨*proof*⟩

lemma *dual-minus-eq* [*simp*]:
dual (*x* − *y*) = − (*dual y* − *dual x*)
⟨*proof*⟩

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* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *undual-Inf-eq* [*simp*]:
undual (*Inf A*) = *Sup* (*undual* ‘ *A*)
⟨*proof*⟩

lemma *dual-Sup-eq* [*simp*]:
dual (*Sup A*) = *Inf* (*dual* ‘ *A*)
⟨*proof*⟩

instantiation *dual* :: (*Inf*) *Sup*
begin

lift-definition *Sup-dual* :: *'a dual set* ⇒ *'a dual*
is *Inf* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *undual-Sup-eq* [*simp*]:
undual (*Sup A*) = *Inf* (*undual* ' *A*)
⟨*proof*⟩

lemma *dual-Inf-eq* [*simp*]:
dual (*Inf A*) = *Sup* (*dual* ' *A*)
⟨*proof*⟩

instance *dual* :: (*complete-lattice*) *complete-lattice*
⟨*proof*⟩

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*⟩

lemma *lfp-dual-gfp*:
lfp f = *undual* (*gfp g*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *gfp-dual-lfp*:
gfp f = *undual* (*lfp g*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eqpoll-empty-iff-empty* [*simp*]: *A* ≈ {} ⟷ *A*={}
⟨*proof*⟩

lemma *lepoll-empty-iff-empty* [*simp*]: *A* ≲ {} ⟷ *A* = {}
⟨*proof*⟩

lemma *not-lesspoll-empty*: ¬ *A* ≺ {}
⟨*proof*⟩

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*⟩

lemma *eqpoll-iff-card-of-ordIso*: *A* ≈ *B* ⟷ *ordIso2* (*card-of A*) (*card-of B*)
⟨*proof*⟩

lemma *eqpoll-refl* [*iff*]: *A* ≈ *A*
⟨*proof*⟩

lemma *eqpoll-finite-iff*: *A* ≈ *B* ⟹ *finite A* ⟷ *finite B*
⟨*proof*⟩

lemma *eqpoll-iff-card*:
assumes *finite A finite B*
shows *A* ≈ *B* ⟷ *card A* = *card B*
⟨*proof*⟩

lemma *eqpoll-singleton-iff*: *A* ≈ {*x*} ⟷ (∃*u*. *A* = {*u*})
⟨*proof*⟩

lemma *eqpoll-doubleton-iff*: *A* ≈ {*x*,*y*} ⟷ (∃*u v*. *A* = {*u*,*v*} ∧ (*u*=*v* ⟷ *x*=*y*))
⟨*proof*⟩

lemma *lepoll-antisym*:
assumes $A \lesssim B$ $B \lesssim A$ **shows** $A \approx B$
⟨*proof*⟩

lemma *lepoll-trans* [*trans*]:
assumes $A \lesssim B$ $B \lesssim C$ **shows** $A \lesssim C$
⟨*proof*⟩

lemma *lepoll-trans1* [*trans*]: $[\![A \approx B;\ B \lesssim C]\!] \Longrightarrow A \lesssim C$
⟨*proof*⟩

lemma *lepoll-trans2* [*trans*]: $[\![A \lesssim B;\ B \approx C]\!] \Longrightarrow A \lesssim C$
⟨*proof*⟩

lemma *eqpoll-sym*: $A \approx B \Longrightarrow B \approx A$
⟨*proof*⟩

lemma *eqpoll-trans* [*trans*]: $[\![A \approx B;\ B \approx C]\!] \Longrightarrow A \approx C$
⟨*proof*⟩

lemma *eqpoll-imp-lepoll*: $A \approx B \Longrightarrow A \lesssim B$
⟨*proof*⟩

lemma *subset-imp-lepoll*: $A \subseteq B \Longrightarrow A \lesssim B$
⟨*proof*⟩

lemma *lepoll-refl* [*iff*]: $A \lesssim A$
⟨*proof*⟩

lemma *lepoll-iff*: $A \lesssim B \longleftrightarrow (\exists\, g.\ A \subseteq g\ `\ B)$
⟨*proof*⟩

lemma *empty-lepoll* [*iff*]: $\{\} \lesssim A$
⟨*proof*⟩

lemma *subset-image-lepoll*: $B \subseteq f\ `\ A \Longrightarrow B \lesssim A$
⟨*proof*⟩

lemma *image-lepoll*: $f\ `\ A \lesssim A$
⟨*proof*⟩

lemma *infinite-le-lepoll*: *infinite* $A \longleftrightarrow (UNIV{::}nat\ set) \lesssim A$
⟨*proof*⟩

lemma *lepoll-Pow-self*: $A \lesssim Pow\ A$
⟨*proof*⟩

lemma *eqpoll-iff-bijections*:
$A \approx B \longleftrightarrow (\exists f\ g.\ (\forall x \in A.\ f\ x \in B \wedge g(f\ x) = x) \wedge (\forall y \in B.\ g\ y \in A \wedge f(g$

$y) = y))$
⟨*proof*⟩

lemma *lepoll-restricted-funspace*:
$\{f.\ f\ `\ A \subseteq B \wedge \{x.\ f\ x \neq k\ x\} \subseteq A \wedge \mathit{finite}\ \{x.\ f\ x \neq k\ x\}\} \lesssim \mathit{Fpow}\ (A \times B)$
⟨*proof*⟩

lemma *singleton-lepoll*: $\{x\} \lesssim \mathit{insert}\ y\ A$
⟨*proof*⟩

lemma *singleton-eqpoll*: $\{x\} \approx \{y\}$
⟨*proof*⟩

lemma *subset-singleton-iff-lepoll*: $(\exists x.\ S \subseteq \{x\}) \longleftrightarrow S \lesssim \{()\}$
⟨*proof*⟩

lemma *infinite-insert-lepoll*:
assumes *infinite A* **shows** $\mathit{insert}\ a\ A \lesssim A$
⟨*proof*⟩

lemma *infinite-insert-eqpoll*: $\mathit{infinite}\ A \Longrightarrow \mathit{insert}\ a\ A \approx A$
⟨*proof*⟩

lemma *finite-lepoll-infinite*:
assumes *infinite A finite B* **shows** $B \lesssim A$
⟨*proof*⟩

lemma *countable-lepoll*: $[\![\mathit{countable}\ A;\ B \lesssim A]\!] \Longrightarrow \mathit{countable}\ B$
⟨*proof*⟩

lemma *countable-eqpoll*: $[\![\mathit{countable}\ A;\ B \approx A]\!] \Longrightarrow \mathit{countable}\ B$
⟨*proof*⟩

34.2 The strict relation

lemma *lesspoll-not-refl* [*iff*]: $\sim (i \prec i)$
⟨*proof*⟩

lemma *lesspoll-imp-lepoll*: $A \prec B \Longrightarrow A \lesssim B$
⟨*proof*⟩

lemma *lepoll-iff-leqpoll*: $A \lesssim B \longleftrightarrow A \prec B \mid A \approx B$
⟨*proof*⟩

lemma *lesspoll-trans* [*trans*]: $[\![X \prec Y;\ Y \prec Z]\!] \Longrightarrow X \prec Z$
⟨*proof*⟩

lemma *lesspoll-trans1* [*trans*]: $[\![X \lesssim Y;\ Y \prec Z]\!] \Longrightarrow X \prec Z$
⟨*proof*⟩

lemma *lesspoll-trans2* [*trans*]: $[\![X \prec Y;\ Y \lesssim Z]\!] \Longrightarrow X \prec Z$
⟨*proof*⟩

lemma *eq-lesspoll-trans* [*trans*]: $[\![X \approx Y;\ Y \prec Z]\!] \Longrightarrow X \prec Z$
⟨*proof*⟩

lemma *lesspoll-eq-trans* [*trans*]: $[\![X \prec Y;\ Y \approx Z]\!] \Longrightarrow X \prec Z$
⟨*proof*⟩

lemma *lesspoll-Pow-self*: $A \prec Pow\ A$
⟨*proof*⟩

lemma *finite-lesspoll-infinite*:
assumes *infinite A finite B* **shows** $B \prec A$
⟨*proof*⟩

lemma *countable-lesspoll*: $[\![countable\ A;\ B \prec A]\!] \Longrightarrow countable\ B$
⟨*proof*⟩

lemma *lepoll-iff-card-le*: $[\![finite\ A;\ finite\ B]\!] \Longrightarrow A \lesssim B \longleftrightarrow card\ A \leq card\ B$
⟨*proof*⟩

lemma *lepoll-iff-finite-card*: $A \lesssim \{..<n::nat\} \longleftrightarrow finite\ A \wedge card\ A \leq n$
⟨*proof*⟩

lemma *eqpoll-iff-finite-card*: $A \approx \{..<n::nat\} \longleftrightarrow finite\ A \wedge card\ A = n$
⟨*proof*⟩

lemma *lesspoll-iff-finite-card*: $A \prec \{..<n::nat\} \longleftrightarrow finite\ A \wedge card\ A < n$
⟨*proof*⟩

34.3 Mapping by an injection

lemma *inj-on-image-eqpoll-self*: $inj\text{-}on\ f\ A \Longrightarrow f\ `\ A \approx A$
⟨*proof*⟩

lemma *inj-on-image-lepoll-1* [*simp*]:
assumes *inj-on f A* **shows** $f\ `\ A \lesssim B \longleftrightarrow A \lesssim B$
⟨*proof*⟩

lemma *inj-on-image-lepoll-2* [*simp*]:
assumes *inj-on f B* **shows** $A \lesssim f\ `\ B \longleftrightarrow A \lesssim B$
⟨*proof*⟩

lemma *inj-on-image-lesspoll-1* [*simp*]:
assumes *inj-on f A* **shows** $f\ `\ A \prec B \longleftrightarrow A \prec B$
⟨*proof*⟩

lemma *inj-on-image-lesspoll-2* [*simp*]:
assumes *inj-on f B* **shows** $A \prec f\ `\ B \longleftrightarrow A \prec B$
⟨*proof*⟩

lemma *inj-on-image-eqpoll-1* [*simp*]:
assumes *inj-on f A* **shows** $f\ `\ A \approx B \longleftrightarrow A \approx B$
⟨*proof*⟩

lemma *inj-on-image-eqpoll-2* [*simp*]:
assumes *inj-on f B* **shows** $A \approx f\ `\ B \longleftrightarrow A \approx B$
⟨*proof*⟩

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*⟩

lemma *insert-eqpoll-insertD*: ⟦*insert u A* $\approx$ *insert v B*; $u \notin A$; $v \notin B$⟧ $\Longrightarrow A \approx B$
⟨*proof*⟩

lemma *insert-lepoll-cong*:
assumes $A \lesssim B$ $b \notin B$ **shows** *insert a A* $\lesssim$ *insert b B*
⟨*proof*⟩

lemma *insert-eqpoll-cong*:
⟦$A \approx B$; $a \notin A$; $b \notin B$⟧ $\Longrightarrow$ *insert a A* $\approx$ *insert b B*
⟨*proof*⟩

lemma *insert-eqpoll-insert-iff*:
⟦$a \notin A$; $b \notin B$⟧ $\Longrightarrow$ *insert a A* $\approx$ *insert b B* $\longleftrightarrow$ $A \approx B$
⟨*proof*⟩

lemma *insert-lepoll-insert-iff*:
⟦$a \notin A$; $b \notin B$⟧ $\Longrightarrow$ (*insert a A* $\lesssim$ *insert b B*) $\longleftrightarrow$ ($A \lesssim B$)
⟨*proof*⟩

lemma *less-imp-insert-lepoll*:
assumes $A \prec B$ **shows** *insert a A* $\lesssim B$
⟨*proof*⟩

lemma *finite-insert-lepoll*: *finite A* $\Longrightarrow$ (*insert a A* $\lesssim A$) $\longleftrightarrow$ ($a \in A$)
⟨*proof*⟩

34.5 Binary sums and unions

lemma *Un-lepoll-mono*:
assumes $A \lesssim C$ $B \lesssim D$ *disjnt C D* **shows** $A \cup B \lesssim C \cup D$
⟨*proof*⟩

lemma *Un-eqpoll-cong*: $[\![A \approx C;\ B \approx D;\ \mathit{disjnt}\ A\ B;\ \mathit{disjnt}\ C\ D]\!] \Longrightarrow A \cup B \approx C \cup D$
⟨*proof*⟩

lemma *sum-lepoll-mono*:
assumes $A \lesssim C$ $B \lesssim D$ **shows** $A <+> B \lesssim C <+> D$
⟨*proof*⟩

lemma *sum-eqpoll-cong*: $[\![A \approx C;\ B \approx D]\!] \Longrightarrow A <+> B \approx C <+> D$
⟨*proof*⟩

34.6 Binary Cartesian products

lemma *times-square-lepoll*: $A \lesssim A \times A$
⟨*proof*⟩

lemma *times-commute-eqpoll*: $A \times B \approx B \times A$
⟨*proof*⟩

lemma *times-assoc-eqpoll*: $(A \times B) \times C \approx A \times (B \times C)$
⟨*proof*⟩

lemma *times-singleton-eqpoll*: $\{a\} \times A \approx A$
⟨*proof*⟩

lemma *times-lepoll-mono*:
assumes $A \lesssim C$ $B \lesssim D$ **shows** $A \times B \lesssim C \times D$
⟨*proof*⟩

lemma *times-eqpoll-cong*: $[\![A \approx C;\ B \approx D]\!] \Longrightarrow A \times B \approx C \times D$
⟨*proof*⟩

lemma
assumes $B \neq \{\}$ **shows** *lepoll-times1*: $A \lesssim A \times B$ **and** *lepoll-times2*: $A \lesssim B \times A$
⟨*proof*⟩

lemma *times-0-eqpoll*: $\{\} \times A \approx \{\}$
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *sum-times-distrib-eqpoll*: $(A <+> B) \times C \approx (A \times C) <+> (B \times C)$
⟨*proof*⟩

lemma *Sigma-eqpoll-cong*:
assumes *h*: *bij-betw h A C* **and** *BD*: $\bigwedge x.\ x \in A \Longrightarrow B\ x \approx D\ (h\ x)$
shows *Sigma A B* ≈ *Sigma C D*
⟨*proof*⟩

lemma *prod-insert-eqpoll*:
assumes $a \notin A$ **shows** *insert a A* × *B* ≈ *B* <+> *A* × *B*
⟨*proof*⟩

34.7 General Unions

lemma *Union-eqpoll-Times*:
assumes *B*: $\bigwedge x.\ x \in A \Longrightarrow F\ x \approx B$ **and** *disj*: *pairwise* (λ*x y*. *disjnt* (*F x*) (*F y*)) *A*
shows $(\bigcup x{\in}A.\ F\ x) \approx A \times B$
⟨*proof*⟩

lemma *UN-lepoll-UN*:
assumes *A*: $\bigwedge x.\ x \in A \Longrightarrow B\ x \lesssim C\ x$
and *disj*: *pairwise* (λ*x y*. *disjnt* (*C x*) (*C y*)) *A*
shows $\bigcup\ (B`A) \lesssim \bigcup\ (C`A)$
⟨*proof*⟩

lemma *UN-eqpoll-UN*:
assumes *A*: $\bigwedge x.\ x \in A \Longrightarrow B\ x \approx C\ x$
and *B*: *pairwise* (λ*x y*. *disjnt* (*B x*) (*B y*)) *A*
and *C*: *pairwise* (λ*x y*. *disjnt* (*C x*) (*C y*)) *A*
shows $(\bigcup x{\in}A.\ B\ x) \approx (\bigcup x{\in}A.\ C\ x)$
⟨*proof*⟩

34.8 General Cartesian products (Pi)

lemma *PiE-sing-eqpoll-self*: $(\{a\} \to_E B) \approx B$
⟨*proof*⟩

lemma *lepoll-funcset-right*:
assumes $B \lesssim B'$ **shows** $A \to_E B \lesssim A \to_E B'$
⟨*proof*⟩

lemma *lepoll-funcset-left*:
assumes $B \neq \{\}$ $A \lesssim A'$
shows $A \to_E B \lesssim A' \to_E B$
⟨*proof*⟩

lemma *lepoll-funcset*:
$[\![B \neq \{\};\ A \lesssim A';\ B \lesssim B']\!] \Longrightarrow A \to_E B \lesssim A' \to_E B'$
⟨*proof*⟩

lemma *lepoll-PiE*:
assumes $\bigwedge i.\ i \in A \Longrightarrow B\ i \lesssim C\ i$
shows $PiE\ A\ B \lesssim PiE\ A\ C$
⟨*proof*⟩

lemma *card-le-PiE-subindex*:
assumes $A \subseteq A'\ Pi_E\ A'\ B \neq \{\}$
shows $PiE\ A\ B \lesssim PiE\ A'\ B$
⟨*proof*⟩

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*⟩

proposition *finite-PiE-iff*:
finite$(PiE\ I\ S) \longleftrightarrow PiE\ I\ S = \{\} \vee$ *finite* $\{i \in I.\ \sim(\exists a.\ S\ i \subseteq \{a\})\} \wedge (\forall i \in I.$ *finite*$(S\ i))$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

corollary *finite-funcset-iff*:
finite$(I \to_E S) \longleftrightarrow (\exists a.\ S \subseteq \{a\}) \vee I = \{\} \vee$ *finite* $I \wedge$ *finite* S
⟨*proof*⟩

34.9 Misc other resultd

lemma *lists-lepoll-mono*:
assumes $A \lesssim B$ **shows** *lists* $A \lesssim$ *lists* B
⟨*proof*⟩

lemma *lepoll-lists*: $A \lesssim$ *lists* A
⟨*proof*⟩

Dedekind's definition of infinite set

lemma *infinite-iff-psubset*: *infinite* $A \longleftrightarrow (\exists B.\ B \subset A \wedge A \approx B)$
⟨*proof*⟩

lemma *infinite-iff-psubset-le*: *infinite* $A \longleftrightarrow (\exists B.\ B \subset A \wedge A \lesssim B)$
⟨*proof*⟩

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*⟩

end

theory *Extended*
  imports *Simps-Case-Conv*
begin

datatype *'a extended* = *Fin 'a* | *Pinf* (‹∞›) | *Minf* (‹−∞›)

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
  ⟨*proof*⟩

end

instance *extended* :: (*linorder*)*linorder*
  ⟨*proof*⟩

lemma *Minf-le*[*simp*]: *Minf* ≤ *y*
⟨*proof*⟩
lemma *le-Pinf*[*simp*]: *x* ≤ *Pinf*
⟨*proof*⟩
lemma *le-Minf*[*simp*]: *x* ≤ *Minf* ⟷ *x* = *Minf*
⟨*proof*⟩
lemma *Pinf-le*[*simp*]: *Pinf* ≤ *x* ⟷ *x* = *Pinf*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

instantiation *extended* :: (*zero*)*zero*
begin
definition *0 = Fin(0::'a)*
instance ⟨*proof*⟩
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*
⟨*proof*⟩

instantiation *extended* :: (*one*)*one*
begin
definition *1 = Fin(1::'a)*
instance ⟨*proof*⟩
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*
⟨*proof*⟩

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 ⟨*proof*⟩

end

instance *extended* :: (*ab-semigroup-add*)*ab-semigroup-add*
⟨*proof*⟩

instance *extended* :: (*ordered-ab-semigroup-add*)*ordered-ab-semigroup-add*
⟨*proof*⟩

instance *extended* :: (*comm-monoid-add*)*comm-monoid-add*
⟨*proof*⟩

instantiation *extended* :: (*uminus*)*uminus*
begin

fun *uminus-extended* **where**
− (Fin x) = Fin (− x) |
− Pinf = Minf |
− Minf = Pinf

instance ⟨*proof*⟩

end

instantiation *extended* :: (*ab-group-add*)*minus*
begin
definition *x − y = x + −*(*y*::*'a extended*)

instance ⟨*proof*⟩
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
⟨*proof*⟩

Numerals:

instance *extended* :: ({*ab-semigroup-add*,*one*})*numeral* ⟨*proof*⟩

lemma *Fin-numeral*[*code-post*]: *Fin*(*numeral w*) = *numeral w*
⟨*proof*⟩

lemma *Fin-neg-numeral*[*code-post*]: *Fin* (− *numeral w*) = − *numeral w*
⟨*proof*⟩

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

⟨*proof*⟩
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*⟩

end

instance *extended* :: ({*conditionally-complete-lattice*,*linorder*}) *complete-linorder*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *INF-nat-binary*:
inf A (*INF x*∈*Collect* ((<) (*0*::*nat*)). *B*) = (*inf A B*::′*a*::*countable-complete-lattice*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *sup-continuous-mono*:
mono F **if** *sup-continuous F*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sup-continuous-compose*:
assumes *f*: *sup-continuous f* **and** *g*: *sup-continuous g*
shows *sup-continuous* (λ*x*. *f* (*g x*))
⟨*proof*⟩

lemma *sup-continuous-sup*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous g* ⟹ *sup-continuous* (λ*x*. *sup* (*f x*) (*g x*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *sup-continuous-and*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∧ *Q x*)
⟨*proof*⟩

lemma *sup-continuous-or*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∨ *Q x*)
⟨*proof*⟩

lemma *sup-continuous-lfp*:
assumes *sup-continuous F* **shows** *lfp F* = (*SUP i*. (*F* ^^ *i*) *bot*) (**is** *lfp F* = *?U*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *inf-continuous-mono*:
mono F **if** *inf-continuous F*
⟨*proof*⟩

lemma [*order-continuous-intros*]:
shows *inf-continuous-const*: *inf-continuous* (*λx*. *c*)
and *inf-continuous-id*: *inf-continuous* (*λx*. *x*)
and *inf-continuous-apply*: *inf-continuous* (*λf*. *f x*)
and *inf-continuous-fun*: (⋀*s*. *inf-continuous* (*λx*. *P x s*)) ⟹ *inf-continuous P*
and *inf-continuous-If*: *inf-continuous F* ⟹ *inf-continuous G* ⟹ *inf-continuous* (*λf*. *if C then F f else G f*)
⟨*proof*⟩

lemma *inf-continuous-inf*[*order-continuous-intros*]:
inf-continuous f ⟹ *inf-continuous g* ⟹ *inf-continuous* (*λx*. *inf* (*f x*) (*g x*))
⟨*proof*⟩

lemma *inf-continuous-sup*[*order-continuous-intros*]:
fixes P Q :: $'a$ :: *countable-complete-lattice* $\Rightarrow$ $'b$ :: *countable-complete-distrib-lattice*
assumes P: *inf-continuous* P **and** Q: *inf-continuous* Q
shows *inf-continuous* $(\lambda x.\ sup\ (P\ x)\ (Q\ x))$
⟨*proof*⟩

lemma *inf-continuous-and*[*order-continuous-intros*]:
inf-continuous $P \Longrightarrow$ *inf-continuous* $Q \Longrightarrow$ *inf-continuous* $(\lambda x.\ P\ x \wedge Q\ x)$
⟨*proof*⟩

lemma *inf-continuous-or*[*order-continuous-intros*]:
inf-continuous $P \Longrightarrow$ *inf-continuous* $Q \Longrightarrow$ *inf-continuous* $(\lambda x.\ P\ x \vee Q\ x)$
⟨*proof*⟩

lemma *inf-continuous-compose*:
assumes f: *inf-continuous* f **and** g: *inf-continuous* g
shows *inf-continuous* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma *inf-continuous-gfp*:
assumes *inf-continuous* F **shows** *gfp* $F = (INF\ i.\ (F \frown i)\ top)$ (**is** *gfp* $F = ?U$)
⟨*proof*⟩

lemma *gfp-transfer*:
assumes α: *inf-continuous* α **and** f: *inf-continuous* f **and** g: *inf-continuous* g
assumes [*simp*]: $\alpha\ top = top$ $\bigwedge x.\ \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (gfp\ f) = gfp\ g$
⟨*proof*⟩

lemma *gfp-transfer-bounded*:
assumes P: $P\ (f\ top)$ $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$ $\bigwedge M.\ antimono\ M \Longrightarrow (\bigwedge i.\ P\ (M\ i)) \Longrightarrow P\ (INF\ i{::}nat.\ M\ i)$
assumes α: $\bigwedge M.\ antimono\ M \Longrightarrow (\bigwedge i{::}nat.\ P\ (M\ i)) \Longrightarrow \alpha\ (INF\ i.\ M\ i) = (INF\ i.\ \alpha\ (M\ i))$
assumes f: *inf-continuous* f **and** g: *inf-continuous* g
assumes [*simp*]: $\bigwedge x.\ P\ x \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
assumes *g-bound*: $\bigwedge x.\ g\ x \leq \alpha\ (f\ top)$
shows $\alpha\ (gfp\ f) = gfp\ g$
⟨*proof*⟩

35.1.1 Least fixed points in countable complete lattices

definition (**in** *countable-complete-lattice*) *cclfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where $cclfp\ f = (SUP\ i.\ (f \frown i)\ bot)$

lemma *cclfp-unfold*:
assumes *sup-continuous* F **shows** $cclfp\ F = F\ (cclfp\ F)$
⟨*proof*⟩

lemma *cclfp-lowerbound*: **assumes** f: *mono f* **and** A: $f\ A \leq A$ **shows** *cclfp f* $\leq A$
⟨*proof*⟩

lemma *cclfp-transfer*:
assumes *sup-continuous* α *mono f*
assumes α *bot* = *bot* $\bigwedge x.\ \alpha\ (f\ x) = g\ (\alpha\ x)$
shows α (*cclfp f*) = *cclfp g*
⟨*proof*⟩

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.* $\sum i{<}n.\ f\ i$)
⟨*proof*⟩

lemma *suminf-eq-SUP*: *suminf f* = (*SUP n.* $\sum i{<}n.\ f\ i$)
⟨*proof*⟩

end

36.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *enat* = *UNIV* :: *nat option set* ⟨*proof*⟩

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 ⟨*proof*⟩

end

instance *enat* :: *countable*
⟨*proof*⟩

old-rep-datatype *enat* ∞ :: *enat*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *not-enat-eq* [*iff*]: (∀ *y*. *x* ≠ *enat y*) = (*x* = ∞)
⟨*proof*⟩

lemma *enat-ex-split*: (∃ *c*::*enat*. *P c*) ⟷ *P* ∞ ∨ (∃ *c*::*nat*. *P c*)
⟨*proof*⟩

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*⟩

end

definition *eSuc* :: *enat* ⇒ *enat* **where**
eSuc i = (*case i of enat n* ⇒ *enat* (*Suc n*) | ∞ ⇒ ∞)

lemma *enat-0* [*code-post*]: *enat 0* = *0*
⟨*proof*⟩

lemma *enat-1* [*code-post*]: *enat 1* = *1*

⟨*proof*⟩

lemma *enat-0-iff*: *enat x = 0 ⟷ x = 0 0 = enat x ⟷ x = 0*
⟨*proof*⟩

lemma *enat-1-iff*: *enat x = 1 ⟷ x = 1 1 = enat x ⟷ x = 1*
⟨*proof*⟩

lemma *one-eSuc*: *1 = eSuc 0*
⟨*proof*⟩

lemma *infinity-ne-i0* [*simp*]: *(∞::enat) ≠ 0*
⟨*proof*⟩

lemma *i0-ne-infinity* [*simp*]: *0 ≠ (∞::enat)*
⟨*proof*⟩

lemma *zero-one-enat-neq*:
¬ 0 = (1::enat)
¬ 1 = (0::enat)
⟨*proof*⟩

lemma *infinity-ne-i1* [*simp*]: *(∞::enat) ≠ 1*
⟨*proof*⟩

lemma *i1-ne-infinity* [*simp*]: *1 ≠ (∞::enat)*
⟨*proof*⟩

lemma *eSuc-enat*: *eSuc (enat n) = enat (Suc n)*
⟨*proof*⟩

lemma *eSuc-infinity* [*simp*]: *eSuc ∞ = ∞*
⟨*proof*⟩

lemma *eSuc-ne-0* [*simp*]: *eSuc n ≠ 0*
⟨*proof*⟩

lemma *zero-ne-eSuc* [*simp*]: *0 ≠ eSuc n*
⟨*proof*⟩

lemma *eSuc-inject* [*simp*]: *eSuc m = eSuc n ⟷ m = n*
⟨*proof*⟩

lemma *eSuc-enat-iff*: *eSuc x = enat y ⟷ (∃ n. y = Suc n ∧ x = enat n)*
⟨*proof*⟩

lemma *enat-eSuc-iff*: *enat y = eSuc x ⟷ (∃ n. y = Suc n ∧ enat n = x)*
⟨*proof*⟩

36.3 Addition

instantiation *enat* :: *comm-monoid-add*
begin

definition [*nitpick-simp*]:
$m + n = (case\ m\ of\ \infty \Rightarrow \infty \mid enat\ m \Rightarrow (case\ n\ of\ \infty \Rightarrow \infty \mid enat\ n \Rightarrow enat\ (m + n)))$

lemma *plus-enat-simps* [*simp*, *code*]:
fixes q :: *enat*
shows $enat\ m + enat\ n = enat\ (m + n)$
and $\infty + q = \infty$
and $q + \infty = \infty$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *eSuc-plus-1*:
$eSuc\ n = n + 1$
⟨*proof*⟩

lemma *plus-1-eSuc*:
$1 + q = eSuc\ q$
$q + 1 = eSuc\ q$
⟨*proof*⟩

lemma *iadd-Suc*: $eSuc\ m + n = eSuc\ (m + n)$
⟨*proof*⟩

lemma *iadd-Suc-right*: $m + eSuc\ n = eSuc\ (m + n)$
⟨*proof*⟩

36.4 Multiplication

instantiation *enat* :: {*comm-semiring-1*, *semiring-no-zero-divisors*}
begin

definition *times-enat-def* [*nitpick-simp*]:
$m * n = (case\ m\ of\ \infty \Rightarrow if\ n = 0\ then\ 0\ else\ \infty \mid enat\ m \Rightarrow (case\ n\ of\ \infty \Rightarrow if\ m = 0\ then\ 0\ else\ \infty \mid enat\ n \Rightarrow enat\ (m * n)))$

lemma *times-enat-simps* [*simp*, *code*]:
$enat\ m * enat\ n = enat\ (m * n)$
$\infty * \infty = (\infty :: enat)$
$\infty * enat\ n = (if\ n = 0\ then\ 0\ else\ \infty)$
$enat\ m * \infty = (if\ m = 0\ then\ 0\ else\ \infty)$

⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *mult-eSuc*: $eSuc\ m * n = n + m * n$
⟨*proof*⟩

lemma *mult-eSuc-right*: $m * eSuc\ n = m + m * n$
⟨*proof*⟩

lemma *of-nat-eq-enat*: $of\text{-}nat\ n = enat\ n$
⟨*proof*⟩

instance *enat* :: *semiring-char-0*
⟨*proof*⟩

lemma *imult-is-infinity*: $((a::enat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$
⟨*proof*⟩

36.5 Numerals

lemma *numeral-eq-enat*:
$numeral\ k = enat\ (numeral\ k)$
⟨*proof*⟩

lemma *enat-numeral* [*code-abbrev*]:
$enat\ (numeral\ k) = numeral\ k$
⟨*proof*⟩

lemma *infinity-ne-numeral* [*simp*]: $(\infty::enat) \neq numeral\ k$
⟨*proof*⟩

lemma *numeral-ne-infinity* [*simp*]: $numeral\ k \neq (\infty::enat)$
⟨*proof*⟩

lemma *eSuc-numeral* [*simp*]: $eSuc\ (numeral\ k) = numeral\ (k + Num.One)$
⟨*proof*⟩

36.6 Subtraction

instantiation *enat* :: *minus*
begin

definition *diff-enat-def*:
$a - b = (case\ a\ of\ (enat\ x) \Rightarrow (case\ b\ of\ (enat\ y) \Rightarrow enat\ (x - y) \mid \infty \Rightarrow 0)$
$\quad \mid \infty \Rightarrow \infty)$

instance ⟨*proof*⟩

end

lemma *idiff-enat-enat* [*simp*, *code*]: *enat a* − *enat b* = *enat* (*a* − *b*)
⟨*proof*⟩

lemma *idiff-infinity* [*simp*, *code*]: ∞ − *n* = (∞::*enat*)
⟨*proof*⟩

lemma *idiff-infinity-right* [*simp*, *code*]: *enat a* − ∞ = *0*
⟨*proof*⟩

lemma *idiff-0* [*simp*]: (*0*::*enat*) − *n* = *0*
⟨*proof*⟩

lemmas *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

lemma *idiff-0-right* [*simp*]: (*n*::*enat*) − *0* = *n*
⟨*proof*⟩

lemmas *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

lemma *idiff-self* [*simp*]: *n* ≠ ∞ ⟹ (*n*::*enat*) − *n* = *0*
⟨*proof*⟩

lemma *eSuc-minus-eSuc* [*simp*]: *eSuc n* − *eSuc m* = *n* − *m*
⟨*proof*⟩

lemma *eSuc-minus-1* [*simp*]: *eSuc n* − *1* = *n*
⟨*proof*⟩

36.7 Ordering

instantiation *enat* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:
m ≤ *n* = (*case n of enat n1* ⇒ (*case m of enat m1* ⇒ *m1* ≤ *n1* | ∞ ⇒ *False*)
| ∞ ⇒ *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*

$q \leq (\infty::enat)$
$q < (\infty::enat) \longleftrightarrow q \neq \infty$
$(\infty::enat) \leq q \longleftrightarrow q = \infty$
$(\infty::enat) < q \longleftrightarrow False$
⟨*proof*⟩

lemma *numeral-le-enat-iff* [*simp*]:
shows $numeral\ m \leq enat\ n \longleftrightarrow numeral\ m \leq n$
⟨*proof*⟩

lemma *numeral-less-enat-iff* [*simp*]:
shows $numeral\ m < enat\ n \longleftrightarrow numeral\ m < n$
⟨*proof*⟩

lemma *enat-ord-code* [*code*]:
$enat\ m \leq enat\ n \longleftrightarrow m \leq n$
$enat\ m < enat\ n \longleftrightarrow m < n$
$q \leq (\infty::enat) \longleftrightarrow True$
$enat\ m < \infty \longleftrightarrow True$
$\infty \leq enat\ n \longleftrightarrow False$
$(\infty::enat) < q \longleftrightarrow False$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *enat* :: *dioid*
⟨*proof*⟩

instance *enat* :: {*linordered-nonzero-semiring*, *strict-ordered-comm-monoid-add*}
⟨*proof*⟩

lemma *add-diff-assoc-enat*: $z \leq y \Longrightarrow x + (y - z) = x + y - (z::enat)$
⟨*proof*⟩

lemma *enat-ord-number* [*simp*]:
$(numeral\ m :: enat) \leq numeral\ n \longleftrightarrow (numeral\ m :: nat) \leq numeral\ n$
$(numeral\ m :: enat) < numeral\ n \longleftrightarrow (numeral\ m :: nat) < numeral\ n$
⟨*proof*⟩

lemma *infinity-ileE* [*elim*!]: $\infty \leq enat\ m \Longrightarrow R$
⟨*proof*⟩

lemma *infinity-ilessE* [*elim*!]: $\infty < enat\ m \Longrightarrow R$
⟨*proof*⟩

lemma *eSuc-ile-mono* [*simp*]: $eSuc\ n \leq eSuc\ m \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *eSuc-mono* [*simp*]: $eSuc\ n < eSuc\ m \longleftrightarrow n < m$
⟨*proof*⟩

lemma *ile-eSuc* [*simp*]: $n \leq eSuc\ n$
⟨*proof*⟩

lemma *not-eSuc-ilei0* [*simp*]: $\neg\ eSuc\ n \leq 0$
⟨*proof*⟩

lemma *i0-iless-eSuc* [*simp*]: $0 < eSuc\ n$
⟨*proof*⟩

lemma *iless-eSuc0*[*simp*]: $(n < eSuc\ 0) = (n = 0)$
⟨*proof*⟩

lemma *ileI1*: $m < n \Longrightarrow eSuc\ m \leq n$
⟨*proof*⟩

lemma *Suc-ile-eq*: $enat\ (Suc\ m) \leq n \longleftrightarrow enat\ m < n$
⟨*proof*⟩

lemma *iless-Suc-eq* [*simp*]: $enat\ m < eSuc\ n \longleftrightarrow enat\ m \leq n$
⟨*proof*⟩

lemma *imult-infinity*: $(0::enat) < n \Longrightarrow \infty * n = \infty$
⟨*proof*⟩

lemma *imult-infinity-right*: $(0::enat) < n \Longrightarrow n * \infty = \infty$
⟨*proof*⟩

lemma *enat-0-less-mult-iff*: $(0 < (m::enat) * n) = (0 < m \wedge 0 < n)$
⟨*proof*⟩

lemma *mono-eSuc*: *mono eSuc*
⟨*proof*⟩

lemma *min-enat-simps* [*simp*]:
$min\ (enat\ m)\ (enat\ n) = enat\ (min\ m\ n)$
$min\ q\ 0 = 0$
$min\ 0\ q = 0$
$min\ q\ (\infty::enat) = q$
$min\ (\infty::enat)\ q = q$
⟨*proof*⟩

lemma *max-enat-simps* [*simp*]:

$max\ (enat\ m)\ (enat\ n) = enat\ (max\ m\ n)$
$max\ q\ 0 = q$
$max\ 0\ q = q$
$max\ q\ \infty = (\infty::enat)$
$max\ \infty\ q = (\infty::enat)$
⟨*proof*⟩

lemma *enat-ile*: $n \leq enat\ m \Longrightarrow \exists\, k.\ n = enat\ k$
⟨*proof*⟩

lemma *enat-iless*: $n < enat\ m \Longrightarrow \exists\, k.\ n = enat\ k$
⟨*proof*⟩

lemma *iadd-le-enat-iff*:
$x + y \leq enat\ n \longleftrightarrow (\exists\, y'\ x'.\ x = enat\ x' \wedge y = enat\ y' \wedge x' + y' \leq n)$
⟨*proof*⟩

lemma *chain-incr*: $\forall\, i.\ \exists\, j.\ Y\ i < Y\ j \Longrightarrow \exists\, j.\ enat\ k < Y\ j$
⟨*proof*⟩

lemma *eSuc-max*: $eSuc\ (max\ x\ y) = max\ (eSuc\ x)\ (eSuc\ y)$
⟨*proof*⟩

lemma *eSuc-Max*:
assumes *finite A* $A \neq \{\}$
shows $eSuc\ (Max\ A) = Max\ (eSuc\ `\ A)$
⟨*proof*⟩

instantiation *enat* :: {*order-bot*, *order-top*}
begin

definition *bot-enat* :: *enat* **where** *bot-enat* = *0*
definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
⟨*proof*⟩

end

lemma *finite-enat-bounded*:
assumes *le-fin*: $\bigwedge y.\ y \in A \Longrightarrow y \leq enat\ n$
shows *finite A*
⟨*proof*⟩

36.8 Cancellation simprocs

lemma *add-diff-cancel-enat*[*simp*]: $x \neq \infty \Longrightarrow x + y - x = (y::enat)$
⟨*proof*⟩

lemma *enat-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::enat) \vee b = c$
⟨*proof*⟩

lemma *enat-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::enat) \vee b \leq c$
⟨*proof*⟩

lemma *enat-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::enat) \wedge b < c$
⟨*proof*⟩

lemma *plus-eq-infty-iff-enat*: $(m::enat) + n = \infty \longleftrightarrow m{=}\infty \vee n{=}\infty$
⟨*proof*⟩

⟨*ML*⟩

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

36.9 Well-ordering

lemma *less-enatE*:
$[\![n < enat\ m;\ \bigwedge k.\ [\![n = enat\ k;\ k < m]\!] \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *less-infinityE*:
$[\![n < \infty;\ \bigwedge k.\ n = enat\ k \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *enat-less-induct*:
assumes $\bigwedge n.\ \forall m::enat.\ m < n \longrightarrow P\ m \Longrightarrow P\ n$
shows $P\ n$
⟨*proof*⟩

instance *enat* :: *wellorder*
⟨*proof*⟩

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*⟩

end

instance *enat* :: *complete-linorder* ⟨*proof*⟩

lemma *eSuc-Sup*: *A* ≠ {} ⟹ *eSuc* (*Sup A*) = *Sup* (*eSuc* ‘ *A*)
⟨*proof*⟩

lemma *sup-continuous-eSuc*: *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *eSuc* (*f x*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *i0-lb* : (*0*::*enat*) ≤ *n*
⟨*proof*⟩

lemma *ile0-eq*: *n* ≤ (*0*::*enat*) ⟷ *n* = *0*
⟨*proof*⟩

lemma *not-iless0*: ¬ *n* < (*0*::*enat*)
⟨*proof*⟩

lemma *i0-less*[*simp*]: (*0*::*enat*) < *n* ⟷ *n* ≠ *0*
⟨*proof*⟩

lemma *imult-is-0*: ((*m*::*enat*) ∗ *n* = *0*) = (*m* = *0* ∨ *n* = *0*)
⟨*proof*⟩

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*⟩

lemma (**in** *conditionally-complete-linorder*) *le-cSUP-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq Sup\ (f\ `\ A) \longleftrightarrow (\forall\, y{<}x.\ \exists\, i{\in}A.\ y < f\ i)$
⟨*proof*⟩

lemma *le-cSup-iff-less*:
fixes $x :: 'a :: \{$*conditionally-complete-linorder*, *dense-linorder*$\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq (SUP\ i{\in}A.\ f\ i) \longleftrightarrow (\forall\, y{<}x.\ \exists\, i{\in}A.\ y \leq f\ i)$
⟨*proof*⟩

lemma *le-Sup-iff-less*:
fixes $x :: 'a :: \{$*complete-linorder*, *dense-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*)
⟨*proof*⟩

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*⟩

lemma (**in** *conditionally-complete-linorder*) *cINF-le-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow Inf\ (f\ `\ A) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ y > f\ i)$
⟨*proof*⟩

lemma *cInf-le-iff-less*:
fixes $x :: 'a :: \{$*conditionally-complete-linorder*, *dense-linorder*$\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow (INF\ i{\in}A.\ f\ i) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ f\ i \leq y)$
⟨*proof*⟩

lemma *Inf-le-iff-less*:
fixes $x :: 'a :: \{$*complete-linorder*, *dense-linorder*$\}$
shows $(INF\ i{\in}A.\ f\ i) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ f\ i \leq y)$
⟨*proof*⟩

lemma *SUP-pair*:
fixes $f :: - \Rightarrow - \Rightarrow -$ :: *complete-lattice*
shows $(SUP\ i \in A.\ SUP\ j \in B.\ f\ i\ j) = (SUP\ p \in A \times B.\ f\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

lemma *INF-pair*:
fixes $f :: - \Rightarrow - \Rightarrow -$ :: *complete-lattice*
shows $(INF\ i \in A.\ INF\ j \in B.\ f\ i\ j) = (INF\ p \in A \times B.\ f\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

lemma *INF-Sigma*:
fixes *f* :: - ⇒ - ⇒ - :: *complete-lattice*
shows (*INF i* ∈ *A*. *INF j* ∈ *B i*. *f i j*) = (*INF p* ∈ *Sigma A B*. *f* (*fst p*) (*snd p*))
⟨*proof*⟩

37.0.1 *Liminf* and *Limsup*

definition *Liminf* :: *'a filter* ⇒ (*'a* ⇒ *'b*) ⇒ *'b* :: *complete-lattice* **where**
Liminf F f = (*SUP P*∈{*P*. *eventually P F*}. *INF x*∈{*x*. *P x*}. *f x*)

definition *Limsup* :: *'a filter* ⇒ (*'a* ⇒ *'b*) ⇒ *'b* :: *complete-lattice* **where**
Limsup F f = (*INF P*∈{*P*. *eventually P F*}. *SUP x*∈{*x*. *P x*}. *f x*)

abbreviation *liminf* ≡ *Liminf sequentially*

abbreviation *limsup* ≡ *Limsup sequentially*

lemma *Liminf-eqI*:
(⋀*P*. *eventually P F* ⟹ *Inf* (*f* ' (*Collect P*)) ≤ *x*) ⟹
(⋀*y*. (⋀*P*. *eventually P F* ⟹ *Inf* (*f* ' (*Collect P*)) ≤ *y*) ⟹ *x* ≤ *y*) ⟹ *Liminf F f* = *x*
⟨*proof*⟩

lemma *Limsup-eqI*:
(⋀*P*. *eventually P F* ⟹ *x* ≤ *Sup* (*f* ' (*Collect P*))) ⟹
(⋀*y*. (⋀*P*. *eventually P F* ⟹ *y* ≤ *Sup* (*f* ' (*Collect P*))) ⟹ *y* ≤ *x*) ⟹ *Limsup F f* = *x*
⟨*proof*⟩

lemma *liminf-SUP-INF*: *liminf f* = (*SUP n*. *INF m*∈{*n*..}. *f m*)
⟨*proof*⟩

lemma *limsup-INF-SUP*: *limsup f* = (*INF n*. *SUP m*∈{*n*..}. *f m*)
⟨*proof*⟩

lemma *mem-limsup-iff*: *x* ∈ *limsup A* ⟷ (∃ *F n in sequentially*. *x* ∈ *A n*)
⟨*proof*⟩

lemma *mem-liminf-iff*: *x* ∈ *liminf A* ⟷ (∀ *F n in sequentially*. *x* ∈ *A n*)
⟨*proof*⟩

lemma *Limsup-const*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Limsup F* (λ*x*. *c*) = *c*
⟨*proof*⟩

lemma *Liminf-const*:
assumes *ntriv*: ¬ *trivial-limit F*

shows *Liminf F* (λ*x*. *c*) = *c*
⟨*proof*⟩

lemma *Liminf-mono*:
assumes *ev*: *eventually* (λ*x*. *f x* ≤ *g x*) *F*
shows *Liminf F f* ≤ *Liminf F g*
⟨*proof*⟩

lemma *Liminf-eq*:
assumes *eventually* (λ*x*. *f x* = *g x*) *F*
shows *Liminf F f* = *Liminf F g*
⟨*proof*⟩

lemma *Limsup-mono*:
assumes *ev*: *eventually* (λ*x*. *f x* ≤ *g x*) *F*
shows *Limsup F f* ≤ *Limsup F g*
⟨*proof*⟩

lemma *Limsup-eq*:
assumes *eventually* (λ*x*. *f x* = *g x*) *net*
shows *Limsup net f* = *Limsup net g*
⟨*proof*⟩

lemma *Liminf-bot*[*simp*]: *Liminf bot f* = *top*
⟨*proof*⟩

lemma *Limsup-bot*[*simp*]: *Limsup bot f* = *bot*
⟨*proof*⟩

lemma *Liminf-le-Limsup*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F f* ≤ *Limsup F f*
⟨*proof*⟩

lemma *Liminf-bounded*:
assumes *le*: *eventually* (λ*n*. *C* ≤ *X n*) *F*
shows *C* ≤ *Liminf F X*
⟨*proof*⟩

lemma *Limsup-bounded*:
assumes *le*: *eventually* (λ*n*. *X n* ≤ *C*) *F*
shows *Limsup F X* ≤ *C*
⟨*proof*⟩

lemma *le-Limsup*:
assumes *F*: *F* ≠ *bot* **and** *x*: ∀ *x in F*. *l* ≤ *f x*
shows *l* ≤ *Limsup F f*
⟨*proof*⟩

lemma *Liminf-le*:
assumes *F*: $F \neq bot$ **and** *x*: $\forall_F\ x\ in\ F.\ f\ x \leq l$
shows $Liminf\ F\ f \leq l$
⟨*proof*⟩

lemma *le-Liminf-iff*:
fixes *X* :: - ⇒ - :: *complete-linorder*
shows $C \leq Liminf\ F\ X \longleftrightarrow (\forall\, y{<}C.\ eventually\ (\lambda x.\ y < X\ x)\ F)$
⟨*proof*⟩

lemma *Limsup-le-iff*:
fixes *X* :: - ⇒ - :: *complete-linorder*
shows $C \geq Limsup\ F\ X \longleftrightarrow (\forall\, y{>}C.\ eventually\ (\lambda x.\ y > X\ x)\ F)$
⟨*proof*⟩

lemma *less-LiminfD*:
$y < Liminf\ F\ (f :: - \Rightarrow 'a :: complete\text{-}linorder) \Longrightarrow eventually\ (\lambda x.\ f\ x > y)\ F$
⟨*proof*⟩

lemma *Limsup-lessD*:
$y > Limsup\ F\ (f :: - \Rightarrow 'a :: complete\text{-}linorder) \Longrightarrow eventually\ (\lambda x.\ f\ x < y)\ F$
⟨*proof*⟩

lemma *lim-imp-Liminf*:
fixes *f* :: $'a \Rightarrow$ - :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
assumes *lim*: $(f \longrightarrow f0)\ F$
shows $Liminf\ F\ f = f0$
⟨*proof*⟩

lemma *lim-imp-Limsup*:
fixes *f* :: $'a \Rightarrow$ - :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
assumes *lim*: $(f \longrightarrow f0)\ F$
shows $Limsup\ F\ f = f0$
⟨*proof*⟩

lemma *Liminf-eq-Limsup*:
fixes *f0* :: $'a$ :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
and *lim*: $Liminf\ F\ f = f0\ Limsup\ F\ f = f0$
shows $(f \longrightarrow f0)\ F$
⟨*proof*⟩

lemma *tendsto-iff-Liminf-eq-Limsup*:
fixes *f0* :: $'a$ :: {*complete-linorder*,*linorder-topology*}
shows ¬ *trivial-limit F* ⟹ $(f \longrightarrow f0)\ F \longleftrightarrow (Liminf\ F\ f = f0 \wedge Limsup\ F\ f = f0)$
⟨*proof*⟩

lemma *liminf-subseq-mono*:
fixes $X :: nat \Rightarrow 'a :: complete\text{-}linorder$
assumes *strict-mono r*
shows $liminf\ X \leq liminf\ (X \circ r)$
⟨*proof*⟩

lemma *limsup-subseq-mono*:
fixes $X :: nat \Rightarrow 'a :: complete\text{-}linorder$
assumes *strict-mono r*
shows $limsup\ (X \circ r) \leq limsup\ X$
⟨*proof*⟩

lemma *continuous-on-imp-continuous-within*:
$continuous\text{-}on\ s\ f \Longrightarrow t \subseteq s \Longrightarrow x \in s \Longrightarrow continuous\ (at\ x\ within\ t)\ f$
⟨*proof*⟩

lemma *Liminf-compose-continuous-mono*:
fixes $f :: 'a::\{complete\text{-}linorder, linorder\text{-}topology\} \Rightarrow 'b::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: $F \neq bot$
shows $Liminf\ F\ (\lambda n.\ f\ (g\ n)) = f\ (Liminf\ F\ g)$
⟨*proof*⟩

lemma *Limsup-compose-continuous-mono*:
fixes $f :: 'a::\{complete\text{-}linorder, linorder\text{-}topology\} \Rightarrow 'b::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: $F \neq bot$
shows $Limsup\ F\ (\lambda n.\ f\ (g\ n)) = f\ (Limsup\ F\ g)$
⟨*proof*⟩

lemma *Liminf-compose-continuous-antimono*:
fixes $f :: 'a::\{complete\text{-}linorder,linorder\text{-}topology\} \Rightarrow 'b::\{complete\text{-}linorder,linorder\text{-}topology\}$
assumes *c*: *continuous-on UNIV f*
and *am*: *antimono f*
and *F*: $F \neq bot$
shows $Liminf\ F\ (\lambda n.\ f\ (g\ n)) = f\ (Limsup\ F\ g)$
⟨*proof*⟩

lemma *Limsup-compose-continuous-antimono*:
fixes $f :: 'a::\{complete\text{-}linorder, linorder\text{-}topology\} \Rightarrow 'b::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes *c*: *continuous-on UNIV f* **and** *am*: *antimono f* **and** *F*: $F \neq bot$
shows $Limsup\ F\ (\lambda n.\ f\ (g\ n)) = f\ (Liminf\ F\ g)$
⟨*proof*⟩

lemma *Liminf-filtermap-le*: $Liminf\ (filtermap\ f\ F)\ g \leq Liminf\ F\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *Limsup-filtermap-ge*: $Limsup\ (filtermap\ f\ F)\ g \geq Limsup\ F\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *Liminf-least*: $(\bigwedge P.\ \textit{eventually}\ P\ F \Longrightarrow (\textit{INF}\ x{\in}\textit{Collect}\ P.\ f\ x) \leq x) \Longrightarrow \textit{Liminf}\ F\ f \leq x$
⟨*proof*⟩

lemma *Limsup-greatest*: $(\bigwedge P.\ \textit{eventually}\ P\ F \Longrightarrow x \leq (\textit{SUP}\ x{\in}\textit{Collect}\ P.\ f\ x)) \Longrightarrow \textit{Limsup}\ F\ f \geq x$
⟨*proof*⟩

lemma *Liminf-filtermap-ge*: $\textit{inj}\ f \Longrightarrow \textit{Liminf}\ (\textit{filtermap}\ f\ F)\ g \geq \textit{Liminf}\ F\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *Limsup-filtermap-le*: $\textit{inj}\ f \Longrightarrow \textit{Limsup}\ (\textit{filtermap}\ f\ F)\ g \leq \textit{Limsup}\ F\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *Liminf-filtermap-eq*: $\textit{inj}\ f \Longrightarrow \textit{Liminf}\ (\textit{filtermap}\ f\ F)\ g = \textit{Liminf}\ F\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *Limsup-filtermap-eq*: $\textit{inj}\ f \Longrightarrow \textit{Limsup}\ (\textit{filtermap}\ f\ F)\ g = \textit{Limsup}\ F\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

37.1 More Limits

lemma *convergent-limsup-cl*:
fixes $X :: \textit{nat} \Rightarrow {'a}{::}\{\textit{complete-linorder},\textit{linorder-topology}\}$
shows $\textit{convergent}\ X \Longrightarrow \textit{limsup}\ X = \textit{lim}\ X$
⟨*proof*⟩

lemma *convergent-liminf-cl*:
fixes $X :: \textit{nat} \Rightarrow {'a}{::}\{\textit{complete-linorder},\textit{linorder-topology}\}$
shows $\textit{convergent}\ X \Longrightarrow \textit{liminf}\ X = \textit{lim}\ X$
⟨*proof*⟩

lemma *lim-increasing-cl*:
assumes $\bigwedge n\ m.\ n \geq m \Longrightarrow f\ n \geq f\ m$
obtains l **where** $f \longrightarrow (l{::}{'a}{::}\{\textit{complete-linorder},\textit{linorder-topology}\})$
⟨*proof*⟩

lemma *lim-decreasing-cl*:
assumes $\bigwedge n\ m.\ n \geq m \Longrightarrow f\ n \leq f\ m$
obtains l **where** $f \longrightarrow (l{::}{'a}{::}\{\textit{complete-linorder},\textit{linorder-topology}\})$
⟨*proof*⟩

lemma *compact-complete-linorder*:
fixes $X :: \textit{nat} \Rightarrow {'a}{::}\{\textit{complete-linorder},\textit{linorder-topology}\}$
shows $\exists\, l\ r.\ \textit{strict-mono}\ r \wedge (X \circ r) \longrightarrow l$

⟨*proof*⟩

lemma *tendsto-Limsup*:
fixes *f* :: - ⇒ *'a* :: {*complete-linorder*,*linorder-topology*}
shows *F* ≠ *bot* ⟹ *Limsup F f* = *Liminf F f* ⟹ (*f* ⟶ *Limsup F f*) *F*
⟨*proof*⟩

lemma *tendsto-Liminf*:
fixes *f* :: - ⇒ *'a* :: {*complete-linorder*,*linorder-topology*}
shows *F* ≠ *bot* ⟹ *Limsup F f* = *Liminf F f* ⟹ (*f* ⟶ *Liminf F f*) *F*
⟨*proof*⟩

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* ⇒ *nat* ⇒ *'a*::*ordered-comm-monoid-add*
shows (⋀*n*. *n* ∈ *A* ⟹ *mono* (*f n*)) ⟹ *mono* (λ*i*. ∑ *n*∈*A*. *f n i*)
⟨*proof*⟩

lemma *incseq-sumI*:
fixes *f* :: *nat* ⇒ *'a*::*ordered-comm-monoid-add*
assumes ⋀*i*. *0* ≤ *f i*
shows *incseq* (λ*i*. *sum f* {..< *i*})
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

instantiation *enat* :: *linorder-topology*
begin

definition *open-enat* :: *enat set* ⇒ *bool* **where**
open-enat = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
⟨*proof*⟩

end

lemma *open-enat*: *open* {*enat n*}
⟨*proof*⟩

lemma *open-enat-iff*:
fixes *A* :: *enat set*
shows *open A* ⟷ (∞ ∈ *A* ⟶ (∃ *n*::*nat*. {*n* <..} ⊆ *A*))
⟨*proof*⟩

lemma *nhds-enat*: *nhds x* = (*if x* = ∞ *then INF i*. *principal* {*enat i*..} *else principal*
{*x*})
⟨*proof*⟩

instance *enat* :: *topological-comm-monoid-add*
⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *ereal* :: *infinity*
begin

definition (∞::*ereal*) = *PInfty*
instance ⟨*proof*⟩

end

declare [[*coercion ereal* :: *real* ⇒ *ereal*]]

lemma *ereal-uminus-uminus*[*simp*]:
  fixes *a* :: *ereal*
  shows − (− *a*) = *a*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

declare
  PInfty-eq-infinity[*code-post*]
  MInfty-eq-minfinity[*code-post*]

lemma [*code-unfold*]:
$\infty = PInfty$
$- PInfty = MInfty$
⟨*proof*⟩

lemma *inj-ereal*[*simp*]: *inj-on ereal A*
⟨*proof*⟩

lemma *ereal-cases*[*cases type*: *ereal*]:
obtains (*real*) r **where** $x = ereal\ r$
| (*PInf*) $x = \infty$
| (*MInf*) $x = -\infty$
⟨*proof*⟩

lemmas *ereal2-cases* = *ereal-cases*[*case-product ereal-cases*]
lemmas *ereal3-cases* = *ereal2-cases*[*case-product ereal-cases*]

lemma *ereal-all-split*: $\bigwedge P.\ (\forall\, x{::}ereal.\ P\ x) \longleftrightarrow P\ \infty \wedge (\forall\, x.\ P\ (ereal\ x)) \wedge P\ (-\infty)$
⟨*proof*⟩

lemma *ereal-ex-split*: $\bigwedge P.\ (\exists\, x{::}ereal.\ P\ x) \longleftrightarrow P\ \infty \vee (\exists\, x.\ P\ (ereal\ x)) \vee P\ (-\infty)$
⟨*proof*⟩

lemma *ereal-uminus-eq-iff*[*simp*]:
fixes $a\ b :: ereal$
shows $-a = -b \longleftrightarrow a = b$
⟨*proof*⟩

function *real-of-ereal* :: *ereal* ⇒ *real* **where**
real-of-ereal (*ereal r*) = *r*
| *real-of-ereal* ∞ = *0*
| *real-of-ereal* (−∞) = *0*
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *real-of-ereal*[*simp*]:
real-of-ereal (− *x* :: *ereal*) = − (*real-of-ereal x*)
⟨*proof*⟩

lemma *range-ereal*[*simp*]: *range ereal* = *UNIV* − {∞, −∞}
⟨*proof*⟩

lemma *ereal-range-uminus*[*simp*]: *range uminus* = (*UNIV*::*ereal set*)
⟨*proof*⟩

instantiation *ereal* :: *abs*
begin

function *abs-ereal* **where**
  $|ereal\ r| = ereal\ |r|$
| $|-\infty| = (\infty::ereal)$
| $|\infty| = (\infty::ereal)$
⟨*proof*⟩
termination ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *abs-eq-infinity-cases*[*elim*!]:
  fixes $x :: ereal$
  assumes $|x| = \infty$
  obtains $x = \infty \mid x = -\infty$
  ⟨*proof*⟩

lemma *abs-neq-infinity-cases*[*elim*!]:
  fixes $x :: ereal$
  assumes $|x| \neq \infty$
  obtains r **where** $x = ereal\ r$
  ⟨*proof*⟩

lemma *abs-ereal-uminus*[*simp*]:
  fixes $x :: ereal$
  shows $|-\ x| = |x|$
  ⟨*proof*⟩

lemma *ereal-infinity-cases*:
  fixes $a :: ereal$
  shows $a \neq \infty \Longrightarrow a \neq -\infty \Longrightarrow |a| \neq \infty$
  ⟨*proof*⟩

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**
  $ereal\ r + ereal\ p = ereal\ (r + p)$
| $\infty + a = (\infty::ereal)$
| $a + \infty = (\infty::ereal)$
| $ereal\ r + -\infty = -\infty$
| $-\infty + ereal\ p = -(\infty::ereal)$
| $-\infty + -\infty = -(\infty::ereal)$

⟨*proof*⟩
termination ⟨*proof*⟩

lemma *Infty-neq-0*[*simp*]:
(∞::*ereal*) ≠ *0* *0* ≠ (∞::*ereal*)
−(∞::*ereal*) ≠ *0* *0* ≠ −(∞::*ereal*)
⟨*proof*⟩

lemma *ereal-eq-0*[*simp*]:
ereal r = *0* ⟷ *r* = *0*
0 = *ereal r* ⟷ *r* = *0*
⟨*proof*⟩

lemma *ereal-eq-1*[*simp*]:
ereal r = *1* ⟷ *r* = *1*
1 = *ereal r* ⟷ *r* = *1*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *ereal-0-plus* [*simp*]: *ereal 0* + *x* = *x*
and *plus-ereal-0* [*simp*]: *x* + *ereal 0* = *x*
⟨*proof*⟩

instance *ereal* :: *numeral* ⟨*proof*⟩

lemma *real-of-ereal-0*[*simp*]: *real-of-ereal* (*0*::*ereal*) = *0*
⟨*proof*⟩

lemma *abs-ereal-zero*[*simp*]: |*0*| = (*0*::*ereal*)
⟨*proof*⟩

lemma *ereal-uminus-zero*[*simp*]: − *0* = (*0*::*ereal*)
⟨*proof*⟩

lemma *ereal-uminus-zero-iff*[*simp*]:
fixes *a* :: *ereal*
shows −*a* = *0* ⟷ *a* = *0*
⟨*proof*⟩

lemma *ereal-plus-eq-PInfty*[*simp*]:
fixes *a b* :: *ereal*
shows *a* + *b* = ∞ ⟷ *a* = ∞ ∨ *b* = ∞
⟨*proof*⟩

lemma *ereal-plus-eq-MInfty*[*simp*]:

fixes a b :: *ereal*
shows $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$
⟨*proof*⟩

lemma *ereal-add-cancel-left*:
fixes a b :: *ereal*
assumes $a \neq -\infty$
shows $a + b = a + c \longleftrightarrow a = \infty \vee b = c$
⟨*proof*⟩

lemma *ereal-add-cancel-right*:
fixes a b :: *ereal*
assumes $a \neq -\infty$
shows $b + a = c + a \longleftrightarrow a = \infty \vee b = c$
⟨*proof*⟩

lemma *ereal-real*: *ereal* (*real-of-ereal* x) = (*if* $|x| = \infty$ *then* 0 *else* x)
⟨*proof*⟩

lemma *real-of-ereal-add*:
fixes a b :: *ereal*
shows *real-of-ereal* $(a + b) =$
(*if* $(|a| = \infty) \wedge (|b| = \infty) \vee (|a| \neq \infty) \wedge (|b| \neq \infty)$ *then real-of-ereal* a + *real-of-ereal* b *else* 0)
⟨*proof*⟩

38.1.2 Linear order on *ereal*

instantiation *ereal* :: *linorder*
begin

function *less-ereal*
where
ereal $x <$ *ereal* $y \longleftrightarrow x < y$
| $(\infty$::*ereal*$) < a \longleftrightarrow$ *False*
| $a < -(\infty$::*ereal*$) \longleftrightarrow$ *False*
| *ereal* $x < \infty \longleftrightarrow$ *True*
| $-\infty <$ *ereal* $r \longleftrightarrow$ *True*
| $-\infty < (\infty$::*ereal*$) \longleftrightarrow$ *True*
⟨*proof*⟩
termination ⟨*proof*⟩

definition $x \leq (y$::*ereal*$) \longleftrightarrow x < y \vee x = y$

lemma *ereal-infty-less*[*simp*]:
fixes x :: *ereal*
shows $x < \infty \longleftrightarrow (x \neq \infty)$
$-\infty < x \longleftrightarrow (x \neq -\infty)$
⟨*proof*⟩

lemma *ereal-infty-less-eq*[*simp*]:
fixes x :: *ereal*
shows $\infty \leq x \longleftrightarrow x = \infty$
and $x \leq -\infty \longleftrightarrow x = -\infty$
⟨*proof*⟩

lemma *ereal-less*[*simp*]:
ereal $r < 0 \longleftrightarrow (r < 0)$
$0 <$ *ereal* $r \longleftrightarrow (0 < r)$
ereal $r < 1 \longleftrightarrow (r < 1)$
$1 <$ *ereal* $r \longleftrightarrow (1 < r)$
$0 < (\infty :: ereal)$
$-(\infty :: ereal) < 0$
⟨*proof*⟩

lemma *ereal-less-eq*[*simp*]:
$x \leq (\infty :: ereal)$
$-(\infty :: ereal) \leq x$
ereal $r \leq$ *ereal* $p \longleftrightarrow r \leq p$
ereal $r \leq 0 \longleftrightarrow r \leq 0$
$0 \leq$ *ereal* $r \longleftrightarrow 0 \leq r$
ereal $r \leq 1 \longleftrightarrow r \leq 1$
$1 \leq$ *ereal* $r \longleftrightarrow 1 \leq r$
⟨*proof*⟩

lemma *ereal-infty-less-eq2*:
$a \leq b \Longrightarrow a = \infty \Longrightarrow b = (\infty :: ereal)$
$a \leq b \Longrightarrow b = -\infty \Longrightarrow a = -(\infty :: ereal)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *ereal-dense2*: $x < y \Longrightarrow \exists z.\ x <$ *ereal* $z \wedge$ *ereal* $z < y$
⟨*proof*⟩

instance *ereal* :: *dense-linorder*
⟨*proof*⟩

instance *ereal* :: *ordered-comm-monoid-add*
⟨*proof*⟩

lemma *ereal-one-not-less-zero-ereal*[*simp*]: $\neg\ 1 < (0 :: ereal)$
⟨*proof*⟩

lemma *real-of-ereal-positive-mono*:

fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow y \neq \infty \Longrightarrow$ *real-of-ereal* $x \leq$ *real-of-ereal* y
⟨*proof*⟩

lemma *ereal-MInfty-lessI*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq -\infty \Longrightarrow -\infty < a$
⟨*proof*⟩

lemma *ereal-less-PInfty*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq \infty \Longrightarrow a < \infty$
⟨*proof*⟩

lemma *ereal-less-ereal-Ex*:
fixes $a\ b$ :: *ereal*
shows $x <$ *ereal* $r \longleftrightarrow x = -\infty \vee (\exists\, p.\ p < r \wedge x =$ *ereal* $p)$
⟨*proof*⟩

lemma *less-PInf-Ex-of-nat*: $x \neq \infty \longleftrightarrow (\exists\, n{::}nat.\ x <$ *ereal* $($*real* $n))$
⟨*proof*⟩

lemma *ereal-add-strict-mono2*:
fixes $a\ b\ c\ d$ :: *ereal*
assumes $a < b$ **and** $c < d$
shows $a + c < b + d$
⟨*proof*⟩

lemma *ereal-minus-le-minus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $-\ a \leq -\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

lemma *ereal-minus-less-minus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $-\ a < -\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *ereal-le-real-iff*:
$x \leq$ *real-of-ereal* $y \longleftrightarrow (|y| \neq \infty \longrightarrow$ *ereal* $x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$
⟨*proof*⟩

lemma *real-le-ereal-iff*:
real-of-ereal $y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq$ *ereal* $x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$
⟨*proof*⟩

lemma *ereal-less-real-iff*:
$x <$ *real-of-ereal* $y \longleftrightarrow (|y| \neq \infty \longrightarrow$ *ereal* $x < y) \wedge (|y| = \infty \longrightarrow x < 0)$
⟨*proof*⟩

lemma *real-less-ereal-iff*:
real-of-ereal $y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y <$ *ereal* $x) \wedge (|y| = \infty \longrightarrow 0 < x)$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *le-ereal-less*: $a \leq$ *ereal* $x \Longrightarrow x < y \Longrightarrow a <$ *ereal* y
⟨*proof*⟩

lemma *less-ereal-le*: $a <$ *ereal* $x \Longrightarrow x \leq y \Longrightarrow a <$ *ereal* y
⟨*proof*⟩

lemma *ereal-le-le*: *ereal* $y \leq a \Longrightarrow x \leq y \Longrightarrow$ *ereal* $x \leq a$
⟨*proof*⟩

lemma *ereal-le-less*: *ereal* $y \leq a \Longrightarrow x < y \Longrightarrow$ *ereal* $x < a$
⟨*proof*⟩

lemma *ereal-less-le*: *ereal* $y < a \Longrightarrow x \leq y \Longrightarrow$ *ereal* $x < a$
⟨*proof*⟩

lemma *real-of-ereal-pos*:
fixes $x ::$ *ereal*
shows $0 \leq x \Longrightarrow 0 \leq$ *real-of-ereal* x
⟨*proof*⟩

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*]: $0 \leq x \Longrightarrow |x :: ereal| = x$
⟨*proof*⟩

lemma *abs-ereal-less0*[*simp*]: $x < 0 \Longrightarrow |x :: ereal| = -x$
⟨*proof*⟩

lemma *abs-ereal-pos*[*simp*]: $0 \leq |x :: ereal|$
⟨*proof*⟩

lemma *ereal-abs-leI*:
fixes $x\ y ::$ *ereal*
shows $[\![x \leq y; -x \leq y]\!] \Longrightarrow |x| \leq y$
⟨*proof*⟩

lemma *ereal-abs-add*:
fixes *a b*::*ereal*

shows $abs(a+b) \leq abs\ a + abs\ b$
⟨*proof*⟩

lemma *real-of-ereal-le-0*[*simp*]: $real\text{-}of\text{-}ereal\ (x :: ereal) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$
⟨*proof*⟩

lemma *abs-real-of-ereal*[*simp*]: $|real\text{-}of\text{-}ereal\ (x :: ereal)| = real\text{-}of\text{-}ereal\ |x|$
⟨*proof*⟩

lemma *zero-less-real-of-ereal*:
fixes $x :: ereal$
shows $0 < real\text{-}of\text{-}ereal\ x \longleftrightarrow 0 < x \wedge x \neq \infty$
⟨*proof*⟩

lemma *ereal-0-le-uminus-iff*[*simp*]:
fixes $a :: ereal$
shows $0 \leq - a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *ereal-uminus-le-0-iff*[*simp*]:
fixes $a :: ereal$
shows $- a \leq 0 \longleftrightarrow 0 \leq a$
⟨*proof*⟩

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 $a + c < b + d$
⟨*proof*⟩

lemma *ereal-less-add*:
fixes $a\ b\ c :: ereal$
shows $|a| \neq \infty \Longrightarrow c < b \Longrightarrow a + c < a + b$
⟨*proof*⟩

lemma *ereal-uminus-eq-reorder*: $- a = b \longleftrightarrow a = (-b::ereal)$
⟨*proof*⟩

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** $a::ereal$
⟨*proof*⟩

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*⟩

lemma *ereal-top*:
fixes x :: *ereal*
assumes $\bigwedge B.\ x \geq ereal\ B$
shows $x = \infty$
⟨*proof*⟩

lemma
shows *ereal-max*[*simp*]: $ereal\ (max\ x\ y) = max\ (ereal\ x)\ (ereal\ y)$
and *ereal-min*[*simp*]: $ereal\ (min\ x\ y) = min\ (ereal\ x)\ (ereal\ y)$
⟨*proof*⟩

lemma *ereal-max-0*: $max\ 0\ (ereal\ r) = ereal\ (max\ 0\ r)$
⟨*proof*⟩

lemma
fixes f :: $nat \Rightarrow ereal$
shows *ereal-incseq-uminus*[*simp*]: $incseq\ (\lambda x.\ -f\ x) \longleftrightarrow decseq\ f$
and *ereal-decseq-uminus*[*simp*]: $decseq\ (\lambda x.\ -f\ x) \longleftrightarrow incseq\ f$
⟨*proof*⟩

lemma *incseq-ereal*: $incseq\ f \Longrightarrow incseq\ (\lambda x.\ ereal\ (f\ x))$
⟨*proof*⟩

lemma *sum-ereal*[*simp*]: $(\sum x{\in}A.\ ereal\ (f\ x)) = ereal\ (\sum x{\in}A.\ f\ x)$
⟨*proof*⟩

lemma *sum-list-ereal* [*simp*]: $sum\text{-}list\ (map\ (\lambda x.\ ereal\ (f\ x))\ xs) = ereal\ (sum\text{-}list\ (map\ f\ xs))$
⟨*proof*⟩

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*⟩

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*⟩

lemma *sum-real-of-ereal*:
fixes f :: $'i \Rightarrow ereal$
assumes $\bigwedge x.\ x \in S \Longrightarrow |f\ x| \neq \infty$

shows $(\sum x{\in}S.\ \textit{real-of-ereal}\ (f\ x)) = \textit{real-of-ereal}\ (\textit{sum}\ f\ S)$
⟨*proof*⟩

38.1.3 Multiplication

instantiation *ereal* :: {*comm-monoid-mult*,*sgn*}
begin

function *sgn-ereal* :: *ereal* ⇒ *ereal* **where**
  $\textit{sgn}\ (\textit{ereal}\ r) = \textit{ereal}\ (\textit{sgn}\ r)$
| $\textit{sgn}\ (\infty{::}\textit{ereal}) = 1$
| $\textit{sgn}\ (-\infty{::}\textit{ereal}) = -1$
⟨*proof*⟩
termination ⟨*proof*⟩

function *times-ereal* **where**
  $\textit{ereal}\ r * \textit{ereal}\ p = \textit{ereal}\ (r * p)$
| $\textit{ereal}\ r * \infty = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ \infty\ \textit{else}\ -\infty)$
| $\infty * \textit{ereal}\ r = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ \infty\ \textit{else}\ -\infty)$
| $\textit{ereal}\ r * -\infty = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ -\infty\ \textit{else}\ \infty)$
| $-\infty * \textit{ereal}\ r = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ -\infty\ \textit{else}\ \infty)$
| $(\infty{::}\textit{ereal}) * \infty = \infty$
| $-(\infty{::}\textit{ereal}) * \infty = -\infty$
| $(\infty{::}\textit{ereal}) * -\infty = -\infty$
| $-(\infty{::}\textit{ereal}) * -\infty = \infty$
⟨*proof*⟩
termination ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma [*simp*]:
  shows *ereal-1-times*: $\textit{ereal}\ 1 * x = x$
  and *times-ereal-1*: $x * \textit{ereal}\ 1 = x$
⟨*proof*⟩

lemma *one-not-le-zero-ereal*[*simp*]: ¬ $(1 \le (0{::}\textit{ereal}))$
  ⟨*proof*⟩

lemma *real-ereal-1*[*simp*]: $\textit{real-of-ereal}\ (1{::}\textit{ereal}) = 1$
  ⟨*proof*⟩

lemma *real-of-ereal-le-1*:
  fixes *a* :: *ereal*
  shows $a \le 1 \Longrightarrow \textit{real-of-ereal}\ a \le 1$
  ⟨*proof*⟩

lemma *abs-ereal-one*[*simp*]: $|1| = (1::ereal)$
⟨*proof*⟩

lemma *ereal-mult-zero*[*simp*]:
fixes $a :: ereal$
shows $a * 0 = 0$
⟨*proof*⟩

lemma *ereal-zero-mult*[*simp*]:
fixes $a :: ereal$
shows $0 * a = 0$
⟨*proof*⟩

lemma *ereal-m1-less-0*[*simp*]: $-(1::ereal) < 0$
⟨*proof*⟩

lemma *ereal-times*[*simp*]:
$1 \neq (\infty::ereal)$ $(\infty::ereal) \neq 1$
$1 \neq -(\infty::ereal)$ $-(\infty::ereal) \neq 1$
⟨*proof*⟩

lemma *ereal-plus-1*[*simp*]:
$1 + ereal\ r = ereal\ (r + 1)$
$ereal\ r + 1 = ereal\ (r + 1)$
$1 + -(\infty::ereal) = -\infty$
$-(\infty::ereal) + 1 = -\infty$
⟨*proof*⟩

lemma *ereal-zero-times*[*simp*]:
fixes $a\ b :: ereal$
shows $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$
⟨*proof*⟩

lemma *ereal-mult-eq-PInfty*[*simp*]:
$a * b = (\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b > 0) \vee (a > 0 \wedge b = \infty) \vee (a = -\infty \wedge b < 0) \vee (a < 0 \wedge b = -\infty)$
⟨*proof*⟩

lemma *ereal-mult-eq-MInfty*[*simp*]:
$a * b = -(\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b < 0) \vee (a < 0 \wedge b = \infty) \vee (a = -\infty \wedge b > 0) \vee (a > 0 \wedge b = -\infty)$
⟨*proof*⟩

lemma *ereal-abs-mult*: $|x * y :: ereal| = |x| * |y|$
⟨*proof*⟩

lemma *ereal-0-less-1*[*simp*]: $0 < (1::ereal)$

⟨*proof*⟩

lemma *ereal-mult-minus-left*[*simp*]:
fixes *a b* :: *ereal*
shows $-a * b = -(a * b)$
⟨*proof*⟩

lemma *ereal-mult-minus-right*[*simp*]:
fixes *a b* :: *ereal*
shows $a * -b = -(a * b)$
⟨*proof*⟩

lemma *ereal-mult-infty*[*simp*]:
$a * (\infty::ereal) = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
⟨*proof*⟩

lemma *ereal-infty-mult*[*simp*]:
$(\infty::ereal) * a = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
⟨*proof*⟩

lemma *ereal-mult-strict-right-mono*:
assumes $a < b$
and $0 < c$
and $c < (\infty::ereal)$
shows $a * c < b * c$
⟨*proof*⟩

lemma *ereal-mult-strict-left-mono*:
$a < b \Longrightarrow 0 < c \Longrightarrow c < (\infty::ereal) \Longrightarrow c * a < c * b$
⟨*proof*⟩

lemma *ereal-mult-right-mono*:
fixes *a b c* :: *ereal*
assumes $a \leq b\ 0 \leq c$
shows $a * c \leq b * c$
⟨*proof*⟩

lemma *ereal-mult-left-mono*:
fixes *a b c* :: *ereal*
shows $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
⟨*proof*⟩

lemma *ereal-mult-mono*:
fixes *a b c d*::*ereal*
assumes $b \geq 0\ c \geq 0\ a \leq b\ c \leq d$
shows $a * c \leq b * d$
⟨*proof*⟩

lemma *ereal-mult-mono'*:

fixes $a\ b\ c\ d$::*ereal*
assumes $a \geq 0\ c \geq 0\ a \leq b\ c \leq d$
shows $a * c \leq b * d$
⟨*proof*⟩

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*⟩

lemma *ereal-mult-mono-strict'*:
fixes $a\ b\ c\ d$::*ereal*
assumes $a > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
⟨*proof*⟩

lemma *zero-less-one-ereal*[*simp*]: $0 \leq (1$::*ereal*$)$
⟨*proof*⟩

lemma *ereal-0-le-mult*[*simp*]: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * (b :: ereal)$
⟨*proof*⟩

lemma *ereal-right-distrib*:
fixes $r\ a\ b :: ereal$
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow r * (a + b) = r * a + r * b$
⟨*proof*⟩

lemma *ereal-left-distrib*:
fixes $r\ a\ b :: ereal$
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow (a + b) * r = a * r + b * r$
⟨*proof*⟩

lemma *ereal-mult-le-0-iff*:
fixes $a\ b :: ereal$
shows $a * b \leq 0 \longleftrightarrow (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$
⟨*proof*⟩

lemma *ereal-zero-le-0-iff*:
fixes $a\ b :: ereal$
shows $0 \leq a * b \longleftrightarrow (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$
⟨*proof*⟩

lemma *ereal-mult-less-0-iff*:
fixes $a\ b :: ereal$
shows $a * b < 0 \longleftrightarrow (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$
⟨*proof*⟩

lemma *ereal-zero-less-0-iff*:

fixes $a\ b :: ereal$
shows $0 < a * b \longleftrightarrow (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$
⟨*proof*⟩

lemma *ereal-left-mult-cong*:
fixes $a\ b\ c :: ereal$
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow a * c = b * d$
⟨*proof*⟩

lemma *ereal-right-mult-cong*:
fixes $a\ b\ c :: ereal$
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow c * a = d * b$
⟨*proof*⟩

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 $(a + b) * c = a * c + b * c$
⟨*proof*⟩

lemma *numeral-eq-ereal* [*simp*]: $numeral\ w = ereal\ (numeral\ w)$
⟨*proof*⟩

lemma *m1-ereal-less-iff* [*simp*]:
$((-1::ereal) < numeral\ a) \longleftrightarrow ((-1::real) < numeral\ a)$
⟨*proof*⟩

lemma *m1-ereal-le-iff* [*simp*]:
$((-1::ereal) \leq numeral\ a) \longleftrightarrow ((-1::real) \leq numeral\ a)$
⟨*proof*⟩

lemma *m1-ereal-eq-iff* [*simp*]:
$((-1::ereal) = numeral\ a) \longleftrightarrow ((-1::real) = numeral\ a)$
⟨*proof*⟩

lemma *ereal-less-m1-iff* [*simp*]:
$(numeral\ a < (-1::ereal)) \longleftrightarrow (numeral\ a < (-1::real))$
⟨*proof*⟩

lemma *ereal-le-m1-iff* [*simp*]:
$(numeral\ a \leq (-1::ereal)) \longleftrightarrow (numeral\ a \leq (-1::real))$
⟨*proof*⟩

lemma *ereal-eq-m1-iff* [*simp*]:
$(numeral\ a = (-1::ereal)) \longleftrightarrow (numeral\ a = (-1::real))$
⟨*proof*⟩

lemma *distrib-left-ereal-nn*:
$c \geq 0 \Longrightarrow (x + y) * ereal\ c = x * ereal\ c + y * ereal\ c$
⟨*proof*⟩

lemma *sum-ereal-right-distrib*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow r * sum\ f\ A = (\sum n{\in}A.\ r * f\ n)$
⟨*proof*⟩

lemma *sum-ereal-left-distrib*:
$(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow sum\ f\ A * r = (\sum n{\in}A.\ f\ n * r :: ereal)$
⟨*proof*⟩

lemma *sum-distrib-right-ereal*:
$c \geq 0 \Longrightarrow sum\ f\ A * ereal\ c = (\sum x{\in}A.\ f\ x * c :: ereal)$
⟨*proof*⟩

lemma *ereal-le-epsilon*:
fixes $x\ y :: ereal$
assumes $\bigwedge e.\ 0 < e \Longrightarrow x \leq y + e$
shows $x \leq y$
⟨*proof*⟩

lemma *ereal-le-epsilon2*:
fixes $x\ y :: ereal$
assumes $\bigwedge e{::}real.\ 0 < e \Longrightarrow x \leq y + ereal\ e$
shows $x \leq y$
⟨*proof*⟩

lemma *ereal-le-real*:
fixes $x\ y :: ereal$
assumes $\bigwedge z.\ x \leq ereal\ z \Longrightarrow y \leq ereal\ z$
shows $y \leq x$
⟨*proof*⟩

lemma *prod-ereal-0*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\prod i{\in}A.\ f\ i) = 0 \longleftrightarrow finite\ A \wedge (\exists\, i{\in}A.\ f\ i = 0)$
⟨*proof*⟩

lemma *prod-ereal-pos*:
fixes $f :: 'a \Rightarrow ereal$
assumes $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$
shows $0 \leq (\prod i{\in}I.\ f\ i)$
⟨*proof*⟩

lemma *prod-PInf*:
fixes $f :: 'a \Rightarrow ereal$
assumes $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$

shows $(\prod i{\in}I.\ f\ i) = \infty \longleftrightarrow \mathit{finite}\ I \wedge (\exists\, i{\in}I.\ f\ i = \infty) \wedge (\forall\, i{\in}I.\ f\ i \neq 0)$
⟨*proof*⟩

lemma *prod-ereal*: $(\prod i{\in}A.\ \mathit{ereal}\ (f\ i)) = \mathit{ereal}\ (\mathit{prod}\ f\ A)$
⟨*proof*⟩

38.1.4 Power

lemma *ereal-power*[*simp*]: $(\mathit{ereal}\ x)\ \hat{}\ n = \mathit{ereal}\ (x\hat{}n)$
⟨*proof*⟩

lemma *ereal-power-PInf*[*simp*]: $(\infty{::}\mathit{ereal})\ \hat{}\ n = (\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ \infty)$
⟨*proof*⟩

lemma *ereal-power-uminus*[*simp*]:
fixes $x :: \mathit{ereal}$
shows $(-\ x)\ \hat{}\ n = (\mathit{if}\ \mathit{even}\ n\ \mathit{then}\ x\ \hat{}\ n\ \mathit{else}\ -\ (x\hat{}n))$
⟨*proof*⟩

lemma *ereal-power-numeral*[*simp*]:
$(\mathit{numeral}\ \mathit{num} :: \mathit{ereal})\ \hat{}\ n = \mathit{ereal}\ (\mathit{numeral}\ \mathit{num}\ \hat{}\ n)$
⟨*proof*⟩

lemma *zero-le-power-ereal*[*simp*]:
fixes $a :: \mathit{ereal}$
assumes $0 \leq a$
shows $0 \leq a\ \hat{}\ n$
⟨*proof*⟩

38.1.5 Subtraction

lemma *ereal-minus-minus-image*[*simp*]:
fixes $S :: \mathit{ereal}\ \mathit{set}$
shows $\mathit{uminus}\ `\ \mathit{uminus}\ `\ S = S$
⟨*proof*⟩

lemma *ereal-uminus-lessThan*[*simp*]:
fixes $a :: \mathit{ereal}$
shows $\mathit{uminus}\ `\ \{..{<}a\} = \{-a{<}..\}$
⟨*proof*⟩

lemma *ereal-uminus-greaterThan*[*simp*]: $\mathit{uminus}\ `\ \{(a{::}\mathit{ereal}){<}..\} = \{..{<}-a\}$
⟨*proof*⟩

instantiation *ereal* :: *minus*
begin

definition $x - y = x + -(y{::}\mathit{ereal})$
instance ⟨*proof*⟩

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
⟨*proof*⟩

lemma *ereal-x-minus-x*[*simp*]: *x − x = (if |x| = ∞ then ∞ else 0::ereal)*
⟨*proof*⟩

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 = −∞)
⟨*proof*⟩

lemma *ereal-eq-minus*:
fixes *x y z* :: *ereal*
shows *|y| ≠ ∞ ⟹ x = z − y ⟷ x + y = z*
⟨*proof*⟩

lemma *ereal-less-minus-iff*:
fixes *x y z* :: *ereal*
shows *x < z − y ⟷*
(y = ∞ ⟶ z = ∞ ∧ x ≠ ∞) ∧
(y = −∞ ⟶ x ≠ ∞) ∧
(|y| ≠ ∞⟶ x + y < z)
⟨*proof*⟩

lemma *ereal-less-minus*:
fixes *x y z* :: *ereal*
shows *|y| ≠ ∞ ⟹ x < z − y ⟷ x + y < z*
⟨*proof*⟩

lemma *ereal-le-minus-iff*:
fixes *x y z* :: *ereal*
shows *x ≤ z − y ⟷ (y = ∞ ⟶ z ≠ ∞ ⟶ x = −∞) ∧ (|y| ≠ ∞ ⟶ x + y ≤ z)*
⟨*proof*⟩

lemma *ereal-le-minus*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x \leq z - y \longleftrightarrow x + y \leq z$
⟨*proof*⟩

lemma *ereal-minus-less-iff*:
fixes $x\ y\ z :: ereal$
shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$
⟨*proof*⟩

lemma *ereal-minus-less*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x - y < z \longleftrightarrow x < z + y$
⟨*proof*⟩

lemma *ereal-minus-le-iff*:
fixes $x\ y\ z :: ereal$
shows $x - y \leq z \longleftrightarrow$
$(y = -\infty \longrightarrow z = \infty) \wedge$
$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$
$(|y| \neq \infty \longrightarrow x \leq z + y)$
⟨*proof*⟩

lemma *ereal-minus-le*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x - y \leq z \longleftrightarrow x \leq z + y$
⟨*proof*⟩

lemma *ereal-minus-eq-minus-iff*:
fixes $a\ b\ c :: ereal$
shows $a - b = a - c \longleftrightarrow$
$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$
⟨*proof*⟩

lemma *ereal-add-le-add-iff*:
fixes $a\ b\ c :: ereal$
shows $c + a \leq c + b \longleftrightarrow$
$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
⟨*proof*⟩

lemma *ereal-add-le-add-iff2*:
fixes $a\ b\ c :: ereal$
shows $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
⟨*proof*⟩

lemma *ereal-mult-le-mult-iff*:
fixes $a\ b\ c :: ereal$
shows $|c| \neq \infty \Longrightarrow c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$

⟨*proof*⟩

lemma *ereal-minus-mono*:
fixes $A\ B\ C\ D$ :: *ereal* **assumes** $A \leq B\ D \leq C$
shows $A - C \leq B - D$
⟨*proof*⟩

lemma *ereal-mono-minus-cancel*:
fixes $a\ b\ c$ :: *ereal*
shows $c - a \leq c - b \Longrightarrow 0 \leq c \Longrightarrow c < \infty \Longrightarrow b \leq a$
⟨*proof*⟩

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)
⟨*proof*⟩

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*$)$
⟨*proof*⟩

lemma *ereal-diff-positive*:
fixes $a\ b$ :: *ereal* **shows** $a \leq b \Longrightarrow 0 \leq b - a$
⟨*proof*⟩

lemma *ereal-between*:
fixes $x\ e$ :: *ereal*
assumes $|x| \neq \infty$ **and** $0 < e$
shows $x - e < x$
and $x < x + e$
⟨*proof*⟩

lemma *ereal-minus-eq-PInfty-iff*:
fixes $x\ y$ :: *ereal*
shows $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$
⟨*proof*⟩

lemma *ereal-diff-add-eq-diff-diff-swap*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - (y + z) = x - y - z$
⟨*proof*⟩

lemma *ereal-diff-add-assoc2*:
fixes $x\ y\ z$ :: *ereal*
shows $x + y - z = x - z + y$
⟨*proof*⟩

lemma *ereal-add-uminus-conv-diff*: **fixes** $x\ y\ z$ :: *ereal* **shows** $-x + y = y - x$

⟨*proof*⟩

lemma *ereal-minus-diff-eq*:
fixes *x y* :: *ereal*
shows ⟦ $x = \infty \longrightarrow y \neq \infty$; $x = -\infty \longrightarrow y \neq -\infty$ ⟧ $\Longrightarrow -(x - y) = y - x$
⟨*proof*⟩

lemma *ediff-le-self* [*simp*]: $x - y \leq (x :: enat)$
⟨*proof*⟩

lemma *ereal-abs-diff*:
fixes *a b*::*ereal*
shows $abs(a-b) \leq abs\ a + abs\ b$
⟨*proof*⟩

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* (−∞::*ereal*) = *0*
⟨*proof*⟩
termination ⟨*proof*⟩

definition *x div y* = *x* ∗ *inverse* (*y* :: *ereal*)

instance ⟨*proof*⟩

end

lemma *real-of-ereal-inverse*[*simp*]:
fixes *a* :: *ereal*
shows *real-of-ereal* (*inverse a*) = *1* / *real-of-ereal a*
⟨*proof*⟩

lemma *ereal-inverse*[*simp*]:
inverse (*0*::*ereal*) = ∞
inverse (*1*::*ereal*) = *1*
⟨*proof*⟩

lemma *ereal-divide*[*simp*]:
ereal r / *ereal p* = (*if p* = *0 then ereal r* ∗ ∞ *else ereal* (*r* / *p*))
⟨*proof*⟩

lemma *ereal-divide-same*[*simp*]:
fixes *x* :: *ereal*

shows $x \,/\, x = (\mathit{if}\ |x| = \infty \vee x = 0\ \mathit{then}\ 0\ \mathit{else}\ 1)$
⟨*proof*⟩

lemma *ereal-inv-inv*[*simp*]:
fixes $x :: \mathit{ereal}$
shows $\mathit{inverse}\ (\mathit{inverse}\ x) = (\mathit{if}\ x \neq -\infty\ \mathit{then}\ x\ \mathit{else}\ \infty)$
⟨*proof*⟩

lemma *ereal-inverse-minus*[*simp*]:
fixes $x :: \mathit{ereal}$
shows $\mathit{inverse}\ (-x) = (\mathit{if}\ x = 0\ \mathit{then}\ \infty\ \mathit{else}\ -\mathit{inverse}\ x)$
⟨*proof*⟩

lemma *ereal-uminus-divide*[*simp*]:
fixes $x\ y :: \mathit{ereal}$
shows $-x \,/\, y = -(x \,/\, y)$
⟨*proof*⟩

lemma *ereal-divide-Infty*[*simp*]:
fixes $x :: \mathit{ereal}$
shows $x \,/\, \infty = 0\ x \,/\, -\infty = 0$
⟨*proof*⟩

lemma *ereal-divide-one*[*simp*]: $x \,/\, 1 = (x{::}\mathit{ereal})$
⟨*proof*⟩

lemma *ereal-divide-ereal*[*simp*]: $\infty \,/\, \mathit{ereal}\ r = (\mathit{if}\ 0 \leq r\ \mathit{then}\ \infty\ \mathit{else}\ -\infty)$
⟨*proof*⟩

lemma *ereal-inverse-nonneg-iff*: $0 \leq \mathit{inverse}\ (x :: \mathit{ereal}) \longleftrightarrow 0 \leq x \vee x = -\infty$
⟨*proof*⟩

lemma *inverse-ereal-ge0I*: $0 \leq (x :: \mathit{ereal}) \Longrightarrow 0 \leq \mathit{inverse}\ x$
⟨*proof*⟩

lemma *zero-le-divide-ereal*[*simp*]:
fixes $a :: \mathit{ereal}$
assumes $0 \leq a$ **and** $0 \leq b$
shows $0 \leq a \,/\, b$
⟨*proof*⟩

lemma *ereal-le-divide-pos*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y \leq z \,/\, x \longleftrightarrow x * y \leq z$
⟨*proof*⟩

lemma *ereal-divide-le-pos*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow z \,/\, x \leq y \longleftrightarrow z \leq x * y$

⟨*proof*⟩

lemma *ereal-le-divide-neg*:
fixes $x\ y\ z$ :: *ereal*
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow y \leq z\ /\ x \longleftrightarrow z \leq x * y$
⟨*proof*⟩

lemma *ereal-divide-le-neg*:
fixes $x\ y\ z$ :: *ereal*
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow z\ /\ x \leq y \longleftrightarrow x * y \leq z$
⟨*proof*⟩

lemma *ereal-inverse-antimono-strict*:
fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x < y \Longrightarrow inverse\ y < inverse\ x$
⟨*proof*⟩

lemma *ereal-inverse-antimono*:
fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow inverse\ y \leq inverse\ x$
⟨*proof*⟩

lemma *inverse-inverse-Pinfty-iff*[*simp*]:
fixes x :: *ereal*
shows $inverse\ x = \infty \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *ereal-inverse-eq-0*:
fixes x :: *ereal*
shows $inverse\ x = 0 \longleftrightarrow x = \infty \vee x = -\infty$
⟨*proof*⟩

lemma *ereal-0-gt-inverse*:
fixes x :: *ereal*
shows $0 < inverse\ x \longleftrightarrow x \neq \infty \wedge 0 \leq x$
⟨*proof*⟩

lemma *ereal-inverse-le-0-iff*:
fixes x :: *ereal*
shows $inverse\ x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$
⟨*proof*⟩

lemma *ereal-divide-eq-0-iff*: $x\ /\ y = 0 \longleftrightarrow x = 0 \vee |y :: ereal| = \infty$
⟨*proof*⟩

lemma *ereal-mult-less-right*:
fixes $a\ b\ c$ :: *ereal*
assumes $b * a < c * a\ 0 < a\ a < \infty$
shows $b < c$

⟨*proof*⟩

lemma *ereal-mult-divide*:
fixes *a b* :: *ereal*
shows $0 < b \Longrightarrow b < \infty \Longrightarrow b * (a \,/\, b) = a$
⟨*proof*⟩

lemma *ereal-power-divide*:
fixes *x y* :: *ereal*
shows $y \neq 0 \Longrightarrow (x \,/\, y) \,\hat{}\, n = x\hat{}n \,/\, y\hat{}n$
⟨*proof*⟩

lemma *divide-pos-0-ereal* [*simp*]: $x > 0 \Longrightarrow x \,/\, (0 :: ereal) = \infty$
⟨*proof*⟩

lemma *divide-pos-0-ereal'* [*simp*]: $x > 0 \Longrightarrow x \,/\, (ereal\ 0) = \infty$
⟨*proof*⟩

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*⟩

lemma *ereal-divide-right-mono*[*simp*]:
fixes *x y z* :: *ereal*
assumes $x \leq y$
and $0 < z$
shows $x \,/\, z \leq y \,/\, z$
⟨*proof*⟩

lemma *ereal-divide-left-mono*[*simp*]:
fixes *x y z* :: *ereal*
assumes $y \leq x$
and $0 < z$
and $0 < x * y$
shows $z \,/\, x \leq z \,/\, y$
⟨*proof*⟩

lemma *ereal-divide-zero-left*[*simp*]:
fixes *a* :: *ereal*
shows $0 \,/\, a = 0$
⟨*proof*⟩

lemma *ereal-times-divide-eq-left*[*simp*]:
fixes *a b c* :: *ereal*
shows $b \,/\, c * a = b * a \,/\, c$
⟨*proof*⟩

lemma *ereal-times-divide-eq*: $a * (b \,/\, c :: ereal) = a * b \,/\, c$
⟨*proof*⟩

lemma *ereal-inverse-real* [*simp*]: $|z| \neq \infty \Longrightarrow z \neq 0 \Longrightarrow ereal\ (inverse\ (real\text{-}of\text{-}ereal\ z)) = inverse\ z$
⟨*proof*⟩

lemma *ereal-inverse-mult*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ (a * (b::ereal)) = inverse\ a * inverse\ b$
⟨*proof*⟩

lemma *inverse-eq-infinity-iff-eq-zero* [*simp*]:
$1/(x::ereal) = \infty \longleftrightarrow x = 0$
⟨*proof*⟩

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 $c * (a + b) = c * a + c * b$
⟨*proof*⟩

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 $c * (a - b) = c * a - c * b$
⟨*proof*⟩

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 $(a - b) * c = a * c - b * c$
⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *ereal* :: *complete-lattice*
begin

definition *bot* = (−∞::*ereal*)
definition *top* = (∞::*ereal*)

definition *Sup S* = (*SOME x* :: *ereal*. (∀ *y*∈*S*. *y* ≤ *x*) ∧ (∀ *z*. (∀ *y*∈*S*. *y* ≤ *z*) ⟶ *x* ≤ *z*))
definition *Inf S* = (*SOME x* :: *ereal*. (∀ *y*∈*S*. *x* ≤ *y*) ∧ (∀ *z*. (∀ *y*∈*S*. *z* ≤ *y*) ⟶ *z* ≤ *x*))

lemma *ereal-complete-Sup*:
fixes *S* :: *ereal set*
shows ∃ *x*. (∀ *y*∈*S*. *y* ≤ *x*) ∧ (∀ *z*. (∀ *y*∈*S*. *y* ≤ *z*) ⟶ *x* ≤ *z*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ereal-complete-Inf*:
∃ *x*. (∀ *y*∈*S*::*ereal set*. *x* ≤ *y*) ∧ (∀ *z*. (∀ *y*∈*S*. *z* ≤ *y*) ⟶ *z* ≤ *x*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *ereal* :: *complete-linorder* ⟨*proof*⟩

instance *ereal* :: *linear-continuum*
⟨*proof*⟩

lemma *min-PInf* [*simp*]: *min* (∞::*ereal*) *x* = *x*
⟨*proof*⟩

lemma *min-PInf2* [*simp*]: *min x* (∞::*ereal*) = *x*
⟨*proof*⟩

lemma *max-PInf* [*simp*]: *max* (∞::*ereal*) *x* = ∞
⟨*proof*⟩

lemma *max-PInf2* [*simp*]: *max x* (∞::*ereal*) = ∞
⟨*proof*⟩

lemma *min-MInf* [*simp*]: *min* (−∞::*ereal*) *x* = −∞
⟨*proof*⟩

lemma *min-MInf2* [*simp*]: *min x* (−∞::*ereal*) = −∞
⟨*proof*⟩

lemma *max-MInf* [*simp*]: *max* (−∞::*ereal*) *x* = *x*
⟨*proof*⟩

lemma *max-MInf2* [*simp*]: *max x* (−∞::*ereal*) = *x*
⟨*proof*⟩

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*<..})
⟨*proof*⟩

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*⟩

lemma *real-greaterThanLessThan-inter*:
real-of-ereal ' {*N*<..<*M*::*ereal*} = *real-of-ereal* ' {−∞<..<*M*} ∩ *real-of-ereal* ' {*N*<..<∞}
⟨*proof*⟩

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*⟩

lemma *real-image-ereal-ivl*:
fixes *a b*::*ereal*
shows
real-of-ereal ' {*a*<..<*b*} =
(*if a* < *b then* (*if a* = −∞ *then if b* = ∞ *then UNIV else* {..<*real-of-ereal b*}
else if b = ∞ *then* {*real-of-ereal a*<..} *else* {*real-of-ereal a* <..< *real-of-ereal b*})
else {})
⟨*proof*⟩

lemma fixes *a b c*::*ereal*
shows *not-inftyI*: $a < b \Longrightarrow b < c \Longrightarrow abs\ b \neq \infty$
⟨*proof*⟩

context
fixes *r s t*::*real*
begin

lemma *interval-Ioo-neq-Ioi*: $\{r<..<s\} \neq \{t<..\}$
⟨*proof*⟩

lemma *interval-Ioo-neq-Iio*: $\{r<..<s\} \neq \{..<t\}$
⟨*proof*⟩

lemma *interval-neq-ioo-UNIV*: $\{r<..<s\} \neq UNIV$
and *interval-Ioi-neq-UNIV*: $\{r<..\} \neq UNIV$
and *interval-Iio-neq-UNIV*: $\{..<r\} \neq UNIV$
⟨*proof*⟩

lemma *interval-Ioi-neq-Iio*: $\{r<..\} \neq \{..<s\}$
⟨*proof*⟩

lemma *interval-empty-neq-Ioi*: $\{\} \neq \{r<..\}$
and *interval-empty-neq-Iio*: $\{\} \neq \{..<r\}$
⟨*proof*⟩

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 $(\{r<..<s\} = \{t<..<u\}) = (r \geq s \wedge u \leq t \vee r = t \wedge s = u)$
⟨*proof*⟩

lemma *real-of-ereal-image-greaterThanLessThan-iff*:
real-of-ereal ‘ $\{a <..< b\}$ = *real-of-ereal* ‘ $\{c <..< d\} \longleftrightarrow (a \geq b \wedge c \geq d \vee a = c \wedge b = d)$
⟨*proof*⟩

lemma *uminus-image-real-of-ereal-image-greaterThanLessThan*:
uminus ‘ *real-of-ereal* ‘ $\{l <..< u\}$ = *real-of-ereal* ‘ $\{-u <..< -l\}$
⟨*proof*⟩

lemma *add-image-real-of-ereal-image-greaterThanLessThan*:

(+) c ‘ *real-of-ereal* ‘ $\{l <..< u\}$ = *real-of-ereal* ‘ $\{c + l <..< c + u\}$
⟨*proof*⟩

lemma *add2-image-real-of-ereal-image-greaterThanLessThan*:
$(\lambda x.\ x + c)$ ‘ *real-of-ereal* ‘ $\{l <..< u\}$ = *real-of-ereal* ‘ $\{l + c <..< u + c\}$
⟨*proof*⟩

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*⟩

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*⟩

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 $a \neq -\infty$
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *continuous-on-ereal*[*continuous-intros*]:
assumes *f*: *continuous-on s f* **shows** *continuous-on s* $(\lambda x.$ *ereal* $(f\ x))$
⟨*proof*⟩

lemma *tendsto-ereal*[*tendsto-intros*, *simp*, *intro*]: $(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.$ *ereal* $(f\ x)) \longrightarrow$ *ereal* $x)\ F$
⟨*proof*⟩

lemma *tendsto-uminus-ereal*[*tendsto-intros*, *simp*, *intro*]:

assumes $(f \longrightarrow x)\ F$
shows $((\lambda x.\ -\ f\ x{::}ereal) \longrightarrow -\ x)\ F$
⟨*proof*⟩

lemma *at-infty-ereal-eq-at-top*: $at\ \infty = filtermap\ ereal\ at\text{-}top$
⟨*proof*⟩

lemma *ereal-Lim-uminus*: $(f \longrightarrow f0)\ net \longleftrightarrow ((\lambda x.\ -\ f\ x{::}ereal) \longrightarrow -\ f0)\ net$
⟨*proof*⟩

lemma *ereal-divide-less-iff*: $0 < (c{::}ereal) \Longrightarrow c < \infty \Longrightarrow a\ /\ c < b \longleftrightarrow a < b * c$
⟨*proof*⟩

lemma *ereal-less-divide-iff*: $0 < (c{::}ereal) \Longrightarrow c < \infty \Longrightarrow a < b\ /\ c \longleftrightarrow a * c < b$
⟨*proof*⟩

lemma *tendsto-cmult-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes c: $|c| \neq \infty$ **and** f: $(f \longrightarrow x)\ F$
shows $((\lambda x.\ c * f\ x{::}ereal) \longrightarrow c * x)\ F$
⟨*proof*⟩

lemma *tendsto-cmult-ereal-not-0*[*tendsto-intros*, *simp*, *intro*]:
assumes $x \neq 0$ **and** f: $(f \longrightarrow x)\ F$
shows $((\lambda x.\ c * f\ x{::}ereal) \longrightarrow c * x)\ F$
⟨*proof*⟩

lemma *tendsto-cadd-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes c: $y \neq -\infty\ x \neq -\infty$ **and** f: $(f \longrightarrow x)\ F$
shows $((\lambda x.\ f\ x + y{::}ereal) \longrightarrow x + y)\ F$
⟨*proof*⟩

lemma *tendsto-add-left-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes c: $|y| \neq \infty$ **and** f: $(f \longrightarrow x)\ F$
shows $((\lambda x.\ f\ x + y{::}ereal) \longrightarrow x + y)\ F$
⟨*proof*⟩

lemma *continuous-at-ereal*[*continuous-intros*]: $continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ ereal\ (f\ x))$
⟨*proof*⟩

lemma *ereal-Sup*:
assumes $*$: $|SUP\ a{\in}A.\ ereal\ a| \neq \infty$
shows $ereal\ (Sup\ A) = (SUP\ a{\in}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-SUP*: $|SUP\ a{\in}A.\ ereal\ (f\ a)| \neq \infty \Longrightarrow ereal\ (SUP\ a{\in}A.\ f\ a) = (SUP\ a{\in}A.\ ereal\ (f\ a))$
⟨*proof*⟩

lemma *ereal-Inf*:
assumes ∗: $|INF\ a{\in}A.\ ereal\ a| \neq \infty$
shows $ereal\ (Inf\ A) = (INF\ a{\in}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-Inf'*:
assumes ∗: *bdd-below A* $A \neq \{\}$
shows $ereal\ (Inf\ A) = (INF\ a{\in}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-INF*: $|INF\ a{\in}A.\ ereal\ (f\ a)| \neq \infty \Longrightarrow ereal\ (INF\ a{\in}A.\ f\ a) = (INF\ a{\in}A.\ ereal\ (f\ a))$
⟨*proof*⟩

lemma *ereal-Sup-uminus-image-eq*: $Sup\ (uminus\ `\ S{::}ereal\ set) = -\ Inf\ S$
⟨*proof*⟩

lemma *ereal-SUP-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(SUP\ x{\in}S.\ uminus\ (f\ x)) = -\ (INF\ x{\in}S.\ f\ x)$
⟨*proof*⟩

lemma *ereal-inj-on-uminus*[*intro*, *simp*]: *inj-on uminus* $(A :: ereal\ set)$
⟨*proof*⟩

lemma *ereal-Inf-uminus-image-eq*: $Inf\ (uminus\ `\ S{::}ereal\ set) = -\ Sup\ S$
⟨*proof*⟩

lemma *ereal-INF-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(INF\ x{\in}S.\ -\ f\ x) = -\ (SUP\ x{\in}S.\ f\ x)$
⟨*proof*⟩

lemma *ereal-SUP-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall\, a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |Sup\ (f\ `\ A)| \neq \infty$
⟨*proof*⟩

lemma *ereal-INF-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall\, a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |Inf\ (f\ `\ A)| \neq \infty$
⟨*proof*⟩

lemma *ereal-image-uminus-shift*:
fixes X Y :: *ereal set*
shows *uminus* ‘ $X = Y \longleftrightarrow X =$ *uminus* ‘ Y
⟨*proof*⟩

lemma *Sup-eq-MInfty*:
fixes S :: *ereal set*
shows *Sup* $S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$
⟨*proof*⟩

lemma *Inf-eq-PInfty*:
fixes S :: *ereal set*
shows *Inf* $S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$
⟨*proof*⟩

lemma *Inf-eq-MInfty*:
fixes S :: *ereal set*
shows $-\infty \in S \Longrightarrow$ *Inf* $S = -\infty$
⟨*proof*⟩

lemma *Sup-eq-PInfty*:
fixes S :: *ereal set*
shows $\infty \in S \Longrightarrow$ *Sup* $S = \infty$
⟨*proof*⟩

lemma *not-MInfty-nonneg*[*simp*]: $0 \leq (x::ereal) \Longrightarrow x \neq -\infty$
⟨*proof*⟩

lemma *Sup-ereal-close*:
fixes e :: *ereal*
assumes $0 < e$
and S: $|Sup\ S| \neq \infty$ $S \neq \{\}$
shows $\exists\, x{\in}S.$ *Sup* $S - e < x$
⟨*proof*⟩

lemma *Inf-ereal-close*:
fixes e :: *ereal*
assumes $|Inf\ X| \neq \infty$
and $0 < e$
shows $\exists\, x{\in}X.\ x <$ *Inf* $X + e$
⟨*proof*⟩

lemma *SUP-PInfty*:
$(\bigwedge n::nat.\ \exists\, i{\in}A.\ ereal\ (real\ n) \leq f\ i) \Longrightarrow (SUP\ i{\in}A.\ f\ i :: ereal) = \infty$
⟨*proof*⟩

lemma *SUP-nat-Infty*: $(SUP\ i.\ ereal\ (real\ i)) = \infty$
⟨*proof*⟩

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*⟩

lemma *SUP-ereal-add-right*:
fixes $c :: ereal$
shows $I \neq \{\} \Longrightarrow c \neq -\infty \Longrightarrow (SUP\ i{\in}I.\ c + f\ i) = c + (SUP\ i{\in}I.\ f\ i)$
⟨*proof*⟩

lemma *SUP-ereal-minus-right*:
assumes $I \neq \{\}$ $c \neq -\infty$
shows $(SUP\ i{\in}I.\ c - f\ i :: ereal) = c - (INF\ i{\in}I.\ f\ i)$
⟨*proof*⟩

lemma *SUP-ereal-minus-left*:
assumes $I \neq \{\}$ $c \neq \infty$
shows $(SUP\ i{\in}I.\ f\ i - c :: ereal) = (SUP\ i{\in}I.\ f\ i) - c$
⟨*proof*⟩

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*⟩

lemma *SUP-ereal-le-addI*:
fixes $f :: 'i \Rightarrow ereal$
assumes $\bigwedge i.\ f\ i + y \leq z$ **and** $y \neq -\infty$
shows $Sup\ (f\ `\ UNIV) + y \leq z$
⟨*proof*⟩

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*⟩

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*⟩

lemma *INF-eq-minf*: $(INF\ i{\in}I.\ f\ i :: ereal) \neq -\infty \longleftrightarrow (\exists\, b{>}{-\infty}.\ \forall\, i{\in}I.\ b \leq f\ i)$
⟨*proof*⟩

lemma *INF-ereal-add-left*:

assumes $I \neq \{\}$ $c \neq -\infty$ $\bigwedge x.\ x \in I \Longrightarrow 0 \leq f\ x$
shows $(INF\ i{\in}I.\ f\ i + c :: ereal) = (INF\ i{\in}I.\ f\ i) + c$
⟨*proof*⟩

lemma *INF-ereal-add-right*:
assumes $I \neq \{\}$ $c \neq -\infty$ $\bigwedge x.\ x \in I \Longrightarrow 0 \leq f\ x$
shows $(INF\ i{\in}I.\ c + f\ i :: ereal) = c + (INF\ i{\in}I.\ f\ i)$
⟨*proof*⟩

lemma *INF-ereal-add-directed*:
fixes $f\ g :: 'a \Rightarrow ereal$
assumes *nonneg*: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$ $\bigwedge i.\ i \in I \Longrightarrow 0 \leq g\ i$
assumes *directed*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists\, k{\in}I.\ f\ i + g\ j \geq f\ k + g\ k$
shows $(INF\ i{\in}I.\ f\ i + g\ i) = (INF\ i{\in}I.\ f\ i) + (INF\ i{\in}I.\ g\ i)$
⟨*proof*⟩

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*⟩

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 $(SUP\ i.\ f\ i + g\ i) = Sup\ (f\ `\ UNIV) + Sup\ (g\ `\ UNIV)$
⟨*proof*⟩

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 $(SUP\ i.\ \sum n{\in}A.\ f\ n\ i) = (\sum n{\in}A.\ Sup\ ((f\ n)\ `\ UNIV))$
⟨*proof*⟩

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*⟩

lemma *countable-approach*:
fixes $x :: ereal$
assumes $x \neq -\infty$
shows $\exists f.\ incseq\ f \wedge (\forall\, i{::}nat.\ f\ i < x) \wedge (f \longrightarrow x)$
⟨*proof*⟩

lemma *Sup-countable-SUP*:
assumes $A \neq \{\}$
shows $\exists f{::}nat \Rightarrow ereal.\ incseq\ f \wedge range\ f \subseteq A \wedge Sup\ A = (SUP\ i.\ f\ i)$
⟨*proof*⟩

lemma *Inf-countable-INF*:
assumes $A \neq \{\}$ **shows** $\exists f{::}nat \Rightarrow ereal.\ decseq\ f \wedge range\ f \subseteq A \wedge Inf\ A = (INF\ i.\ f\ i)$
⟨*proof*⟩

lemma *SUP-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f{::}nat \Rightarrow ereal.\ range\ f \subseteq g`A \wedge Sup\ (g\ `\ A) = Sup\ (f\ `\ UNIV)$
⟨*proof*⟩

38.5 Relation to *enat*

definition $ereal\text{-}of\text{-}enat\ n = (case\ n\ of\ enat\ n \Rightarrow ereal\ (real\ n) \mid \infty \Rightarrow \infty)$

declare [[*coercion ereal-of-enat* :: $enat \Rightarrow ereal$]]
declare [[*coercion* $(\lambda n.\ ereal\ (real\ n))$ :: $nat \Rightarrow ereal$]]

lemma *ereal-of-enat-simps*[*simp*]:
$ereal\text{-}of\text{-}enat\ (enat\ n) = ereal\ n$
$ereal\text{-}of\text{-}enat\ \infty = \infty$
⟨*proof*⟩

lemma *ereal-of-enat-le-iff*[*simp*]: $ereal\text{-}of\text{-}enat\ m \leq ereal\text{-}of\text{-}enat\ n \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *ereal-of-enat-less-iff*[*simp*]: $ereal\text{-}of\text{-}enat\ m < ereal\text{-}of\text{-}enat\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *numeral-le-ereal-of-enat-iff*[*simp*]: $numeral\ m \leq ereal\text{-}of\text{-}enat\ n \longleftrightarrow numeral\ m \leq n$
⟨*proof*⟩

lemma *numeral-less-ereal-of-enat-iff*[*simp*]: $numeral\ m < ereal\text{-}of\text{-}enat\ n \longleftrightarrow numeral\ m < n$
⟨*proof*⟩

lemma *ereal-of-enat-ge-zero-cancel-iff*[*simp*]: $0 \leq ereal\text{-}of\text{-}enat\ n \longleftrightarrow 0 \leq n$
⟨*proof*⟩

lemma *ereal-of-enat-gt-zero-cancel-iff*[*simp*]: $0 < ereal\text{-}of\text{-}enat\ n \longleftrightarrow 0 < n$
⟨*proof*⟩

lemma *ereal-of-enat-zero*[*simp*]: $ereal\text{-}of\text{-}enat\ 0 = 0$
⟨*proof*⟩

lemma *ereal-of-enat-inf*[*simp*]: *ereal-of-enat n* = ∞ ⟷ *n* = ∞
⟨*proof*⟩

lemma *ereal-of-enat-add*: *ereal-of-enat* (*m* + *n*) = *ereal-of-enat m* + *ereal-of-enat n*
⟨*proof*⟩

lemma *ereal-of-enat-sub*:
assumes *n* ≤ *m*
shows *ereal-of-enat* (*m* − *n*) = *ereal-of-enat m* − *ereal-of-enat n*
⟨*proof*⟩

lemma *ereal-of-enat-mult*:
ereal-of-enat (*m* ∗ *n*) = *ereal-of-enat m* ∗ *ereal-of-enat n*
⟨*proof*⟩

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* ≥ *0*
⟨*proof*⟩

lemma *ereal-of-enat-Sup*:
assumes *A* ≠ {} **shows** *ereal-of-enat* (*Sup A*) = (*SUP a* ∈ *A*. *ereal-of-enat a*)
⟨*proof*⟩

lemma *ereal-of-enat-SUP*:
A ≠ {} ⟹ *ereal-of-enat* (*SUP a*∈*A*. *f a*) = (*SUP a* ∈ *A*. *ereal-of-enat* (*f a*))
⟨*proof*⟩

38.6 Limits on *ereal*

lemma *open-PInfty*: *open A* ⟹ ∞ ∈ *A* ⟹ (∃ *x*. {*ereal x*<..} ⊆ *A*)
⟨*proof*⟩

lemma *open-MInfty*: *open A* ⟹ −∞ ∈ *A* ⟹ (∃ *x*. {..<*ereal x*} ⊆ *A*)
⟨*proof*⟩

lemma *open-ereal-vimage*: *open S* ⟹ *open* (*ereal* −‘ *S*)
⟨*proof*⟩

lemma *open-ereal*: *open S* ⟹ *open* (*ereal* ‘ *S*)
⟨*proof*⟩

lemma *open-image-real-of-ereal*:
fixes *X*::*ereal set*
assumes *open X*
assumes *infty*: ∞ ∉ *X* −∞ ∉ *X*

shows *open* (*real-of-ereal* ‘ *X*)
⟨*proof*⟩

lemma *eventually-finite*:
fixes *x* :: *ereal*
assumes |*x*| ≠ ∞ (*f* ⟶ *x*) *F*
shows *eventually* (λ*x*. |*f x*| ≠ ∞) *F*
⟨*proof*⟩

lemma *open-ereal-def*:
open A ⟷ *open* (*ereal* −‘ *A*) ∧ (∞ ∈ *A* ⟶ (∃ *x*. {*ereal x* <..} ⊆ *A*)) ∧ (−∞ ∈ *A* ⟶ (∃ *x*. {..<*ereal x*} ⊆ *A*))
(**is** *open A* ⟷ *?rhs*)
⟨*proof*⟩

lemma *open-PInfty2*:
assumes *open A* **and** ∞ ∈ *A*
obtains *x* **where** {*ereal x*<..} ⊆ *A*
⟨*proof*⟩

lemma *open-MInfty2*:
assumes *open A* **and** −∞ ∈ *A*
obtains *x* **where** {..<*ereal x*} ⊆ *A*
⟨*proof*⟩

lemma *ereal-openE*:
assumes *open A*
obtains *x y* **where** *open* (*ereal* −‘ *A*)
and ∞ ∈ *A* ⟹ {*ereal x*<..} ⊆ *A*
and −∞ ∈ *A* ⟹ {..<*ereal y*} ⊆ *A*
⟨*proof*⟩

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* ∈ *S*
and |*x*| ≠ ∞
obtains *e* **where** *e* > *0* **and** {*x*−*e* <..< *x*+*e*} ⊆ *S*
⟨*proof*⟩

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** $\{a <..< b\} \subseteq S$
⟨*proof*⟩

38.6.1 Convergent sequences

lemma *lim-real-of-ereal*[*simp*]:
assumes *lim*: $(f \longrightarrow ereal\ x)\ net$
shows $((\lambda x.\ real\text{-}of\text{-}ereal\ (f\ x)) \longrightarrow x)\ net$
⟨*proof*⟩

lemma *lim-ereal*[*simp*]: $((\lambda n.\ ereal\ (f\ n)) \longrightarrow ereal\ x)\ net \longleftrightarrow (f \longrightarrow x)\ net$
⟨*proof*⟩

lemma *convergent-real-imp-convergent-ereal*:
assumes *convergent a*
shows *convergent* $(\lambda n.\ ereal\ (a\ n))$ **and** $lim\ (\lambda n.\ ereal\ (a\ n)) = ereal\ (lim\ a)$
⟨*proof*⟩

lemma *tendsto-PInfty*: $(f \longrightarrow \infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
⟨*proof*⟩

lemma *tendsto-PInfty'*: $(f \longrightarrow \infty)\ F = (\forall r{>}c.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
⟨*proof*⟩

lemma *tendsto-PInfty-eq-at-top*:
$((\lambda z.\ ereal\ (f\ z)) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
⟨*proof*⟩

lemma *tendsto-MInfty*: $(f \longrightarrow -\infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ f\ x < ereal\ r)\ F)$
⟨*proof*⟩

lemma *tendsto-MInfty'*: $(f \longrightarrow -\infty)\ F = (\forall r{<}c.\ eventually\ (\lambda x.\ ereal\ r > f\ x)\ F)$
⟨*proof*⟩

lemma *Lim-PInfty*: $f \longrightarrow \infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ f\ n \geq ereal\ B)$
⟨*proof*⟩

lemma *Lim-MInfty*: $f \longrightarrow -\infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ ereal\ B \geq f\ n)$
⟨*proof*⟩

lemma *Lim-bounded-PInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ f\ n \leq ereal\ B) \Longrightarrow l \neq \infty$
⟨*proof*⟩

lemma *Lim-bounded-MInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ ereal\ B \leq f\ n) \Longrightarrow l \neq -\infty$
⟨*proof*⟩

lemma *tendsto-zero-erealI*:
assumes $\bigwedge e.\ e > 0 \Longrightarrow eventually\ (\lambda x.\ |f\ x| < ereal\ e)\ F$
shows $(f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *Lim-bounded-PInfty2*: $f \longrightarrow l \Longrightarrow \forall\, n{\geq}N.\ f\ n \leq ereal\ B \Longrightarrow l \neq \infty$
⟨*proof*⟩

lemma *real-of-ereal-mult*[*simp*]:
fixes $a\ b :: ereal$
shows $real\text{-}of\text{-}ereal\ (a * b) = real\text{-}of\text{-}ereal\ a * real\text{-}of\text{-}ereal\ b$
⟨*proof*⟩

lemma *real-of-ereal-eq-0*:
fixes $x :: ereal$
shows $real\text{-}of\text{-}ereal\ x = 0 \longleftrightarrow x = \infty \vee x = -\infty \vee x = 0$
⟨*proof*⟩

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*⟩

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*⟩

lemma *ereal-mult-cancel-left*:
fixes $a\ b\ c :: ereal$
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
⟨*proof*⟩

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*⟩

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*⟩

lemma *ereal-inj-affinity*:
fixes $m\ t$ :: *ereal*
assumes $|m| \neq \infty$
and $m \neq 0$
and $|t| \neq \infty$
shows *inj-on* $(\lambda x.\ m * x + t)\ A$
⟨*proof*⟩

lemma *ereal-PInfty-eq-plus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$
⟨*proof*⟩

lemma *ereal-MInfty-eq-plus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$
⟨*proof*⟩

lemma *ereal-less-divide-pos*:
fixes $x\ y$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y < z\ /\ x \longleftrightarrow x * y < z$
⟨*proof*⟩

lemma *ereal-divide-less-pos*:
fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y\ /\ x < z \longleftrightarrow y < x * z$
⟨*proof*⟩

lemma *ereal-divide-eq*:
fixes $a\ b\ c$ :: *ereal*
shows $b \neq 0 \Longrightarrow |b| \neq \infty \Longrightarrow a\ /\ b = c \longleftrightarrow a = b * c$
⟨*proof*⟩

lemma *ereal-inverse-not-MInfty*[*simp*]: *inverse* $(a$::*ereal*$) \neq -\infty$
⟨*proof*⟩

lemma *ereal-mult-m1*[*simp*]: $x * ereal\ (-1) = -x$
⟨*proof*⟩

lemma *ereal-real'*:
assumes $|x| \neq \infty$
shows *ereal* (*real-of-ereal* x) $= x$
⟨*proof*⟩

lemma *real-ereal-id*: *real-of-ereal* $\circ$ *ereal* $= id$
⟨*proof*⟩

lemma *open-image-ereal*: *open*(*UNIV*−{ ∞ , (−∞ :: *ereal*)})
⟨*proof*⟩

lemma *ereal-le-distrib*:
fixes *a b c* :: *ereal*
shows *c* ∗ (*a* + *b*) ≤ *c* ∗ *a* + *c* ∗ *b*
⟨*proof*⟩

lemma *ereal-pos-distrib*:
fixes *a b c* :: *ereal*
assumes *0* ≤ *c*
and *c* ≠ ∞
shows *c* ∗ (*a* + *b*) = *c* ∗ *a* + *c* ∗ *b*
⟨*proof*⟩

lemma *ereal-LimI-finite*:
fixes *x* :: *ereal*
assumes |*x*| ≠ ∞
and ⋀*r*. *0* < *r* ⟹ ∃ *N*. ∀ *n*≥*N*. *u n* < *x* + *r* ∧ *x* < *u n* + *r*
shows *u* ⟶ *x*
⟨*proof*⟩

lemma *tendsto-obtains-N*:
assumes *f* ⟶ *f0 open S f0* ∈ *S*
obtains *N* **where** ∀ *n*≥*N*. *f n* ∈ *S*
⟨*proof*⟩

lemma *ereal-LimI-finite-iff*:
fixes *x* :: *ereal*
assumes |*x*| ≠ ∞
shows *u* ⟶ *x* ⟷ (∀ *r*. *0* < *r* ⟶ (∃ *N*. ∀ *n*≥*N*. *u n* < *x* + *r* ∧ *x* < *u n* + *r*))
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *ereal-Limsup-uminus*:
fixes *f* :: *'a* ⇒ *ereal*
shows *Limsup net* (λ*x*. − (*f x*)) = − *Liminf net f*
⟨*proof*⟩

lemma *liminf-bounded-iff*:
fixes *x* :: *nat* ⇒ *ereal*
shows *C* ≤ *liminf x* ⟷ (∀ *B*<*C*. ∃ *N*. ∀ *n*≥*N*. *B* < *x n*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *Liminf-add-le*:
fixes *f g* :: - ⇒ *ereal*

assumes *F*: $F \neq bot$
assumes *ev*: $eventually\ (\lambda x.\ 0 \leq f\ x)\ F\ eventually\ (\lambda x.\ 0 \leq g\ x)\ F$
shows $Liminf\ F\ f + Liminf\ F\ g \leq Liminf\ F\ (\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *Sup-ereal-mult-right'*:
assumes *nonempty*: $Y \neq \{\}$
and *x*: $x \geq 0$
shows $(SUP\ i{\in}Y.\ f\ i) * ereal\ x = (SUP\ i{\in}Y.\ f\ i * ereal\ x)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *Sup-ereal-mult-left'*:
$[\![\ Y \neq \{\};\ x \geq 0\ ]\!] \Longrightarrow ereal\ x * (SUP\ i{\in}Y.\ f\ i) = (SUP\ i{\in}Y.\ ereal\ x * f\ i)$
⟨*proof*⟩

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 $sup\text{-}continuous\ (\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *sup-continuous-mult-right*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow sup\text{-}continuous\ f \Longrightarrow sup\text{-}continuous\ (\lambda x.\ f\ x * c :: ereal)$
⟨*proof*⟩

lemma *sup-continuous-mult-left*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow sup\text{-}continuous\ f \Longrightarrow sup\text{-}continuous\ (\lambda x.\ c * f\ x :: ereal)$
⟨*proof*⟩

lemma *sup-continuous-ereal-of-enat*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f*
shows $sup\text{-}continuous\ (\lambda x.\ ereal\text{-}of\text{-}enat\ (f\ x))$
⟨*proof*⟩

38.6.2 Sums

lemma *sums-ereal-positive*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
shows $f\ sums\ (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *summable-ereal-pos*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
shows *summable f*
⟨*proof*⟩

lemma *sums-ereal*: $(\lambda x.\ ereal\ (f\ x))\ sums\ ereal\ x \longleftrightarrow f\ sums\ x$
⟨*proof*⟩

lemma *suminf-ereal-eq-SUP*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows $(\sum x.\ f\ x) = (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

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 $suminf\ f \le x$
⟨*proof*⟩

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 $suminf\ f + y \le x$
⟨*proof*⟩

lemma *suminf-upper*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n.\ 0 \le f\ n$
shows $(\sum n{<}N.\ f\ n) \le (\sum n.\ f\ n)$
⟨*proof*⟩

lemma *suminf-0-le*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n.\ 0 \le f\ n$
shows $0 \le (\sum n.\ f\ n)$
⟨*proof*⟩

lemma *suminf-le-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge N.\ f\ N \le g\ N$
and $\bigwedge N.\ 0 \le f\ N$
shows $suminf\ f \le suminf\ g$
⟨*proof*⟩

lemma *suminf-half-series-ereal*: $(\sum n.\ (1/2 :: ereal)\ \hat{}\ Suc\ n) = 1$
⟨*proof*⟩

lemma *suminf-add-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i\ \bigwedge i.\ 0 \le g\ i$
shows $(\sum i.\ f\ i + g\ i) = suminf\ f + suminf\ g$

⟨*proof*⟩

lemma *suminf-cmult-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$ **and** $0 \le a$
shows $(\sum i.\ a * f\ i) = a * suminf\ f$
⟨*proof*⟩

lemma *suminf-PInfty*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
and $suminf\ f \neq \infty$
shows $f\ i \neq \infty$
⟨*proof*⟩

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*⟩

lemma *summable-ereal*:
assumes $\bigwedge i.\ 0 \le f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows *summable f*
⟨*proof*⟩

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*⟩

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*⟩

lemma *suminf-ereal-PInf* [*simp*]: $(\sum x.\ \infty :: ereal) = \infty$
⟨*proof*⟩

lemma *summable-real-of-ereal*:
fixes $f :: nat \Rightarrow ereal$
assumes *f*: $\bigwedge i.\ 0 \le f\ i$
and *fin*: $(\sum i.\ f\ i) \neq \infty$
shows *summable* $(\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i))$
⟨*proof*⟩

lemma *suminf-SUP-eq*:
fixes $f :: nat \Rightarrow nat \Rightarrow ereal$
assumes $\bigwedge i.\ incseq\ (\lambda n.\ f\ n\ i)$
and $\bigwedge n\ i.\ 0 \le f\ n\ i$
shows $(\sum i.\ SUP\ n.\ f\ n\ i) = (SUP\ n.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

lemma *suminf-sum-ereal*:
fixes $f :: - \Rightarrow - \Rightarrow ereal$
assumes *nonneg*: $\bigwedge i\ a.\ a \in A \Longrightarrow 0 \le f\ i\ a$
shows $(\sum i.\ \sum a{\in}A.\ f\ i\ a) = (\sum a{\in}A.\ \sum i.\ f\ i\ a)$
⟨*proof*⟩

lemma *suminf-ereal-eq-0*:
fixes $f :: nat \Rightarrow ereal$
assumes *nneg*: $\bigwedge i.\ 0 \le f\ i$
shows $(\sum i.\ f\ i) = 0 \longleftrightarrow (\forall\, i.\ f\ i = 0)$
⟨*proof*⟩

lemma *suminf-ereal-offset-le*:
fixes $f :: nat \Rightarrow ereal$
assumes *f*: $\bigwedge i.\ 0 \le f\ i$
shows $(\sum i.\ f\ (i + k)) \le suminf\ f$
⟨*proof*⟩

lemma *sums-suminf-ereal*: $f\ sums\ x \Longrightarrow (\sum i.\ ereal\ (f\ i)) = ereal\ x$
⟨*proof*⟩

lemma *suminf-ereal'*: $summable\ f \Longrightarrow (\sum i.\ ereal\ (f\ i)) = ereal\ (\sum i.\ f\ i)$
⟨*proof*⟩

lemma *suminf-ereal-finite*: $summable\ f \Longrightarrow (\sum i.\ ereal\ (f\ i)) \neq \infty$
⟨*proof*⟩

lemma *suminf-ereal-finite-neg*:
assumes *summable f*
shows $(\sum x.\ ereal\ (f\ x)) \neq -\infty$
⟨*proof*⟩

lemma *SUP-ereal-add-directed*:
fixes $f\ g :: 'a \Rightarrow ereal$
assumes *nonneg*: $\bigwedge i.\ i \in I \Longrightarrow 0 \le f\ i \bigwedge i.\ i \in I \Longrightarrow 0 \le g\ i$
assumes *directed*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists\, k{\in}I.\ f\ i + g\ j \le f\ k + g\ k$
shows $(SUP\ i{\in}I.\ f\ i + g\ i) = (SUP\ i{\in}I.\ f\ i) + (SUP\ i{\in}I.\ g\ i)$
⟨*proof*⟩

lemma *SUP-ereal-sum-directed*:
fixes $f\ g :: 'a \Rightarrow 'b \Rightarrow ereal$

assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N\ i\ j.\ N \subseteq A \Longrightarrow i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ n\ i \leq f\ n\ k \wedge f\ n\ j \leq f\ n\ k$
assumes *nonneg*: $\bigwedge n\ i.\ i \in I \Longrightarrow n \in A \Longrightarrow 0 \leq f\ n\ i$
shows $(SUP\ i{\in}I.\ \sum n{\in}A.\ f\ n\ i) = (\sum n{\in}A.\ SUP\ i{\in}I.\ f\ n\ i)$
⟨*proof*⟩

lemma *suminf-SUP-eq-directed*:
fixes $f :: - \Rightarrow nat \Rightarrow ereal$
assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ i\ n \leq f\ k\ n \wedge f\ j\ n \leq f\ k\ n$
assumes *nonneg*: $\bigwedge n\ i.\ 0 \leq f\ n\ i$
shows $(\sum i.\ SUP\ n{\in}I.\ f\ n\ i) = (SUP\ n{\in}I.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

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*⟩

lemma *continuous-within-ereal*[*intro*, *simp*]: $x \in A \Longrightarrow continuous\ (at\ x\ within\ A)\ ereal$
⟨*proof*⟩

lemma *ereal-open-uminus*:
fixes $S :: ereal\ set$
assumes *open S*
shows *open* $(uminus\ `\ S)$
⟨*proof*⟩

lemma *ereal-uminus-complement*:
fixes $S :: ereal\ set$
shows $uminus\ `\ (-\ S) = -\ uminus\ `\ S$
⟨*proof*⟩

lemma *ereal-closed-uminus*:
fixes $S :: ereal\ set$
assumes *closed S*
shows *closed* $(uminus\ `\ S)$
⟨*proof*⟩

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*⟩

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*⟩

lemma *open-uminus-iff*:
fixes S :: *ereal set*
shows *open* $(uminus\ `\ S) \longleftrightarrow$ *open* S
⟨*proof*⟩

lemma *ereal-Liminf-uminus*:
fixes f :: $'a \Rightarrow$ *ereal*
shows *Liminf net* $(\lambda x.\ -\ (f\ x)) = -$ *Limsup net f*
⟨*proof*⟩

lemma *Liminf-PInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes ¬ *trivial-limit net*
shows $(f \longrightarrow \infty)$ *net* $\longleftrightarrow$ *Liminf net f* $= \infty$
⟨*proof*⟩

lemma *Limsup-MInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes ¬ *trivial-limit net*
shows $(f \longrightarrow -\infty)$ *net* $\longleftrightarrow$ *Limsup net f* $= -\infty$
⟨*proof*⟩

lemma *convergent-ereal*: — RENAME
fixes X :: *nat* $\Rightarrow$ $'a$ :: {*complete-linorder*,*linorder-topology*}
shows *convergent* $X \longleftrightarrow$ *limsup* $X =$ *liminf* X
⟨*proof*⟩

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*⟩

lemma *liminf-PInfty*:
fixes X :: *nat* $\Rightarrow$ *ereal*
shows $X \longrightarrow \infty \longleftrightarrow$ *liminf* $X = \infty$
⟨*proof*⟩

lemma *limsup-MInfty*:
fixes X :: *nat* $\Rightarrow$ *ereal*

shows $X \longrightarrow -\infty \longleftrightarrow limsup\ X = -\infty$
⟨*proof*⟩

lemma *SUP-eq-LIMSEQ*:
assumes *mono f*
shows $(SUP\ n.\ ereal\ (f\ n)) = ereal\ x \longleftrightarrow f \longrightarrow x$
⟨*proof*⟩

lemma *liminf-ereal-cminus*:
fixes $f :: nat \Rightarrow ereal$
assumes $c \neq -\infty$
shows $liminf\ (\lambda x.\ c - f\ x) = c - limsup\ f$
⟨*proof*⟩

38.6.3 Continuity

lemma *continuous-at-of-ereal*:
$|x0 :: ereal| \neq \infty \Longrightarrow continuous\ (at\ x0)\ real\text{-}of\text{-}ereal$
⟨*proof*⟩

lemma *nhds-ereal*: $nhds\ (ereal\ r) = filtermap\ ereal\ (nhds\ r)$
⟨*proof*⟩

lemma *at-ereal*: $at\ (ereal\ r) = filtermap\ ereal\ (at\ r)$
⟨*proof*⟩

lemma *at-left-ereal*: $at\text{-}left\ (ereal\ r) = filtermap\ ereal\ (at\text{-}left\ r)$
⟨*proof*⟩

lemma *at-right-ereal*: $at\text{-}right\ (ereal\ r) = filtermap\ ereal\ (at\text{-}right\ r)$
⟨*proof*⟩

lemma
shows *at-left-PInf*: $at\text{-}left\ \infty = filtermap\ ereal\ at\text{-}top$
and *at-right-MInf*: $at\text{-}right\ (-\infty) = filtermap\ ereal\ at\text{-}bot$
⟨*proof*⟩

lemma *ereal-tendsto-simps1*:
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}left\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at\text{-}left\ x)$
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}right\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at\text{-}right\ x)$
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}left\ (\infty::ereal)) \longleftrightarrow (f \longrightarrow y)\ at\text{-}top$
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}right\ (-\infty::ereal)) \longleftrightarrow (f \longrightarrow y)\ at\text{-}bot$
⟨*proof*⟩

lemma *ereal-tendsto-simps2*:
$((ereal \circ f) \longrightarrow ereal\ a)\ F \longleftrightarrow (f \longrightarrow a)\ F$
$((ereal \circ f) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x :> at\text{-}top)$
$((ereal \circ f) \longrightarrow -\infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x :> at\text{-}bot)$
⟨*proof*⟩

lemma *inverse-infty-ereal-tendsto-0*: *inverse* $-\infty\rightarrow$ (*0*::*ereal*)
⟨*proof*⟩

lemma *inverse-ereal-tendsto-pos*:
fixes *x* :: *ereal* **assumes** *0* < *x*
shows *inverse* $-x\rightarrow$ *inverse x*
⟨*proof*⟩

lemma *inverse-ereal-tendsto-at-right-0*: (*inverse* ⟶ ∞) (*at-right* (*0*::*ereal*))
⟨*proof*⟩

lemmas *ereal-tendsto-simps* = *ereal-tendsto-simps1 ereal-tendsto-simps2*

lemma *continuous-at-iff-ereal*:
fixes *f* :: *'a*::*t2-space* ⇒ *real*
shows *continuous* (*at x0 within s*) *f* ⟷ *continuous* (*at x0 within s*) (*ereal* ∘ *f*)
⟨*proof*⟩

lemma *continuous-on-iff-ereal*:
fixes *f* :: *'a*::*t2-space* => *real*
assumes *open A*
shows *continuous-on A f* ⟷ *continuous-on A* (*ereal* ∘ *f*)
⟨*proof*⟩

lemma *continuous-on-real*: *continuous-on* (*UNIV* − {∞, −∞::*ereal*}) *real-of-ereal*
⟨*proof*⟩

lemma *continuous-on-iff-real*:
fixes *f* :: *'a*::*t2-space* ⇒ *ereal*
assumes ⋀*x*. *x* ∈ *A* ⟹ |*f x*| ≠ ∞
shows *continuous-on A f* ⟷ *continuous-on A* (*real-of-ereal* ∘ *f*)
⟨*proof*⟩

lemma *continuous-uminus-ereal* [*continuous-intros*]: *continuous-on* (*A* :: *ereal set*) *uminus*
⟨*proof*⟩

lemma *ereal-uminus-atMost* [*simp*]: *uminus* ' {..(*a*::*ereal*)} = {−*a*..}
⟨*proof*⟩

lemma *continuous-on-inverse-ereal* [*continuous-intros*]:
continuous-on {*0*::*ereal* ..} *inverse*
⟨*proof*⟩

lemma *continuous-inverse-ereal-nonpos*: *continuous-on* ({..<*0*} :: *ereal set*) *inverse*
⟨*proof*⟩

lemma *tendsto-inverse-ereal*:
assumes $(f \longrightarrow (c :: ereal))\ F$
assumes *eventually* $(\lambda x.\ f\ x \geq 0)\ F$
shows $((\lambda x.\ inverse\ (f\ x)) \longrightarrow inverse\ c)\ F$
⟨*proof*⟩

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*⟩

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*⟩

lemma *Liminf-ereal-mult-left*:
assumes $F \neq bot\ (c::real) \geq 0$
shows $Liminf\ F\ (\lambda n.\ ereal\ c * f\ n) = ereal\ c * Liminf\ F\ f$
⟨*proof*⟩

lemma *Limsup-ereal-mult-left*:
assumes $F \neq bot\ (c::real) \geq 0$
shows $Limsup\ F\ (\lambda n.\ ereal\ c * f\ n) = ereal\ c * Limsup\ F\ f$
⟨*proof*⟩

lemma *limsup-ereal-mult-right*:
$(c::real) \geq 0 \Longrightarrow limsup\ (\lambda n.\ f\ n * ereal\ c) = limsup\ f * ereal\ c$
⟨*proof*⟩

lemma *limsup-ereal-mult-left*:
$(c::real) \geq 0 \Longrightarrow limsup\ (\lambda n.\ ereal\ c * f\ n) = ereal\ c * limsup\ f$
⟨*proof*⟩

lemma *Limsup-add-ereal-right*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Limsup\ F\ (\lambda n.\ g\ n + (c :: ereal)) = Limsup\ F\ g + c$
⟨*proof*⟩

lemma *Limsup-add-ereal-left*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Limsup\ F\ (\lambda n.\ (c :: ereal) + g\ n) = c + Limsup\ F\ g$
⟨*proof*⟩

lemma *Liminf-add-ereal-right*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Liminf\ F\ (\lambda n.\ g\ n + (c :: ereal)) = Liminf\ F\ g + c$
⟨*proof*⟩

lemma *Liminf-add-ereal-left*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Liminf\ F\ (\lambda n.\ (c :: ereal) + g\ n) = c + Liminf\ F\ g$
⟨*proof*⟩

lemma
assumes $F \neq bot$
assumes *nonneg*: *eventually* $(\lambda x.\ f\ x \geq (0{::}ereal))\ F$
shows *Liminf-inverse-ereal*: $Liminf\ F\ (\lambda x.\ inverse\ (f\ x)) = inverse\ (Limsup\ F\ f)$
and *Limsup-inverse-ereal*: $Limsup\ F\ (\lambda x.\ inverse\ (f\ x)) = inverse\ (Liminf\ F\ f)$
⟨*proof*⟩

lemma *ereal-diff-le-mono-left*: $[\![\ x \leq z;\ 0 \leq y\ ]\!] \Longrightarrow x - y \leq (z :: ereal)$
⟨*proof*⟩

lemma *neg-0-less-iff-less-erea* [*simp*]: $0 < - a \longleftrightarrow (a :: ereal) < 0$
⟨*proof*⟩

lemma *not-infty-ereal*: $|x| \neq \infty \longleftrightarrow (\exists x'.\ x = ereal\ x')$
⟨*proof*⟩

lemma *neq-PInf-trans*: **fixes** $x\ y :: ereal$ **shows** $[\![\ y \neq \infty;\ x \leq y\ ]\!] \Longrightarrow x \neq \infty$
⟨*proof*⟩

lemma *mult-2-ereal*: $ereal\ 2 * x = x + x$
⟨*proof*⟩

lemma *ereal-diff-le-self*: $0 \leq y \Longrightarrow x - y \leq (x :: ereal)$
⟨*proof*⟩

lemma *ereal-le-add-self*: $0 \leq y \Longrightarrow x \leq x + (y :: ereal)$
⟨*proof*⟩

lemma *ereal-le-add-self2*: $0 \leq y \Longrightarrow x \leq y + (x :: ereal)$
⟨*proof*⟩

lemma *ereal-diff-nonpos*:
fixes $a\ b :: ereal$ **shows** $[\![\ a \leq b;\ a = \infty \Longrightarrow b \neq \infty;\ a = -\infty \Longrightarrow b \neq -\infty\ ]\!] \Longrightarrow a - b \leq 0$
⟨*proof*⟩

lemma *minus-ereal-0* [*simp*]: $x - ereal\ 0 = x$
⟨*proof*⟩

lemma *ereal-diff-eq-0-iff*: **fixes** $a\ b :: ereal$
shows $(|a| = \infty \Longrightarrow |b| \neq \infty) \Longrightarrow a - b = 0 \longleftrightarrow a = b$
⟨*proof*⟩

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*⟩

lemma *ereal-divide-le-posI*:
fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow z \neq -\infty \Longrightarrow z \leq x * y \Longrightarrow z\ /\ x \leq y$
⟨*proof*⟩

lemma *add-diff-eq-ereal*:
fixes $x\ y\ z$ :: *ereal*
shows $x + (y - z) = x + y - z$
⟨*proof*⟩

lemma *ereal-diff-gr0*:
fixes $a\ b$ :: *ereal*
shows $a < b \Longrightarrow 0 < b - a$
⟨*proof*⟩

lemma *ereal-minus-minus*:
fixes $x\ y\ z$ :: *ereal* **shows**
$(|y| = \infty \Longrightarrow |z| \neq \infty) \Longrightarrow x - (y - z) = x + z - y$
⟨*proof*⟩

lemma *diff-diff-commute-ereal*:
fixes $x\ y\ z$ :: *ereal*
shows $x - y - z = x - z - y$
⟨*proof*⟩

lemma *ereal-diff-eq-MInfty-iff*:
fixes $x\ y$ :: *ereal*
shows $x - y = -\infty \longleftrightarrow x = -\infty \wedge y \neq -\infty \vee y = \infty \wedge |x| \neq \infty$
⟨*proof*⟩

lemma *ereal-diff-add-inverse*:
fixes $x\ y$ :: *ereal*
shows $|x| \neq \infty \Longrightarrow x + y - x = y$
⟨*proof*⟩

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*⟩

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* $(\lambda z.\ f\ z - g\ z :: ereal)$
⟨*proof*⟩

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* ⇒ *'a* ⇒ *real* **where** *indicat-real S* ≡ *indicator S*

lemma *indicator-simps*[*simp*]:
$x \in S \Longrightarrow$ *indicator S x = 1*
$x \notin S \Longrightarrow$ *indicator S x = 0*
⟨*proof*⟩

lemma *indicator-pos-le*[*intro*, *simp*]: $(0 :: 'a :: linordered\text{-}semidom) \leq$ *indicator S x*
and *indicator-le-1*[*intro*, *simp*]: *indicator S x* $\leq (1 :: 'a :: linordered\text{-}semidom)$
⟨*proof*⟩

lemma *indicator-abs-le-1*: $|indicator\ S\ x| \leq (1 :: 'a :: linordered\text{-}idom)$
⟨*proof*⟩

lemma *indicator-eq-0-iff*: *indicator A x* $= (0 :: 'a :: zero\text{-}neq\text{-}one) \longleftrightarrow x \notin A$
⟨*proof*⟩

lemma *indicator-eq-1-iff*: *indicator A x* $= (1 :: 'a :: zero\text{-}neq\text{-}one) \longleftrightarrow x \in A$
⟨*proof*⟩

lemma *indicator-UNIV* [*simp*]: *indicator UNIV* $= (\lambda x.\ 1)$

⟨*proof*⟩

lemma *indicator-leI*:
(*x* ∈ *A* ⟹ *y* ∈ *B*) ⟹ (*indicator A x* :: *'a*::*linordered-nonzero-semiring*) ≤ *indicator B y*
⟨*proof*⟩

lemma *split-indicator*: *P* (*indicator S x*) ⟷ ((*x* ∈ *S* ⟶ *P 1*) ∧ (*x* ∉ *S* ⟶ *P 0*))
⟨*proof*⟩

lemma *split-indicator-asm*: *P* (*indicator S x*) ⟷ (¬ (*x* ∈ *S* ∧ ¬ *P 1* ∨ *x* ∉ *S* ∧ ¬ *P 0*))
⟨*proof*⟩

lemma *indicator-inter-arith*: *indicator* (*A* ∩ *B*) *x* = *indicator A x* ∗ (*indicator B x*::*'a*::*semiring-1*)
⟨*proof*⟩

lemma *indicator-union-arith*:
indicator (*A* ∪ *B*) *x* = *indicator A x* + *indicator B x* − *indicator A x* ∗ (*indicator B x* :: *'a*::*ring-1*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *indicator-disj-union*:
A ∩ *B* = {} ⟹ *indicator* (*A* ∪ *B*) *x* = (*indicator A x* + *indicator B x* :: *'a*::*linordered-semidom*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *indicator-times*:
indicator (*A* × *B*) *x* = *indicator A* (*fst x*) ∗ (*indicator B* (*snd x*) :: *'a*::*semiring-1*)
⟨*proof*⟩

lemma *indicator-sum*:
indicator (*A* <+> *B*) *x* = (*case x of Inl x* ⇒ *indicator A x* | *Inr x* ⇒ *indicator B x*)
⟨*proof*⟩

lemma *indicator-image*: *inj f* $\Longrightarrow$ *indicator* (*f* ' *X*) (*f x*) = (*indicator X x*::-::*zero-neq-one*)
⟨*proof*⟩

lemma *indicator-vimage*: *indicator* (*f* −' *A*) *x* = *indicator A* (*f x*)
⟨*proof*⟩

lemma *mult-indicator-cong*:
fixes *f g* :: - ⇒ 'a :: *semiring-1*
shows (⋀*x*. *x* ∈ *A* $\Longrightarrow$ *f x* = *g x*) $\Longrightarrow$ *indicator A x* ∗ *f x* = *indicator A x* ∗ *g x*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sum-indicator-eq-card*:
assumes *finite A*
shows (∑ *x* ∈ *A*. *indicator B x*) = *card* (*A Int B*)
⟨*proof*⟩

lemma *sum-indicator-scaleR*[*simp*]:
finite A $\Longrightarrow$
(∑ *x* ∈ *A*. *indicator* (*B x*) (*g x*) ∗$_R$ *f x*) = (∑ *x* ∈ {*x*∈*A*. *g x* ∈ *B x*}. *f x* :: 'a::*real-vector*)
⟨*proof*⟩

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*⟩

lemma *LIMSEQ-indicator-UN*:
(λ*k*. *indicator* (⋃ *i*<*k*. *A i*) *x* :: 'a::{*topological-space*,*zero-neq-one*}) ⟶ *indicator* (⋃ *i*. *A i*) *x*
⟨*proof*⟩

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*⟩

lemma *LIMSEQ-indicator-INT*:

$(\lambda k.\ \mathit{indicator}\ (\bigcap i{<}k.\ A\ i)\ x :: 'a{::}\{\mathit{topological\text{-}space}, \mathit{zero\text{-}neq\text{-}one}\}) \longrightarrow \mathit{indicator}\ (\bigcap i.\ A\ i)\ x$
⟨*proof*⟩

lemma *indicator-add*:
$A \cap B = \{\} \Longrightarrow (\mathit{indicator}\ A\ x{::}{-}{::}\mathit{monoid\text{-}add}) + \mathit{indicator}\ B\ x = \mathit{indicator}\ (A \cup B)\ x$
⟨*proof*⟩

lemma *of-real-indicator*: $\mathit{of\text{-}real}\ (\mathit{indicator}\ A\ x) = \mathit{indicator}\ A\ x$
⟨*proof*⟩

lemma *real-of-nat-indicator*: $\mathit{real}\ (\mathit{indicator}\ A\ x :: \mathit{nat}) = \mathit{indicator}\ A\ x$
⟨*proof*⟩

lemma *abs-indicator*: $|\mathit{indicator}\ A\ x :: 'a{::}\mathit{linordered\text{-}idom}| = \mathit{indicator}\ A\ x$
⟨*proof*⟩

lemma *mult-indicator-subset*:
$A \subseteq B \Longrightarrow \mathit{indicator}\ A\ x * \mathit{indicator}\ B\ x = (\mathit{indicator}\ A\ x :: 'a{::}\mathit{comm\text{-}semiring\text{-}1})$
⟨*proof*⟩

lemma *indicator-times-eq-if*:
fixes $f :: 'a \Rightarrow 'b{::}\mathit{comm\text{-}ring\text{-}1}$
shows $\mathit{indicator}\ S\ x * f\ x = (\mathit{if}\ x \in S\ \mathit{then}\ f\ x\ \mathit{else}\ 0)\ f\ x * \mathit{indicator}\ S\ x = (\mathit{if}\ x \in S\ \mathit{then}\ f\ x\ \mathit{else}\ 0)$
⟨*proof*⟩

lemma *indicator-scaleR-eq-if*:
fixes $f :: 'a \Rightarrow 'b{::}\mathit{real\text{-}vector}$
shows $\mathit{indicator}\ S\ x *_R f\ x = (\mathit{if}\ x \in S\ \mathit{then}\ f\ x\ \mathit{else}\ 0)$
⟨*proof*⟩

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*⟩

The indicator function of the union of a disjoint family of sets is the sum over all the individual indicators.

lemma *indicator-UN-disjoint*:
$\mathit{finite}\ A \Longrightarrow \mathit{disjoint\text{-}family\text{-}on}\ f\ A \Longrightarrow \mathit{indicator}\ (\bigcup (f\ `\ A))\ x = (\sum y{\in}A.\ \mathit{indicator}\ (f\ y)\ x)$
⟨*proof*⟩

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 $a = b + (a-b)$
⟨*proof*⟩

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 $F \neq bot \Longrightarrow Limsup\ F\ (\lambda x.\ c + f\ x) = c + Limsup\ F\ f$
⟨*proof*⟩

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 $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ c + f\ x) = c + Liminf\ F\ f$
⟨*proof*⟩

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 $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ f\ x + c) = Liminf\ F\ f + c$
⟨*proof*⟩

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*⟩

lemma *suminf-offset*:
fixes $f\ g :: nat \Rightarrow 'a :: \{t2\text{-}space, topological\text{-}comm\text{-}monoid\text{-}add\}$
shows $summable\ (\lambda j.\ f\ (j + i)) \Longrightarrow suminf\ f = (\sum j.\ f\ (j + i)) + (\sum j{<}i.\ f\ j)$
⟨*proof*⟩

lemma *eventually-at-left-1*: $(\bigwedge z::real.\ 0 < z \Longrightarrow z < 1 \Longrightarrow P\ z) \Longrightarrow eventually\ P\ (at\text{-}left\ 1)$
⟨*proof*⟩

lemma *mult-eq-1*:
fixes $a\ b :: 'a :: \{ordered\text{-}semiring, comm\text{-}monoid\text{-}mult\}$
shows $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow b \leq 1 \Longrightarrow a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$
⟨*proof*⟩

lemma *ereal-add-diff-cancel*:
fixes $a\ b :: ereal$
shows $|b| \neq \infty \Longrightarrow (a + b) - b = a$
⟨*proof*⟩

lemma *add-top*:
fixes $x :: 'a::\{order\text{-}top,\ ordered\text{-}comm\text{-}monoid\text{-}add\}$
shows $0 \leq x \Longrightarrow x + top = top$
⟨*proof*⟩

lemma *top-add*:
fixes $x :: 'a::\{order\text{-}top,\ ordered\text{-}comm\text{-}monoid\text{-}add\}$
shows $0 \leq x \Longrightarrow top + x = top$
⟨*proof*⟩

lemma *le-lfp*: $mono\ f \Longrightarrow x \leq lfp\ f \Longrightarrow f\ x \leq lfp\ f$
⟨*proof*⟩

lemma *lfp-transfer*:
assumes α: *sup-continuous* α **and** f: *sup-continuous* f **and** *mg*: *mono* g
assumes *bot*: $\alpha\ bot \leq lfp\ g$ **and** *eq*: $\bigwedge x.\ x \leq lfp\ f \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (lfp\ f) = lfp\ g$
⟨*proof*⟩

lemma *sup-continuous-applyD*: $sup\text{-}continuous\ f \Longrightarrow sup\text{-}continuous\ (\lambda x.\ f\ x\ h)$
⟨*proof*⟩

lemma *sup-continuous-SUP*[*order-continuous-intros*]:
fixes $M :: - \Rightarrow - \Rightarrow 'a::complete\text{-}lattice$
assumes M: $\bigwedge i.\ i \in I \Longrightarrow sup\text{-}continuous\ (M\ i)$
shows $sup\text{-}continuous\ (SUP\ i{\in}I.\ M\ i)$
⟨*proof*⟩

lemma *sup-continuous-apply-SUP*[*order-continuous-intros*]:
fixes $M :: - \Rightarrow - \Rightarrow 'a::complete\text{-}lattice$
shows $(\bigwedge i.\ i \in I \Longrightarrow sup\text{-}continuous\ (M\ i)) \Longrightarrow sup\text{-}continuous\ (\lambda x.\ SUP\ i{\in}I.\ M\ i\ x)$
⟨*proof*⟩

lemma *sup-continuous-lfp'*[*order-continuous-intros*]:
assumes *1*: *sup-continuous* f
assumes *2*: $\bigwedge g.\ sup\text{-}continuous\ g \Longrightarrow sup\text{-}continuous\ (f\ g)$
shows *sup-continuous* $(lfp\ f)$
⟨*proof*⟩

lemma *sup-continuous-lfp''*[*order-continuous-intros*]:
assumes *1*: $\bigwedge s.\ sup\text{-}continuous\ (f\ s)$
assumes *2*: $\bigwedge g.\ sup\text{-}continuous\ g \Longrightarrow sup\text{-}continuous\ (\lambda s.\ f\ s\ (g\ s))$
shows *sup-continuous* $(\lambda x.\ lfp\ (f\ x))$

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *continuous-on-cmult-ereal*:
|*c*::*ereal*| ≠ ∞ ⟹ *continuous-on A f* ⟹ *continuous-on A* (λ*x*. *c* ∗ *f x*)
⟨*proof*⟩

lemma *real-of-nat-Sup*:
assumes *A* ≠ {} *bdd-above A*
shows *of-nat* (*Sup A*) = (*SUP a*∈*A*. *of-nat a* :: *real*)
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUP-sup-const1*:
I ≠ {} ⟹ (*SUP i*∈*I*. *sup c* (*f i*)) = *sup c* (*SUP i*∈*I*. *f i*)
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUP-sup-const2*:
I ≠ {} ⟹ (*SUP i*∈*I*. *sup* (*f i*) *c*) = *sup* (*SUP i*∈*I*. *f i*) *c*
⟨*proof*⟩

lemma *one-less-of-natD*:
assumes (*1*::*'a*::*linordered-semidom*) < *of-nat n* **shows** *1* < *n*
⟨*proof*⟩

40.1 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = {*x* :: *ereal*. *0* ≤ *x*}
morphisms *enn2ereal e2ennreal'*
⟨*proof*⟩

definition *e2ennreal x* = *e2ennreal'* (*max 0 x*)

lemma *enn2ereal-range*: *e2ennreal* ‘ {*0*..} = *UNIV*
⟨*proof*⟩

lemma *type-definition-ennreal'*: *type-definition enn2ereal e2ennreal* {*x*. *0* ≤ *x*}
⟨*proof*⟩

setup-lifting *type-definition-ennreal'*

declare [[*coercion e2ennreal*]]

instantiation *ennreal* :: *complete-linorder*
begin

lift-definition *top-ennreal* :: *ennreal* **is** *top* ⟨*proof*⟩
lift-definition *bot-ennreal* :: *ennreal* **is** *0* ⟨*proof*⟩
lift-definition *sup-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *sup* ⟨*proof*⟩
lift-definition *inf-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *inf* ⟨*proof*⟩

lift-definition *Inf-ennreal* :: *ennreal set* ⇒ *ennreal* **is** *Inf*
⟨*proof*⟩

lift-definition *Sup-ennreal* :: *ennreal set* ⇒ *ennreal* **is** *sup 0* ∘ *Sup*
⟨*proof*⟩

lift-definition *less-eq-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *bool* **is** (≤) ⟨*proof*⟩
lift-definition *less-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *bool* **is** (<) ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *pcr-ennreal-enn2ereal*[*simp*]: *pcr-ennreal* (*enn2ereal x*) *x*
⟨*proof*⟩

lemma *rel-fun-eq-pcr-ennreal*: *rel-fun* (=) *pcr-ennreal f g* ⟷ *f* = *enn2ereal* ∘ *g*
⟨*proof*⟩

instantiation *ennreal* :: *infinity*
begin

definition *infinity-ennreal* :: *ennreal*
where [*simp*]: ∞ = (*top*::*ennreal*)

instance ⟨*proof*⟩

end

instantiation *ennreal* :: {*semiring-1-no-zero-divisors*, *comm-semiring-1*}
begin

lift-definition *one-ennreal* :: *ennreal* **is** *1* ⟨*proof*⟩
lift-definition *zero-ennreal* :: *ennreal* **is** *0* ⟨*proof*⟩
lift-definition *plus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** (+) ⟨*proof*⟩
lift-definition *times-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** (∗) ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *ennreal* :: *minus*
begin

lift-definition *minus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** λ*a b*. *max 0* (*a* − *b*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *ennreal* :: *numeral* ⟨*proof*⟩

instantiation *ennreal* :: *inverse*
begin

lift-definition *inverse-ennreal* :: *ennreal* ⇒ *ennreal* **is** *inverse*
⟨*proof*⟩

definition *divide-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal*
where *x div y* = *x* ∗ *inverse* (*y* :: *ennreal*)

instance ⟨*proof*⟩

end

lemma *ennreal-zero-less-one*: *0* < (*1*::*ennreal*) — TODO: remove
⟨*proof*⟩

instance *ennreal* :: *dioid*
⟨*proof*⟩

instance *ennreal* :: *ordered-comm-semiring*
⟨*proof*⟩

instance *ennreal* :: *linordered-nonzero-semiring*
⟨*proof*⟩

instance *ennreal* :: *strict-ordered-ab-semigroup-add*
⟨*proof*⟩

declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *e2ennreal-neg*: *x* ≤ *0* ⟹ *e2ennreal x* = *0*
⟨*proof*⟩

lemma *e2ennreal-mono*: *x* ≤ *y* ⟹ *e2ennreal x* ≤ *e2ennreal y*
⟨*proof*⟩

lemma *enn2ereal-nonneg*[*simp*]: $0 \leq$ *enn2ereal* x
⟨*proof*⟩

lemma *ereal-ennreal-cases*:
obtains b **where** $0 \leq a$ $a =$ *enn2ereal* $b \mid a < 0$
⟨*proof*⟩

lemma *rel-fun-liminf*[*transfer-rule*]: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal* *liminf* *liminf*
⟨*proof*⟩

lemma *rel-fun-limsup*[*transfer-rule*]: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal* *limsup* *limsup*
⟨*proof*⟩

lemma *sum-enn2ereal*[*simp*]: $(\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i) \Longrightarrow (\sum i{\in}I.$ *enn2ereal* $(f\ i))$ = *enn2ereal* (*sum* f I)
⟨*proof*⟩

lemma *transfer-e2ennreal-sum* [*transfer-rule*]:
rel-fun (*rel-fun* (=) *pcr-ennreal*) (*rel-fun* (=) *pcr-ennreal*) *sum* *sum*
⟨*proof*⟩

lemma *enn2ereal-of-nat*[*simp*]: *enn2ereal* (*of-nat* n) = *ereal* n
⟨*proof*⟩

lemma *enn2ereal-numeral*[*simp*]: *enn2ereal* (*numeral* a) = *numeral* a
⟨*proof*⟩

lemma *transfer-numeral*[*transfer-rule*]: *pcr-ennreal* (*numeral* a) (*numeral* a)
⟨*proof*⟩

40.2 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty{::}ennreal) \vee b = c$
⟨*proof*⟩

lemma *ennreal-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty{::}ennreal) \vee b \leq c$
⟨*proof*⟩

lemma *ereal-add-left-cancel-less*:
fixes a b c :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$
⟨*proof*⟩

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty{::}ennreal) \wedge b < c$
⟨*proof*⟩

⟨*ML*⟩

40.3 Order with top

lemma *ennreal-zero-less-top*[*simp*]: $0 < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-one-less-top*[*simp*]: $1 < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-zero-neq-top*[*simp*]: $0 \neq (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-top-neq-zero*[*simp*]: $(top::ennreal) \neq 0$
⟨*proof*⟩

lemma *ennreal-top-neq-one*[*simp*]: $top \neq (1::ennreal)$
⟨*proof*⟩

lemma *ennreal-one-neq-top*[*simp*]: $1 \neq (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-add-less-top*[*simp*]:
fixes $a\ b :: ennreal$
shows $a + b < top \longleftrightarrow a < top \wedge b < top$
⟨*proof*⟩

lemma *ennreal-add-eq-top*[*simp*]:
fixes $a\ b :: ennreal$
shows $a + b = top \longleftrightarrow a = top \vee b = top$
⟨*proof*⟩

lemma *ennreal-sum-less-top*[*simp*]:
fixes $f :: 'a \Rightarrow ennreal$
shows *finite* $I \Longrightarrow (\sum i{\in}I.\ f\ i) < top \longleftrightarrow (\forall\, i{\in}I.\ f\ i < top)$
⟨*proof*⟩

lemma *ennreal-sum-eq-top*[*simp*]:
fixes $f :: 'a \Rightarrow ennreal$
shows *finite* $I \Longrightarrow (\sum i{\in}I.\ f\ i) = top \longleftrightarrow (\exists\, i{\in}I.\ f\ i = top)$
⟨*proof*⟩

lemma *ennreal-mult-eq-top-iff*:
fixes $a\ b :: ennreal$
shows $a * b = top \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
⟨*proof*⟩

lemma *ennreal-top-eq-mult-iff*:
fixes $a\ b :: ennreal$
shows $top = a * b \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
⟨*proof*⟩

lemma *ennreal-mult-less-top*:
fixes $a\ b :: ennreal$
shows $a * b < top \longleftrightarrow (a = 0 \vee b = 0 \vee (a < top \wedge b < top))$
⟨*proof*⟩

lemma *top-power-ennreal*: $top \hat{} n = (if\ n = 0\ then\ 1\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-prod-eq-0*[*simp*]:
fixes $f :: 'a \Rightarrow ennreal$
shows $(prod\ f\ A = 0) = (finite\ A \wedge (\exists i{\in}A.\ f\ i = 0))$
⟨*proof*⟩

lemma *ennreal-prod-eq-top*:
fixes $f :: 'a \Rightarrow ennreal$
shows $(\prod i{\in}I.\ f\ i) = top \longleftrightarrow (finite\ I \wedge ((\forall i{\in}I.\ f\ i \neq 0) \wedge (\exists i{\in}I.\ f\ i = top)))$
⟨*proof*⟩

lemma *ennreal-top-mult*: $top * a = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-top*: $a * top = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *enn2ereal-eq-top-iff*[*simp*]: $enn2ereal\ x = \infty \longleftrightarrow x = top$
⟨*proof*⟩

lemma *enn2ereal-top*[*simp*]: $enn2ereal\ top = \infty$
⟨*proof*⟩

lemma *e2ennreal-infty*[*simp*]: $e2ennreal\ \infty = top$
⟨*proof*⟩

lemma *ennreal-top-minus*[*simp*]: $top - x = (top::ennreal)$
⟨*proof*⟩

lemma *minus-top-ennreal*: $x - top = (if\ x = top\ then\ top\ else\ 0:: ennreal)$
⟨*proof*⟩

lemma *bot-ennreal*: $bot = (0::ennreal)$
⟨*proof*⟩

lemma *ennreal-of-nat-neq-top*[*simp*]: $of\text{-}nat\ i \neq (top::ennreal)$
⟨*proof*⟩

lemma *numeral-eq-of-nat*: $(numeral\ a::ennreal) = of\text{-}nat\ (numeral\ a)$
⟨*proof*⟩

lemma *of-nat-less-top*: $of\text{-}nat\ i < (top::ennreal)$

⟨*proof*⟩

lemma *top-neq-numeral*[*simp*]: $top \neq (numeral\ i :: ennreal)$
⟨*proof*⟩

lemma *ennreal-numeral-less-top*[*simp*]: $numeral\ i < (top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-add-bot*[*simp*]: $bot + x = (x :: ennreal)$
⟨*proof*⟩

lemma *add-top-right-ennreal* [*simp*]: $x + top = (top :: ennreal)$
⟨*proof*⟩

lemma *add-top-left-ennreal* [*simp*]: $top + x = (top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-top-mult-left* [*simp*]: $x \neq 0 \Longrightarrow x * top = (top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-top-mult-right* [*simp*]: $x \neq 0 \Longrightarrow top * x = (top :: ennreal)$
⟨*proof*⟩

lemma *power-top-ennreal* [*simp*]: $n > 0 \Longrightarrow top \mathbin{\hat{}} n = (top :: ennreal)$
⟨*proof*⟩

lemma *power-eq-top-ennreal-iff*: $x \mathbin{\hat{}} n = top \longleftrightarrow x = (top :: ennreal) \wedge n > 0$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *power-mono-ennreal*: $x \leq y \Longrightarrow x \mathbin{\hat{}} n \leq (y \mathbin{\hat{}} n :: ennreal)$
⟨*proof*⟩

instance *ennreal* :: *semiring-char-0*
⟨*proof*⟩

40.4 Arithmetic

lemma *ennreal-minus-zero*[*simp*]: $a - (0 :: ennreal) = a$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel-right*[*simp*]:
fixes $x\ y\ z :: ennreal$ **shows** $y \neq top \Longrightarrow (x + y) - y = x$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel-left*[*simp*]:
fixes $x\ y\ z :: ennreal$ **shows** $y \neq top \Longrightarrow (y + x) - y = x$
⟨*proof*⟩

lemma
fixes $a\ b :: ennreal$
shows $a - b = 0 \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *ennreal-minus-cancel*:
fixes $a\ b\ c :: ennreal$
shows $c \neq top \Longrightarrow a \leq c \Longrightarrow b \leq c \Longrightarrow c - a = c - b \Longrightarrow a = b$
⟨*proof*⟩

lemma *sup-const-add-ennreal*:
fixes $a\ b\ c :: ennreal$
shows $sup\ (c + a)\ (c + b) = c + sup\ a\ b$
⟨*proof*⟩

lemma *ennreal-diff-add-assoc*:
fixes $a\ b\ c :: ennreal$
shows $a \leq b \Longrightarrow c + b - a = c + (b - a)$
⟨*proof*⟩

lemma *mult-divide-eq-ennreal*:
fixes $a\ b :: ennreal$
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b) / b = a$
⟨*proof*⟩

lemma *divide-mult-eq*: $a \neq 0 \Longrightarrow a \neq \infty \Longrightarrow x * a / (b * a) = x / (b::ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-divide-eq*:
fixes $a\ b :: ennreal$
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b) / b = a$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel*:
fixes $a\ b :: ennreal$
shows $b \neq \infty \Longrightarrow (a + b) - b = a$
⟨*proof*⟩

lemma *ennreal-minus-eq-0*:
$a - b = 0 \Longrightarrow a \leq (b::ennreal)$
⟨*proof*⟩

lemma *ennreal-mono-minus-cancel*:
fixes $a\ b\ c :: ennreal$

shows $a - b \leq a - c \Longrightarrow a < top \Longrightarrow b \leq a \Longrightarrow c \leq a \Longrightarrow c \leq b$
⟨*proof*⟩

lemma *ennreal-mono-minus*:
fixes $a\ b\ c$ :: *ennreal*
shows $c \leq b \Longrightarrow a - b \leq a - c$
⟨*proof*⟩

lemma *ennreal-minus-pos-iff*:
fixes $a\ b$ :: *ennreal*
shows $a < top \vee b < top \Longrightarrow 0 < a - b \Longrightarrow b < a$
⟨*proof*⟩

lemma *ennreal-inverse-top*[*simp*]: *inverse top* = (*0*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-inverse-zero*[*simp*]: *inverse 0* = (*top*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-top-divide*: *top* / (*x*::*ennreal*) = (*if x* = *top then 0 else top*)
⟨*proof*⟩

lemma *ennreal-zero-divide*[*simp*]: *0* / (*x*::*ennreal*) = *0*
⟨*proof*⟩

lemma *ennreal-divide-zero*[*simp*]: *x* / (*0*::*ennreal*) = (*if x* = *0 then 0 else top*)
⟨*proof*⟩

lemma *ennreal-divide-top*[*simp*]: *x* / (*top*::*ennreal*) = *0*
⟨*proof*⟩

lemma *ennreal-times-divide*: $a * (b / c) = a * b / (c$::*ennreal*$)$
⟨*proof*⟩

lemma *ennreal-zero-less-divide*: $0 < a / b \longleftrightarrow (0 < a \wedge b < (top$::*ennreal*$))$
⟨*proof*⟩

lemma *add-divide-distrib-ennreal*: $(a + b) / c = a / c + b / (c$ :: *ennreal*$)$
⟨*proof*⟩

lemma *divide-right-mono-ennreal*:
fixes $a\ b\ c$ :: *ennreal*
shows $a \leq b \Longrightarrow a / c \leq b / c$
⟨*proof*⟩

lemma *ennreal-mult-strict-right-mono*: $(a$::*ennreal*$) < c \Longrightarrow 0 < b \Longrightarrow b < top$
$\Longrightarrow a * b < c * b$
⟨*proof*⟩

lemma *ennreal-indicator-less*[*simp*]:
indicator A x ≤ (*indicator B x*::*ennreal*) ⟷ (*x* ∈ *A* ⟶ *x* ∈ *B*)
⟨*proof*⟩

lemma *ennreal-inverse-positive*: *0* < *inverse x* ⟷ (*x*::*ennreal*) ≠ *top*
⟨*proof*⟩

lemma *ennreal-inverse-mult'*: ((*0* < *b* ∨ *a* < *top*) ∧ (*0* < *a* ∨ *b* < *top*)) ⟹ *inverse* (*a* * *b*::*ennreal*) = *inverse a* * *inverse b*
⟨*proof*⟩

lemma *ennreal-inverse-mult*: *a* < *top* ⟹ *b* < *top* ⟹ *inverse* (*a* * *b*::*ennreal*) = *inverse a* * *inverse b*
⟨*proof*⟩

lemma *ennreal-inverse-1*[*simp*]: *inverse* (*1*::*ennreal*) = *1*
⟨*proof*⟩

lemma *ennreal-inverse-eq-0-iff*[*simp*]: *inverse* (*a*::*ennreal*) = *0* ⟷ *a* = *top*
⟨*proof*⟩

lemma *ennreal-inverse-eq-top-iff*[*simp*]: *inverse* (*a*::*ennreal*) = *top* ⟷ *a* = *0*
⟨*proof*⟩

lemma *ennreal-divide-eq-0-iff*[*simp*]: (*a*::*ennreal*) / *b* = *0* ⟷ (*a* = *0* ∨ *b* = *top*)
⟨*proof*⟩

lemma *ennreal-divide-eq-top-iff*: (*a*::*ennreal*) / *b* = *top* ⟷ ((*a* ≠ *0* ∧ *b* = *0*) ∨ (*a* = *top* ∧ *b* ≠ *top*))
⟨*proof*⟩

lemma *one-divide-one-divide-ennreal*[*simp*]: *1* / (*1* / *c*) = (*c*::*ennreal*)
including *ennreal.lifting*
⟨*proof*⟩

lemma *ennreal-mult-left-cong*:
((*a*::*ennreal*) ≠ *0* ⟹ *b* = *c*) ⟹ *a* * *b* = *a* * *c*
⟨*proof*⟩

lemma *ennreal-mult-right-cong*:
((*a*::*ennreal*) ≠ *0* ⟹ *b* = *c*) ⟹ *b* * *a* = *c* * *a*
⟨*proof*⟩

lemma *ennreal-zero-less-mult-iff*: *0* < *a* * *b* ⟷ *0* < *a* ∧ *0* < (*b*::*ennreal*)
⟨*proof*⟩

lemma *less-diff-eq-ennreal*:
fixes *a b c* :: *ennreal*
shows *b* < *top* ∨ *c* < *top* ⟹ *a* < *b* − *c* ⟷ *a* + *c* < *b*

⟨*proof*⟩

lemma *diff-add-cancel-ennreal*:
fixes *a b* :: *ennreal* **shows** $a \leq b \Longrightarrow b - a + a = b$
⟨*proof*⟩

lemma *ennreal-diff-self*[*simp*]: $a \neq top \Longrightarrow a - a = (0::ennreal)$
⟨*proof*⟩

lemma *ennreal-minus-mono*:
fixes *a b c* :: *ennreal*
shows $a \leq c \Longrightarrow d \leq b \Longrightarrow a - b \leq c - d$
⟨*proof*⟩

lemma *ennreal-minus-eq-top*[*simp*]: $a - (b::ennreal) = top \longleftrightarrow a = top$
⟨*proof*⟩

lemma *ennreal-divide-self*[*simp*]: $a \neq 0 \Longrightarrow a < top \Longrightarrow a \,/\, a = (1::ennreal)$
⟨*proof*⟩

40.5 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: *real* ⇒ *ennreal* **is** *sup 0* ∘ *ereal*
⟨*proof*⟩

declare [[*coercion ennreal*]]

lemma *ennreal-cong*: $x = y \Longrightarrow ennreal\ x = ennreal\ y$
⟨*proof*⟩

lemma *ennreal-cases*[*cases type*: *ennreal*]:
fixes *x* :: *ennreal*
obtains (*real*) *r* :: *real* **where** $0 \leq r\ x = ennreal\ r \mid (top)\ x = top$
⟨*proof*⟩

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]
lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neq-top*[*simp*]: $ennreal\ r \neq top$
⟨*proof*⟩

lemma *top-neq-ennreal*[*simp*]: $top \neq ennreal\ r$
⟨*proof*⟩

lemma *ennreal-less-top*[*simp*]: $ennreal\ x < top$
⟨*proof*⟩

lemma *ennreal-neg*: $x \leq 0 \Longrightarrow ennreal\ x = 0$
⟨*proof*⟩

lemma *ennreal-inj*[*simp*]:
$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ a = ennreal\ b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *ennreal-le-iff*[*simp*]: $0 \leq y \Longrightarrow ennreal\ x \leq ennreal\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *le-ennreal-iff*: $0 \leq r \Longrightarrow x \leq ennreal\ r \longleftrightarrow (\exists\, q{\geq}0.\ x = ennreal\ q \wedge q \leq r)$
⟨*proof*⟩

lemma *ennreal-less-iff*: $0 \leq r \Longrightarrow ennreal\ r < ennreal\ q \longleftrightarrow r < q$
⟨*proof*⟩

lemma *ennreal-eq-zero-iff*[*simp*]: $0 \leq x \Longrightarrow ennreal\ x = 0 \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *ennreal-less-zero-iff*[*simp*]: $0 < ennreal\ x \longleftrightarrow 0 < x$
⟨*proof*⟩

lemma *ennreal-lessI*: $0 < q \Longrightarrow r < q \Longrightarrow ennreal\ r < ennreal\ q$
⟨*proof*⟩

lemma *ennreal-leI*: $x \leq y \Longrightarrow ennreal\ x \leq ennreal\ y$
⟨*proof*⟩

lemma *enn2ereal-ennreal*[*simp*]: $0 \leq x \Longrightarrow enn2ereal\ (ennreal\ x) = x$
⟨*proof*⟩

lemma *e2ennreal-enn2ereal*[*simp*]: $e2ennreal\ (enn2ereal\ x) = x$
⟨*proof*⟩

lemma *enn2ereal-e2ennreal*: $x \geq 0 \Longrightarrow enn2ereal\ (e2ennreal\ x) = x$
⟨*proof*⟩

lemma *e2ennreal-ereal* [*simp*]: $e2ennreal\ (ereal\ x) = ennreal\ x$
⟨*proof*⟩

lemma *ennreal-0*[*simp*]: $ennreal\ 0 = 0$
⟨*proof*⟩

lemma *ennreal-1*[*simp*]: $ennreal\ 1 = 1$
⟨*proof*⟩

lemma *ennreal-eq-0-iff*: $ennreal\ x = 0 \longleftrightarrow x \leq 0$
⟨*proof*⟩

lemma *ennreal-le-iff2*: $ennreal\ x \leq ennreal\ y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge$

$y \leq 0))$
⟨*proof*⟩

lemma *ennreal-eq-1*[*simp*]: $ennreal\ x = 1 \longleftrightarrow x = 1$
⟨*proof*⟩

lemma *ennreal-le-1*[*simp*]: $ennreal\ x \leq 1 \longleftrightarrow x \leq 1$
⟨*proof*⟩

lemma *ennreal-ge-1*[*simp*]: $ennreal\ x \geq 1 \longleftrightarrow x \geq 1$
⟨*proof*⟩

lemma *one-less-ennreal*[*simp*]: $1 < ennreal\ x \longleftrightarrow 1 < x$
⟨*proof*⟩

lemma *ennreal-plus*[*simp*]:
$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ (a + b) = ennreal\ a + ennreal\ b$
⟨*proof*⟩

lemma *add-mono-ennreal*: $x < ennreal\ y \Longrightarrow x' < ennreal\ y' \Longrightarrow x + x' < ennreal\ (y + y')$
⟨*proof*⟩

lemma *sum-ennreal*[*simp*]: $(\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i) \Longrightarrow (\sum i{\in}I.\ ennreal\ (f\ i)) = ennreal\ (sum\ f\ I)$
⟨*proof*⟩

lemma *sum-list-ennreal*[*simp*]:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x \geq 0$
shows $sum\text{-}list\ (map\ (\lambda x.\ ennreal\ (f\ x))\ xs) = ennreal\ (sum\text{-}list\ (map\ f\ xs))$
⟨*proof*⟩

lemma *ennreal-of-nat-eq-real-of-nat*: $of\text{-}nat\ i = ennreal\ (of\text{-}nat\ i)$
⟨*proof*⟩

lemma *of-nat-le-ennreal-iff*[*simp*]: $0 \leq r \Longrightarrow of\text{-}nat\ i \leq ennreal\ r \longleftrightarrow of\text{-}nat\ i \leq r$
⟨*proof*⟩

lemma *ennreal-le-of-nat-iff*[*simp*]: $ennreal\ r \leq of\text{-}nat\ i \longleftrightarrow r \leq of\text{-}nat\ i$
⟨*proof*⟩

lemma *ennreal-indicator*: $ennreal\ (indicator\ A\ x) = indicator\ A\ x$
⟨*proof*⟩

lemma *ennreal-numeral*[*simp*]: $ennreal\ (numeral\ n) = numeral\ n$
⟨*proof*⟩

lemma *ennreal-less-numeral-iff* [*simp*]: $ennreal\ n < numeral\ w \longleftrightarrow n < numeral$

w
⟨*proof*⟩

lemma *numeral-less-ennreal-iff* [*simp*]: *numeral w* < *ennreal n* ⟷ *numeral w* < *n*
⟨*proof*⟩

lemma *numeral-le-ennreal-iff* [*simp*]: *numeral n* ≤ *ennreal m* ⟷ *numeral n* ≤ *m*
⟨*proof*⟩

lemma *min-ennreal*: *0* ≤ *x* ⟹ *0* ≤ *y* ⟹ *min* (*ennreal x*) (*ennreal y*) = *ennreal* (*min x y*)
⟨*proof*⟩

lemma *ennreal-half*[*simp*]: *ennreal* (*1*/*2*) = *inverse 2*
⟨*proof*⟩

lemma *ennreal-minus*: *0* ≤ *q* ⟹ *ennreal r* − *ennreal q* = *ennreal* (*r* − *q*)
⟨*proof*⟩

lemma *ennreal-minus-top*[*simp*]: *ennreal a* − *top* = *0*
⟨*proof*⟩

lemma *e2eenreal-enn2ereal-diff* [*simp*]:
e2ennreal(*enn2ereal x* − *enn2ereal y*) = *x* − *y* **for** *x y*
⟨*proof*⟩

lemma *ennreal-mult*: *0* ≤ *a* ⟹ *0* ≤ *b* ⟹ *ennreal* (*a* ∗ *b*) = *ennreal a* ∗ *ennreal b*
⟨*proof*⟩

lemma *ennreal-mult′*: *0* ≤ *a* ⟹ *ennreal* (*a* ∗ *b*) = *ennreal a* ∗ *ennreal b*
⟨*proof*⟩

lemma *indicator-mult-ennreal*: *indicator A x* ∗ *ennreal r* = *ennreal* (*indicator A x* ∗ *r*)
⟨*proof*⟩

lemma *ennreal-mult′′*: *0* ≤ *b* ⟹ *ennreal* (*a* ∗ *b*) = *ennreal a* ∗ *ennreal b*
⟨*proof*⟩

lemma *numeral-mult-ennreal*: *0* ≤ *x* ⟹ *numeral b* ∗ *ennreal x* = *ennreal* (*numeral b* ∗ *x*)
⟨*proof*⟩

lemma *ennreal-power*: *0* ≤ *r* ⟹ *ennreal r* ^ *n* = *ennreal* (*r* ^ *n*)
⟨*proof*⟩

lemma *power-eq-top-ennreal*: *x* ^ *n* = *top* ⟷ (*n* ≠ *0* ∧ (*x*::*ennreal*) = *top*)

⟨*proof*⟩

lemma *inverse-ennreal*: $0 < r \Longrightarrow inverse\ (ennreal\ r) = ennreal\ (inverse\ r)$
⟨*proof*⟩

lemma *divide-ennreal*: $0 \leq r \Longrightarrow 0 < q \Longrightarrow ennreal\ r\ /\ ennreal\ q = ennreal\ (r\ /\ q)$
⟨*proof*⟩

lemma *ennreal-inverse-power*: $inverse\ (x \hat{}\ n :: ennreal) = inverse\ x \hat{}\ n$
⟨*proof*⟩

lemma *power-divide-distrib-ennreal* [*algebra-simps*]:
$(x\ /\ y) \hat{}\ n = x \hat{}\ n\ /\ (y \hat{}\ n :: ennreal)$
⟨*proof*⟩

lemma *ennreal-divide-numeral*: $0 \leq x \Longrightarrow ennreal\ x\ /\ numeral\ b = ennreal\ (x\ /\ numeral\ b)$
⟨*proof*⟩

lemma *prod-ennreal*: $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow (\prod i{\in}A.\ ennreal\ (f\ i)) = ennreal\ (prod\ f\ A)$
⟨*proof*⟩

lemma *prod-mono-ennreal*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \leq (g\ x :: ennreal)$
shows $prod\ f\ A \leq prod\ g\ A$
⟨*proof*⟩

lemma *mult-right-ennreal-cancel*: $a * ennreal\ c = b * ennreal\ c \longleftrightarrow (a = b \vee c \leq 0)$
⟨*proof*⟩

lemma *ennreal-le-epsilon*:
$(\bigwedge e{::}real.\ y < top \Longrightarrow 0 < e \Longrightarrow x \leq y + ennreal\ e) \Longrightarrow x \leq y$
⟨*proof*⟩

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*⟩

lemma *ennreal-Ex-less-of-nat*: $(x{::}ennreal) < top \Longrightarrow \exists n.\ x < of\text{-}nat\ n$
⟨*proof*⟩

40.6 Coercion from *ennreal* to *real*

definition *enn2real* $x = real\text{-}of\text{-}ereal\ (enn2ereal\ x)$

lemma *enn2real-nonneg*[*simp*]: $0 \leq$ *enn2real* x
⟨*proof*⟩

lemma *enn2real-mono*: $a \leq b \Longrightarrow b < \mathit{top} \Longrightarrow \mathit{enn2real}\ a \leq \mathit{enn2real}\ b$
⟨*proof*⟩

lemma *enn2real-of-nat*[*simp*]: *enn2real* (*of-nat* n) $= n$
⟨*proof*⟩

lemma *enn2real-ennreal*[*simp*]: $0 \leq r \Longrightarrow \mathit{enn2real}\ (\mathit{ennreal}\ r) = r$
⟨*proof*⟩

lemma *ennreal-enn2real*[*simp*]: $r < \mathit{top} \Longrightarrow \mathit{ennreal}\ (\mathit{enn2real}\ r) = r$
⟨*proof*⟩

lemma *real-of-ereal-enn2ereal*[*simp*]: *real-of-ereal* (*enn2ereal* x) $=$ *enn2real* x
⟨*proof*⟩

lemma *enn2real-top*[*simp*]: *enn2real top* $= 0$
⟨*proof*⟩

lemma *enn2real-0*[*simp*]: *enn2real* $0 = 0$
⟨*proof*⟩

lemma *enn2real-1*[*simp*]: *enn2real* $1 = 1$
⟨*proof*⟩

lemma *enn2real-numeral*[*simp*]: *enn2real* (*numeral* n) $=$ (*numeral* n)
⟨*proof*⟩

lemma *enn2real-mult*: *enn2real* $(a * b) =$ *enn2real* $a *$ *enn2real* b
⟨*proof*⟩

lemma *enn2real-leI*: $0 \leq B \Longrightarrow x \leq \mathit{ennreal}\ B \Longrightarrow \mathit{enn2real}\ x \leq B$
⟨*proof*⟩

lemma *enn2real-positive-iff*: $0 <$ *enn2real* $x \longleftrightarrow (0 < x \wedge x < \mathit{top})$
⟨*proof*⟩

lemma *enn2real-eq-posreal-iff*[*simp*]: $c > 0 \Longrightarrow \mathit{enn2real}\ x = c \longleftrightarrow x = c$
⟨*proof*⟩

lemma *ennreal-enn2real-if*: *ennreal* (*enn2real* r) $=$ (*if* $r =$ *top then* 0 *else* r)
⟨*proof*⟩

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 ∞ = ∞*
⟨*proof*⟩

lemma *ennreal-of-enat-enat*[*simp*]: *ennreal-of-enat (enat n) = of-nat n*
⟨*proof*⟩

lemma *ennreal-of-enat-0*[*simp*]: *ennreal-of-enat 0 = 0*
⟨*proof*⟩

lemma *ennreal-of-enat-1*[*simp*]: *ennreal-of-enat 1 = 1*
⟨*proof*⟩

lemma *ennreal-top-neq-of-nat*[*simp*]: *(top::ennreal) ≠ of-nat i*
⟨*proof*⟩

lemma *ennreal-of-enat-inj*[*simp*]: *ennreal-of-enat i = ennreal-of-enat j ⟷ i = j*
⟨*proof*⟩

lemma *ennreal-of-enat-le-iff*[*simp*]: *ennreal-of-enat m ≤ ennreal-of-enat n ⟷ m ≤ n*
⟨*proof*⟩

lemma *of-nat-less-ennreal-of-nat*[*simp*]: *of-nat n ≤ ennreal-of-enat x ⟷ of-nat n ≤ x*
⟨*proof*⟩

lemma *ennreal-of-enat-Sup*: *ennreal-of-enat (Sup X) = (SUP x∈X. ennreal-of-enat x)*
⟨*proof*⟩

lemma *ennreal-of-enat-eSuc*[*simp*]: *ennreal-of-enat (eSuc x) = 1 + ennreal-of-enat x*
⟨*proof*⟩

lemma *ennreal-of-enat-plus*[*simp*]: *‹ennreal-of-enat (a+b) = ennreal-of-enat a + ennreal-of-enat b›*
⟨*proof*⟩

lemma *sum-ennreal-of-enat*[*simp*]: *(∑ i∈I. ennreal-of-enat (f i)) = ennreal-of-enat (sum f I)*
⟨*proof*⟩

40.8 Topology on *ennreal*

lemma *enn2ereal-Iio*: *enn2ereal* $-`$ $\{..<a\} = (\textit{if } 0 \leq a \textit{ then } \{..<$ *e2ennreal* $a\} \textit{ else } \{\})$
⟨*proof*⟩

lemma *enn2ereal-Ioi*: *enn2ereal* $-`$ $\{a <..\} = (\textit{if } 0 \leq a \textit{ then } \{$*e2ennreal* $a <..\} \textit{ else } \textit{UNIV})$
⟨*proof*⟩

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*⟩

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*
⟨*proof*⟩

lemma *continuous-at-e2ennreal*: *continuous* (*at x within A*) *e2ennreal*
⟨*proof*⟩

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*
⟨*proof*⟩

lemma *continuous-at-enn2ereal*: *continuous* (*at x within A*) *enn2ereal*
⟨*proof*⟩

lemma *sup-continuous-e2ennreal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *e2ennreal* (*f x*))
⟨*proof*⟩

lemma *sup-continuous-enn2ereal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *enn2ereal* (*f x*))
⟨*proof*⟩

lemma *sup-continuous-mult-left-ennreal'*:
fixes *c* :: *ennreal*
shows *sup-continuous* (λ*x*. *c* ∗ *x*)
⟨*proof*⟩

lemma *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *c* ∗ *f x* :: *ennreal*)
⟨*proof*⟩

lemma *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *f x* ∗ *c* :: *ennreal*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *transfer-enn2ereal-continuous-on* [*transfer-rule*]:
rel-fun (=) (*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *continuous-on continuous-on*
⟨*proof*⟩

lemma *transfer-sup-continuous*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *sup-continuous sup-continuous*
⟨*proof*⟩

lemma *continuous-on-ennreal*[*tendsto-intros*]:
continuous-on A f ⟹ *continuous-on A* (λ*x*. *ennreal* (*f x*))
⟨*proof*⟩

lemma *tendsto-ennrealD*:
assumes *lim*: ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F*
assumes ∗: ∀ *F x in F*. *0* ≤ *f x* **and** *x*: *0* ≤ *x*
shows (*f* ⟶ *x*) *F*
⟨*proof*⟩

lemma *tendsto-ennreal-iff* [*simp*]:
‹((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F* ⟷ (*f* ⟶ *x*) *F*› (**is** ‹*?P* ⟷ *?Q*›)
if ‹∀ *F x in F*. *0* ≤ *f x*› ‹*0* ≤ *x*›
⟨*proof*⟩

lemma *tendsto-enn2ereal-iff*[*simp*]: ((λ*i*. *enn2ereal* (*f i*)) ⟶ *enn2ereal x*) *F* ⟷
(*f* ⟶ *x*) *F*
⟨*proof*⟩

lemma *ennreal-tendsto-0-iff*: (⋀*n*. *f n* ≥ *0*) ⟹ ((λ*n*. *ennreal* (*f n*)) ⟶ *0*)
⟷ (*f* ⟶ *0*)
⟨*proof*⟩

lemma *continuous-on-add-ennreal*:
fixes *f g* :: *'a*::*topological-space* ⇒ *ennreal*
shows *continuous-on A f* ⟹ *continuous-on A g* ⟹ *continuous-on A* (λ*x*. *f x*
+ *g x*)
⟨*proof*⟩

lemma *continuous-on-inverse-ennreal*[*continuous-intros*]:
fixes *f* :: *'a*::*topological-space* ⇒ *ennreal*

shows *continuous-on A f* $\Longrightarrow$ *continuous-on A* ($\lambda x.$ *inverse* (*f x*))
⟨*proof*⟩

instance *ennreal* :: *topological-comm-monoid-add*
⟨*proof*⟩

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *'a*::*complete-lattice* $\Rightarrow$ *ennreal*
shows *sup-continuous f* $\Longrightarrow$ *sup-continuous g* $\Longrightarrow$ *sup-continuous* ($\lambda x.$ *f x* + *g x*)
⟨*proof*⟩

lemma *ennreal-suminf-lessD*: ($\sum$ *i. f i* :: *ennreal*) < *x* $\Longrightarrow$ *f i* < *x*
⟨*proof*⟩

lemma *sums-ennreal*[*simp*]: ($\bigwedge i.$ *0* $\leq$ *f i*) $\Longrightarrow$ *0* $\leq$ *x* $\Longrightarrow$ ($\lambda i.$ *ennreal* (*f i*)) *sums ennreal x* $\longleftrightarrow$ *f sums x*
⟨*proof*⟩

lemma *summable-suminf-not-top*: ($\bigwedge i.$ *0* $\leq$ *f i*) $\Longrightarrow$ ($\sum i.$ *ennreal* (*f i*)) $\neq$ *top* $\Longrightarrow$ *summable f*
⟨*proof*⟩

lemma *suminf-ennreal*[*simp*]:
($\bigwedge i.$ *0* $\leq$ *f i*) $\Longrightarrow$ ($\sum i.$ *ennreal* (*f i*)) $\neq$ *top* $\Longrightarrow$ ($\sum i.$ *ennreal* (*f i*)) = *ennreal* ($\sum i.$ *f i*)
⟨*proof*⟩

lemma *sums-enn2ereal*[*simp*]: ($\lambda i.$ *enn2ereal* (*f i*)) *sums enn2ereal x* $\longleftrightarrow$ *f sums x*
⟨*proof*⟩

lemma *suminf-enn2ereal*[*simp*]: ($\sum i.$ *enn2ereal* (*f i*)) = *enn2ereal* (*suminf f*)
⟨*proof*⟩

lemma *transfer-e2ennreal-suminf* [*transfer-rule*]: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal suminf suminf*
⟨*proof*⟩

lemma *ennreal-suminf-cmult*[*simp*]: ($\sum i.$ *r* * *f i*) = *r* * ($\sum i.$ *f i*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-suminf-multc*[*simp*]: ($\sum i.$ *f i* * *r*) = ($\sum i.$ *f i*::*ennreal*) * *r*
⟨*proof*⟩

lemma *ennreal-suminf-divide*[*simp*]: ($\sum i.$ *f i* / *r*) = ($\sum i.$ *f i*::*ennreal*) / *r*
⟨*proof*⟩

lemma *ennreal-suminf-neq-top*: *summable f* $\Longrightarrow$ ($\bigwedge i.$ *0* $\leq$ *f i*) $\Longrightarrow$ ($\sum i.$ *ennreal*

$(f\ i)) \neq top$
⟨*proof*⟩

lemma *suminf-ennreal-eq*:
$(\bigwedge i.\ 0 \leq f\ i) \Longrightarrow f\ sums\ x \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ x$
⟨*proof*⟩

lemma *ennreal-suminf-bound-add*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge N.\ (\sum n{<}N.\ f\ n) + y \leq x) \Longrightarrow suminf\ f + y \leq x$
⟨*proof*⟩

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*⟩

lemma *INF-ennreal-add-const*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(INF\ i.\ f\ i + c) = (INF\ i.\ f\ i) + c$
⟨*proof*⟩

lemma *INF-ennreal-const-add*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(INF\ i.\ c + f\ i) = c + (INF\ i.\ f\ i)$
⟨*proof*⟩

lemma *SUP-mult-left-ennreal*: $c * (SUP\ i{\in}I.\ f\ i) = (SUP\ i{\in}I.\ c * f\ i :: ennreal)$
⟨*proof*⟩

lemma *SUP-mult-right-ennreal*: $(SUP\ i{\in}I.\ f\ i) * c = (SUP\ i{\in}I.\ f\ i * c :: ennreal)$
⟨*proof*⟩

lemma *SUP-divide-ennreal*: $(SUP\ i{\in}I.\ f\ i)\ /\ c = (SUP\ i{\in}I.\ f\ i\ /\ c :: ennreal)$
⟨*proof*⟩

lemma *ennreal-SUP-of-nat-eq-top*: $(SUP\ x.\ of\text{-}nat\ x :: ennreal) = top$
⟨*proof*⟩

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*⟩

lemma *ennreal-INF-const-minus*:
fixes $f :: 'a \Rightarrow ennreal$

shows $I \neq \{\} \Longrightarrow (SUP\ x{\in}I.\ c - f\ x) = c - (INF\ x{\in}I.\ f\ x)$
⟨*proof*⟩

lemma *of-nat-Sup-ennreal*:
assumes $A \neq \{\}$ *bdd-above* A
shows *of-nat* (*Sup* A) = (*SUP* $a{\in}A$. *of-nat* a :: *ennreal*)
⟨*proof*⟩

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*⟩

lemma *ennreal-SUP-add*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows *incseq* $f \Longrightarrow$ *incseq* $g \Longrightarrow (SUP\ i.\ f\ i + g\ i) = Sup\ (f\ `\ UNIV) + Sup\ (g\ `\ UNIV)$
⟨*proof*⟩

lemma *ennreal-SUP-sum*:
fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
shows $(\bigwedge i.\ i \in I \Longrightarrow incseq\ (f\ i)) \Longrightarrow (SUP\ n.\ \sum i{\in}I.\ f\ i\ n) = (\sum i{\in}I.\ SUP\ n.\ f\ i\ n)$
⟨*proof*⟩

lemma *ennreal-liminf-minus*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge n.\ f\ n \leq c) \Longrightarrow liminf\ (\lambda n.\ c - f\ n) = c - limsup\ f$
⟨*proof*⟩

lemma *ennreal-continuous-on-cmult*:
$(c{::}ennreal) < top \Longrightarrow$ *continuous-on* $A\ f \Longrightarrow$ *continuous-on* $A\ (\lambda x.\ c * f\ x)$
⟨*proof*⟩

lemma *ennreal-tendsto-cmult*:
$(c{::}ennreal) < top \Longrightarrow (f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ c * f\ x) \longrightarrow c * x)\ F$
⟨*proof*⟩

lemma *tendsto-ennrealI*[*intro*, *simp*, *tendsto-intros*]:
$(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ ennreal\ (f\ x)) \longrightarrow ennreal\ x)\ F$
⟨*proof*⟩

lemma *tendsto-enn2erealI* [*tendsto-intros*]:
assumes $(f \longrightarrow l)\ F$
shows $((\lambda i.\ enn2ereal(f\ i)) \longrightarrow enn2ereal\ l)\ F$
⟨*proof*⟩

lemma *tendsto-e2ennrealI* [*tendsto-intros*]:
assumes $(f \longrightarrow l)\ F$
shows $((\lambda i.\ e2ennreal(f\ i)) \longrightarrow e2ennreal\ l)\ F$
⟨*proof*⟩

lemma *ennreal-suminf-minus*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(\bigwedge i.\ g\ i \leq f\ i) \Longrightarrow suminf\ f \neq top \Longrightarrow suminf\ g \neq top \Longrightarrow (\sum i.\ f\ i - g\ i) = suminf\ f - suminf\ g$
⟨*proof*⟩

lemma *ennreal-Sup-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f{::}nat \Rightarrow ennreal.\ incseq\ f \wedge range\ f \subseteq A \wedge Sup\ A = (SUP\ i.\ f\ i)$
⟨*proof*⟩

lemma *ennreal-Inf-countable-INF*:
$A \neq \{\} \Longrightarrow \exists f{::}nat \Rightarrow ennreal.\ decseq\ f \wedge range\ f \subseteq A \wedge Inf\ A = (INF\ i.\ f\ i)$
⟨*proof*⟩

lemma *ennreal-SUP-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f{::}nat \Rightarrow ennreal.\ range\ f \subseteq g`A \wedge Sup\ (g\ `\ A) = Sup\ (f\ `\ UNIV)$
⟨*proof*⟩

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n{::}nat.\ of\text{-}nat\ n :: ennreal) \longrightarrow top$
⟨*proof*⟩

lemma *SUP-sup-continuous-ennreal*:
fixes $f :: ennreal \Rightarrow 'a{::}complete\text{-}lattice$
assumes f: *sup-continuous* f **and** $I \neq \{\}$
shows $(SUP\ i{\in}I.\ f\ (g\ i)) = f\ (SUP\ i{\in}I.\ g\ i)$
⟨*proof*⟩

lemma *ennreal-suminf-SUP-eq*:
fixes $f :: nat \Rightarrow nat \Rightarrow ennreal$
shows $(\bigwedge i.\ incseq\ (\lambda n.\ f\ n\ i)) \Longrightarrow (\sum i.\ SUP\ n.\ f\ n\ i) = (SUP\ n.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

lemma *ennreal-SUP-add-left*:
fixes $c :: ennreal$
shows $I \neq \{\} \Longrightarrow (SUP\ i{\in}I.\ f\ i + c) = (SUP\ i{\in}I.\ f\ i) + c$
⟨*proof*⟩

lemma *ennreal-SUP-const-minus*:
fixes $f :: 'a \Rightarrow ennreal$
shows $I \neq \{\} \Longrightarrow c < top \Longrightarrow (INF\ x{\in}I.\ c - f\ x) = c - (SUP\ x{\in}I.\ f\ x)$
⟨*proof*⟩

lemma *isCont-ennreal*[*simp*]: ‹*isCont ennreal x*›

⟨*proof*⟩

lemma *isCont-ennreal-of-enat*[*simp*]: ‹*isCont ennreal-of-enat x*›
⟨*proof*⟩

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 $\exists\, i \in A.\ f\ i < x + e$
⟨*proof*⟩

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*⟩

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*⟩

lemma *ennreal-approx-INF*:
fixes *x*::*ennreal*
assumes *f-bound*: $\bigwedge i.\ i \in A \Longrightarrow x \leq f\ i$
assumes *approx*: $\bigwedge e.\ (e{::}real) > 0 \Longrightarrow \exists\, i \in A.\ f\ i \leq x + e$
shows $x = (INF\ i \in A.\ f\ i)$
⟨*proof*⟩

lemma *ennreal-approx-unit*:
$(\bigwedge a{::}ennreal.\ 0 < a \Longrightarrow a < 1 \Longrightarrow a * z \leq y) \Longrightarrow z \leq y$
⟨*proof*⟩

lemma *suminf-ennreal2*:
$(\bigwedge i.\ 0 \leq f\ i) \Longrightarrow summable\ f \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
⟨*proof*⟩

lemma *less-top-ennreal*: $x < top \longleftrightarrow (\exists\, r{\geq}0.\ x = ennreal\ r)$
⟨*proof*⟩

lemma *enn2real-less-iff*[*simp*]: $x < top \Longrightarrow enn2real\ x < c \longleftrightarrow x < c$
⟨*proof*⟩

lemma *enn2real-le-iff*[*simp*]: $[\![x < top;\ c > 0]\!] \Longrightarrow enn2real\ x \leq c \longleftrightarrow x \leq c$
⟨*proof*⟩

lemma *enn2real-less*:
assumes *enn2real* $e < r$ $e \neq top$ **shows** $e < ennreal\ r$
⟨*proof*⟩

lemma *enn2real-le*:
assumes *enn2real* $e \leq r$ $e \neq top$ **shows** $e \leq ennreal\ r$
⟨*proof*⟩

lemma *tendsto-top-iff-ennreal*:
fixes $f :: 'a \Rightarrow ennreal$
shows $(f \longrightarrow top)\ F \longleftrightarrow (\forall l{\geq}0.\ eventually\ (\lambda x.\ ennreal\ l < f\ x)\ F)$
⟨*proof*⟩

lemma *ennreal-tendsto-top-eq-at-top*:
$((\lambda z.\ ennreal\ (f\ z)) \longrightarrow top)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
⟨*proof*⟩

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:
fixes $f :: - \Rightarrow ennreal$
shows $Limsup\ F\ f = 0 \Longrightarrow (f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *diff-le-self-ennreal*[*simp*]: $a - b \leq (a{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-ineq-diff-add*: $b \leq a \Longrightarrow a = b + (a - b{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-strict-left-mono*: $(a{::}ennreal) < c \Longrightarrow 0 < b \Longrightarrow b < top \Longrightarrow b * a < b * c$
⟨*proof*⟩

lemma *ennreal-between*: $0 < e \Longrightarrow 0 < x \Longrightarrow x < top \Longrightarrow x - e < (x{::}ennreal)$
⟨*proof*⟩

lemma *minus-less-iff-ennreal*: $b < top \Longrightarrow b \leq a \Longrightarrow a - b < c \longleftrightarrow a < c + (b{::}ennreal)$
⟨*proof*⟩

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*⟩

lifting-update *ennreal.lifting*
lifting-forget *ennreal.lifting*

40.10 *ennreal* theorems

lemma *neq-top-trans*: **fixes** $x\ y :: ennreal$ **shows** $[\![y \neq top; x \leq y]\!] \Longrightarrow x \neq top$
⟨*proof*⟩

lemma *diff-diff-ennreal*: **fixes** $a\ b :: ennreal$ **shows** $a \leq b \Longrightarrow b \neq \infty \Longrightarrow b - (b - a) = a$
⟨*proof*⟩

lemma *ennreal-less-one-iff*[*simp*]: $ennreal\ x < 1 \longleftrightarrow x < 1$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *zero-minus-ennreal*[*simp*]: $0 - (a{::}ennreal) = 0$
including *ennreal.lifting*
⟨*proof*⟩

lemma *diff-diff-commute-ennreal*:
fixes $a\ b\ c :: ennreal$ **shows** $a - b - c = a - c - b$
⟨*proof*⟩

lemma *diff-gr0-ennreal*: $b < (a{::}ennreal) \Longrightarrow 0 < a - b$
including *ennreal.lifting* ⟨*proof*⟩

lemma *divide-le-posI-ennreal*:
fixes $x\ y\ z :: ennreal$
shows $x > 0 \Longrightarrow z \leq x * y \Longrightarrow z\ /\ x \leq y$
⟨*proof*⟩

lemma *add-diff-eq-ennreal*:
fixes $x\ y\ z :: ennreal$
shows $z \leq y \Longrightarrow x + (y - z) = x + y - z$
⟨*proof*⟩

lemma *add-diff-inverse-ennreal*:
fixes $x\ y :: ennreal$ **shows** $x \leq y \Longrightarrow x + (y - x) = y$
⟨*proof*⟩

lemma *add-diff-eq-iff-ennreal*[*simp*]:
fixes $x\ y :: ennreal$ **shows** $x + (y - x) = y \longleftrightarrow x \leq y$

⟨*proof*⟩

lemma *add-diff-le-ennreal*: $a + b - c \leq a + (b - c::ennreal)$
⟨*proof*⟩

lemma *diff-eq-0-ennreal*: $a < top \Longrightarrow a \leq b \Longrightarrow a - b = (0::ennreal)$
⟨*proof*⟩

lemma *diff-diff-ennreal'*: **fixes** $x\ y\ z :: ennreal$ **shows** $z \leq y \Longrightarrow y - z \leq x \Longrightarrow x - (y - z) = x + z - y$
⟨*proof*⟩

lemma *diff-diff-ennreal''*: **fixes** $x\ y\ z :: ennreal$
shows $z \leq y \Longrightarrow x - (y - z) = (if\ y - z \leq x\ then\ x + z - y\ else\ 0)$
⟨*proof*⟩

lemma *power-less-top-ennreal*: **fixes** $x :: ennreal$ **shows** $x \hat{} n < top \longleftrightarrow x < top \vee n = 0$
⟨*proof*⟩

lemma *ennreal-divide-times*: $(a\ /\ b) * c = a * (c\ /\ b :: ennreal)$
⟨*proof*⟩

lemma *diff-less-top-ennreal*: $a - b < top \longleftrightarrow\ a < (top :: ennreal)$
⟨*proof*⟩

lemma *divide-less-ennreal*: $b \neq 0 \Longrightarrow b < top \Longrightarrow a\ /\ b < c \longleftrightarrow a < (c * b :: ennreal)$
⟨*proof*⟩

lemma *one-less-numeral*[*simp*]: $1 < (numeral\ n::ennreal) \longleftrightarrow (num.One < n)$
⟨*proof*⟩

lemma *divide-eq-1-ennreal*: $a\ /\ b = (1::ennreal) \longleftrightarrow (b \neq top \wedge b \neq 0 \wedge b = a)$
⟨*proof*⟩

lemma *ennreal-mult-cancel-left*: $(a * b = a * c) = (a = top \wedge b \neq 0 \wedge c \neq 0 \vee a = 0 \vee b = (c::ennreal))$
⟨*proof*⟩

lemma *ennreal-minus-if*: $ennreal\ a - ennreal\ b = ennreal\ (if\ 0 \leq b\ then\ (if\ b \leq a\ then\ a - b\ else\ 0)\ else\ a)$
⟨*proof*⟩

lemma *ennreal-plus-if*: $ennreal\ a + ennreal\ b = ennreal\ (if\ 0 \leq a\ then\ (if\ 0 \leq b\ then\ a + b\ else\ a)\ else\ b)$
⟨*proof*⟩

lemma *ennreal-diff-le-mono-left*: $a \leq b \Longrightarrow a - c \leq (b::ennreal)$

⟨*proof*⟩

lemma *ennreal-minus-le-iff*: $a - b \leq c \longleftrightarrow (a \leq b + (c::ennreal) \wedge (a = top \wedge b = top \longrightarrow c = top))$
⟨*proof*⟩

lemma *ennreal-le-minus-iff*: $a \leq b - c \longleftrightarrow (a + c \leq (b::ennreal) \vee (a = 0 \wedge b \leq c))$
⟨*proof*⟩

lemma *diff-add-eq-diff-diff-swap-ennreal*: $x - (y + z :: ennreal) = x - y - z$
⟨*proof*⟩

lemma *diff-add-assoc2-ennreal*: $b \leq a \Longrightarrow (a - b + c::ennreal) = a + c - b$
⟨*proof*⟩

lemma *diff-gt-0-iff-gt-ennreal*: $0 < a - b \longleftrightarrow (a = top \wedge b = top \vee b < (a::ennreal))$
⟨*proof*⟩

lemma *diff-eq-0-iff-ennreal*: $(a - b::ennreal) = 0 \longleftrightarrow (a < top \wedge a \leq b)$
⟨*proof*⟩

lemma *add-diff-self-ennreal*: $a + (b - a::ennreal) = (if\ a \leq b\ then\ b\ else\ a)$
⟨*proof*⟩

lemma *diff-add-self-ennreal*: $(b - a + a::ennreal) = (if\ a \leq b\ then\ b\ else\ a)$
⟨*proof*⟩

lemma *ennreal-minus-cancel-iff*:
fixes $a\ b\ c :: ennreal$
shows $a - b = a - c \longleftrightarrow (b = c \vee (a \leq b \wedge a \leq c) \vee a = top)$
⟨*proof*⟩

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 $a * (b - c) = a * b - a * c$
⟨*proof*⟩

lemma *SUP-diff-ennreal*:
$c < top \Longrightarrow (SUP\ i{\in}I.\ f\ i - c :: ennreal) = (SUP\ i{\in}I.\ f\ i) - c$
⟨*proof*⟩

lemma *ennreal-SUP-add-right*:
fixes $c :: ennreal$ **shows** $I \neq \{\} \Longrightarrow c + (SUP\ i{\in}I.\ f\ i) = (SUP\ i{\in}I.\ c + f\ i)$
⟨*proof*⟩

lemma *SUP-add-directed-ennreal*:

fixes *f g* :: - ⇒ *ennreal*
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*⟩

lemma *enn2real-eq-0-iff*: *enn2real x* = *0* ⟷ *x* = *0* ∨ *x* = *top*
⟨*proof*⟩

lemma *continuous-on-diff-ennreal*:
continuous-on A f ⟹ *continuous-on A g* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *f x* ≠ *top*) ⟹ (⋀*x*. *x* ∈ *A* ⟹ *g x* ≠ *top*) ⟹ *continuous-on A* (λ*z*. *f z* − *g z*::*ennreal*)
including *ennreal.lifting*
⟨*proof*⟩

lemma *tendsto-diff-ennreal*:
(*f* ⟶ *x*) *F* ⟹ (*g* ⟶ *y*) *F* ⟹ *x* ≠ *top* ⟹ *y* ≠ *top* ⟹ ((λ*z*. *f z* − *g z*::*ennreal*) ⟶ *x* − *y*) *F*
⟨*proof*⟩

declare *lim-real-of-ereal* [*tendsto-intros*]

lemma *tendsto-enn2real* [*tendsto-intros*]:
assumes (*u* ⟶ *ennreal l*) *F l* ≥ *0*
shows ((λ*n*. *enn2real* (*u n*)) ⟶ *l*) *F*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *floorlog-bounds*:
$b \mathbin{\hat{}} (\mathit{floorlog}\ b\ x - 1) \leq x \wedge x < b \mathbin{\hat{}} (\mathit{floorlog}\ b\ x)$ **if** $x > 0\ b > 1$
⟨*proof*⟩

lemma *floorlog-power* [*simp*]:
$\mathit{floorlog}\ b\ (a * b \mathbin{\hat{}} c) = \mathit{floorlog}\ b\ a + c$ **if** $a > 0\ b > 1$
⟨*proof*⟩

lemma *floor-log-add-eqI*:
$\lfloor \mathit{log}\ b\ (a + r) \rfloor = \lfloor \mathit{log}\ b\ a \rfloor$ **if** $b > 1\ a \geq 1\ 0 \leq r\ r < 1$
for $a\ b :: \mathit{nat}$ **and** $r :: \mathit{real}$
⟨*proof*⟩

lemma *floor-log-div*:
$\lfloor \mathit{log}\ b\ x \rfloor = \lfloor \mathit{log}\ b\ (x\ \mathit{div}\ b) \rfloor + 1$ **if** $b > 1\ x > 0\ x\ \mathit{div}\ b > 0$
for $b\ x :: \mathit{nat}$
⟨*proof*⟩

lemma *compute-floorlog* [*code*]:
$\mathit{floorlog}\ b\ x = (\mathit{if}\ x > 0 \wedge b > 1\ \mathit{then}\ \mathit{floorlog}\ b\ (x\ \mathit{div}\ b) + 1\ \mathit{else}\ 0)$
⟨*proof*⟩

lemma *floor-log-eq-if*:
$\lfloor \mathit{log}\ b\ x \rfloor = \lfloor \mathit{log}\ b\ y \rfloor$ **if** $x\ \mathit{div}\ b = y\ \mathit{div}\ b\ b > 1\ x > 0\ x\ \mathit{div}\ b \geq 1$
for $b\ x\ y :: \mathit{nat}$
⟨*proof*⟩

lemma *floorlog-eq-if*:
$\mathit{floorlog}\ b\ x = \mathit{floorlog}\ b\ y$ **if** $x\ \mathit{div}\ b = y\ \mathit{div}\ b\ b > 1\ x > 0\ x\ \mathit{div}\ b \geq 1$
for $b\ x\ y :: \mathit{nat}$
⟨*proof*⟩

lemma *floorlog-leD*:
$\mathit{floorlog}\ b\ x \leq w \Longrightarrow b > 1 \Longrightarrow x < b \mathbin{\hat{}} w$
⟨*proof*⟩

lemma *floorlog-leI*:
$x < b \mathbin{\hat{}} w \Longrightarrow 0 \leq w \Longrightarrow b > 1 \Longrightarrow \mathit{floorlog}\ b\ x \leq w$
⟨*proof*⟩

lemma *floorlog-eq-zero-iff*:
$\mathit{floorlog}\ b\ x = 0 \longleftrightarrow b \leq 1 \vee x \leq 0$
⟨*proof*⟩

lemma *floorlog-le-iff*:
$\mathit{floorlog}\ b\ x \leq w \longleftrightarrow b \leq 1 \vee b > 1 \wedge 0 \leq w \wedge x < b \mathbin{\hat{}} w$
⟨*proof*⟩

lemma *floorlog-ge-SucI*:

Suc w ≤ *floorlog b x* **if** *b* ^ *w* ≤ *x b* > *1*
⟨*proof*⟩

lemma *floorlog-geI*:
w ≤ *floorlog b x* **if** *b* ^ (*w* − *1*) ≤ *x b* > *1*
⟨*proof*⟩

lemma *floorlog-geD*:
b ^ (*w* − *1*) ≤ *x* **if** *w* ≤ *floorlog b x w* > *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ceillog2-le1-eq-0* [*simp*]: *n* ≤ *1* ⟹ *ceillog2 n* = *0*
⟨*proof*⟩

lemma *ceillog2-2-power* [*simp*]: *ceillog2* (*2* ^ *n*) = *n*
⟨*proof*⟩

lemma *ceillog2-ge-log*:
assumes *n* > *0*
shows *real* (*ceillog2 n*) ≥ *log 2* (*real n*)
⟨*proof*⟩

lemma *ceillog2-less-log*:
assumes *n* > *0*
shows *real* (*ceillog2 n*) < *log 2* (*real n*) + *1*
⟨*proof*⟩

lemma *ceillog2-le-iff*:
assumes *n* > *0*
shows *ceillog2 n* ≤ *l* ⟷ *n* ≤ *2* ^ *l*
⟨*proof*⟩

lemma *ceillog2-ge-iff*:
assumes *n* > *0*
shows *ceillog2 n* ≥ *l* ⟷ *2* ^ *l* < *2* ∗ *n*
⟨*proof*⟩

lemma *le-two-power-ceillog2*: *n* ≤ *2* ^ *ceillog2 n*
⟨*proof*⟩

lemma *two-power-ceillog2-gt*:
assumes $n > 0$
shows $2 * n > 2 \hat{} ceillog2\ n$
⟨*proof*⟩

lemma *ceillog2-eqI*:
assumes $n \leq 2 \hat{} l\ 2 \hat{} l < 2 * n$
shows $ceillog2\ n = l$
⟨*proof*⟩

lemma *ceillog2-rec-even*:
assumes $k > 0$
shows $ceillog2\ (2 * k) = Suc\ (ceillog2\ k)$
⟨*proof*⟩

lemma *ceillog2-mono*:
assumes $m \leq n$
shows $ceillog2\ m \leq ceillog2\ n$
⟨*proof*⟩

lemma *ceillog2-rec-odd*:
assumes $k > 0$
shows $ceillog2\ (Suc\ (2 * k)) = Suc\ (ceillog2\ (Suc\ k))$
⟨*proof*⟩

lemma *ceillog2-rec*:
$ceillog2\ n = (if\ n \leq 1\ then\ 0\ else\ 1 + ceillog2\ ((n + 1)\ div\ 2))$
⟨*proof*⟩

lemma *funpow-div2-ceillog2-le-1*:
$((\lambda n.\ (n + 1)\ div\ 2) \hat{\hat{}} ceillog2\ n)\ n \leq 1$
⟨*proof*⟩

fun *ceillog2-aux* :: $nat \Rightarrow nat \Rightarrow nat$ **where**
$ceillog2\text{-}aux\ acc\ n = (if\ n \leq 1\ then\ acc\ else\ ceillog2\text{-}aux\ (acc + 1)\ ((n + 1)\ div\ 2))$

lemmas [*simp del*] = *ceillog2-aux.simps*

lemma *ceillog2-aux-correct*: $ceillog2\text{-}aux\ acc\ n = ceillog2\ n + acc$
⟨*proof*⟩

lemma *ceillog2-code* [*code*]: $ceillog2\ n = ceillog2\text{-}aux\ 0\ n$
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *bitlen-zero* [*simp*]:
  bitlen 0 = *0*
  ⟨*proof*⟩

lemma *bitlen-nonneg*:
  0 ≤ *bitlen x*
  ⟨*proof*⟩

lemma *bitlen-bounds*:
  2 ^ *nat* (*bitlen x* − *1*) ≤ *x* ∧ *x* < *2* ^ *nat* (*bitlen x*) **if** *x* > *0*
⟨*proof*⟩

lemma *bitlen-pow2* [*simp*]:
  bitlen (*b* ∗ *2* ^ *c*) = *bitlen b* + *c* **if** *b* > *0*
  ⟨*proof*⟩

lemma *compute-bitlen* [*code*]:
  bitlen x = (*if x* > *0 then bitlen* (*x div 2*) + *1 else 0*)
  ⟨*proof*⟩

lemma *bitlen-eq-zero-iff*:
  bitlen x = *0* ⟷ *x* ≤ *0*
  ⟨*proof*⟩

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*⟩

lemma *bitlen-le-iff-floorlog*:
  bitlen x ≤ *w* ⟷ *w* ≥ *0* ∧ *floorlog 2* (*nat x*) ≤ *nat w*
  ⟨*proof*⟩

lemma *bitlen-le-iff-power*:
  bitlen x ≤ *w* ⟷ *w* ≥ *0* ∧ *x* < *2* ^ *nat w*
  ⟨*proof*⟩

lemma *less-power-nat-iff-bitlen*:
  x < *2* ^ *w* ⟷ *bitlen* (*int x*) ≤ *w*
  ⟨*proof*⟩

lemma *bitlen-ge-iff-power*:
$w \leq$ *bitlen* $x \longleftrightarrow w \leq 0 \vee 2 \mathbin{\hat{}} (nat\ w - 1) \leq x$
⟨*proof*⟩

lemma *bitlen-twopow-add-eq*:
bitlen $(2 \mathbin{\hat{}} w + b) = w + 1$ **if** $0 \leq b\ b < 2 \mathbin{\hat{}} w$
⟨*proof*⟩

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*⟩

lemma *add-inf-distrib-right*: $inf\ a\ b + c = inf\ (a + c)\ (b + c)$
⟨*proof*⟩

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*⟩

lemma *add-sup-distrib-right*: $sup\ a\ b + c = sup\ (a + c)\ (b + c)$
⟨*proof*⟩

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

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*⟩

lemma *sup-eq-neg-inf*: *sup a b* = − *inf* (− *a*) (− *b*)
⟨*proof*⟩

lemma *neg-inf-eq-sup*: − *inf a b* = *sup* (− *a*) (− *b*)
⟨*proof*⟩

lemma *diff-inf-eq-sup*: *a* − *inf b c* = *a* + *sup* (− *b*) (− *c*)
⟨*proof*⟩

lemma *neg-sup-eq-inf*: − *sup a b* = *inf* (− *a*) (− *b*)
⟨*proof*⟩

lemma *diff-sup-eq-inf*: *a* − *sup b c* = *a* + *inf* (− *b*) (− *c*)
⟨*proof*⟩

lemma *add-eq-inf-sup*: *a* + *b* = *sup a b* + *inf a b*
⟨*proof*⟩

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*⟩

lemma *nprt-neg*: *nprt* (− *x*) = − *pprt x*
⟨*proof*⟩

lemma *prts*: *a* = *pprt a* + *nprt a*
⟨*proof*⟩

lemma *zero-le-pprt*[*simp*]: *0* ≤ *pprt a*
⟨*proof*⟩

lemma *nprt-le-zero*[*simp*]: *nprt a* ≤ *0*
⟨*proof*⟩

lemma *le-eq-neg*: *a* ≤ − *b* ⟷ *a* + *b* ≤ *0*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *pprt-0*[*simp*]: *pprt 0* = *0* ⟨*proof*⟩

lemma *nprt-0*[*simp*]: *nprt 0 = 0* ⟨*proof*⟩

lemma *pprt-eq-id* [*simp*, *no-atp*]: *0 ≤ x ⟹ pprt x = x*
⟨*proof*⟩

lemma *nprt-eq-id* [*simp*, *no-atp*]: *x ≤ 0 ⟹ nprt x = x*
⟨*proof*⟩

lemma *pprt-eq-0* [*simp*, *no-atp*]: *x ≤ 0 ⟹ pprt x = 0*
⟨*proof*⟩

lemma *nprt-eq-0* [*simp*, *no-atp*]: *0 ≤ x ⟹ nprt x = 0*
⟨*proof*⟩

lemma *sup-0-imp-0*:
assumes *sup a (− a) = 0*
shows *a = 0*
⟨*proof*⟩

lemma *inf-0-imp-0*: *inf a (− a) = 0 ⟹ a = 0*
⟨*proof*⟩

lemma *inf-0-eq-0* [*simp*]: *inf a (− a) = 0 ⟷ a = 0*
⟨*proof*⟩

lemma *sup-0-eq-0* [*simp*]: *sup a (− a) = 0 ⟷ a = 0*
⟨*proof*⟩

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: *0 ≤ a + a ⟷ 0 ≤ a*
(**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *double-zero* [*simp*]: *a + a = 0 ⟷ a = 0*
⟨*proof*⟩

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]: *0 < a + a ⟷ 0 < a*
⟨*proof*⟩

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: *a + a ≤ 0 ⟷ a ≤ 0*
⟨*proof*⟩

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]: *a + a < 0 ⟷ a < 0*
⟨*proof*⟩

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*⟩

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
⟨*proof*⟩

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
⟨*proof*⟩

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
⟨*proof*⟩

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
⟨*proof*⟩

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*⟩

subclass *ordered-ab-group-add-abs*
⟨*proof*⟩

end

lemma *sup-eq-if*:
fixes $a :: 'a::\{lattice\text{-}ab\text{-}group\text{-}add,linorder\}$
shows $sup\ a\ (- a) = (if\ a < 0\ then\ - a\ else\ a)$
⟨*proof*⟩

lemma *abs-if-lattice*:
fixes $a :: 'a::\{lattice\text{-}ab\text{-}group\text{-}add\text{-}abs,linorder\}$

shows $|a| = (\mathit{if}\ a < 0\ \mathit{then} - a\ \mathit{else}\ a)$
⟨*proof*⟩

lemma *estimate-by-abs*:
fixes *a b c* :: *'a*::*lattice-ab-group-add-abs*
assumes $a + b \leq c$
shows $a \leq c + |b|$
⟨*proof*⟩

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

end

lemma *abs-le-mult*:
fixes *a b* :: *'a*::*lattice-ring*
shows $|a * b| \leq |a| * |b|$
⟨*proof*⟩

instance *lattice-ring* ⊆ *ordered-ring-abs*
⟨*proof*⟩

lemma *mult-le-prts*:
fixes *a b* :: *'a*::*lattice-ring*
assumes $a1 \leq a$
and $a \leq a2$
and $b1 \leq b$
and $b \leq b2$
shows $a * b \leq$
*pprt a2 * pprt b2 + pprt a1 * nprt b2 + nprt a2 * pprt b1 + nprt a1 * nprt b1*
⟨*proof*⟩

lemma *mult-ge-prts*:
fixes *a b* :: *'a*::*lattice-ring*
assumes $a1 \leq a$
and $a \leq a2$
and $b1 \leq b$
and $b \leq b2$
shows $a * b \geq$
*nprt a1 * pprt b2 + nprt a2 * nprt b2 + pprt a1 * pprt b1 + pprt a2 * nprt b1*
⟨*proof*⟩

instance *int* :: *lattice-ring*
⟨*proof*⟩

instance *real* :: *lattice-ring*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *zero-float*[*simp*]: *0* ∈ *float*
⟨*proof*⟩

lemma *one-float*[*simp*]: *1* ∈ *float*
⟨*proof*⟩

lemma *numeral-float*[*simp*]: *numeral i* ∈ *float*
⟨*proof*⟩

lemma *neg-numeral-float*[*simp*]: − *numeral i* ∈ *float*

⟨*proof*⟩

lemma *real-of-int-float*[*simp*]: *real-of-int x* ∈ *float* **for** *x* :: *int*
⟨*proof*⟩

lemma *real-of-nat-float*[*simp*]: *real x* ∈ *float* **for** *x* :: *nat*
⟨*proof*⟩

lemma *two-powr-int-float*[*simp*]: *2 powr* (*real-of-int i*) ∈ *float* **for** *i* :: *int*
⟨*proof*⟩

lemma *two-powr-nat-float*[*simp*]: *2 powr* (*real i*) ∈ *float* **for** *i* :: *nat*
⟨*proof*⟩

lemma *two-powr-minus-int-float*[*simp*]: *2 powr* − (*real-of-int i*) ∈ *float* **for** *i* :: *int*
⟨*proof*⟩

lemma *two-powr-minus-nat-float*[*simp*]: *2 powr* − (*real i*) ∈ *float* **for** *i* :: *nat*
⟨*proof*⟩

lemma *two-powr-numeral-float*[*simp*]: *2 powr numeral i* ∈ *float*
⟨*proof*⟩

lemma *two-powr-neg-numeral-float*[*simp*]: *2 powr* − *numeral i* ∈ *float*
⟨*proof*⟩

lemma *two-pow-float*[*simp*]: *2* ^ *n* ∈ *float*
⟨*proof*⟩

lemma *plus-float*[*simp*]: *r* ∈ *float* ⟹ *p* ∈ *float* ⟹ *r* + *p* ∈ *float*
⟨*proof*⟩

lemma *uminus-float*[*simp*]: *x* ∈ *float* ⟹ −*x* ∈ *float*
⟨*proof*⟩

lemma *times-float*[*simp*]: *x* ∈ *float* ⟹ *y* ∈ *float* ⟹ *x* ∗ *y* ∈ *float*
⟨*proof*⟩

lemma *minus-float*[*simp*]: *x* ∈ *float* ⟹ *y* ∈ *float* ⟹ *x* − *y* ∈ *float*
⟨*proof*⟩

lemma *abs-float*[*simp*]: *x* ∈ *float* ⟹ |*x*| ∈ *float*
⟨*proof*⟩

lemma *sgn-of-float*[*simp*]: *x* ∈ *float* ⟹ *sgn x* ∈ *float*
⟨*proof*⟩

lemma *div-power-2-float*[*simp*]: *x* ∈ *float* ⟹ *x* / *2*^*d* ∈ *float*

⟨*proof*⟩

lemma *div-power-2-int-float*[*simp*]: $x \in \mathit{float} \Longrightarrow x \mathbin{/} (2::\mathit{int})\hat{}d \in \mathit{float}$
⟨*proof*⟩

lemma *div-numeral-Bit0-float*[*simp*]:
assumes $x \mathbin{/} \mathit{numeral}\ n \in \mathit{float}$
shows $x \mathbin{/} (\mathit{numeral}\ (\mathit{Num.Bit0}\ n)) \in \mathit{float}$
⟨*proof*⟩

lemma *div-neg-numeral-Bit0-float*[*simp*]:
assumes $x \mathbin{/} \mathit{numeral}\ n \in \mathit{float}$
shows $x \mathbin{/} (-\ \mathit{numeral}\ (\mathit{Num.Bit0}\ n)) \in \mathit{float}$
⟨*proof*⟩

lemma *power-float*[*simp*]:
assumes $a \in \mathit{float}$
shows $a \hat{}\ b \in \mathit{float}$
⟨*proof*⟩

lift-definition *Float* :: *int* ⇒ *int* ⇒ *float* **is** $\lambda(m::\mathit{int})\ (e::\mathit{int}).\ m * 2\ \mathit{powr}\ e$
⟨*proof*⟩
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*)))
⟨*proof*⟩

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* ⟨*proof*⟩
declare *zero-float.rep-eq*[*simp*]

lift-definition *one-float* :: *float* **is** *1* ⟨*proof*⟩
declare *one-float.rep-eq*[*simp*]

lift-definition *plus-float* :: *float* ⇒ *float* ⇒ *float* **is** (+) ⟨*proof*⟩
declare *plus-float.rep-eq*[*simp*]

lift-definition *times-float* :: *float* ⇒ *float* ⇒ *float* **is** (∗) ⟨*proof*⟩
declare *times-float.rep-eq*[*simp*]

lift-definition *minus-float* :: *float* ⇒ *float* ⇒ *float* **is** (−) ⟨*proof*⟩
declare *minus-float.rep-eq*[*simp*]

lift-definition *uminus-float* :: *float* ⇒ *float* **is** *uminus* ⟨*proof*⟩
declare *uminus-float.rep-eq*[*simp*]

lift-definition *abs-float* :: *float* ⇒ *float* **is** *abs* ⟨*proof*⟩
declare *abs-float.rep-eq*[*simp*]

lift-definition *sgn-float* :: *float* ⇒ *float* **is** *sgn* ⟨*proof*⟩
declare *sgn-float.rep-eq*[*simp*]

lift-definition *equal-float* :: *float* ⇒ *float* ⇒ *bool* **is** (=) :: *real* ⇒ *real* ⇒ *bool* ⟨*proof*⟩

lift-definition *less-eq-float* :: *float* ⇒ *float* ⇒ *bool* **is** (≤) ⟨*proof*⟩
declare *less-eq-float.rep-eq*[*simp*]

lift-definition *less-float* :: *float* ⇒ *float* ⇒ *bool* **is** (<) ⟨*proof*⟩
declare *less-float.rep-eq*[*simp*]

instance
⟨*proof*⟩

end

lemma *real-of-float* [*simp*]: *real-of-float* (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *real-of-float-of-int-eq* [*simp*]: *real-of-float* (*of-int z*) = *of-int z*
⟨*proof*⟩

lemma *Float-0-eq-0*[*simp*]: *Float 0 e* = *0*
⟨*proof*⟩

lemma *real-of-float-power*[*simp*]: *real-of-float* (*f^n*) = *real-of-float f^n* **for** *f* :: *float*
⟨*proof*⟩

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*
⟨*proof*⟩

instance *float* :: *unbounded-dense-linorder*
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *float-numeral*[*simp*]: *real-of-float* (*numeral x* :: *float*) = *numeral x*
⟨*proof*⟩

lemma *transfer-numeral* [*transfer-rule*]:
rel-fun (=) *pcr-float* (*numeral* :: - ⇒ *real*) (*numeral* :: - ⇒ *float*)
⟨*proof*⟩

lemma *float-neg-numeral*[*simp*]: *real-of-float* (− *numeral x* :: *float*) = − *numeral x*
⟨*proof*⟩

lemma *transfer-neg-numeral* [*transfer-rule*]:
rel-fun (=) *pcr-float* (− *numeral* :: - ⇒ *real*) (− *numeral* :: - ⇒ *float*)
⟨*proof*⟩

lemma *float-of-numeral*: *numeral k* = *float-of* (*numeral k*)
and *float-of-neg-numeral*: − *numeral k* = *float-of* (− *numeral k*)
⟨*proof*⟩

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 ⟨*proof*⟩

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
(λx. Quickcheck-Exhaustive.full-exhaustive (λy. f (valtermify-float x y)) d) d

instance ⟨*proof*⟩

end

instantiation *float* :: *random*
begin

definition *Quickcheck-Random.random i =*
scomp (Quickcheck-Random.random (2 ^ nat-of-natural i))
(λman. scomp (Quickcheck-Random.random i) (λexp. Pair (valtermify-float man exp)))

instance ⟨*proof*⟩

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*⟩

lemma *int-cancel-factors*:
fixes *n* :: *int*
assumes *1 < r*
shows *n = 0* ∨ (∃ *k i*. *n = k * r ^ i* ∧ ¬ *r dvd k*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *mantissa-noteq-0*: *f ≠ 0 ⟹ mantissa f ≠ 0*
⟨*proof*⟩

lemma *mantissa-eq-zero-iff*: *mantissa x = 0 ⟷ x = 0*
(**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *mantissa-pos-iff*: *0 < mantissa x ⟷ 0 < x*
⟨*proof*⟩

lemma *mantissa-nonneg-iff*: $0 \leq$ *mantissa* $x \longleftrightarrow 0 \leq x$
⟨*proof*⟩

lemma *mantissa-neg-iff*: $0 >$ *mantissa* $x \longleftrightarrow 0 > x$
⟨*proof*⟩

lemma
fixes $m\ e$ :: *int*
defines $f \equiv$ *float-of* $(m * 2\ powr\ e)$
assumes *dvd*: $\neg\ 2\ dvd\ m$
shows *mantissa-float*: *mantissa* $f = m$ (**is** *?M*)
and *exponent-float*: $m \neq 0 \Longrightarrow$ *exponent* $f = e$ (**is** - $\Longrightarrow$ *?E*)
⟨*proof*⟩

43.5 Compute arithmetic operations

lemma *Float-mantissa-exponent*: *Float* (*mantissa* f) (*exponent* f) $= f$
⟨*proof*⟩

lemma *Float-cases* [*cases type*: *float*]:
fixes f :: *float*
obtains (*Float*) $m\ e$ :: *int* **where** $f =$ *Float* $m\ e$
⟨*proof*⟩

lemma *denormalize-shift*:
assumes *f-def*: $f =$ *Float* $m\ e$
and *not-0*: $f \neq 0$
obtains i **where** $m =$ *mantissa* $f * 2\ \hat{}\ i\ e =$ *exponent* $f - i$
⟨*proof*⟩

context
begin

qualified lemma *compute-float-zero*[*code-unfold*, *code*]: $0 =$ *Float* $0\ 0$
⟨*proof*⟩ **lemma** *compute-float-one*[*code-unfold*, *code*]: $1 =$ *Float* $1\ 0$
⟨*proof*⟩

lift-definition *normfloat* :: *float* ⇒ *float* **is** $\lambda x.\ x$ ⟨*proof*⟩
lemma *normloat-id*[*simp*]: *normfloat* $x = x$ ⟨*proof*⟩ **lemma** *compute-normfloat*[*code*]:
normfloat (*Float* $m\ e$) =
(*if* m *mod* $2 = 0 \wedge m \neq 0$ *then normfloat* (*Float* (m *div* 2) ($e + 1$))
else if $m = 0$ *then* 0 *else Float* $m\ e$)
⟨*proof*⟩ **lemma** *compute-float-numeral*[*code-abbrev*]: *Float* (*numeral* k) $0 =$ *numeral* k
⟨*proof*⟩ **lemma** *compute-float-neg-numeral*[*code-abbrev*]: *Float* ($-$ *numeral* k) $0 = -$ *numeral* k
⟨*proof*⟩ **lemma** *compute-float-uminus*[*code*]: $-$ *Float m1 e1* = *Float* ($-$ *m1*) *e1*
⟨*proof*⟩ **lemma** *compute-float-times*[*code*]: *Float m1 e1* $*$ *Float m2 e2* = *Float*

$(m1 * m2)\ (e1 + e2)$
$\langle proof \rangle$ **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)*
$\langle proof \rangle$ **lemma** *compute-float-minus*[*code*]: $f - g = f + (-g)$ **for** $f\ g$ :: *float*
$\langle proof \rangle$ **lemma** *compute-float-sgn*[*code*]:
sgn (Float m1 e1) = (if 0 < m1 then 1 else if m1 < 0 then −1 else 0)
$\langle proof \rangle$

lift-definition *is-float-pos* :: *float* ⇒ *bool* **is** (<) *0* :: *real* ⇒ *bool* $\langle proof \rangle$ **lemma** *compute-is-float-pos*[*code*]: *is-float-pos (Float m e)* ⟷ *0 < m*
$\langle proof \rangle$

lift-definition *is-float-nonneg* :: *float* ⇒ *bool* **is** (≤) *0* :: *real* ⇒ *bool* $\langle proof \rangle$ **lemma** *compute-is-float-nonneg*[*code*]: *is-float-nonneg (Float m e)* ⟷ *0 ≤ m*
$\langle proof \rangle$

lift-definition *is-float-zero* :: *float* ⇒ *bool* **is** (=) *0* :: *real* ⇒ *bool* $\langle proof \rangle$ **lemma** *compute-is-float-zero*[*code*]: *is-float-zero (Float m e)* ⟷ *0 = m*
$\langle proof \rangle$ **lemma** *compute-float-abs*[*code*]: *|Float m e| = Float |m| e*
$\langle proof \rangle$ **lemma** *compute-float-eq*[*code*]: *equal-class.equal f g = is-float-zero (f − g)*
$\langle proof \rangle$

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* = $\lfloor x * 2\ powr\ prec \rfloor * 2\ powr\ -prec$

definition *round-up* :: *int* ⇒ *real* ⇒ *real*
where *round-up prec x* = $\lceil x * 2\ powr\ prec \rceil * 2\ powr\ -prec$

lemma *round-down-float*[*simp*]: *round-down prec x* ∈ *float*

⟨*proof*⟩

lemma *round-up-float*[*simp*]: *round-up prec x* ∈ *float*
⟨*proof*⟩

lemma *round-up*: *x* ≤ *round-up prec x*
⟨*proof*⟩

lemma *round-down*: *round-down prec x* ≤ *x*
⟨*proof*⟩

lemma *round-up-0*[*simp*]: *round-up p 0* = *0*
⟨*proof*⟩

lemma *round-down-0*[*simp*]: *round-down p 0* = *0*
⟨*proof*⟩

lemma *round-up-diff-round-down*: *round-up prec x* − *round-down prec x* ≤ *2 powr* −*prec*
⟨*proof*⟩

lemma *round-down-shift*: *round-down p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-down* (*p* + *k*) *x*
⟨*proof*⟩

lemma *round-up-shift*: *round-up p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-up* (*p* + *k*) *x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *round-up-mono*: *x* ≤ *y* ⟹ *round-up p x* ≤ *round-up p y*
⟨*proof*⟩

lemma *round-up-le1*:
assumes *x* ≤ *1 prec* ≥ *0*
shows *round-up prec x* ≤ *1*
⟨*proof*⟩

lemma *round-up-less1*:
assumes *x* < *1* / *2 p* > *0*
shows *round-up p x* < *1*
⟨*proof*⟩

lemma *round-down-ge1*:
assumes *x*: *x* ≥ *1*
assumes *prec*: *p* ≥ − *log 2 x*

shows $1 \leq$ *round-down p x*
⟨*proof*⟩

lemma *round-up-le0*: $x \leq 0 \Longrightarrow$ *round-up p x* ≤ 0
⟨*proof*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

lift-definition *float-up* :: *int* ⇒ *float* ⇒ *float* **is** *round-up* ⟨*proof*⟩
declare *float-up.rep-eq*[*simp*]

lemma *round-up-correct*: *round-up e f* − *f* ∈ {0..2 *powr* −*e*}
⟨*proof*⟩

lemma *float-up-correct*: *real-of-float* (*float-up e f*) − *real-of-float f* ∈ {0..2 *powr* −*e*}
⟨*proof*⟩

lift-definition *float-down* :: *int* ⇒ *float* ⇒ *float* **is** *round-down* ⟨*proof*⟩
declare *float-down.rep-eq*[*simp*]

lemma *round-down-correct*: *f* − (*round-down e f*) ∈ {0..2 *powr* −*e*}
⟨*proof*⟩

lemma *float-down-correct*: *real-of-float f* − *real-of-float* (*float-down e f*) ∈ {0..2 *powr* −*e*}
⟨*proof*⟩

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*⟩

lemma *abs-round-down-le*: $|f - (\text{round-down } e\ f)| \leq 2\ \text{powr}\ -e$
⟨*proof*⟩

lemma *abs-round-up-le*: $|f - (\text{round-up } e\ f)| \leq 2\ \text{powr}\ -e$
⟨*proof*⟩

lemma *round-down-nonneg*: $0 \leq s \Longrightarrow 0 \leq \text{round-down } p\ s$
⟨*proof*⟩

lemma *ceil-divide-floor-conv*:
assumes $b \neq 0$
shows $\lceil \text{real-of-int } a\ /\ \text{real-of-int } b \rceil =$
(*if* b *dvd* a *then* a *div* b *else* $\lfloor \text{real-of-int } a\ /\ \text{real-of-int } b \rfloor + 1$)
⟨*proof*⟩ **lemma** *compute-float-up*[*code*]: *float-up* $p\ x = -\ \text{float-down}\ p\ (-x)$
⟨*proof*⟩

end

lemma *bitlen-Float*:
fixes m e
defines [*THEN meta-eq-to-obj-eq*]: $f \equiv \text{Float } m\ e$
shows *bitlen* $|\text{mantissa } f| + \text{exponent } f = (\text{if } m = 0 \text{ then } 0 \text{ else bitlen } |m| + e)$
⟨*proof*⟩

lemma *float-gt1-scale*:
assumes $1 \leq \text{Float } m\ e$
shows $0 \leq e + (\text{bitlen } m - 1)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *truncate-down-le*: *x* ≤ *y* ⟹ *truncate-down prec x* ≤ *y*
⟨*proof*⟩

lemma *truncate-down-zero*[*simp*]: *truncate-down prec* 0 = 0
⟨*proof*⟩

lemma *truncate-down-float*[*simp*]: *truncate-down p x* ∈ *float*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *truncate-up-le*: *x* ≤ *y* ⟹ *x* ≤ *truncate-up prec y*
⟨*proof*⟩

lemma *truncate-up-zero*[*simp*]: *truncate-up prec* 0 = 0
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *truncate-up-float*[*simp*]: *truncate-up p x* ∈ *float*
⟨*proof*⟩

lemma *mult-powr-eq*: 0 < *b* ⟹ *b* ≠ 1 ⟹ 0 < *x* ⟹ *x* ∗ *b powr y* = *b powr* (*y* + *log b x*)
⟨*proof*⟩

lemma *truncate-down-pos*:
assumes *x* > 0
shows *truncate-down p x* > 0
⟨*proof*⟩

lemma *truncate-down-nonneg*: 0 ≤ *y* ⟹ 0 ≤ *truncate-down prec y*
⟨*proof*⟩

lemma *truncate-down-ge1*: 1 ≤ *x* ⟹ 1 ≤ *truncate-down p x*
⟨*proof*⟩

lemma *truncate-up-nonpos*: *x* ≤ 0 ⟹ *truncate-up prec x* ≤ 0
⟨*proof*⟩

lemma *truncate-up-le1*:
assumes *x* ≤ 1
shows *truncate-up p x* ≤ 1
⟨*proof*⟩

lemma *truncate-down-shift-int*:
truncate-down p (*x* ∗ 2 *powr real-of-int k*) = *truncate-down p x* ∗ 2 *powr k*
⟨*proof*⟩

lemma *truncate-down-shift-nat*: *truncate-down p* (*x* ∗ 2 *powr real k*) = *truncate-down p x* ∗ 2 *powr k*

⟨*proof*⟩

lemma *truncate-up-shift-int*: *truncate-up p* (*x* ∗ *2 powr real-of-int k*) = *truncate-up p x* ∗ *2 powr k*
⟨*proof*⟩

lemma *truncate-up-shift-nat*: *truncate-up p* (*x* ∗ *2 powr real k*) = *truncate-up p x* ∗ *2 powr k*
⟨*proof*⟩

43.10 Truncating Floats

lift-definition *float-round-up* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-up*
⟨*proof*⟩

lemma *float-round-up*: *real-of-float x* ≤ *real-of-float* (*float-round-up prec x*)
⟨*proof*⟩

lemma *float-round-up-zero*[*simp*]: *float-round-up prec 0* = *0*
⟨*proof*⟩

lift-definition *float-round-down* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-down*
⟨*proof*⟩

lemma *float-round-down*: *real-of-float* (*float-round-down prec x*) ≤ *real-of-float x*
⟨*proof*⟩

lemma *float-round-down-zero*[*simp*]: *float-round-down prec 0* = *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩ **lemma** *compute-float-round-up*[*code*]:
float-round-up prec x = − *float-round-down prec* (−*x*)

⟨*proof*⟩

end

lemma *truncate-up-nonneg-mono*:
assumes $0 \leq x$ $x \leq y$
shows *truncate-up prec x* ≤ *truncate-up prec y*
⟨*proof*⟩

lemma *truncate-up-switch-sign-mono*:
assumes $x \leq 0$ $0 \leq y$
shows *truncate-up prec x* ≤ *truncate-up prec y*
⟨*proof*⟩

lemma *truncate-down-switch-sign-mono*:
assumes $x \leq 0$
and $0 \leq y$
and $x \leq y$
shows *truncate-down prec x* ≤ *truncate-down prec y*
⟨*proof*⟩

lemma *truncate-down-nonneg-mono*:
assumes $0 \leq x$ $x \leq y$
shows *truncate-down prec x* ≤ *truncate-down prec y*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *truncate-down-mono*: $x \leq y \Longrightarrow$ *truncate-down p x* ≤ *truncate-down p y*
⟨*proof*⟩

lemma *truncate-up-mono*: $x \leq y \Longrightarrow$ *truncate-up p x* ≤ *truncate-up p y*
⟨*proof*⟩

lemma *truncate-up-nonneg*: $0 \leq$ *truncate-up p x* **if** $0 \leq x$
⟨*proof*⟩

lemma *truncate-up-pos*: $0 <$ *truncate-up p x* **if** $0 < x$
⟨*proof*⟩

lemma *truncate-up-less-zero-iff*[*simp*]: *truncate-up p x* < 0 ⟷ $x < 0$
⟨*proof*⟩

lemma *truncate-up-nonneg-iff*[*simp*]: *truncate-up p x* ≥ 0 ⟷ $x \geq 0$
⟨*proof*⟩

lemma *truncate-down-less-zero-iff*[*simp*]: *truncate-down p x* < 0 ⟷ $x < 0$

⟨*proof*⟩

lemma *truncate-down-nonneg-iff*[*simp*]: *truncate-down p x* ≥ *0* ⟷ *x* ≥ *0*
⟨*proof*⟩

lemma *truncate-down-eq-zero-iff*[*simp*]: *truncate-down prec x* = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *truncate-up-eq-zero-iff*[*simp*]: *truncate-up prec x* = *0* ⟷ *x* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *real-div-nat-eq-floor-of-divide*: *a div b* = *real-of-int* ⌊*a* / *b*⌋ **for** *a b* :: *nat*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lift-definition *lapprox-posrat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float*
is λ*prec* (*x*::*nat*) (*y*::*nat*). *truncate-down prec* (*x* / *y*)
⟨*proof*⟩

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)))
   ⟨proof⟩

end

lift-definition rapprox-posrat :: nat ⇒ nat ⇒ nat ⇒ float
  is λprec (x::nat) (y::nat). truncate-up prec (x / y)
  ⟨proof⟩

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⟩

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⟩

lemma rapprox-posrat-less1:
  0 ≤ x ⟹ 0 < y ⟹ 2 * x < y ⟹ real-of-float (rapprox-posrat n x y) < 1
  ⟨proof⟩

lift-definition lapprox-rat :: nat ⇒ int ⇒ int ⇒ float is
  λprec (x::int) (y::int). truncate-down prec (x / y)
  ⟨proof⟩

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*))))
⟨*proof*⟩

lift-definition *rapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float* **is**
λ*prec* (*x*::*int*) (*y*::*int*). *truncate-up prec* (*x* / *y*)
⟨*proof*⟩

lemma *rapprox-rat* = *rapprox-posrat*
⟨*proof*⟩

lemma *lapprox-rat* = *lapprox-posrat*
⟨*proof*⟩ **lemma** *compute-rapprox-rat*[*code*]:
rapprox-rat prec x y = − *lapprox-rat prec* (−*x*) *y*
⟨*proof*⟩ **lemma** *compute-truncate-down*[*code*]:
truncate-down p (*Ratreal r*) = (*let* (*a*, *b*) = *quotient-of r in lapprox-rat p a b*)
⟨*proof*⟩ **lemma** *compute-truncate-up*[*code*]:
truncate-up p (*Ratreal r*) = (*let* (*a*, *b*) = *quotient-of r in rapprox-rat p a b*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lift-definition *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *real-divr*
⟨*proof*⟩ **lemma** *compute-float-divr*[*code*]:
float-divr prec x y = − *float-divl prec* (−*x*) *y*
⟨*proof*⟩

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*]: $x \in$ *float* $\Longrightarrow$ $y \in$ *float* $\Longrightarrow$ *plus-down p x y* $\in$ *float*
⟨*proof*⟩

lemma *float-plus-up-float*[*intro*, *simp*]: $x \in$ *float* $\Longrightarrow$ $y \in$ *float* $\Longrightarrow$ *plus-up p x y* $\in$ *float*
⟨*proof*⟩

lift-definition *float-plus-down* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-down* ⟨*proof*⟩

lift-definition *float-plus-up* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-up* ⟨*proof*⟩

lemma *plus-down*: *plus-down prec x y* $\leq x + y$
and *plus-up*: $x + y \leq$ *plus-up prec x y*
⟨*proof*⟩

lemma *float-plus-down*: *real-of-float* (*float-plus-down prec x y*) $\leq x + y$
and *float-plus-up*: $x + y \leq$ *real-of-float* (*float-plus-up prec x y*)
⟨*proof*⟩

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* $(-x)$ $(-y)$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *abs-real-le-2-powr-bitlen*[*simp*]: $|$*real-of-int m2*$| <$ *2 powr real-of-int* (*bitlen* $|m2|$)
⟨*proof*⟩

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*)| ≤ *1*
and *leqp*: *q* ≤ *p*
shows ⌊(*a* + *b*) * *2 powr q*⌋ = ⌊(*2* * *ai* + *sgn b*) * *2 powr* (*q* − *p* − *1*)⌋
⟨*proof*⟩

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*⟩

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*⟩

lemma *compute-float-plus-down-naive*: *float-plus-down p x y* = *float-round-down p* (*x* + *y*)
⟨*proof*⟩ **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*⟩ **lemma** *compute-float-plus-up*[*code*]: *float-plus-up p x y* = − *float-plus-down p* (−*x*) (−*y*)
⟨*proof*⟩

lemma *mantissa-zero*: *mantissa 0 = 0*
⟨*proof*⟩ **lemma** *compute-float-less*[*code*]: $a < b \longleftrightarrow$ *is-float-pos* (*float-plus-down 0 b* (− *a*))
⟨*proof*⟩ **lemma** *compute-float-le*[*code*]: $a \leq b \longleftrightarrow$ *is-float-nonneg* (*float-plus-down 0 b* (− *a*))
⟨*proof*⟩

end

lemma *plus-down-mono*: *plus-down p a b* ≤ *plus-down p c d* **if** $a + b \leq c + d$
⟨*proof*⟩

lemma *plus-up-mono*: *plus-up p a b* ≤ *plus-up p c d* **if** $a + b \leq c + d$
⟨*proof*⟩

43.14 Approximate Multiplication

lemma *mult-mono-nonpos-nonneg*: $a * b \leq c * d$
if $a \leq c$ $a \leq 0$ $0 \leq d$ $d \leq b$ **for** *a b c d*::′*a*::*ordered-ring*
⟨*proof*⟩

lemma *mult-mono-nonneg-nonpos*: $b * a \leq d * c$
if $a \leq c$ $c \leq 0$ $0 \leq d$ $d \leq b$ **for** *a b c d*::′*a*::*ordered-ring*
⟨*proof*⟩

lemma *mult-mono-nonpos-nonpos*: $a * b \leq c * d$
if $a \geq c$ $a \leq 0$ $b \geq d$ $d \leq 0$ **for** *a b c d*::*real*
⟨*proof*⟩

lemma *mult-float-mono1*:
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-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*)))
⟨*proof*⟩

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*)))
$\leq$ *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*)))
⟨*proof*⟩

43.15 Approximate Power

lemma *div2-less-self*[*termination-simp*]: *odd n* $\Longrightarrow$ *n div 2* $<$ *n* **for** *n* :: *nat*
⟨*proof*⟩

fun *power-down* :: *nat* $\Rightarrow$ *real* $\Rightarrow$ *nat* $\Rightarrow$ *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* $\Rightarrow$ *real* $\Rightarrow$ *nat* $\Rightarrow$ *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* $\Rightarrow$ *float* $\Rightarrow$ *nat* $\Rightarrow$ *float* **is** *power-up*
⟨*proof*⟩

lift-definition *power-down-fl* :: *nat* $\Rightarrow$ *float* $\Rightarrow$ *nat* $\Rightarrow$ *float* **is** *power-down*
⟨*proof*⟩

lemma *power-float-transfer*[*transfer-rule*]:
(*rel-fun pcr-float* (*rel-fun* (=) *pcr-float*)) (^) (^)
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *power-down-pos*: *0 < x ⟹ 0 < power-down p x n*
⟨*proof*⟩

lemma *power-down-nonneg*: *0 ≤ x ⟹ 0 ≤ power-down p x n*
⟨*proof*⟩

lemma *power-down*: *0 ≤ x ⟹ power-down p x n ≤ x ^ n*
⟨*proof*⟩

lemma *power-up*: *0 ≤ x ⟹ x ^ n ≤ power-up p x n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *power-up-fl*: *0 ≤ x ⟹ x ^ n ≤ power-up-fl p x n*
⟨*proof*⟩

lemma *real-power-up-fl*: *real-of-float (power-up-fl p x n) = power-up p x n*
⟨*proof*⟩

lemma *real-power-down-fl*: *real-of-float (power-down-fl p x n) = power-down p x n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *power-down-eq-zero-iff*[*simp*]: *power-down prec b n = 0 ⟷ b = 0 ∧ n ≠ 0*
⟨*proof*⟩

lemma *power-down-nonneg-iff*[*simp*]:
power-down prec b n ≥ 0 ⟷ even n ∨ b ≥ 0
⟨*proof*⟩

lemma *power-down-neg-iff*[*simp*]:
power-down prec b n $< 0 \longleftrightarrow$
$b < 0 \wedge$ *odd n*
⟨*proof*⟩

lemma *power-down-nonpos-iff*[*simp*]:
notes [*simp del*] = *power-down-neg-iff power-down-eq-zero-iff*
shows *power-down prec b n* $\leq 0 \longleftrightarrow b < 0 \wedge$ *odd n* $\vee\ b = 0 \wedge n \neq 0$
⟨*proof*⟩

lemma *power-down-mono*:
power-down prec a n $\leq$ *power-down prec b n*
if $((0 \leq a \wedge a \leq b) \vee (odd\ n \wedge a \leq b) \vee (even\ n \wedge a \leq 0 \wedge b \leq a))$
⟨*proof*⟩

lemma *power-up-even-nonneg*: *even n* $\Longrightarrow 0 \leq$ *power-up p x n*
⟨*proof*⟩

lemma *power-up-eq-zero-iff*[*simp*]: *power-up prec b n* $= 0 \longleftrightarrow b = 0 \wedge n \neq 0$
⟨*proof*⟩

lemma *power-up-nonneg-iff*[*simp*]:
power-up prec b n $\geq 0 \longleftrightarrow$ *even n* $\vee\ b \geq 0$
⟨*proof*⟩

lemma *power-up-neg-iff*[*simp*]:
power-up prec b n $< 0 \longleftrightarrow b < 0 \wedge$ *odd n*
⟨*proof*⟩

lemma *power-up-nonpos-iff*[*simp*]:
notes [*simp del*] = *power-up-neg-iff power-up-eq-zero-iff*
shows *power-up prec b n* $\leq 0 \longleftrightarrow b < 0 \wedge$ *odd n* $\vee\ b = 0 \wedge n \neq 0$
⟨*proof*⟩

lemma *power-up-mono*:
power-up prec a n $\leq$ *power-up prec b n*
if $((0 \leq a \wedge a \leq b) \vee (odd\ n \wedge a \leq b) \vee (even\ n \wedge a \leq 0 \wedge b \leq a))$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *real-of-Float-int*[*simp*]: *real-of-float (Float n 0) = real n*
⟨*proof*⟩

lemma *float-zero*[*simp*]: *real-of-float (Float 0 e) = 0*
⟨*proof*⟩

lemma *abs-div-2-less*: *a ≠ 0 ⟹ a ≠ −1 ⟹ |(a::int) div 2| < |a|*
⟨*proof*⟩

lemma *lapprox-rat*: *real-of-float (lapprox-rat prec x y) ≤ real-of-int x / real-of-int y*
⟨*proof*⟩

lemma *mult-div-le*:
fixes *a b :: int*
assumes *b > 0*
shows *a ≥ b * (a div b)*
⟨*proof*⟩

lemma *lapprox-rat-nonneg*:
assumes *0 ≤ x* **and** *0 ≤ y*
shows *0 ≤ real-of-float (lapprox-rat n x y)*
⟨*proof*⟩

lemma *rapprox-rat*: *real-of-int x / real-of-int y ≤ real-of-float (rapprox-rat prec x y)*
⟨*proof*⟩

lemma *rapprox-rat-le1*:
assumes *0 ≤ x 0 < y x ≤ y*
shows *real-of-float (rapprox-rat n x y) ≤ 1*
⟨*proof*⟩

lemma *rapprox-rat-nonneg-nonpos*: *0 ≤ x ⟹ y ≤ 0 ⟹ real-of-float (rapprox-rat n x y) ≤ 0*
⟨*proof*⟩

lemma *rapprox-rat-nonpos-nonneg*: *x ≤ 0 ⟹ 0 ≤ y ⟹ real-of-float (rapprox-rat n x y) ≤ 0*
⟨*proof*⟩

lemma *real-divl*: *real-divl prec x y ≤ x / y*
⟨*proof*⟩

lemma *real-divr*: $x \mathbin{/} y \leq$ *real-divr prec x y*
⟨*proof*⟩

lemma *float-divl*: *real-of-float* (*float-divl prec x y*) $\leq x \mathbin{/} y$
⟨*proof*⟩

lemma *real-divl-lower-bound*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq$ *real-divl prec x y*
⟨*proof*⟩

lemma *float-divl-lower-bound*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq$ *real-of-float* (*float-divl prec x y*)
⟨*proof*⟩

lemma *exponent-1*: *exponent 1* $=$ *0*
⟨*proof*⟩

lemma *mantissa-1*: *mantissa 1* $=$ *1*
⟨*proof*⟩

lemma *bitlen-1*: *bitlen 1* $=$ *1*
⟨*proof*⟩

lemma *float-upper-bound*: $x \leq$ *2 powr* (*bitlen* $|$*mantissa x*$|$ $+$ *exponent x*)
⟨*proof*⟩

lemma *real-divl-pos-less1-bound*:
assumes $0 < x$ $x \leq 1$
shows $1 \leq$ *real-divl prec 1 x*
⟨*proof*⟩

lemma *float-divl-pos-less1-bound*:
$0 <$ *real-of-float x* $\Longrightarrow$ *real-of-float x* $\leq 1 \Longrightarrow$ *prec* $\geq 1 \Longrightarrow$
$1 \leq$ *real-of-float* (*float-divl prec 1 x*)
⟨*proof*⟩

lemma *float-divr*: *real-of-float x* $\mathbin{/}$ *real-of-float y* $\leq$ *real-of-float* (*float-divr prec x y*)
⟨*proof*⟩

lemma *real-divr-pos-less1-lower-bound*:
assumes $0 < x$
and $x \leq 1$
shows $1 \leq$ *real-divr prec 1 x*
⟨*proof*⟩

lemma *float-divr-pos-less1-lower-bound*: $0 < x \Longrightarrow x \leq 1 \Longrightarrow 1 \leq$ *float-divr prec 1 x*
⟨*proof*⟩

lemma *real-divr-nonpos-pos-upper-bound*: $x \leq 0 \Longrightarrow 0 \leq y \Longrightarrow$ *real-divr prec x y* ≤ 0
⟨*proof*⟩

lemma *float-divr-nonpos-pos-upper-bound*:
real-of-float x $\leq 0 \Longrightarrow 0 \leq$ *real-of-float y* $\Longrightarrow$ *real-of-float* (*float-divr prec x y*) ≤ 0
⟨*proof*⟩

lemma *real-divr-nonneg-neg-upper-bound*: $0 \leq x \Longrightarrow y \leq 0 \Longrightarrow$ *real-divr prec x y* ≤ 0
⟨*proof*⟩

lemma *float-divr-nonneg-neg-upper-bound*:
$0 \leq$ *real-of-float x* $\Longrightarrow$ *real-of-float y* $\leq 0 \Longrightarrow$ *real-of-float* (*float-divr prec x y*) ≤ 0
⟨*proof*⟩

lemma *Float-le-zero-iff*: *Float a b* $\leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *real-of-float-pprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*pprt a*) = *pprt* (*real-of-float a*)
⟨*proof*⟩

lemma *real-of-float-nprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*nprt a*) = *nprt* (*real-of-float a*)
⟨*proof*⟩

context
begin

lift-definition *int-floor-fl* :: *float* ⇒ *int* **is** *floor* ⟨*proof*⟩ **lemma** *compute-int-floor-fl*[*code*]:
int-floor-fl (*Float m e*) = (*if* $0 \leq e$ *then* $m * 2 \hat{} nat\ e$ *else m div* ($2 \hat{} (nat\ (-e))$))
⟨*proof*⟩

lift-definition *floor-fl* :: *float* ⇒ *float* **is** $\lambda x.$ *real-of-int* $\lfloor x \rfloor$
⟨*proof*⟩ **lemma** *compute-floor-fl*[*code*]:
floor-fl (*Float m e*) = (*if* $0 \leq e$ *then Float m e else Float* (*m div* ($2 \hat{} (nat\ (-e))$)) *0*)
⟨*proof*⟩

end

lemma *floor-fl*: *real-of-float* (*floor-fl x*) $\leq$ *real-of-float x*
⟨*proof*⟩

lemma *int-floor-fl*: *real-of-int* (*int-floor-fl x*) ≤ *real-of-float x*
⟨*proof*⟩

lemma *floor-pos-exp*: *exponent* (*floor-fl x*) ≥ *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *plus-fun-apply* [*simp*]:
(*f* + *g*) *x* = *f x* + *g x*
⟨*proof*⟩

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

definition *0* = (λ*x*. *0*)
instance ⟨*proof*⟩

end

lemma *zero-fun-apply* [*simp*]:
0 x = 0
⟨*proof*⟩

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

definition *f ∗ g* = (λ*x*. *f x ∗ g x*)
instance ⟨*proof*⟩

end

lemma *times-fun-apply* [*simp*]:
(f ∗ g) x = f x ∗ g x
⟨*proof*⟩

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

definition *1* = (λ*x*. *1*)
instance ⟨*proof*⟩

end

lemma *one-fun-apply* [*simp*]:
1 x = 1
⟨*proof*⟩

Additive structures

instance *fun* :: (*type*, *semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *monoid-add*) *monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *group-add*) *group-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-group-add*) *ab-group-add*
⟨*proof*⟩

Multiplicative structures

instance *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *monoid-mult*) *monoid-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*
⟨*proof*⟩

Misc

instance *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* ⟨*proof*⟩

instance *fun* :: (*type*, *mult-zero*) *mult-zero*
⟨*proof*⟩

instance *fun* :: (*type*, *zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

Ring structures

instance *fun* :: (*type*, *semiring*) *semiring*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring*) *comm-semiring*
⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0*) *semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0*) *comm-semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0-cancel*) *semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-1*) *semiring-1* ⟨*proof*⟩

lemma *numeral-fun*:
‹*numeral n* = (λ*x*::'*a*. *numeral n*)›

⟨*proof*⟩

lemma *numeral-fun-apply* [*simp*]:
‹*numeral n x* = *numeral n*›
⟨*proof*⟩

lemma *of-nat-fun*: *of-nat n* = (λ*x*::′*a*. *of-nat n*)
⟨*proof*⟩

lemma *of-nat-fun-apply* [*simp*]:
of-nat n x = *of-nat n*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-1*) *comm-semiring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-1-cancel*) *semiring-1-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-1-cancel*) *comm-semiring-1-cancel*
⟨*proof*⟩

instance *fun* :: (*type*, *semiring-char-0*) *semiring-char-0*
⟨*proof*⟩

instance *fun* :: (*type*, *ring*) *ring* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-ring*) *comm-ring* ⟨*proof*⟩

instance *fun* :: (*type*, *ring-1*) *ring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-ring-1*) *comm-ring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *ring-char-0*) *ring-char-0* ⟨*proof*⟩

Ordered structures

instance *fun* :: (*type*, *ordered-ab-semigroup-add*) *ordered-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-ab-semigroup-add*) *ordered-cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ab-semigroup-add-imp-le*) *ordered-ab-semigroup-add-imp-le*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-monoid-add*) *ordered-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-comm-monoid-add*) *ordered-cancel-comm-monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ab-group-add*) *ordered-ab-group-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-semiring*) *ordered-semiring*
⟨*proof*⟩

instance *fun* :: (*type*, *dioid*) *dioid*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-semiring*) *ordered-comm-semiring*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-semiring*) *ordered-cancel-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-comm-semiring*) *ordered-cancel-comm-semiring*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ring*) *ordered-ring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-ring*) *ordered-comm-ring* ⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *inverse-fun-apply* [*simp*]:
inverse f x = *inverse* (*f x*)
⟨*proof*⟩

lemma *divide-fun-apply* [*simp*]:
$(f\ /\ g)\ x = f\ x\ /\ g\ x$
⟨*proof*⟩

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* :: $('b \Rightarrow 'a{::}field) \Rightarrow bool$ **where**
$zero\text{-}free\ f \equiv \neg\ (\exists x.\ f\ x = 0)$

lemma *fun-left-inverse*:
fixes $f :: 'b \Rightarrow 'a{::}field$
shows $zero\text{-}free\ f \Longrightarrow inverse\ f * f = 1$
⟨*proof*⟩

lemma *fun-right-inverse*:
fixes $f :: 'b \Rightarrow 'a{::}field$
shows $zero\text{-}free\ f \Longrightarrow f * inverse\ f = 1$
⟨*proof*⟩

lemma *fun-divide-inverse*:
fixes $f\ g :: 'b \Rightarrow 'a{::}field$
shows $f\ /\ g = f * inverse\ g$
⟨*proof*⟩

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*]: $(c *_R f)\ x = c *_R (f\ x)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *fun* :: (*type*, *real-vector*) *real-vector*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *less-funE*:
assumes *less-fun f g*
obtains *k* **where** *f k* < *g k* **and** ⋀*k'*. *k'* < *k* ⟹ *f k'* = *g k'*
⟨*proof*⟩

lemma *less-fun-asym*:
assumes *less-fun f g*
shows ¬ *less-fun g f*
⟨*proof*⟩

lemma *less-fun-irrefl*:
¬ *less-fun f f*
⟨*proof*⟩

lemma *less-fun-trans*:
assumes *less-fun f g* **and** *less-fun g h*
shows *less-fun f h*
⟨*proof*⟩

lemma *order-less-fun*:
class.order (λ*f g*. *less-fun f g* ∨ *f* = *g*) *less-fun*
⟨*proof*⟩

lemma *less-fun-trichotomy*:
assumes *finite* {*k*. *f k* ≠ *g k*}

shows *less-fun f g ∨ f = g ∨ less-fun g f*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eventually-going-toI* [*intro*]:
assumes *eventually P F*
shows *eventually* (λ*x*. *P* (*f x*)) (*f going-to F*)
⟨*proof*⟩

lemma *filterlim-going-toI-weak* [*intro*]: *filterlim f F* (*f going-to F within A*)

⟨*proof*⟩

lemma *going-to-mono*: $F \leq G \Longrightarrow A \subseteq B \Longrightarrow f$ *going-to* F *within* $A \leq f$ *going-to* G *within* B
⟨*proof*⟩

lemma *going-to-inf*:
f *going-to* $(inf\ F\ G)$ *within* $A = inf\ (f$ *going-to* F *within* $A)\ (f$ *going-to* G *within* $A)$
⟨*proof*⟩

lemma *going-to-sup*:
f *going-to* $(sup\ F\ G)$ *within* $A \geq sup\ (f$ *going-to* F *within* $A)\ (f$ *going-to* G *within* $A)$
⟨*proof*⟩

lemma *going-to-top* [*simp*]: f *going-to top within* $A = principal\ A$
⟨*proof*⟩

lemma *going-to-bot* [*simp*]: f *going-to bot within* $A = bot$
⟨*proof*⟩

lemma *going-to-principal*:
f *going-to principal* A *within* $B = principal\ (f\ -`\ A \cap B)$
⟨*proof*⟩

lemma *going-to-within-empty* [*simp*]: f *going-to* F *within* $\{\} = bot$
⟨*proof*⟩

lemma *going-to-within-union* [*simp*]:
f *going-to* F *within* $(A \cup B) = sup\ (f$ *going-to* F *within* $A)\ (f$ *going-to* F *within* $B)$
⟨*proof*⟩

lemma *eventually-going-to-at-top-linorder*:
fixes $f :: 'a \Rightarrow 'b :: linorder$
shows $eventually\ P\ (f$ *going-to at-top within* $A) \longleftrightarrow (\exists\, C.\ \forall\, x{\in}A.\ f\ x \geq C \longrightarrow P\ x)$
⟨*proof*⟩

lemma *eventually-going-to-at-bot-linorder*:
fixes $f :: 'a \Rightarrow 'b :: linorder$
shows $eventually\ P\ (f$ *going-to at-bot within* $A) \longleftrightarrow (\exists\, C.\ \forall\, x{\in}A.\ f\ x \leq C \longrightarrow P\ x)$
⟨*proof*⟩

lemma *eventually-going-to-at-top-dense*:
fixes $f :: 'a \Rightarrow 'b :: \{linorder, no\text{-}top\}$
shows $eventually\ P\ (f$ *going-to at-top within* $A) \longleftrightarrow (\exists\, C.\ \forall\, x{\in}A.\ f\ x > C \longrightarrow$

$P\ x$)
⟨*proof*⟩

lemma *eventually-going-to-at-bot-dense*:
fixes $f :: 'a \Rightarrow 'b :: \{linorder, no\text{-}bot\}$
shows *eventually* P (f *going-to at-bot within* A) $\longleftrightarrow$ $(\exists C.\ \forall x{\in}A.\ f\ x < C \longrightarrow P\ x)$
⟨*proof*⟩

lemma *eventually-going-to-nhds*:
eventually P (f *going-to nhds* a *within* A) $\longleftrightarrow$
$(\exists S.\ open\ S \wedge a \in S \wedge (\forall x{\in}A.\ f\ x \in S \longrightarrow P\ x))$
⟨*proof*⟩

lemma *eventually-going-to-at*:
eventually P (f *going-to* (*at* a *within* B) *within* A) $\longleftrightarrow$
$(\exists S.\ open\ S \wedge a \in S \wedge (\forall x{\in}A.\ f\ x \in B \cap S - \{a\} \longrightarrow P\ x))$
⟨*proof*⟩

lemma *norm-going-to-at-top-eq*: *norm going-to at-top* = *at-infinity*
⟨*proof*⟩

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-monoid-set.F* $f\ \mathbf{1}\ g\ \{a.\ g\ a \neq \mathbf{1}\}$

interpretation F: *comm-monoid-set* $f\ \mathbf{1}$
⟨*proof*⟩

lemma *expand-superset*:
assumes *finite* A **and** $\{a.\ g\ a \neq \mathbf{1}\} \subseteq A$
shows $G\ g = F.F\ g\ A$
⟨*proof*⟩

lemma *conditionalize*:
assumes *finite A*
shows $F.F\ g\ A = G\ (\lambda a.\ \textit{if}\ a \in A\ \textit{then}\ g\ a\ \textit{else}\ \mathbf{1})$
⟨*proof*⟩

lemma *neutral* [*simp*]:
$G\ (\lambda a.\ \mathbf{1}) = \mathbf{1}$
⟨*proof*⟩

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*⟩

lemma *infinite* [*simp*]:
$\neg$ *finite* $\{a.\ g\ a \neq \mathbf{1}\} \Longrightarrow G\ g = \mathbf{1}$
⟨*proof*⟩

lemma *cong* [*cong*]:
assumes $\bigwedge a.\ g\ a = h\ a$
shows $G\ g = G\ h$
⟨*proof*⟩

lemma *not-neutral-obtains-not-neutral*:
assumes $G\ g \neq \mathbf{1}$
obtains a **where** $g\ a \neq \mathbf{1}$
⟨*proof*⟩

lemma *reindex-cong*:
assumes *bij l*
assumes $g \circ l = h$
shows $G\ g = G\ h$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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* × *?C* ⊆ *D*)
shows $G\ (\lambda(a,\ b).\ G\ (g\ a\ b)) = G\ (\lambda(a,\ b,\ c).\ g\ a\ b\ c)$
⟨*proof*⟩

lemma *delta* [*simp*]:
$G\ (\lambda b.$ *if* $b = a$ *then* $g\ b$ *else* $\mathbf{1}) = g\ a$
⟨*proof*⟩

lemma *delta′* [*simp*]:
$G\ (\lambda b.$ *if* $a = b$ *then* $g\ b$ *else* $\mathbf{1}) = g\ a$
⟨*proof*⟩

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*⟩

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
$\sum a.\ b \rightleftharpoons$ *CONST Sum-any* $(\lambda a.\ b)$

lemma *Sum-any-left-distrib*:
fixes *r* :: *'a* :: *semiring-0*

assumes *finite* $\{a.\ g\ a \neq 0\}$
shows *Sum-any* $g * r = (\sum n.\ g\ n * r)$
⟨*proof*⟩

lemma *Sum-any-right-distrib*:
fixes $r :: 'a ::$ *semiring-0*
assumes *finite* $\{a.\ g\ a \neq 0\}$
shows $r *$ *Sum-any* $g = (\sum n.\ r * g\ n)$
⟨*proof*⟩

lemma *Sum-any-product*:
fixes $f\ g :: 'b \Rightarrow 'a::$*semiring-0*
assumes *finite* $\{a.\ f\ a \neq 0\}$ **and** *finite* $\{b.\ g\ b \neq 0\}$
shows *Sum-any* $f *$ *Sum-any* $g = (\sum a.\ \sum b.\ f\ a * g\ b)$
⟨*proof*⟩

lemma *Sum-any-eq-zero-iff* [*simp*]:
fixes $f :: 'a \Rightarrow nat$
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows *Sum-any* $f = 0 \longleftrightarrow f = (\lambda\text{-}.\ 0)$
⟨*proof*⟩

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*⟩

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
$\prod a.\ b$ == *CONST Prod-any* $(\lambda a.\ b)$

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*⟩

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 $f\ a \neq 0$
⟨*proof*⟩

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*⟩

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* $Y \Longrightarrow \exists x :: 'a.\ x \notin Y$
⟨*proof*⟩

lemma *arb-finite-subset*: *finite* $Y \Longrightarrow \exists X :: 'a\ set.\ Y \cap X = \{\} \wedge$ *finite* $X \wedge n \leq$ *card* X
⟨*proof*⟩

lemma *arb-inj-on-finite-infinite*: *finite*$(A :: 'b\ set) \Longrightarrow \exists f :: 'b \Rightarrow 'a.$ *inj-on* $f\ A$
⟨*proof*⟩

lemma *arb-countable-map*: *finite* $Y \Longrightarrow \exists f :: (nat \Rightarrow 'a).$ *inj* $f \wedge$ *range* $f \subseteq$ *UNIV* $-$ Y
⟨*proof*⟩

end

instance *nat* :: *infinite*
⟨*proof*⟩

instance *int* :: *infinite*
⟨*proof*⟩

instance *rat* :: *infinite*
⟨*proof*⟩

instance *real* :: *infinite*
⟨*proof*⟩

instance *complex* :: *infinite*
⟨*proof*⟩

instance *option* :: (*infinite*) *infinite*
⟨*proof*⟩

instance *prod* :: (*infinite*, *type*) *infinite*
⟨*proof*⟩

instance *list* :: (*type*) *infinite*
⟨*proof*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *set* :: (*zero*) *zero*
begin

definition *set-zero*[*simp*]: (*0*::'*a*::*zero set*) = {*0*}

instance ⟨*proof*⟩

end

instantiation *set* :: (*one*) *one*
begin

definition *set-one*[*simp*]: (*1*::'*a*::*one set*) = {*1*}

instance ⟨*proof*⟩

end

definition *elt-set-plus* :: '*a*::*plus* ⇒ '*a set* ⇒ '*a set* (**infixl** ‹+*o*› *70*)
where *a* +*o* *B* = {*c*. ∃ *b*∈*B*. *c* = *a* + *b*}

definition *elt-set-times* :: '*a*::*times* ⇒ '*a set* ⇒ '*a set* (**infixl** ‹∗*o*› *80*)
where *a* ∗*o* *B* = {*c*. ∃ *b*∈*B*. *c* = *a* ∗ *b*}

abbreviation (*input*) *elt-set-eq* :: '*a* ⇒ '*a set* ⇒ *bool* (**infix** ‹=*o*› *50*)
where *x* =*o* *A* ≡ *x* ∈ *A*

instance *set* :: (*semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *set* :: (*ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *set* :: (*monoid-add*) *monoid-add*
⟨*proof*⟩

instance *set* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *set* :: (*semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instance *set* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
⟨*proof*⟩

instance *set* :: (*monoid-mult*) *monoid-mult*

⟨*proof*⟩

instance *set* :: (*comm-monoid-mult*) *comm-monoid-mult*
⟨*proof*⟩

lemma *sumset-empty* [*simp*]: $A + \{\} = \{\}$ $\{\} + A = \{\}$
⟨*proof*⟩

lemma *Un-set-plus*: $(A \cup B) + C = (A+C) \cup (B+C)$ **and** *set-plus-Un*: $C + (A \cup B) = (C+A) \cup (C+B)$
⟨*proof*⟩

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*: $B + (insert\ a\ A) = (B+A) \cup (((+)a)\ `\ B)$
⟨*proof*⟩

lemma *set-add-0* [*simp*]:
fixes A :: $'a$::*comm-monoid-add set*
shows $\{0\} + A = A$
⟨*proof*⟩

lemma *set-add-0-right* [*simp*]:
fixes A :: $'a$::*comm-monoid-add set*
shows $A + \{0\} = A$
⟨*proof*⟩

lemma *card-plus-sing*:
fixes A :: $'a$::*ab-group-add set*
shows $card\ (A + \{a\}) = card\ A$
⟨*proof*⟩

lemma *set-plus-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a + b \in C + D$
⟨*proof*⟩

lemma *set-plus-elim*:
assumes $x \in A + B$
obtains $a\ b$ **where** $x = a + b$ **and** $a \in A$ **and** $b \in B$
⟨*proof*⟩

lemma *set-plus-intro2* [*intro*]: $b \in C \Longrightarrow a + b \in a +o C$
⟨*proof*⟩

lemma *set-plus-rearrange*: $(a +o C) + (b +o D) = (a + b) +o (C + D)$
for $a\ b$ :: $'a$::*comm-monoid-add*
⟨*proof*⟩

lemma *set-plus-rearrange2*: $a +o (b +o C) = (a + b) +o C$

for a b :: $'a$::*semigroup-add*
⟨*proof*⟩

lemma *set-plus-rearrange3*: $(a +o B) + C = a +o (B + C)$
for a :: $'a$::*semigroup-add*
⟨*proof*⟩

theorem *set-plus-rearrange4*: $C + (a +o D) = a +o (C + D)$
for a :: $'a$::*comm-monoid-add*
⟨*proof*⟩

lemmas *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2 set-plus-rearrange3 set-plus-rearrange4*

lemma *set-plus-mono* [*intro*!]: $C \subseteq D \Longrightarrow a +o C \subseteq a +o D$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-plus-mono3* [*intro*]: $a \in C \Longrightarrow a +o D \subseteq C + D$
⟨*proof*⟩

lemma *set-plus-mono4* [*intro*]: $a \in C \Longrightarrow a +o D \subseteq D + C$
for a :: $'a$::*comm-monoid-add*
⟨*proof*⟩

lemma *set-plus-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a +o B \subseteq C + D$
⟨*proof*⟩

lemma *set-plus-mono-b*: $C \subseteq D \Longrightarrow x \in a +o C \Longrightarrow x \in a +o D$
⟨*proof*⟩

lemma *set-zero-plus* [*simp*]: $0 +o C = C$
for C :: $'a$::*comm-monoid-add set*
⟨*proof*⟩

lemma *set-zero-plus2*: $0 \in A \Longrightarrow B \subseteq A + B$
for A B :: $'a$::*comm-monoid-add set*
⟨*proof*⟩

lemma *set-plus-imp-minus*: $a \in b +o C \Longrightarrow a - b \in C$
for a b :: $'a$::*ab-group-add*
⟨*proof*⟩

lemma *set-minus-imp-plus*: $a - b \in C \Longrightarrow a \in b +o C$
for a b :: $'a$::*ab-group-add*
⟨*proof*⟩

lemma *set-minus-plus*: $a - b \in C \longleftrightarrow a \in b +o C$
for $a\ b :: 'a$::*ab-group-add*
⟨*proof*⟩

lemma *set-times-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a * b \in C * D$
⟨*proof*⟩

lemma *set-times-elim*:
assumes $x \in A * B$
obtains $a\ b$ **where** $x = a * b$ **and** $a \in A$ **and** $b \in B$
⟨*proof*⟩

lemma *set-times-intro2* [*intro*!]: $b \in C \Longrightarrow a * b \in a *o C$
⟨*proof*⟩

lemma *set-times-rearrange*: $(a *o C) * (b *o D) = (a * b) *o (C * D)$
for $a\ b :: 'a$::*comm-monoid-mult*
⟨*proof*⟩

lemma *set-times-rearrange2*: $a *o (b *o C) = (a * b) *o C$
for $a\ b :: 'a$::*semigroup-mult*
⟨*proof*⟩

lemma *set-times-rearrange3*: $(a *o B) * C = a *o (B * C)$
for $a :: 'a$::*semigroup-mult*
⟨*proof*⟩

theorem *set-times-rearrange4*: $C * (a *o D) = a *o (C * D)$
for $a :: 'a$::*comm-monoid-mult*
⟨*proof*⟩

lemmas *set-times-rearranges* = *set-times-rearrange set-times-rearrange2*
set-times-rearrange3 set-times-rearrange4

lemma *set-times-mono* [*intro*]: $C \subseteq D \Longrightarrow a *o C \subseteq a *o D$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-times-mono3* [*intro*]: $a \in C \Longrightarrow a *o D \subseteq C * D$
⟨*proof*⟩

lemma *set-times-mono4* [*intro*]: $a \in C \Longrightarrow a *o D \subseteq D * C$
for $a :: 'a$::*comm-monoid-mult*
⟨*proof*⟩

lemma *set-times-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a *o B \subseteq C * D$
⟨*proof*⟩

lemma *set-one-times* [*simp*]: $1 *o C = C$
for $C :: 'a::comm\text{-}monoid\text{-}mult\ set$
⟨*proof*⟩

lemma *set-times-plus-distrib*: $a *o (b +o C) = (a * b) +o (a *o C)$
for $a\ b :: 'a::semiring$
⟨*proof*⟩

lemma *set-times-plus-distrib2*: $a *o (B + C) = (a *o B) + (a *o C)$
for $a :: 'a::semiring$
⟨*proof*⟩

lemma *set-times-plus-distrib3*: $(a +o C) * D \subseteq a *o D + C * D$
for $a :: 'a::semiring$
⟨*proof*⟩

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 $a :: 'a::ring\text{-}1$
⟨*proof*⟩

lemma *set-neg-intro2*: $a \in C \Longrightarrow - a \in (-1) *o C$
for $a :: 'a::ring\text{-}1$
⟨*proof*⟩

lemma *set-plus-image*: $S + T = (\lambda(x, y).\ x + y)\ `\ (S \times T)$
⟨*proof*⟩

lemma *set-times-image*: $S * T = (\lambda(x, y).\ x * y)\ `\ (S \times T)$
⟨*proof*⟩

lemma *finite-set-plus*: *finite* $s \Longrightarrow$ *finite* $t \Longrightarrow$ *finite* $(s + t)$
⟨*proof*⟩

lemma *finite-set-times*: *finite* $s \Longrightarrow$ *finite* $t \Longrightarrow$ *finite* $(s * t)$
⟨*proof*⟩

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** $- = ?sum\ I$)
⟨*proof*⟩

lemma *sum-set-cond-linear*:
fixes f :: $'a$::*comm-monoid-add set* ⇒ $'b$::*comm-monoid-add set*
assumes [*intro*!]: $\bigwedge A\ B.\ P\ A \implies P\ B \implies P\ (A + B)\ P\ \{0\}$
and f: $\bigwedge A\ B.\ P\ A \implies P\ B \Longrightarrow f\ (A + B) = f\ A + f\ B\ f\ \{0\} = \{0\}$
assumes *all*: $\bigwedge i.\ i \in I \implies P\ (S\ i)$
shows $f\ (sum\ S\ I) = sum\ (f \circ S)\ I$
⟨*proof*⟩

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 $f\ (sum\ S\ I) = sum\ (f \circ S)\ I$
⟨*proof*⟩

lemma *set-times-Un-distrib*:
$A * (B \cup C) = A * B \cup A * C$
$(A \cup B) * C = A * C \cup B * C$
⟨*proof*⟩

lemma *set-times-UNION-distrib*:
$A * \bigcup (M\ `\ I) = (\bigcup i{\in}I.\ A * M\ i)$
$\bigcup (M\ `\ I) * A = (\bigcup i{\in}I.\ M\ i * A)$
⟨*proof*⟩

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*
⟨*proof*⟩

setup-lifting *type-definition-interval*

lift-definition *lower*::($'a$::*preorder*) *interval* ⇒ $'a$ **is** *fst* ⟨*proof*⟩

lift-definition *upper*::($'a$::*preorder*) *interval* ⇒ $'a$ **is** *snd* ⟨*proof*⟩

lemma *interval-eq-iff*: $a = b \longleftrightarrow lower\ a = lower\ b \wedge upper\ a = upper\ b$
⟨*proof*⟩

lemma *interval-eqI*: $lower\ a = lower\ b \Longrightarrow upper\ a = upper\ b \Longrightarrow a = b$
⟨*proof*⟩

lemma *lower-le-upper*[*simp*]: $lower\ i \leq upper\ i$
⟨*proof*⟩

lift-definition *set-of* :: $'a$::*preorder interval* ⇒ $'a$ *set* **is** $\lambda x.\ \{fst\ x\ ..\ snd\ x\}$ ⟨*proof*⟩

lemma *set-of-eq*: $set\text{-}of\ x = \{lower\ x\ ..\ upper\ x\}$
⟨*proof*⟩

context notes [[*typedef-overloaded*]] **begin**

lift-definition(*code-dt*) *Interval'*::$'a$::*preorder* ⇒ $'a$::*preorder* ⇒ $'a$ *interval option*
is $\lambda a\ b.\ if\ a \leq b\ then\ Some\ (a,\ b)\ else\ None$
⟨*proof*⟩

lemma *Interval'-split*:
$P\ (Interval'\ a\ b) \longleftrightarrow$
$(\forall ivl.\ a \leq b \longrightarrow lower\ ivl = a \longrightarrow upper\ ivl = b \longrightarrow P\ (Some\ ivl)) \wedge (\neg a \leq b \longrightarrow P\ None)$
⟨*proof*⟩

lemma *Interval'-split-asm*:
$P\ (Interval'\ a\ b) \longleftrightarrow$
$\neg((\exists ivl.\ a \leq b \wedge lower\ ivl = a \wedge upper\ ivl = b \wedge \neg P\ (Some\ ivl)) \vee (\neg a \leq b \wedge \neg P\ None))$
⟨*proof*⟩

lemmas *Interval'-splits* = *Interval'-split Interval'-split-asm*

lemma *Interval'-eq-Some*: $Interval'\ a\ b = Some\ i \Longrightarrow lower\ i = a \wedge upper\ i = b$
⟨*proof*⟩

end

instantiation *interval* :: ({*preorder*,*equal*}) *equal*
begin

definition *equal-class.equal a b* ≡ $(lower\ a = lower\ b) \wedge (upper\ a = upper\ b)$

instance ⟨*proof*⟩
end

instantiation *interval* :: (*preorder*) *ord* **begin**

definition *less-eq-interval* :: $'a$ *interval* ⇒ $'a$ *interval* ⇒ *bool*
where *less-eq-interval a b* ⟷ $lower\ b \leq lower\ a \wedge upper\ a \leq upper\ b$

definition *less-interval* :: *'a interval* ⇒ *'a interval* ⇒ *bool*
where *less-interval x y* = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)

instance ⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *lower-sup*[*simp*]: *lower* (*sup A B*) = *inf* (*lower A*) (*lower B*)
⟨*proof*⟩

lemma *upper-sup*[*simp*]: *upper* (*sup A B*) = *sup* (*upper A*) (*upper B*)
⟨*proof*⟩

instance ⟨*proof*⟩
end

lemma *set-of-interval-union*: *set-of A* ∪ *set-of B* ⊆ *set-of* (*sup A B*) **for** *A*::*'a*::*lattice interval*
⟨*proof*⟩

lemma *interval-union-commute*: *sup A B* = *sup B A* **for** *A*::*'a*::*lattice interval*
⟨*proof*⟩

lemma *interval-union-mono1*: *set-of a* ⊆ *set-of* (*sup a A*) **for** *A* :: *'a*::*lattice interval*
⟨*proof*⟩

lemma *interval-union-mono2*: *set-of A* ⊆ *set-of* (*sup a A*) **for** *A* :: *'a*::*lattice interval*
⟨*proof*⟩

lift-definition *interval-of* :: *'a*::*preorder* ⇒ *'a interval* **is** λ*x*. (*x*, *x*)
⟨*proof*⟩

lemma *lower-interval-of*[*simp*]: *lower* (*interval-of a*) = *a*
⟨*proof*⟩

lemma *upper-interval-of*[*simp*]: *upper* (*interval-of a*) = *a*
⟨*proof*⟩

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*)
⟨*proof*⟩
lemma *lower-plus*[*simp*]: *lower* (*plus A B*) = *plus* (*lower A*) (*lower B*)
⟨*proof*⟩
lemma *upper-plus*[*simp*]: *upper* (*plus A B*) = *plus* (*upper A*) (*upper B*)
⟨*proof*⟩

instance ⟨*proof*⟩
end

instance *interval* :: ({*ordered-ab-semigroup-add*, *lattice*}) *ordered-ab-semigroup-add*
⟨*proof*⟩

instantiation *interval* :: ({*preorder*,*zero*}) *zero*
begin

lift-definition *zero-interval*::*'a interval* **is** (*0*, *0*) ⟨*proof*⟩
lemma *lower-zero*[*simp*]: *lower 0* = *0*
⟨*proof*⟩
lemma *upper-zero*[*simp*]: *upper 0* = *0*
⟨*proof*⟩
instance ⟨*proof*⟩
end

instance *interval* :: ({*ordered-comm-monoid-add*}) *comm-monoid-add*
⟨*proof*⟩

instance *interval* :: ({*ordered-comm-monoid-add*,*lattice*}) *ordered-comm-monoid-add*
⟨*proof*⟩

instantiation *interval* :: ({*ordered-ab-group-add*}) *uminus*
begin

lift-definition *uminus-interval*::*'a interval* ⇒ *'a interval* **is** λ(*a*, *b*). (−*b*, −*a*)
⟨*proof*⟩
lemma *lower-uminus*[*simp*]: *lower* (− *A*) = − *upper A*
⟨*proof*⟩
lemma *upper-uminus*[*simp*]: *upper* (− *A*) = − *lower A*
⟨*proof*⟩
instance ⟨*proof*⟩
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*)
⟨*proof*⟩
lemma *upper-minus*[*simp*]: *upper* (*minus A B*) = *minus* (*upper A*) (*lower B*)
⟨*proof*⟩

instance ⟨*proof*⟩
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*))))
⟨*proof*⟩

lemma *lower-times*:
lower (*times A B*) = *Min* {*lower A* ∗ *lower B*, *lower A* ∗ *upper B*, *upper A* ∗ *lower B*, *upper A* ∗ *upper B*}
⟨*proof*⟩

lemma *upper-times*:
upper (*times A B*) = *Max* {*lower A* ∗ *lower B*, *lower A* ∗ *upper B*, *upper A* ∗ *lower B*, *upper A* ∗ *upper B*}
⟨*proof*⟩

instance ⟨*proof*⟩
end

lemma *interval-eq-set-of-iff*: *X* = *Y* ⟷ *set-of X* = *set-of Y* **for** *X Y*::*'a*::*order interval*
⟨*proof*⟩

52.1 Membership

abbreviation (**in** *preorder*) *in-interval* (‹(‹*notation*=‹*infix* ∈$_i$››-/ ∈$_i$ -)› [*51*, *51*] *50*)
where *in-interval x X* ≡ *x* ∈ *set-of X*

lemma *in-interval-to-interval*[*intro*!]: *a* ∈$_i$ *interval-of a*
⟨*proof*⟩

lemma *plus-in-intervalI*:
fixes *x y* :: *'a* :: *ordered-ab-semigroup-add*

shows $x \in_i X \Longrightarrow y \in_i Y \Longrightarrow x + y \in_i X + Y$
⟨*proof*⟩

lemma *connected-set-of*[*intro*, *simp*]:
connected (*set-of X*) **for** *X*::*'a*::*linear-continuum-topology interval*
⟨*proof*⟩

lemma *ex-sum-in-interval-lemma*: $\exists\, xa{\in}\{la\ ..\ ua\}.\ \exists\, xb{\in}\{lb\ ..\ ub\}.\ x = xa + xb$
if $la \le ua$ $lb \le ub$ $la + lb \le x$ $x \le ua + ub$
$ua - la \le ub - lb$
for *la b c d*::*'a*::*linordered-ab-group-add*
⟨*proof*⟩

lemma *ex-sum-in-interval*: $\exists\, xa{\ge}la.\ xa \le ua \wedge (\exists\, xb{\ge}lb.\ xb \le ub \wedge x = xa + xb)$
if *a*: $la \le ua$ **and** *b*: $lb \le ub$ **and** *x*: $la + lb \le x$ $x \le ua + ub$
for *la b c d*::*'a*::*linordered-ab-group-add*
⟨*proof*⟩

lemma *Icc-plus-Icc*:
$\{a\ ..\ b\} + \{c\ ..\ d\} = \{a + c\ ..\ b + d\}$
if $a \le b$ $c \le d$
for *a b c d*::*'a*::*linordered-ab-group-add*
⟨*proof*⟩

lemma *set-of-plus*:
fixes *A* :: *'a*::*linordered-ab-group-add interval*
shows *set-of* $(A + B)$ = *set-of A* + *set-of B*
⟨*proof*⟩

lemma *plus-in-intervalE*:
fixes *xy* :: *'a* :: *linordered-ab-group-add*
assumes $xy \in_i X + Y$
obtains *x y* **where** $xy = x + y$ $x \in_i X$ $y \in_i Y$
⟨*proof*⟩

lemma *set-of-uminus*: *set-of* $(-X) = \{-\ x \mid x.\ x \in$ *set-of X*$\}$
for *X* :: *'a* :: *ordered-ab-group-add interval*
⟨*proof*⟩

lemma *uminus-in-intervalI*:
fixes *x* :: *'a* :: *ordered-ab-group-add*
shows $x \in_i X \Longrightarrow -x \in_i -X$
⟨*proof*⟩

lemma *uminus-in-intervalD*:
fixes *x* :: *'a* :: *ordered-ab-group-add*
shows $x \in_i - X \Longrightarrow - x \in_i X$
⟨*proof*⟩

lemma *minus-in-intervalI*:
fixes $x\ y :: 'a :: ordered\text{-}ab\text{-}group\text{-}add$
shows $x \in_i X \Longrightarrow y \in_i Y \Longrightarrow x - y \in_i X - Y$
⟨*proof*⟩

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 $X\ Y :: 'a :: linordered\text{-}ab\text{-}group\text{-}add\ interval$
⟨*proof*⟩

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*⟩

lemma *times-in-intervalE*:
fixes $xy :: 'a :: \{linorder, real\text{-}normed\text{-}algebra, linear\text{-}continuum\text{-}topology\}$
— TODO: linear continuum topology is pretty strong
assumes $xy \in_i X * Y$
obtains $x\ y$ **where** $xy = x * y\ x \in_i X\ y \in_i Y$
⟨*proof*⟩
thm *times-in-intervalE*[*of 1*::*real*]
lemma *set-of-times*: $set\text{-}of\ (X * Y) = \{x * y \mid x\ y.\ x \in set\text{-}of\ X \wedge y \in set\text{-}of\ Y\}$
for $X\ Y{::}'a :: \{linordered\text{-}ring, real\text{-}normed\text{-}algebra, linear\text{-}continuum\text{-}topology\}$ *interval*
⟨*proof*⟩

instance *interval* :: (*linordered-idom*) *cancel-semigroup-add*
⟨*proof*⟩

lemma *interval-mul-commute*: $A * B = B * A$ **for** $A\ B{::}\ 'a{::}linordered\text{-}idom\ interval$
⟨*proof*⟩

lemma *interval-times-zero-right*[*simp*]: $A * 0 = 0$ **for** $A :: 'a{::}linordered\text{-}ring\ interval$
⟨*proof*⟩

lemma *interval-times-zero-left*[*simp*]:
$0 * A = 0$ **for** $A :: 'a{::}linordered\text{-}ring\ interval$
⟨*proof*⟩

instantiation *interval* :: ({*preorder*,*one*}) *one*
begin

lift-definition *one-interval*::$'a$ *interval* **is** $(1, 1)$ ⟨*proof*⟩
lemma *lower-one*[*simp*]: *lower 1* = *1*

⟨*proof*⟩
lemma *upper-one*[*simp*]: *upper 1 = 1*
⟨*proof*⟩
instance ⟨*proof*⟩
end

instance *interval* :: ({*one*, *preorder*, *linorder*, *times*}) *power*
⟨*proof*⟩

lemma *set-of-one*[*simp*]: *set-of* (*1*::*'a*::{*one*, *order*} *interval*) = {*1*}
⟨*proof*⟩

instance *interval* ::
({*linordered-idom*, *real-normed-algebra*, *linear-continuum-topology*}) *monoid-mult*
⟨*proof*⟩

lemma *one-times-ivl-left*[*simp*]: *1* ∗ *A* = *A* **for** *A* :: *'a*::*linordered-idom interval*
⟨*proof*⟩

lemma *one-times-ivl-right*[*simp*]: *A* ∗ *1* = *A* **for** *A* :: *'a*::*linordered-idom interval*
⟨*proof*⟩

lemma *set-of-power-mono*: *a*^*n* ∈ *set-of* (*A*^*n*) **if** *a* ∈ *set-of A*
for *a* :: *'a*::*linordered-idom*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-of-neg-inc*:
set-of $(-A) \subseteq$ *set-of* $(-A')$
if *set-of* $A \subseteq$ *set-of* A'
for A :: *'a*::*ordered-ab-group-add interval*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-of-distrib-left*:
set-of $(B * (A1 + A2)) \subseteq$ *set-of* $(B * A1 + B * A2)$
for *A1* :: *'a*::*linordered-ring interval*
⟨*proof*⟩

lemma *set-of-distrib-right*:
set-of $((A1 + A2) * B) \subseteq$ *set-of* $(A1 * B + A2 * B)$
for *A1 A2 B* :: *'a*::{*linordered-ring*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
⟨*proof*⟩

lemma *set-of-mul-inc-left*:
set-of $(A * B) \subseteq$ *set-of* $(A' * B)$
if *set-of* $A \subseteq$ *set-of* A'
for A :: *'a*::{*linordered-ring*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

instance *interval* :: ({*linordered-semiring*, *zero*, *times*}) *mult-zero*
⟨*proof*⟩

lift-definition *min-interval*::*'a*::*linorder interval* ⇒ *'a interval* ⇒ *'a interval* **is**
λ(*l1*, *u1*). λ(*l2*, *u2*). (*min l1 l2*, *min u1 u2*)
⟨*proof*⟩
lemma *lower-min-interval*[*simp*]: *lower* (*min-interval x y*) = *min* (*lower x*) (*lower y*)

⟨*proof*⟩
lemma *upper-min-interval*[*simp*]: *upper* (*min-interval x y*) = *min* (*upper x*) (*upper y*)
⟨*proof*⟩

lemma *min-intervalI*:
a ∈$_i$ *A* ⟹ *b* ∈$_i$ *B* ⟹ *min a b* ∈$_i$ *min-interval A B*
⟨*proof*⟩

lift-definition *max-interval*::'*a*::*linorder interval* ⇒ '*a interval* ⇒ '*a interval* **is**
λ(*l1*, *u1*). λ(*l2*, *u2*). (*max l1 l2*, *max u1 u2*)
⟨*proof*⟩
lemma *lower-max-interval*[*simp*]: *lower* (*max-interval x y*) = *max* (*lower x*) (*lower y*)
⟨*proof*⟩
lemma *upper-max-interval*[*simp*]: *upper* (*max-interval x y*) = *max* (*upper x*) (*upper y*)
⟨*proof*⟩

lemma *max-intervalI*:
a ∈$_i$ *A* ⟹ *b* ∈$_i$ *B* ⟹ *max a b* ∈$_i$ *max-interval A B*
⟨*proof*⟩

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*|))
⟨*proof*⟩

lemma *lower-abs-interval*[*simp*]:
lower (*abs-interval x*) = (*if lower x* < *0* ∧ *0* < *upper x then 0 else min* |*lower x*| |*upper x*|)
⟨*proof*⟩
lemma *upper-abs-interval*[*simp*]: *upper* (*abs-interval x*) = *max* |*lower x*| |*upper x*|
⟨*proof*⟩

lemma *in-abs-intervalI1*:
lx < *0* ⟹ *0* < *ux* ⟹ *0* ≤ *xa* ⟹ *xa* ≤ *max* (− *lx*) (*ux*) ⟹ *xa* ∈ *abs* ‘ {*lx*..*ux*}
for *xa*::'*a*::*linordered-idom*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-of-abs-interval*: *set-of* (*abs-interval x*) = *abs* ‘ *set-of x*
⟨*proof*⟩

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*))
  ⟨*proof*⟩
end

lemma *split-domain-nonempty*:
  assumes ⋀*I*. *split I* ≠ []
  shows *split-domain split I* ≠ []
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *split-intervalD*: *split-interval X x* = (*A*, *B*) ⟹ *set-of X* ⊆ *set-of A* ∪ *set-of B*
  ⟨*proof*⟩

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*⟩

end

52.2 Quickcheck

lift-definition *Ivl*::*'a* ⇒ *'a*::*preorder* ⇒ *'a interval* **is** λ*a b*. (*min a b*, *b*)

⟨*proof*⟩

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 (λ*x*. *Quickcheck-Exhaustive.exhaustive* (λ*y*. *f* (*Ivl x y*)) *d*) *d*

instance ⟨*proof*⟩

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
(λ*x*. *Quickcheck-Exhaustive.full-exhaustive* (λ*y*. *f* (*valtermify-interval x y*)) *d*) *d*

instance ⟨*proof*⟩

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*)
(λ*man. scomp* (*Quickcheck-Random.random i*) (λ*exp. Pair* (*valtermify-interval man exp*)))

instance ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mid-le*: *lower i* ≤ *mid i mid i* ≤ *upper i*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lower-round-ivl*[*simp*]: *lower* (*round-interval p x*) = *float-round-down p* (*lower x*)
⟨*proof*⟩
lemma *upper-round-ivl*[*simp*]: *upper* (*round-interval p x*) = *float-round-up p* (*upper x*)
⟨*proof*⟩

lemma *round-ivl-correct*: *set-of A* ⊆ *set-of* (*round-interval prec A*)
⟨*proof*⟩

lift-definition *truncate-ivl* :: *nat* ⇒ *real interval* ⇒ *real interval*
is λ*p*. λ(*l*, *u*). (*truncate-down p l*, *truncate-up p u*)
⟨*proof*⟩

lemma *lower-truncate-ivl*[*simp*]: *lower* (*truncate-ivl p x*) = *truncate-down p* (*lower x*)
⟨*proof*⟩
lemma *upper-truncate-ivl*[*simp*]: *upper* (*truncate-ivl p x*) = *truncate-up p* (*upper x*)
⟨*proof*⟩

lemma *truncate-ivl-correct*: *set-of A* ⊆ *set-of* (*truncate-ivl prec A*)
⟨*proof*⟩

lift-definition *real-interval*::*float interval* ⇒ *real interval*
is λ(*l*, *u*). (*real-of-float l*, *real-of-float u*)
⟨*proof*⟩

lemma *lower-real-interval*[*simp*]: *lower* (*real-interval x*) = *lower x*
⟨*proof*⟩
lemma *upper-real-interval*[*simp*]: *upper* (*real-interval x*) = *upper x*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *real-interval-max-interval*[*simp*]:
real-interval (*max-interval a b*) = *max-interval* (*real-interval a*) (*real-interval b*)
⟨*proof*⟩

lemma *in-intervalI*:
$x \in_i X$ **if** *lower* $X \leq x$ $x \leq$ *upper* X
⟨*proof*⟩

abbreviation *in-real-interval* (‹(‹*notation*=‹*infix* $\in_r$››-/ $\in_r$ -)› [*51*, *51*] *50*)
where $x \in_r X \equiv x \in_i$ *real-interval* X

lemma *in-real-intervalI*:
$x \in_r X$ **if** *lower* $X \leq x$ $x \leq$ *upper* X **for** x::*real* **and** X::*float interval*
⟨*proof*⟩

53.2 intros for *real-interval*

lemma *in-round-intervalI*: $x \in_r A \implies x \in_r$ (*round-interval prec A*)
⟨*proof*⟩

lemma *zero-in-float-intervalI*: $0 \in_r 0$
⟨*proof*⟩

lemma *plus-in-float-intervalI*: $a + b \in_r A + B$ **if** $a \in_r A$ $b \in_r B$
⟨*proof*⟩

lemma *minus-in-float-intervalI*: $a - b \in_r A - B$ **if** $a \in_r A$ $b \in_r B$
⟨*proof*⟩

lemma *uminus-in-float-intervalI*: $-a \in_r -A$ **if** $a \in_r A$
⟨*proof*⟩

lemma *real-interval-times*: *real-interval* $(A * B)$ = *real-interval* A $*$ *real-interval* B
⟨*proof*⟩

lemma *times-in-float-intervalI*: $a * b \in_r A * B$ **if** $a \in_r A$ $b \in_r B$
⟨*proof*⟩

lemma *real-interval-abs*: *real-interval* (*abs-interval A*) = *abs-interval* (*real-interval A*)
⟨*proof*⟩

lemma *abs-in-float-intervalI*: *abs* $a \in_r$ *abs-interval* A **if** $a \in_r A$

⟨*proof*⟩

lemma *interval-of*[*intro*,*simp*]: $x \in_r$ *interval-of* x
⟨*proof*⟩

lemma *split-float-interval-realD*: *split-float-interval* $X = (A, B) \Longrightarrow x \in_r X \Longrightarrow x \in_r A \vee x \in_r B$
⟨*proof*⟩

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
⟨*proof*⟩

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* $I \wedge (\forall i <$ *length* $I.\ x\ !\ i \in_r I\ !\ i)$)

definition *all-subset* :: 'a::*order interval list* ⇒ 'a *interval list* ⇒ *bool*
(**infix** ‹(*all'-subset*)› 50)
where I *all-subset* J = (*length* I = *length* $J \wedge (\forall i <$ *length* $I.$ *set-of* $(I!i) \subseteq$ *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
⟨*proof*⟩

lemma *round-interval-mono*: *set-of* (*round-interval prec* X) ⊆ *set-of* (*round-interval prec* Y)
if *set-of* $X \subseteq$ *set-of* Y
⟨*proof*⟩

lemma *Ivl-simps*[*simp*]: *lower* (*Ivl* a b) = *min* a b *upper* (*Ivl* a b) = b
⟨*proof*⟩

lemma *set-of-subset-iff*: *set-of* $X \subseteq$ *set-of* $Y \longleftrightarrow$ *lower* $Y \leq$ *lower* $X \wedge$ *upper* $X \leq$ *upper* Y
for X Y::'a::*linorder interval*
⟨*proof*⟩

lemma *set-of-subset-iff'*:
set-of a ⊆ *set-of* (*b* :: *'a* :: *linorder interval*) ⟷ *a* ≤ *b*
⟨*proof*⟩

lemma *bounds-of-interval-eq-lower-upper*:
bounds-of-interval ivl = (*lower ivl*, *upper ivl*) **if** *lower ivl* ≤ *upper ivl*
⟨*proof*⟩

lemma *real-interval-Ivl*: *real-interval* (*Ivl a b*) = *Ivl a b*
⟨*proof*⟩

lemma *set-of-mul-contains-real-zero*:
0 ∈$_r$ (*A* ∗ *B*) **if** *0* ∈$_r$ *A* ∨ *0* ∈$_r$ *B*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subdivide-interval-correct*:
list-ex (λ*i*. *x* ∈$_r$ *i*) (*subdivide-interval n I*) **if** *x* ∈$_r$ *I* **for** *x*::*real*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *split-domain-correct*:
fixes *x* :: *real list*

assumes *x all-in I*
assumes *split-correct*: $\bigwedge x\ a\ I.\ x \in_r I \Longrightarrow$ *list-ex* $(\lambda i{::}$*float interval*. $x \in_r i)$ (*split I*)
shows *list-ex* $(\lambda s.$ *x all-in s*) (*split-domain split I*)
⟨*proof*⟩

lift-definition(*code-dt*) *inverse-float-interval*::*nat* ⇒ *float interval* ⇒ *float interval option* **is**
λ*prec* (*l*, *u*). *if* $(0 < l \vee u < 0)$ *then Some* (*float-divl prec 1 u*, *float-divr prec 1 l*) *else None*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *inverse-float-interval*:
inverse ' *set-of* (*real-interval X*) ⊆ *set-of* (*real-interval Y*)
if *inverse-float-interval p X* = *Some Y*
⟨*proof*⟩

lemma *inverse-float-intervalI*:
$x \in_r X \Longrightarrow$ *inverse x* ∈ *set-of'* (*inverse-float-interval p X*)
⟨*proof*⟩

lemma *inverse-float-interval-eqI*: *inverse-float-interval p X* = *Some IVL* $\Longrightarrow x \in_r$ *X* $\Longrightarrow$ *inverse x* $\in_r$ *IVL*
⟨*proof*⟩

lemma *real-interval-abs-interval*[*simp*]:
real-interval (*abs-interval x*) = *abs-interval* (*real-interval x*)
⟨*proof*⟩

lift-definition *floor-float-interval*::*float interval* ⇒ *float interval* **is**
$\lambda(l, u).$ (*floor-fl l*, *floor-fl u*)
⟨*proof*⟩

lemma *lower-floor-float-interval*[*simp*]: *lower* (*floor-float-interval x*) = *floor-fl* (*lower x*)
⟨*proof*⟩
lemma *upper-floor-float-interval*[*simp*]: *upper* (*floor-float-interval x*) = *floor-fl* (*upper x*)
⟨*proof*⟩

lemma *floor-float-intervalI*: $\lfloor x \rfloor \in_r$ *floor-float-interval* X **if** $x \in_r X$
⟨*proof*⟩

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*
⟨*proof*⟩

end

54.2 Generic code equations

lemma [*code*]:
size (as :: 'a iarray) = Suc (IArray.length as)
⟨*proof*⟩

lemma [*code*]:
size-iarray f as = Suc (size-list f (IArray.list-of as))
⟨*proof*⟩

lemma [*code*]:
rec-iarray f as = f (IArray.list-of as)
⟨*proof*⟩

lemma [*code*]:
case-iarray f as = f (IArray.list-of as)
⟨*proof*⟩

lemma [*code*]:
set-iarray as = set (IArray.list-of as)
⟨*proof*⟩

lemma [*code*]:
map-iarray f as = IArray (map f (IArray.list-of as))
⟨*proof*⟩

lemma [*code*]:
rel-iarray r as bs = list-all2 r (IArray.list-of as) (IArray.list-of bs)
⟨*proof*⟩

lemma *list-of-code* [*code*]:
IArray.list-of as = map (λn. as !! n) [0 ..< IArray.length as]
⟨*proof*⟩

lemma [*code*]:
HOL.equal as bs ⟷ HOL.equal (IArray.list-of as) (IArray.list-of bs)
⟨*proof*⟩

lemma [*code*]:
IArray.all p = Not ∘ IArray.exists (Not ∘ p)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩ **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*
⟨*proof*⟩

lemma [*code*]:
as !! *n* = *IArray.sub'* (*as*, *integer-of-nat n*)
⟨*proof*⟩ **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*)
⟨*proof*⟩

lemma [*code*]:
IArray.length as = *nat-of-integer* (*IArray.length' as*)
⟨*proof*⟩ **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* ⟨*proof*⟩

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*⟩
including *integer.lifting* ⟨*proof*⟩

lemma [*code*]:
IArray.exists p as ⟷ exists-upto p (length′ as) as
including *integer.lifting* ⟨*proof*⟩

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*
⟨*proof*⟩

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'*(-'))›)
where *O*(*g*) ≡ *bigo at-top g*

abbreviation *smallo-at-top* (‹(‹*indent=2 notation=‹mixfix smallo›*› *o'*(-'))›)
where *o*(*g*) ≡ *smallo at-top g*

abbreviation *bigomega-at-top* (‹(‹*indent=2 notation=‹mixfix bigomega›*›Ω'(-'))›)
where Ω(*g*) ≡ *bigomega at-top g*

abbreviation *smallomega-at-top* (‹(‹*indent=2 notation=‹mixfix smallomega›*›ω'(-'))›)
where ω(*g*) ≡ *smallomega at-top g*

abbreviation *bigtheta-at-top* (‹(‹*indent=2 notation=‹mixfix bigtheta›*›Θ'(-'))›)
where Θ(*g*) ≡ *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* ≤ *F2* ⟹ *L F2 f* ⊆ *L F1 f*
assumes *in-filtermap-iff*:
f' ∈ *L* (*filtermap h' F'*) *g'* ⟷ (λ*x*. *f'* (*h' x*)) ∈ *L' F'* (λ*x*. *g'* (*h' x*))
assumes *filtercomap*:
f' ∈ *L F''* *g'* ⟹ (λ*x*. *f'* (*h' x*)) ∈ *L'* (*filtercomap h' F''*) (λ*x*. *g'* (*h' x*))
assumes *sup*: *f* ∈ *L F1 g* ⟹ *f* ∈ *L F2 g* ⟹ *f* ∈ *L* (*sup F1 F2*) *g*
assumes *in-cong*: *eventually* (λ*x*. *f x* = *g x*) *F* ⟹ *f* ∈ *L F* (*h*) ⟷ *g* ∈ *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*]: $f \in L\ bot\ g$ ⟨*proof*⟩

lemma *filter-mono*: $F1 \leq F2 \Longrightarrow f \in L\ F2\ g \Longrightarrow f \in L\ F1\ g$
⟨*proof*⟩

lemma *cong-ex*:
eventually $(\lambda x.\ f1\ x = f2\ x)\ F \Longrightarrow$ *eventually* $(\lambda x.\ g1\ x = g2\ x)\ F \Longrightarrow$
$f1 \in L\ F\ (g1) \longleftrightarrow f2 \in L\ F\ (g2)$
⟨*proof*⟩

lemma *cong-ex-bigtheta*:
$f1 \in \Theta[F](f2) \Longrightarrow g1 \in \Theta[F](g2) \Longrightarrow f1 \in L\ F\ (g1) \longleftrightarrow f2 \in L\ F\ (g2)$
⟨*proof*⟩

lemma *bigtheta-trans1*:
$f \in L\ F\ (g) \Longrightarrow g \in \Theta[F](h) \Longrightarrow f \in L\ F\ (h)$
⟨*proof*⟩

lemma *bigtheta-trans2*:
$f \in \Theta[F](g) \Longrightarrow g \in L\ F\ (h) \Longrightarrow f \in L\ F\ (h)$
⟨*proof*⟩

lemma *cmult'* [*simp*]: $c \neq 0 \Longrightarrow L\ F\ (\lambda x.\ f\ x * c) = L\ F\ (f)$
⟨*proof*⟩

lemma *cmult-in-iff'* [*simp*]: $c \neq 0 \Longrightarrow (\lambda x.\ f\ x * c) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
⟨*proof*⟩

lemma *cdiv* [*simp*]: $c \neq 0 \Longrightarrow L\ F\ (\lambda x.\ f\ x\ /\ c) = L\ F\ (f)$

⟨*proof*⟩

lemma *cdiv-in-iff'* [*simp*]: $c \neq 0 \Longrightarrow (\lambda x.\ f\ x\ /\ c) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
⟨*proof*⟩

lemma *uminus* [*simp*]: $L\ F\ (\lambda x.\ -g\ x) = L\ F\ (g)$ ⟨*proof*⟩

lemma *uminus-in-iff* [*simp*]: $(\lambda x.\ -f\ x) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
⟨*proof*⟩

lemma *const*: $c \neq 0 \Longrightarrow L\ F\ (\lambda\text{-}.\ c) = L\ F\ (\lambda\text{-}.\ 1)$
⟨*proof*⟩

lemma *const'* [*simp*]: *NO-MATCH 1 c* $\Longrightarrow c \neq 0 \Longrightarrow L\ F\ (\lambda\text{-}.\ c) = L\ F\ (\lambda\text{-}.\ 1)$
⟨*proof*⟩

lemma *const-in-iff*: $c \neq 0 \Longrightarrow (\lambda\text{-}.\ c) \in L\ F\ (f) \longleftrightarrow (\lambda\text{-}.\ 1) \in L\ F\ (f)$
⟨*proof*⟩

lemma *const-in-iff'* [*simp*]: *NO-MATCH 1 c* $\Longrightarrow c \neq 0 \Longrightarrow (\lambda\text{-}.\ c) \in L\ F\ (f) \longleftrightarrow (\lambda\text{-}.\ 1) \in L\ F\ (f)$
⟨*proof*⟩

lemma *plus-subset2*: $g \in o[F](f) \Longrightarrow L\ F\ (f) \subseteq L\ F\ (\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *mult-right* [*simp*]: $f \in L\ F\ (g) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in L\ F\ (\lambda x.\ g\ x * h\ x)$
⟨*proof*⟩

lemma *mult*: *f1* $\in L\ F\ (g1) \Longrightarrow$ *f2* $\in L\ F\ (g2) \Longrightarrow (\lambda x.\ f1\ x * f2\ x) \in L\ F\ (\lambda x.\ g1\ x * g2\ x)$
⟨*proof*⟩

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*⟩

lemma *divide-right*:
assumes *eventually* $(\lambda x.\ h\ x \neq 0)\ F$
assumes $f \in L\ F\ (g)$
shows $(\lambda x.\ f\ x\ /\ h\ x) \in L\ F\ (\lambda x.\ g\ x\ /\ h\ x)$
⟨*proof*⟩

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*⟩

lemma *divide-left*:
assumes *eventually* (λx. *f x* ≠ *0*) *F*
assumes *eventually* (λx. *g x* ≠ *0*) *F*
assumes *g* ∈ *L F*(*f*)
shows (λx. *h x* / *f x*) ∈ *L F* (λx. *h x* / *g x*)
⟨*proof*⟩

lemma *divide-left-iff*:
assumes *eventually* (λx. *f x* ≠ *0*) *F*
assumes *eventually* (λx. *g x* ≠ *0*) *F*
assumes *eventually* (λx. *h x* ≠ *0*) *F*
shows (λx. *h x* / *f x*) ∈ *L F* (λx. *h x* / *g x*) ⟷ *g* ∈ *L F* (*f*)
⟨*proof*⟩

lemma *divide*:
assumes *eventually* (λx. *g1 x* ≠ *0*) *F*
assumes *eventually* (λx. *g2 x* ≠ *0*) *F*
assumes *f1* ∈ *L F* (*f2*) *g2* ∈ *L F* (*g1*)
shows (λx. *f1 x* / *g1 x*) ∈ *L F* (λx. *f2 x* / *g2 x*)
⟨*proof*⟩

lemma *divide-eq1*:
assumes *eventually* (λx. *h x* ≠ *0*) *F*
shows *f* ∈ *L F* (λx. *g x* / *h x*) ⟷ (λx. *f x* ∗ *h x*) ∈ *L F* (*g*)
⟨*proof*⟩

lemma *divide-eq2*:
assumes *eventually* (λx. *h x* ≠ *0*) *F*
shows (λx. *f x* / *h x*) ∈ *L F* (λx. *g x*) ⟷ *f* ∈ *L F* (λx. *g x* ∗ *h x*)
⟨*proof*⟩

lemma *inverse-eq1*:
assumes *eventually* (λx. *g x* ≠ *0*) *F*
shows *f* ∈ *L F* (λx. *inverse* (*g x*)) ⟷ (λx. *f x* ∗ *g x*) ∈ *L F* (λ-. *1*)
⟨*proof*⟩

lemma *inverse-eq2*:
assumes *eventually* (λx. *f x* ≠ *0*) *F*
shows (λx. *inverse* (*f x*)) ∈ *L F* (*g*) ⟷ (λx. *1*) ∈ *L F* (λx. *f x* ∗ *g x*)
⟨*proof*⟩

lemma *inverse-flip*:
assumes *eventually* (λx. *g x* ≠ *0*) *F*
assumes *eventually* (λx. *h x* ≠ *0*) *F*
assumes (λx. *inverse* (*g x*)) ∈ *L F* (*h*)
shows (λx. *inverse* (*h x*)) ∈ *L F* (*g*)
⟨*proof*⟩

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 $(\lambda x.\ t\ x\ (f\ x)) \in L\ F\ (h)$
⟨*proof*⟩

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 $f \in L\ F\ (\lambda x.\ t\ x\ (h\ x))$
⟨*proof*⟩

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 $(\lambda x.\ t\ x\ (f\ x)) \in L\ F\ (h)$
⟨*proof*⟩

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 $f \in L\ F\ (\lambda x.\ t\ x\ (h\ x))$
⟨*proof*⟩

lemma (**in** *landau-symbol*) *mult-in-1*:
assumes $f \in L\ F\ (\lambda\text{-}.\ 1)$ $g \in L\ F\ (\lambda\text{-}.\ 1)$
shows $(\lambda x.\ f\ x * g\ x) \in L\ F\ (\lambda\text{-}.\ 1)$
⟨*proof*⟩

lemma (**in** *landau-symbol*) *of-real-cancel*:
$(\lambda x.\ of\text{-}real\ (f\ x)) \in L\ F\ (\lambda x.\ of\text{-}real\ (g\ x)) \Longrightarrow f \in Lr\ F\ g$
⟨*proof*⟩

lemma (**in** *landau-symbol*) *of-real-iff*:
$(\lambda x.\ of\text{-}real\ (f\ x)) \in L\ F\ (\lambda x.\ of\text{-}real\ (g\ x)) \longleftrightarrow f \in Lr\ F\ g$
⟨*proof*⟩

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 $(\leq)$
⟨*proof*⟩

interpretation *landau-omega*:
landau-pair bigomega smallomega bigomega smallomega bigomega smallomega $(\geq)$
⟨*proof*⟩

context *landau-pair*
begin

lemmas *R-E* = *disjE* [*OF R*, *case-names le ge*]

lemma *bigI*:
$c > 0 \Longrightarrow eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g\ x)))\ F \Longrightarrow f \in L\ F\ (g)$
⟨*proof*⟩

lemma *bigE*:
assumes $f \in L\ F\ (g)$
obtains c **where** $c > 0\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * (norm\ (g\ x))))\ F$
⟨*proof*⟩

lemma *smallI*:
$(\bigwedge c.\ c > 0 \Longrightarrow eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * (norm\ (g\ x))))\ F) \Longrightarrow f \in l\ F\ (g)$
⟨*proof*⟩

lemma *smallD*:

$f \in l\ F\ (g) \Longrightarrow c > 0 \Longrightarrow$ *eventually* $(\lambda x.\ R\ (norm\ (f\ x))\ (c * (norm\ (g\ x))))\ F$
⟨*proof*⟩

lemma *bigE-nonneg-real*:
assumes $f \in Lr\ F\ (g)$ *eventually* $(\lambda x.\ f\ x \geq 0)\ F$
obtains c **where** $c > 0$ *eventually* $(\lambda x.\ R\ (f\ x)\ (c * |g\ x|))\ F$
⟨*proof*⟩

lemma *smallD-nonneg-real*:
assumes $f \in lr\ F\ (g)$ *eventually* $(\lambda x.\ f\ x \geq 0)\ F\ c > 0$
shows *eventually* $(\lambda x.\ R\ (f\ x)\ (c * |g\ x|))\ F$
⟨*proof*⟩

lemma *small-imp-big*: $f \in l\ F\ (g) \Longrightarrow f \in L\ F\ (g)$
⟨*proof*⟩

lemma *small-subset-big*: $l\ F\ (g) \subseteq L\ F\ (g)$
⟨*proof*⟩

lemma *R-refl* [*simp*]: $R\ x\ x$ ⟨*proof*⟩

lemma *R-linear*: $\neg R\ x\ y \Longrightarrow R\ y\ x$
⟨*proof*⟩

lemma *R-trans* [*trans*]: $R\ a\ b \Longrightarrow R\ b\ c \Longrightarrow R\ a\ c$
⟨*proof*⟩

lemma *R-mult-left-mono*: $R\ a\ b \Longrightarrow c \geq 0 \Longrightarrow R\ (c*a)\ (c*b)$
⟨*proof*⟩

lemma *R-mult-right-mono*: $R\ a\ b \Longrightarrow c \geq 0 \Longrightarrow R\ (a*c)\ (b*c)$
⟨*proof*⟩

lemma *big-trans*:
assumes $f \in L\ F\ (g)\ g \in L\ F\ (h)$
shows $f \in L\ F\ (h)$
⟨*proof*⟩

lemma *big-small-trans*:
assumes $f \in L\ F\ (g)\ g \in l\ F\ (h)$
shows $f \in l\ F\ (h)$
⟨*proof*⟩

lemma *small-big-trans*:
assumes $f \in l\ F\ (g)\ g \in L\ F\ (h)$
shows $f \in l\ F\ (h)$
⟨*proof*⟩

lemma *small-trans*:

$f \in l\ F\ (g) \Longrightarrow g \in l\ F\ (h) \Longrightarrow f \in l\ F\ (h)$
⟨*proof*⟩

lemma *small-big-trans'*:
$f \in l\ F\ (g) \Longrightarrow g \in L\ F\ (h) \Longrightarrow f \in L\ F\ (h)$
⟨*proof*⟩

lemma *big-small-trans'*:
$f \in L\ F\ (g) \Longrightarrow g \in l\ F\ (h) \Longrightarrow f \in L\ F\ (h)$
⟨*proof*⟩

lemma *big-subsetI* [*intro*]: $f \in L\ F\ (g) \Longrightarrow L\ F\ (f) \subseteq L\ F\ (g)$
⟨*proof*⟩

lemma *small-subsetI* [*intro*]: $f \in L\ F\ (g) \Longrightarrow l\ F\ (f) \subseteq l\ F\ (g)$
⟨*proof*⟩

lemma *big-refl* [*simp*]: $f \in L\ F\ (f)$
⟨*proof*⟩

lemma *small-refl-iff*: $f \in l\ F\ (f) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *big-small-asymmetric*: $f \in L\ F\ (g) \Longrightarrow g \in l\ F\ (f) \Longrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *small-big-asymmetric*: $f \in l\ F\ (g) \Longrightarrow g \in L\ F\ (f) \Longrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *small-asymmetric*: $f \in l\ F\ (g) \Longrightarrow g \in l\ F\ (f) \Longrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *plus-aux*:
assumes $f \in o[F](g)$
shows $g \in L\ F\ (\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

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*⟩

lemma *bigomega-iff-bigo*: $g \in \Omega[F](f) \longleftrightarrow f \in O[F](g)$
⟨*proof*⟩

lemma *smallomega-iff-smallo*: $g \in \omega[F](f) \longleftrightarrow f \in o[F](g)$
⟨*proof*⟩

context *landau-pair*
begin

lemma *big-mono*:
eventually $(\lambda x.\ R\ (norm\ (f\ x))\ (norm\ (g\ x)))\ F \Longrightarrow f \in L\ F\ (g)$
⟨*proof*⟩

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*⟩

lemma *small-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*⟩

lemma *big-small-mult*:
$f1 \in L\ F\ (g1) \Longrightarrow f2 \in l\ F\ (g2) \Longrightarrow (\lambda x.\ f1\ x * f2\ x) \in l\ F\ (\lambda x.\ g1\ x * g2\ x)$
⟨*proof*⟩

lemma *small-mult*: $f1 \in l\ F\ (g1) \Longrightarrow f2 \in l\ F\ (g2) \Longrightarrow (\lambda x.\ f1\ x * f2\ x) \in l\ F$
$(\lambda x.\ g1\ x * g2\ x)$
⟨*proof*⟩

lemmas *mult* = *big-mult small-big-mult big-small-mult small-mult*

lemma *big-power*:
assumes $f \in L\ F\ (g)$
shows $(\lambda x.\ f\ x\ \hat{}\ m) \in L\ F\ (\lambda x.\ g\ x\ \hat{}\ m)$
⟨*proof*⟩

lemma (**in** *landau-pair*) *small-power*:
assumes $f \in l\ F\ (g)\ m > 0$
shows $(\lambda x.\ f\ x\ \hat{}\ m) \in l\ F\ (\lambda x.\ g\ x\ \hat{}\ m)$
⟨*proof*⟩

lemma *big-power-increasing*:
assumes $(\lambda\text{-}.\ 1) \in L\ F\ f\ m \leq n$
shows $(\lambda x.\ f\ x\ \hat{}\ m) \in L\ F\ (\lambda x.\ f\ x\ \hat{}\ n)$
⟨*proof*⟩

lemma *small-power-increasing*:
assumes $(\lambda\text{-}.\ 1) \in l\ F\ f\ m < n$
shows $(\lambda x.\ f\ x\ \hat{}\ m) \in l\ F\ (\lambda x.\ f\ x\ \hat{}\ n)$
⟨*proof*⟩

sublocale *big*: *landau-symbol* $L\ L'\ Lr$
⟨*proof*⟩

sublocale *small*: *landau-symbol* $l\ l'\ lr$
⟨*proof*⟩

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'''*: $(\lambda\text{-}.\ 1) \in l\ F\ (g) \Longrightarrow f \in L\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x.\ g\ x * h\ x)$
⟨*proof*⟩

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'''*: $h \in l\ F\ (\lambda\text{-}.\ 1) \Longrightarrow f \in L\ F\ (g) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in l\ F\ (g)$
⟨*proof*⟩

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*⟩

lemma *big-prod*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \in L\ F\ (g\ x)$
shows $(\lambda y.\ \prod x{\in}A.\ f\ x\ y) \in L\ F\ (\lambda y.\ \prod x{\in}A.\ g\ x\ y)$
⟨*proof*⟩

lemma *big-prod-in-1*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \in L\ F\ (\lambda\text{-}.\ 1)$
shows $(\lambda y.\ \prod x{\in}A.\ f\ x\ y) \in L\ F\ (\lambda\text{-}.\ 1)$
⟨*proof*⟩

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*⟩

lemma *plus-absorb2*: $g \in o[F](f) \Longrightarrow L\ F\ (\lambda x.\ f\ x + g\ x) = L\ F\ (f)$
⟨*proof*⟩

lemma *diff-absorb1*: $f \in o[F](g) \Longrightarrow L\ F\ (\lambda x.\ f\ x - g\ x) = L\ F\ (g)$
⟨*proof*⟩

lemma *diff-absorb2*: $g \in o[F](f) \Longrightarrow L\ F\ (\lambda x.\ f\ x - g\ x) = L\ F\ (f)$
⟨*proof*⟩

lemmas *absorb* = *plus-absorb1 plus-absorb2 diff-absorb1 diff-absorb2*

end

lemma *bigthetaI* [*intro*]: $f \in O[F](g) \Longrightarrow f \in \Omega[F](g) \Longrightarrow f \in \Theta[F](g)$
⟨*proof*⟩

lemma *bigthetaD1* [*dest*]: $f \in \Theta[F](g) \Longrightarrow f \in O[F](g)$
and *bigthetaD2* [*dest*]: $f \in \Theta[F](g) \Longrightarrow f \in \Omega[F](g)$
⟨*proof*⟩

lemma *bigtheta-refl* [*simp*]: $f \in \Theta[F](f)$
⟨*proof*⟩

lemma *bigtheta-sym*: $f \in \Theta[F](g) \longleftrightarrow g \in \Theta[F](f)$
⟨*proof*⟩

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*⟩

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* (λ*x*. (*norm* (*f x*)) ≤ *c* ∗ (*norm* (*g x*))) *F*
shows *f* ∈ *O*[*F*](*g*)
⟨*proof*⟩

lemma *smallomegaD* [*dest*]:
assumes *f* ∈ ω[*F*](*g*)
shows *eventually* (λ*x*. (*norm* (*f x*)) ≥ *c* ∗ (*norm* (*g x*))) *F*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bigthetaI-cong*: *eventually* (λ*x*. *f x* = *g x*) *F* ⟹ *f* ∈ Θ[*F*](*g*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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 ∈ *L F* (*g*) ⟹ *h* ∈ Θ[*F*](*g*) ⟹ *f* ∈ *L F* (*h*)
⟨*proof*⟩

lemma (**in** *landau-symbol*) *bigtheta-trans2'*:
g ∈ Θ[*F*](*f*) ⟹ *g* ∈ *L F* (*h*) ⟹ *f* ∈ *L F* (*h*)

⟨*proof*⟩

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*: $f \in \omega[F](g) \Longrightarrow h \in o[F](g) \Longrightarrow f \in \omega[F](h)$
⟨*proof*⟩

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*⟩

lemma *bigtheta-divide*:
assumes $f1 \in \Theta(f2)$ $g1 \in \Theta(g2)$
shows $(\lambda x.\ f1\ x\ /\ g1\ x) \in \Theta(\lambda x.\ f2\ x\ /\ g2\ x)$
⟨*proof*⟩

lemma *eventually-nonzero-bigtheta*:
assumes $f \in \Theta[F](g)$
shows $eventually\ (\lambda x.\ f\ x \neq 0)\ F \longleftrightarrow eventually\ (\lambda x.\ g\ x \neq 0)\ F$
⟨*proof*⟩

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*⟩

lemma *bigoI-tendsto*:
assumes $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow c)\ F$
assumes $eventually\ (\lambda x.\ g\ x \neq 0)\ F$
shows $f \in O[F](g)$
⟨*proof*⟩

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*⟩

lemma *bigomegaI-tendsto*:
assumes *c-not-0*: $(c::real) \neq 0$
assumes *lim*: $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow c)\ F$
shows $f \in \Omega[F](g)$
⟨*proof*⟩

lemma *smallomegaI-filterlim-at-top-norm*:
assumes *lim*: $filterlim\ (\lambda x.\ norm\ (f\ x\ /\ g\ x))\ at\text{-}top\ F$
shows $f \in \omega[F](g)$
⟨*proof*⟩

lemma *smallomegaI-filterlim-at-infinity*:
assumes *lim*: $filterlim\ (\lambda x.\ f\ x\ /\ g\ x)\ at\text{-}infinity\ F$
shows $f \in \omega[F](g)$
⟨*proof*⟩

lemma *smallomegaD-filterlim-at-top-norm*:
assumes $f \in \omega[F](g)$
assumes $eventually\ (\lambda x.\ g\ x \neq 0)\ F$
shows $LIM\ x\ F.\ norm\ (f\ x\ /\ g\ x) :> at\text{-}top$
⟨*proof*⟩

lemma *smallomegaD-filterlim-at-infinity*:
assumes $f \in \omega[F](g)$
assumes $eventually\ (\lambda x.\ g\ x \neq 0)\ F$
shows $LIM\ x\ F.\ f\ x\ /\ g\ x :> at\text{-}infinity$
⟨*proof*⟩

lemma *smallomega-1-conv-filterlim*: $f \in \omega[F](\lambda\text{-}.\ 1) \longleftrightarrow filterlim\ f\ at\text{-}infinity\ F$

⟨*proof*⟩

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*⟩

lemma *smalloD-tendsto*:
assumes $f \in o[F](g)$
shows $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow 0)\ F$
⟨*proof*⟩

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*⟩

lemma *bigthetaI-tendsto*:
assumes *c-not-0*: $(c::real) \neq 0$
assumes *lim*: $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow c)\ F$
shows $f \in \Theta[F](g)$
⟨*proof*⟩

lemma *tendsto-add-smallo*:
assumes $(f1 \longrightarrow a)\ F$
assumes $f2 \in o[F](f1)$
shows $((\lambda x.\ f1\ x + f2\ x) \longrightarrow a)\ F$
⟨*proof*⟩

lemma *tendsto-diff-smallo*:
shows $(f1 \longrightarrow a)\ F \Longrightarrow f2 \in o[F](f1) \Longrightarrow ((\lambda x.\ f1\ x - f2\ x) \longrightarrow a)\ F$
⟨*proof*⟩

lemma *tendsto-add-smallo-iff*:
assumes $f2 \in o[F](f1)$
shows $(f1 \longrightarrow a)\ F \longleftrightarrow ((\lambda x.\ f1\ x + f2\ x) \longrightarrow a)\ F$
⟨*proof*⟩

lemma *tendsto-diff-smallo-iff*:
shows $f2 \in o[F](f1) \Longrightarrow (f1 \longrightarrow a)\ F \longleftrightarrow ((\lambda x.\ f1\ x - f2\ x) \longrightarrow a)\ F$
⟨*proof*⟩

lemma *tendsto-divide-smallo*:
assumes $((\lambda x.\ f1\ x\ /\ g1\ x) \longrightarrow a)\ F$
assumes $f2 \in o[F](f1)\ g2 \in o[F](g1)$
assumes *eventually* $(\lambda x.\ g1\ x \neq 0)\ F$
shows $((\lambda x.\ (f1\ x + f2\ x)\ /\ (g1\ x + g2\ x)) \longrightarrow a)\ F$ (**is** $(?f \longrightarrow -)$ -)

⟨*proof*⟩

lemma *bigo-powr*:
fixes $f :: 'a \Rightarrow real$
assumes $f \in O[F](g)$ $p \geq 0$
shows $(\lambda x.\ |f\ x|\ powr\ p) \in O[F](\lambda x.\ |g\ x|\ powr\ p)$
⟨*proof*⟩

lemma *smallo-powr*:
fixes $f :: 'a \Rightarrow real$
assumes $f \in o[F](g)$ $p > 0$
shows $(\lambda x.\ |f\ x|\ powr\ p) \in o[F](\lambda x.\ |g\ x|\ powr\ p)$
⟨*proof*⟩

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*⟩

lemma *bigtheta-powr*:
fixes $f :: 'a \Rightarrow real$
shows $f \in \Theta[F](g) \Longrightarrow (\lambda x.\ |f\ x|\ powr\ p) \in \Theta[F](\lambda x.\ |g\ x|\ powr\ p)$
⟨*proof*⟩

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*⟩

lemma *zero-in-smallo* [*simp*]: $(\lambda\text{-}.\ 0) \in o[F](f)$
⟨*proof*⟩

lemma *zero-in-bigo* [*simp*]: $(\lambda\text{-}.\ 0) \in O[F](f)$
⟨*proof*⟩

lemma *in-bigomega-zero* [*simp*]: $f \in \Omega[F](\lambda x.\ 0)$
⟨*proof*⟩

lemma *in-smallomega-zero* [*simp*]: $f \in \omega[F](\lambda x.\ 0)$
⟨*proof*⟩

lemma *in-smallo-zero-iff* [*simp*]: $f \in o[F](\lambda\text{-}.\ 0) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)$ F
⟨*proof*⟩

lemma *in-bigo-zero-iff* [*simp*]: $f \in O[F](\lambda\text{-}.\ 0) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *zero-in-smallomega-iff* [*simp*]: $(\lambda\text{-}.\ 0) \in \omega[F](f) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *zero-in-bigomega-iff* [*simp*]: $(\lambda\text{-}.\ 0) \in \Omega[F](f) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *zero-in-bigtheta-iff* [*simp*]: $(\lambda\text{-}.\ 0) \in \Theta[F](f) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

lemma *in-bigtheta-zero-iff* [*simp*]: $f \in \Theta[F](\lambda x.\ 0) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
⟨*proof*⟩

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*]: $(\lambda x.\ f\ x * c) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
⟨*proof*⟩

lemma *bigo-const* [*simp*]: $(\lambda\text{-}.\ c) \in O[F](\lambda\text{-}.\ 1)$ ⟨*proof*⟩

lemma *bigo-const-iff* [*simp*]: $(\lambda\text{-}.\ c1) \in O[F](\lambda\text{-}.\ c2) \longleftrightarrow F = bot \vee c1 = 0 \vee c2 \neq 0$
⟨*proof*⟩

lemma *bigomega-const-iff* [*simp*]: $(\lambda\text{-}.\ c1) \in \Omega[F](\lambda\text{-}.\ c2) \longleftrightarrow F = bot \vee c1 \neq 0 \vee c2 = 0$
⟨*proof*⟩

lemma *smallo-real-nat-transfer*:
$(f :: real \Rightarrow real) \in o(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in o(\lambda x.\ g\ (real\ x))$
⟨*proof*⟩

lemma *bigo-real-nat-transfer*:
$(f :: real \Rightarrow real) \in O(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in O(\lambda x.\ g\ (real\ x))$
⟨*proof*⟩

lemma *smallomega-real-nat-transfer*:
$(f :: real \Rightarrow real) \in \omega(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in \omega(\lambda x.\ g\ (real\ x))$
⟨*proof*⟩

lemma *bigomega-real-nat-transfer*:
$(f :: real \Rightarrow real) \in \Omega(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in \Omega(\lambda x.\ g\ (real\ x))$
⟨*proof*⟩

lemma *bigtheta-real-nat-transfer*:
$(f :: real \Rightarrow real) \in \Theta(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in \Theta(\lambda x.\ g\ (real\ x))$
⟨*proof*⟩

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* $(\lambda x{::}'a{::}linordered\text{-}semidom.\ if\ x = a\ then\ f\ x\ else\ g\ x) = L$ *at-top* (g)
⟨*proof*⟩

lemmas *landau-symbols-if-at-top-eq* [*simp*] = *landau-symbols*[*THEN landau-symbol-if-at-top-eq*]

lemma *sum-in-smallo*:
assumes $f \in o[F](h)$ $g \in o[F](h)$
shows $(\lambda x.\ f\ x + g\ x) \in o[F](h)$ $(\lambda x.\ f\ x - g\ x) \in o[F](h)$
⟨*proof*⟩

lemma *big-sum-in-smallo*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \in o[F](g)$
shows $(\lambda x.\ sum\ (\lambda y.\ f\ y\ x)\ A) \in o[F](g)$
⟨*proof*⟩

lemma *sum-in-bigo*:
assumes $f \in O[F](h)$ $g \in O[F](h)$
shows $(\lambda x.\ f\ x + g\ x) \in O[F](h)$ $(\lambda x.\ f\ x - g\ x) \in O[F](h)$
⟨*proof*⟩

lemma *big-sum-in-bigo*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \in O[F](g)$
shows $(\lambda x.\ sum\ (\lambda y.\ f\ y\ x)\ A) \in O[F](g)$
⟨*proof*⟩

lemma *smallo-multiples*:
assumes *f*: $f \in o(real)$ **and** $k{>}0$
shows $(\lambda n.\ f\ (k * n)) \in o(real)$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *mult-cancel-right*:
assumes *f2* $\in \Theta[F](g2)$ **and** *eventually* $(\lambda x.\ g2\ x \neq 0)\ F$
shows $(\lambda x.\ f1\ x * f2\ x) \in L\ F\ (\lambda x.\ g1\ x * g2\ x) \longleftrightarrow f1 \in L\ F\ (g1)$
⟨*proof*⟩

lemma *divide-cancel-right*:
assumes *f2* $\in \Theta[F](g2)$ **and** *eventually* $(\lambda x.\ g2\ x \neq 0)\ F$
shows $(\lambda x.\ f1\ x\ /\ f2\ x) \in L\ F\ (\lambda x.\ g1\ x\ /\ g2\ x) \longleftrightarrow f1 \in L\ F\ (g1)$
⟨*proof*⟩

lemma *divide-cancel-left*:
assumes *f1* $\in \Theta[F](g1)$ **and** *eventually* $(\lambda x.\ g1\ x \neq 0)\ F$
shows $(\lambda x.\ f1\ x\ /\ f2\ x) \in L\ F\ (\lambda x.\ g1\ x\ /\ g2\ x) \longleftrightarrow$
$(\lambda x.\ inverse\ (f2\ x)) \in L\ F\ (\lambda x.\ inverse\ (g2\ x))$
⟨*proof*⟩

end

lemma *powr-smallo-iff*:
assumes *filterlim g at-top F* $F \neq bot$
shows $(\lambda x.\ g\ x\ powr\ p :: real) \in o[F](\lambda x.\ g\ x\ powr\ q) \longleftrightarrow p < q$
⟨*proof*⟩

lemma *powr-bigo-iff*:
assumes *filterlim g at-top F* $F \neq bot$
shows $(\lambda x.\ g\ x\ powr\ p :: real) \in O[F](\lambda x.\ g\ x\ powr\ q) \longleftrightarrow p \leq q$
⟨*proof*⟩

lemma *powr-bigtheta-iff*:
assumes *filterlim g at-top F F* $\neq$ *bot*
shows $(\lambda x.\ g\ x\ powr\ p :: real) \in \Theta[F](\lambda x.\ g\ x\ powr\ q) \longleftrightarrow p = q$
⟨*proof*⟩

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*: $(\lambda x.\ ln\ (f\ x)) \in o(\lambda x.\ ln\ (g\ x))$
assumes *q*: *q* > *0*
shows $(\lambda x.\ f\ x\ powr\ p) \in o(\lambda x.\ g\ x\ powr\ q)$
⟨*proof*⟩

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*: $(\lambda x.\ ln\ (f\ x)) \in o(\lambda x.\ ln\ (g\ x))$
assumes *q*: *q* < *0*
shows $(\lambda x.\ g\ x\ powr\ q) \in o(\lambda x.\ f\ x\ powr\ p)$
⟨*proof*⟩

55.4 Asymptotic Equivalence

named-theorems *asymp-equiv-intros*
named-theorems *asymp-equiv-simps*

definition *asymp-equiv* :: ('*a* ⇒ ('*b* :: *real-normed-field*)) ⇒ '*a filter* ⇒ ('*a* ⇒ '*b*) ⇒ *bool*
(‹(‹*open-block notation*=‹*mixfix asymp-equiv*››- ∼[-] -)› [*51*, *10*, *51*] *50*)
where *f* ∼[*F*] *g* ⟷ ((λ*x*. *if f x* = *0* ∧ *g x* = *0 then 1 else f x* / *g x*) ⟶ *1*) *F*

abbreviation (*input*) *asymp-equiv-at-top* **where**
asymp-equiv-at-top f g ≡ *f* ∼[*at-top*] *g*

bundle *asymp-equiv-syntax*
begin
notation *asymp-equiv-at-top* (**infix** ‹∼› *50*)

end

lemma *asymp-equivI*: $((\lambda x.\ \mathit{if}\ f\ x = 0 \wedge g\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ f\ x\ /\ g\ x) \longrightarrow 1)\ F \Longrightarrow f \sim[F]\ g$
⟨*proof*⟩

lemma *asymp-equivD*: $f \sim[F]\ g \Longrightarrow ((\lambda x.\ \mathit{if}\ f\ x = 0 \wedge g\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ f\ x\ /\ g\ x) \longrightarrow 1)\ F$
⟨*proof*⟩

lemma *asymp-equiv-filtermap-iff*:
$f \sim[\mathit{filtermap}\ h\ F]\ g \longleftrightarrow (\lambda x.\ f\ (h\ x)) \sim[F]\ (\lambda x.\ g\ (h\ x))$
⟨*proof*⟩

lemma *asymp-equiv-refl* [*simp*, *asymp-equiv-intros*]: $f \sim[F]\ f$
⟨*proof*⟩

lemma *asymp-equiv-symI*:
assumes $f \sim[F]\ g$
shows $g \sim[F]\ f$
⟨*proof*⟩

lemma *asymp-equiv-sym*: $f \sim[F]\ g \longleftrightarrow g \sim[F]\ f$
⟨*proof*⟩

lemma *asymp-equivI'*:
assumes $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow 1)\ F$
shows $f \sim[F]\ g$
⟨*proof*⟩

lemma *tendsto-imp-asymp-equiv-const*:
assumes $(f \longrightarrow c)\ F\ c \neq 0$
shows $f \sim[F]\ (\lambda\text{-}.\ c)$
⟨*proof*⟩

lemma *asymp-equiv-cong*:
assumes *eventually* $(\lambda x.\ f1\ x = f2\ x)\ F$ *eventually* $(\lambda x.\ g1\ x = g2\ x)\ F$
shows $f1 \sim[F]\ g1 \longleftrightarrow f2 \sim[F]\ g2$
⟨*proof*⟩

lemma *asymp-equiv-eventually-zeros*:
fixes $f\ g :: 'a \Rightarrow 'b :: \mathit{real\text{-}normed\text{-}field}$
assumes $f \sim[F]\ g$
shows *eventually* $(\lambda x.\ f\ x = 0 \longleftrightarrow g\ x = 0)\ F$
⟨*proof*⟩

lemma *asymp-equiv-transfer*:
assumes $f1 \sim[F]\ g1$ *eventually* $(\lambda x.\ f1\ x = f2\ x)\ F$ *eventually* $(\lambda x.\ g1\ x = g2\ x)\ F$

shows *f2* $\sim[F]$ *g2*
⟨*proof*⟩

lemma *asymp-equiv-transfer-trans* [*trans*]:
assumes $(\lambda x.\ f\ x\ (h1\ x)) \sim[F] (\lambda x.\ g\ x\ (h1\ x))$
assumes *eventually* $(\lambda x.\ h1\ x = h2\ x)\ F$
shows $(\lambda x.\ f\ x\ (h2\ x)) \sim[F] (\lambda x.\ g\ x\ (h2\ x))$
⟨*proof*⟩

lemma *asymp-equiv-trans* [*trans*]:
fixes *f g h*
assumes $f \sim[F]\ g\ g \sim[F]\ h$
shows $f \sim[F]\ h$
⟨*proof*⟩

lemma *asymp-equiv-trans-lift1* [*trans*]:
assumes $a \sim[F]\ f\ b\ b \sim[F]\ c \bigwedge c\ d.\ c \sim[F]\ d \Longrightarrow f\ c \sim[F]\ f\ d$
shows $a \sim[F]\ f\ c$
⟨*proof*⟩

lemma *asymp-equiv-trans-lift2* [*trans*]:
assumes $f\ a \sim[F]\ b\ a \sim[F]\ c \bigwedge c\ d.\ c \sim[F]\ d \Longrightarrow f\ c \sim[F]\ f\ d$
shows $f\ c \sim[F]\ b$
⟨*proof*⟩

lemma *asymp-equivD-const*:
assumes $f \sim[F] (\lambda\text{-}.\ c)$
shows $(f \longrightarrow c)\ F$
⟨*proof*⟩

lemma *asymp-equiv-refl-ev*:
assumes *eventually* $(\lambda x.\ f\ x = g\ x)\ F$
shows $f \sim[F]\ g$
⟨*proof*⟩

lemma *asymp-equiv-nhds-iff*: $f \sim[nhds\ (z :: 'a :: t1\text{-}space)]\ g \longleftrightarrow f \sim[at\ z]\ g \wedge f\ z = g\ z$
⟨*proof*⟩

lemma *asymp-equiv-sandwich*:
fixes *f g h* :: $'a \Rightarrow 'b$ :: {*real-normed-field*, *order-topology*, *linordered-field*}
assumes *eventually* $(\lambda x.\ f\ x \geq 0)\ F$
assumes *eventually* $(\lambda x.\ f\ x \leq g\ x)\ F$
assumes *eventually* $(\lambda x.\ g\ x \leq h\ x)\ F$
assumes $f \sim[F]\ h$
shows $g \sim[F]\ f\ g \sim[F]\ h$
⟨*proof*⟩

lemma *asymp-equiv-imp-eventually-same-sign*:

fixes $f\ g :: real \Rightarrow real$
assumes $f \sim[F]\ g$
shows *eventually* $(\lambda x.\ sgn\ (f\ x) = sgn\ (g\ x))\ F$
⟨*proof*⟩

lemma
fixes $f\ g :: - \Rightarrow real$
assumes $f \sim[F]\ g$
shows *asymp-equiv-eventually-same-sign*: *eventually* $(\lambda x.\ sgn\ (f\ x) = sgn\ (g\ x))\ F$ (**is** *?th1*)
and *asymp-equiv-eventually-neg-iff*: *eventually* $(\lambda x.\ f\ x < 0 \longleftrightarrow g\ x < 0)\ F$ (**is** *?th2*)
and *asymp-equiv-eventually-pos-iff*: *eventually* $(\lambda x.\ f\ x > 0 \longleftrightarrow g\ x > 0)\ F$ (**is** *?th3*)
⟨*proof*⟩

lemma *asymp-equiv-tendsto-transfer*:
assumes $f \sim[F]\ g$ **and** $(f \longrightarrow c)\ F$
shows $(g \longrightarrow c)\ F$
⟨*proof*⟩

lemma *tendsto-asymp-equiv-cong*:
assumes $f \sim[F]\ g$
shows $(f \longrightarrow c)\ F \longleftrightarrow (g \longrightarrow c)\ F$
⟨*proof*⟩

lemma *smallo-imp-eventually-sgn*:
fixes $f\ g :: real \Rightarrow real$
assumes $g \in o(f)$
shows *eventually* $(\lambda x.\ sgn\ (f\ x + g\ x) = sgn\ (f\ x))$ *at-top*
⟨*proof*⟩

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*⟩

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*⟩

end

lemma *asymp-equiv-add-left* [*asymp-equiv-simps*]:

assumes $h \in o[F](g)$
shows $(\lambda x.\ h\ x + f\ x) \sim[F]\ g \longleftrightarrow f \sim[F]\ g$
⟨*proof*⟩

lemma *asymp-equiv-add-right'* [*asymp-equiv-simps*]:
assumes $h \in o[F](g)$
shows $g \sim[F]\ (\lambda x.\ f\ x + h\ x) \longleftrightarrow g \sim[F]\ f$
⟨*proof*⟩

lemma *asymp-equiv-add-left'* [*asymp-equiv-simps*]:
assumes $h \in o[F](g)$
shows $g \sim[F]\ (\lambda x.\ h\ x + f\ x) \longleftrightarrow g \sim[F]\ f$
⟨*proof*⟩

lemma *smallo-imp-asymp-equiv*:
assumes $(\lambda x.\ f\ x - g\ x) \in o[F](g)$
shows $f \sim[F]\ g$
⟨*proof*⟩

lemma *asymp-equiv-uminus* [*asymp-equiv-intros*]:
$f \sim[F]\ g \Longrightarrow (\lambda x.\ -f\ x) \sim[F]\ (\lambda x.\ -g\ x)$
⟨*proof*⟩

lemma *asymp-equiv-uminus-iff* [*asymp-equiv-simps*]:
$(\lambda x.\ -f\ x) \sim[F]\ g \longleftrightarrow f \sim[F]\ (\lambda x.\ -g\ x)$
⟨*proof*⟩

lemma *asymp-equiv-mult* [*asymp-equiv-intros*]:
fixes *f1 f2 g1 g2* :: *'a* ⇒ *'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*⟩

lemma *asymp-equiv-power* [*asymp-equiv-intros*]:
$f \sim[F]\ g \Longrightarrow (\lambda x.\ f\ x\ \hat{}\ n) \sim[F]\ (\lambda x.\ g\ x\ \hat{}\ n)$
⟨*proof*⟩

lemma *asymp-equiv-inverse* [*asymp-equiv-intros*]:
assumes $f \sim[F]\ g$
shows $(\lambda x.\ inverse\ (f\ x)) \sim[F]\ (\lambda x.\ inverse\ (g\ x))$
⟨*proof*⟩

lemma *asymp-equiv-inverse-iff* [*asymp-equiv-simps*]:
$(\lambda x.\ inverse\ (f\ x)) \sim[F]\ (\lambda x.\ inverse\ (g\ x)) \longleftrightarrow f \sim[F]\ g$
⟨*proof*⟩

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*$)$

⟨*proof*⟩

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*⟩

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*⟩

lemma *asymp-equiv-compose'*:
assumes $f \sim[G] g$ *filterlim* $h\ G\ F$
shows $(\lambda x.\ f\ (h\ x)) \sim[F] (\lambda x.\ g\ (h\ x))$
⟨*proof*⟩

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 $(\lambda x.\ f\ x\ powr\ y) \sim[F] (\lambda x.\ g\ x\ powr\ y)$
⟨*proof*⟩

lemma *asymp-equiv-norm* [*asymp-equiv-intros*]:
fixes $f\ g :: 'a \Rightarrow 'b :: real\text{-}normed\text{-}field$
assumes $f \sim[F] g$
shows $(\lambda x.\ norm\ (f\ x)) \sim[F] (\lambda x.\ norm\ (g\ x))$
⟨*proof*⟩

lemma *asymp-equiv-abs-real* [*asymp-equiv-intros*]:
fixes $f\ g :: 'a \Rightarrow real$
assumes $f \sim[F] g$
shows $(\lambda x.\ |f\ x|) \sim[F] (\lambda x.\ |g\ x|)$
⟨*proof*⟩

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*⟩

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*⟩

lemma *asymp-equiv-imp-bigo*:
assumes $f \sim[F] g$
shows $f \in O[F](g)$
⟨*proof*⟩

lemma *asymp-equiv-imp-bigomega*:
$f \sim[F]\ g \Longrightarrow f \in \Omega[F](g)$
⟨*proof*⟩

lemma *asymp-equiv-imp-bigtheta*:
$f \sim[F]\ g \Longrightarrow f \in \Theta[F](g)$
⟨*proof*⟩

lemma *asymp-equiv-at-infinity-transfer*:
assumes $f \sim[F]\ g$ *filterlim f at-infinity F*
shows *filterlim g at-infinity F*
⟨*proof*⟩

lemma *asymp-equiv-at-top-transfer*:
fixes $f\ g :: - \Rightarrow$ *real*
assumes $f \sim[F]\ g$ *filterlim f at-top F*
shows *filterlim g at-top F*
⟨*proof*⟩

lemma *asymp-equiv-at-bot-transfer*:
fixes $f\ g :: - \Rightarrow$ *real*
assumes $f \sim[F]\ g$ *filterlim f at-bot F*
shows *filterlim g at-bot F*
⟨*proof*⟩

lemma *asymp-equivI'-const*:
assumes $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow c)\ F\ c \neq 0$
shows $f \sim[F]\ (\lambda x.\ c * g\ x)$
⟨*proof*⟩

lemma *asymp-equivI'-inverse-const*:
assumes $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow inverse\ c)\ F\ c \neq 0$
shows $(\lambda x.\ c * f\ x) \sim[F]\ g$
⟨*proof*⟩

lemma *filterlim-at-bot-imp-at-infinity*: *filterlim f at-bot F* $\Longrightarrow$ *filterlim f at-infinity F*
for $f :: - \Rightarrow$ *real* ⟨*proof*⟩

lemma *asymp-equiv-imp-diff-smallo*:
assumes $f \sim[F]\ g$
shows $(\lambda x.\ f\ x - g\ x) \in o[F](g)$
⟨*proof*⟩

lemma *asymp-equiv-altdef*:
$f \sim[F]\ g \longleftrightarrow (\lambda x.\ f\ x - g\ x) \in o[F](g)$
⟨*proof*⟩

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*
⟨*proof*⟩

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 $f \sim[F] g$
⟨*proof*⟩

lemma *asymp-equiv-sandwich-real'*:
fixes *f g l u* :: $'a \Rightarrow$ *real*
assumes $f \sim[F] l\ f \sim[F] u$ *eventually* $(\lambda x.\ g\ x \in \{l\ x..u\ x\})$ *F*
shows $f \sim[F] g$
⟨*proof*⟩

lemma *asymp-equiv-sandwich-real''*:
fixes *f g l u* :: $'a \Rightarrow$ *real*
assumes $l1 \sim[F] u1\ u1 \sim[F] l2\ l2 \sim[F] u2$
eventually $(\lambda x.\ f\ x \in \{l1\ x..u1\ x\})$ *F eventually* $(\lambda x.\ g\ x \in \{l2\ x..u2\ x\})$ *F*
shows $f \sim[F] g$
⟨*proof*⟩

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**
$x \leq y \longleftrightarrow$ (*case x of Bot* ⇒ *True* | *Value x* ⇒ (*case y of Bot* ⇒ *False* | *Value y* ⇒ $x \leq y$))

definition *less-bot* **where**
$x < y \longleftrightarrow$ (*case y of Bot* ⇒ *False* | *Value y* ⇒ (*case x of Bot* ⇒ *True* | *Value x* ⇒ $x < y$))

lemma *less-eq-bot-Bot* [*simp*]: *Bot* $\leq x$
⟨*proof*⟩

lemma *less-eq-bot-Bot-code* [*code*]: $Bot \leq x \longleftrightarrow True$
⟨*proof*⟩

lemma *less-eq-bot-Bot-is-Bot*: $x \leq Bot \Longrightarrow x = Bot$
⟨*proof*⟩

lemma *less-eq-bot-Value-Bot* [*simp*, *code*]: $Value\ x \leq Bot \longleftrightarrow False$
⟨*proof*⟩

lemma *less-eq-bot-Value* [*simp*, *code*]: $Value\ x \leq Value\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *less-bot-Bot* [*simp*, *code*]: $x < Bot \longleftrightarrow False$
⟨*proof*⟩

lemma *less-bot-Bot-is-Value*: $Bot < x \Longrightarrow \exists z.\ x = Value\ z$
⟨*proof*⟩

lemma *less-bot-Bot-Value* [*simp*]: $Bot < Value\ x$
⟨*proof*⟩

lemma *less-bot-Bot-Value-code* [*code*]: $Bot < Value\ x \longleftrightarrow True$
⟨*proof*⟩

lemma *less-bot-Value* [*simp*, *code*]: $Value\ x < Value\ y \longleftrightarrow x < y$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *bot* :: (*order*) *order*
⟨*proof*⟩

instance *bot* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *bot* :: (*order*) *bot*
begin
definition *bot* = *Bot*
instance ⟨*proof*⟩
end

instantiation *bot* :: (*top*) *top*
begin
definition *top* = *Value top*
instance ⟨*proof*⟩
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
⟨*proof*⟩

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
⟨*proof*⟩

end

instance *bot* :: (*lattice*) *bounded-lattice-bot*
⟨*proof*⟩

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*]: $x \leq Top$
⟨*proof*⟩

lemma *less-eq-top-Top-code* [*code*]: $x \leq Top \longleftrightarrow True$
⟨*proof*⟩

lemma *less-eq-top-is-Top*: $Top \leq x \Longrightarrow x = Top$
⟨*proof*⟩

lemma *less-eq-top-Top-Value* [*simp*, *code*]: $Top \leq Value\ x \longleftrightarrow False$
⟨*proof*⟩

lemma *less-eq-top-Value-Value* [*simp*, *code*]: $Value\ x \leq Value\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *less-top-Top* [*simp*, *code*]: $Top < x \longleftrightarrow False$
⟨*proof*⟩

lemma *less-top-Top-is-Value*: $x < Top \Longrightarrow \exists z.\ x = Value\ z$
⟨*proof*⟩

lemma *less-top-Value-Top* [*simp*]: $Value\ x < Top$
⟨*proof*⟩

lemma *less-top-Value-Top-code* [*code*]: $Value\ x < Top \longleftrightarrow True$
⟨*proof*⟩

lemma *less-top-Value* [*simp*, *code*]: $Value\ x < Value\ y \longleftrightarrow x < y$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *top* :: (*order*) *order*
⟨*proof*⟩

instance *top* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *top* :: (*order*) *top*
begin

```
  definition top = Top
  instance ⟨proof⟩
end

instantiation top :: (bot) bot
begin
  definition bot = Value bot
  instance ⟨proof⟩
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
  ⟨proof⟩

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
  ⟨proof⟩

end

instance top :: (lattice) bounded-lattice-top
  ⟨proof⟩
```

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*
⟨*proof*⟩

lemma [*simp*]: *y* ≤ *Top*
⟨*proof*⟩

lemma *greater-than-two-values*:
assumes *a* ≠ *b Value a* ≤ *z Value b* ≤ *z*
shows *z* = *Top*
⟨*proof*⟩

lemma *lesser-than-two-values*:
assumes *a* ≠ *b z* ≤ *Value a z* ≤ *Value b*
shows *z* = *Bot*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *flat-complete-lattice* :: (*type*) *bot*
begin

```
  definition bot = Bot
  instance ⟨proof⟩
end

instantiation flat-complete-lattice :: (type) top
begin
  definition top = Top
  instance ⟨proof⟩
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
  ⟨proof⟩

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 $\exists\, v.\ A - \{Bot\} = \{Value\ v\}$ *then Value* (*THE v.* $A - \{Bot\} = \{Value\ v\}$)
else Top)

definition *Inf-flat-complete-lattice*
where
Inf A =
(*if* $A = \{\} \vee A = \{Top\}$ *then Top*
else if $\exists\, v.\ A - \{Top\} = \{Value\ v\}$ *then Value* (*THE v.* $A - \{Top\} = \{Value\ v\}$)
else Bot)

instance
⟨*proof*⟩

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* ⟷ $x \in$ *sset s*

lemma *smember-code*[*code*, *simp*]: *smember x* (*y* ## *s*) = (*if x* = *y then True else smember x s*)
⟨*proof*⟩

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 $P\ y\ s$
⟨*proof*⟩

lemma *smap-ctr*: $smap\ f\ s = x\ \#\#\ s' \longleftrightarrow f\ (shd\ s) = x \wedge smap\ f\ (stl\ s) = s'$
⟨*proof*⟩

57.1 prepend list to stream

primrec $shift :: 'a\ list \Rightarrow 'a\ stream \Rightarrow 'a\ stream$ (**infixr** ‹@−› 65) **where**
$shift\ []\ s = s$
| $shift\ (x\ \#\ xs)\ s = x\ \#\#\ shift\ xs\ s$

lemma *smap-shift*[*simp*]: $smap\ f\ (xs\ @-\ s) = map\ f\ xs\ @-\ smap\ f\ s$
⟨*proof*⟩

lemma *shift-append*[*simp*]: $(xs\ @\ ys)\ @-\ s = xs\ @-\ ys\ @-\ s$
⟨*proof*⟩

lemma *shift-simps*[*simp*]:
$shd\ (xs\ @-\ s) = (if\ xs = []\ then\ shd\ s\ else\ hd\ xs)$
$stl\ (xs\ @-\ s) = (if\ xs = []\ then\ stl\ s\ else\ tl\ xs\ @-\ s)$
⟨*proof*⟩

lemma *sset-shift*[*simp*]: $sset\ (xs\ @-\ s) = set\ xs \cup sset\ s$
⟨*proof*⟩

lemma *shift-left-inj*[*simp*]: $xs\ @-\ s1 = xs\ @-\ s2 \longleftrightarrow s1 = s2$
⟨*proof*⟩

57.2 set of streams with elements in some fixed set

context
notes [[*inductive-internals*]]
begin

coinductive-set
streams :: $'a\ set \Rightarrow 'a\ stream\ set$
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*: $stl\ s \in streams\ S \Longrightarrow shd\ s \in S \Longrightarrow s \in streams\ S$
⟨*proof*⟩

lemma *streamsE*: $s \in streams\ A \Longrightarrow (shd\ s \in A \Longrightarrow stl\ s \in streams\ A \Longrightarrow P) \Longrightarrow P$

⟨*proof*⟩

lemma *Stream-image*: $x \mathbin{\#\#} y \in ((\#\#)\ x')\ `\ Y \longleftrightarrow x = x' \wedge y \in Y$
⟨*proof*⟩

lemma *shift-streams*: $[\![w \in lists\ A;\ s \in streams\ A]\!] \Longrightarrow w\ @{-}\ s \in streams\ A$
⟨*proof*⟩

lemma *streams-Stream*: $x \mathbin{\#\#} s \in streams\ A \longleftrightarrow x \in A \wedge s \in streams\ A$
⟨*proof*⟩

lemma *streams-stl*: $s \in streams\ A \Longrightarrow stl\ s \in streams\ A$
⟨*proof*⟩

lemma *streams-shd*: $s \in streams\ A \Longrightarrow shd\ s \in A$
⟨*proof*⟩

lemma *sset-streams*:
assumes $sset\ s \subseteq A$
shows $s \in streams\ A$
⟨*proof*⟩

lemma *streams-sset*:
assumes $s \in streams\ A$
shows $sset\ s \subseteq A$
⟨*proof*⟩

lemma *streams-iff-sset*: $s \in streams\ A \longleftrightarrow sset\ s \subseteq A$
⟨*proof*⟩

lemma *streams-mono*: $s \in streams\ A \Longrightarrow A \subseteq B \Longrightarrow s \in streams\ B$
⟨*proof*⟩

lemma *streams-mono2*: $S \subseteq T \Longrightarrow streams\ S \subseteq streams\ T$
⟨*proof*⟩

lemma *smap-streams*: $s \in streams\ A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow smap\ f\ s \in streams\ B$
⟨*proof*⟩

lemma *streams-empty*: $streams\ \{\} = \{\}$
⟨*proof*⟩

lemma *streams-UNIV*[*simp*]: $streams\ UNIV = UNIV$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *snth-smap*[*simp*]: *smap f s !! n = f (s !! n)*
⟨*proof*⟩

lemma *shift-snth-less*[*simp*]: *p < length xs ⟹ (xs @− s) !! p = xs ! p*
⟨*proof*⟩

lemma *shift-snth-ge*[*simp*]: *p ≥ length xs ⟹ (xs @− s) !! p = s !! (p − length xs)*
⟨*proof*⟩

lemma *shift-snth*: *(xs @− s) !! n = (if n < length xs then xs ! n else s !! (n − length xs))*
⟨*proof*⟩

lemma *snth-sset*[*simp*]: *s !! n ∈ sset s*
⟨*proof*⟩

lemma *sset-range*: *sset s = range (snth s)*
⟨*proof*⟩

lemma *streams-iff-snth*: *s ∈ streams X ⟷ (∀ n. s !! n ∈ X)*
⟨*proof*⟩

lemma *snth-in*: *s ∈ streams X ⟹ s !! n ∈ X*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *stake-smap*[*simp*]: *stake n (smap f s) = map f (stake n s)*
⟨*proof*⟩

lemma *take-stake*: *take n (stake m s) = stake (min n m) s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sdrop-smap*[*simp*]: *sdrop n* (*smap f s*) = *smap f* (*sdrop n s*)
⟨*proof*⟩

lemma *sdrop-stl*: *sdrop n* (*stl s*) = *stl* (*sdrop n s*)
⟨*proof*⟩

lemma *drop-stake*: *drop n* (*stake m s*) = *stake* (*m* − *n*) (*sdrop n s*)
⟨*proof*⟩

lemma *stake-sdrop*: *stake n s* @− *sdrop n s* = *s*
⟨*proof*⟩

lemma *id-stake-snth-sdrop*:
s = *stake i s* @− *s* !! *i* ## *sdrop* (*Suc i*) *s*
⟨*proof*⟩

lemma *smap-alt*: *smap f s* = *s'* ⟷ (∀ *n*. *f* (*s* !! *n*) = *s'* !! *n*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *stake-invert-Nil*[*iff*]: *stake n s* = [] ⟷ *n* = *0*
⟨*proof*⟩

lemma *sdrop-shift*: *sdrop i* (*w* @− *s*) = *drop i w* @− *sdrop* (*i* − *length w*) *s*
⟨*proof*⟩

lemma *stake-shift*: *stake i* (*w* @− *s*) = *take i w* @ *stake* (*i* − *length w*) *s*
⟨*proof*⟩

lemma *stake-add*[*simp*]: *stake m s* @ *stake n* (*sdrop m s*) = *stake* (*m* + *n*) *s*
⟨*proof*⟩

lemma *sdrop-add*[*simp*]: *sdrop n* (*sdrop m s*) = *sdrop* (*m* + *n*) *s*
⟨*proof*⟩

lemma *sdrop-snth*: *sdrop n s* !! *m* = *s* !! (*n* + *m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sdrop-while-sdrop-LEAST*:

assumes $\exists\, n.\ P\ (s\ !!\ n)$
shows *sdrop-while* $(Not \circ P)\ s = sdrop\ (LEAST\ n.\ P\ (s\ !!\ n))\ s$
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *stream-all-shift*[*simp*]: *stream-all P* (*xs @− s*) = (*list-all P xs* ∧ *stream-all P s*)
⟨*proof*⟩

lemma *stream-all-Stream*: *stream-all P* (*x ## X*) ⟷ *P x* ∧ *stream-all P X*
⟨*proof*⟩

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*⟩

lemma *cycle-Cons*[*code*]: *cycle* (*x # xs*) = *x ## cycle* (*xs* @ [*x*])
⟨*proof*⟩

lemma *cycle-rotated*: ⟦*v* ≠ []; *cycle u* = *v @− s*⟧ ⟹ *cycle* (*tl u* @ [*hd u*]) = *tl v @− s*
⟨*proof*⟩

lemma *stake-append*: *stake n* (*u @− s*) = *take* (*min* (*length u*) *n*) *u* @ *stake* (*n* − *length u*) *s*
⟨*proof*⟩

lemma *stake-cycle-le*[*simp*]:
assumes *u* ≠ [] *n* < *length u*
shows *stake n* (*cycle u*) = *take n u*
⟨*proof*⟩

lemma *stake-cycle-eq*[*simp*]: $u \neq [] \Longrightarrow stake\ (length\ u)\ (cycle\ u) = u$
⟨*proof*⟩

lemma *sdrop-cycle-eq*[*simp*]: $u \neq [] \Longrightarrow sdrop\ (length\ u)\ (cycle\ u) = cycle\ u$
⟨*proof*⟩

lemma *stake-cycle-eq-mod-0*[*simp*]: $[\![u \neq []; n\ mod\ length\ u = 0]\!] \Longrightarrow$
$stake\ n\ (cycle\ u) = concat\ (replicate\ (n\ div\ length\ u)\ u)$
⟨*proof*⟩

lemma *sdrop-cycle-eq-mod-0*[*simp*]: $[\![u \neq []; n\ mod\ length\ u = 0]\!] \Longrightarrow$
$sdrop\ n\ (cycle\ u) = cycle\ u$
⟨*proof*⟩

lemma *stake-cycle*: $u \neq [] \Longrightarrow$
$stake\ n\ (cycle\ u) = concat\ (replicate\ (n\ div\ length\ u)\ u)\ @\ take\ (n\ mod\ length\ u)\ u$
⟨*proof*⟩

lemma *sdrop-cycle*: $u \neq [] \Longrightarrow sdrop\ n\ (cycle\ u) = cycle\ (rotate\ (n\ mod\ length\ u)\ u)$
⟨*proof*⟩

lemma *sset-cycle*[*simp*]:
assumes $xs \neq []$
shows $sset\ (cycle\ xs) = set\ xs$
⟨*proof*⟩

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*: $stake\ (Suc\ n)\ s = stake\ n\ s\ @\ [s\ !!\ n]$
⟨*proof*⟩

lemma *snth-siterate*[*simp*]: $siterate\ f\ x\ !!\ n = (f\widehat{\ }n)\ x$
⟨*proof*⟩

lemma *sdrop-siterate*[*simp*]: $sdrop\ n\ (siterate\ f\ x) = siterate\ f\ ((f\widehat{\ }n)\ x)$
⟨*proof*⟩

lemma *stake-siterate*[*simp*]: $stake\ n\ (siterate\ f\ x) = map\ (\lambda n.\ (f\widehat{\ }n)\ x)\ [0\ ..<\ n]$
⟨*proof*⟩

lemma *sset-siterate*: $sset\ (siterate\ f\ x) = \{(f\widehat{\ }n)\ x \mid n.\ True\}$
⟨*proof*⟩

lemma *smap-siterate*: *smap f* (*siterate f x*) = *siterate f* (*f x*)
⟨*proof*⟩

57.7 stream repeating a single element

abbreviation *sconst* ≡ *siterate id*

lemma *shift-replicate-sconst*[*simp*]: *replicate n x* @− *sconst x* = *sconst x*
⟨*proof*⟩

lemma *sset-sconst*[*simp*]: *sset* (*sconst x*) = {*x*}
⟨*proof*⟩

lemma *sconst-alt*: *s* = *sconst x* ⟷ *sset s* = {*x*}
⟨*proof*⟩

lemma *sconst-cycle*: *sconst x* = *cycle* [*x*]
⟨*proof*⟩

lemma *smap-sconst*: *smap f* (*sconst x*) = *sconst* (*f x*)
⟨*proof*⟩

lemma *sconst-streams*: *x* ∈ *A* ⟹ *sconst x* ∈ *streams A*
⟨*proof*⟩

lemma *streams-empty-iff*: *streams S* = {} ⟷ *S* = {}
⟨*proof*⟩

57.8 stream of natural numbers

abbreviation *fromN* ≡ *siterate Suc*

abbreviation *nats* ≡ *fromN 0*

lemma *sset-fromN*[*simp*]: *sset* (*fromN n*) = {*n* ..}
⟨*proof*⟩

lemma *stream-smap-fromN*: *s* = *smap* (λ*j*. *let i* = *j* − *n in s* !! *i*) (*fromN n*)
⟨*proof*⟩

lemma *stream-smap-nats*: *s* = *smap* (*snth s*) *nats*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *flat-Stream*[*simp*]: *xs* ≠ [] ⟹ *flat* (*xs* ## *ws*) = *xs* @− *flat ws*
⟨*proof*⟩

lemma *flat-unfold*: *shd ws* ≠ [] ⟹ *flat ws* = *shd ws* @− *flat* (*stl ws*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *sset-flat*[*simp*]: ∀ *xs* ∈ *sset s*. *xs* ≠ [] ⟹
sset (*flat s*) = (⋃ *xs* ∈ *sset s*. *set xs*) (**is** *?P* ⟹ *?L* = *?R*)
⟨*proof*⟩

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* ⟹ *stake n s* ! *m* = *s* !! *m*
⟨*proof*⟩

lemma *snth-sset-smerge*: *ss* !! *n* !! *m* ∈ *sset* (*smerge ss*)
⟨*proof*⟩

lemma *sset-smerge*: *sset* (*smerge ss*) = ⋃(*sset* ' (*sset ss*))
⟨*proof*⟩

57.11 product of two streams

definition *sproduct* :: *'a stream* ⇒ *'b stream* ⇒ (*'a* × *'b*) *stream* **where**
sproduct s1 s2 = *smerge* (*smap* (λ*x*. *smap* (*Pair x*) *s2*) *s1*)

lemma *sset-sproduct*: *sset* (*sproduct s1 s2*) = *sset s1* × *sset s2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sinterleave-snth*[*simp*]:
even n ⟹ *sinterleave s1 s2* !! *n* = *s1* !! (*n div 2*)
odd n ⟹ *sinterleave s1 s2* !! *n* = *s2* !! (*n div 2*)
⟨*proof*⟩

lemma *sset-sinterleave*: *sset* (*sinterleave s1 s2*) = *sset s1* ∪ *sset s2*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *snth-szip*[*simp*]: *szip s1 s2* !! *n* = (*s1* !! *n*, *s2* !! *n*)
⟨*proof*⟩

lemma *stake-szip*[*simp*]:
stake n (*szip s1 s2*) = *zip* (*stake n s1*) (*stake n s2*)
⟨*proof*⟩

lemma *sdrop-szip*[*simp*]: *sdrop n* (*szip s1 s2*) = *szip* (*sdrop n s1*) (*sdrop n s2*)
⟨*proof*⟩

lemma *smap-szip-fst*:
smap (λ*x*. *f* (*fst x*)) (*szip s1 s2*) = *smap f s1*
⟨*proof*⟩

lemma *smap-szip-snd*:
smap (λ*x*. *g* (*snd x*)) (*szip s1 s2*) = *smap g s2*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *smap2-szip*:
smap2 f s1 s2 = *smap* (*case-prod f*) (*szip s1 s2*)
⟨*proof*⟩

lemma *smap-smap2*[*simp*]:
smap f (*smap2 g s1 s2*) = *smap2* (λ*x y*. *f* (*g x y*)) *s1 s2*
⟨*proof*⟩

lemma *smap2-alt*:
(*smap2 f s1 s2* = *s*) = (∀ *n*. *f* (*s1* !! *n*) (*s2* !! *n*) = *s* !! *n*)
⟨*proof*⟩

lemma *snth-smap2*[*simp*]:
smap2 f s1 s2 !! *n* = *f* (*s1* !! *n*) (*s2* !! *n*)
⟨*proof*⟩

lemma *stake-smap2*[*simp*]:
stake n (*smap2 f s1 s2*) = *map* (*case-prod f*) (*zip* (*stake n s1*) (*stake n s2*))
⟨*proof*⟩

lemma *sdrop-smap2*[*simp*]:
sdrop n (*smap2 f s1 s2*) = *smap2 f* (*sdrop n s1*) (*sdrop n s2*)
⟨*proof*⟩

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*
⟨*proof*⟩

interpretation *prefix-order*: *order prefix strict-prefix*
⟨*proof*⟩

global-interpretation *prefix-bot*: *ordering-top* ‹λ*xs ys*. *prefix ys xs*› ‹λ*xs ys*. *strict-prefix ys xs*› ‹[]›
⟨*proof*⟩

interpretation *prefix-bot*: *order-bot Nil prefix strict-prefix*

⟨*proof*⟩

lemma *prefixI* [*intro?*]: *ys* = *xs* @ *zs* ⟹ *prefix xs ys*
⟨*proof*⟩

lemma *prefixE* [*elim?*]:
assumes *prefix xs ys*
obtains *zs* **where** *ys* = *xs* @ *zs*
⟨*proof*⟩

lemma *strict-prefixI'* [*intro?*]: *ys* = *xs* @ *z* # *zs* ⟹ *strict-prefix xs ys*
⟨*proof*⟩

lemma *strict-prefixE'* [*elim?*]:
assumes *strict-prefix xs ys*
obtains *z zs* **where** *ys* = *xs* @ *z* # *zs*
⟨*proof*⟩

lemma *strict-prefixI* [*intro?*]: *prefix xs ys* ⟹ *xs* ≠ *ys* ⟹ *strict-prefix xs ys*
⟨*proof*⟩

lemma *strict-prefixE* [*elim?*]:
fixes *xs ys* :: *'a list*
assumes *strict-prefix xs ys*
obtains *prefix xs ys* **and** *xs* ≠ *ys*
⟨*proof*⟩

58.2 Basic properties of prefixes

theorem *Nil-prefix* [*simp*]: *prefix* [] *xs*
⟨*proof*⟩

theorem *prefix-Nil* [*simp*]: (*prefix xs* []) = (*xs* = [])
⟨*proof*⟩

lemma *prefix-snoc* [*simp*]: *prefix xs* (*ys* @ [*y*]) ⟷ *xs* = *ys* @ [*y*] ∨ *prefix xs ys*
⟨*proof*⟩

lemma *Cons-prefix-Cons* [*simp*]: *prefix* (*x* # *xs*) (*y* # *ys*) = (*x* = *y* ∧ *prefix xs ys*)
⟨*proof*⟩

lemma *prefix-code* [*code*]:
prefix [] *xs* ⟷ *True*
prefix (*x* # *xs*) [] ⟷ *False*
prefix (*x* # *xs*) (*y* # *ys*) ⟷ *x* = *y* ∧ *prefix xs ys*
⟨*proof*⟩

lemma *same-prefix-prefix* [*simp*]: *prefix* (*xs* @ *ys*) (*xs* @ *zs*) = *prefix ys zs*

⟨*proof*⟩

lemma *same-prefix-nil* [*simp*]: *prefix* (*xs* @ *ys*) *xs* = (*ys* = [])
⟨*proof*⟩

lemma *prefix-prefix* [*simp*]: *prefix xs ys* ⟹ *prefix xs* (*ys* @ *zs*)
⟨*proof*⟩

lemma *append-prefixD*: *prefix* (*xs* @ *ys*) *zs* ⟹ *prefix xs zs*
⟨*proof*⟩

theorem *prefix-Cons*: *prefix xs* (*y* # *ys*) = (*xs* = [] ∨ (∃ *zs*. *xs* = *y* # *zs* ∧ *prefix zs ys*))
⟨*proof*⟩

theorem *prefix-append*:
prefix xs (*ys* @ *zs*) = (*prefix xs ys* ∨ (∃ *us*. *xs* = *ys* @ *us* ∧ *prefix us zs*))
⟨*proof*⟩

lemma *append-one-prefix*:
prefix xs ys ⟹ *length xs* < *length ys* ⟹ *prefix* (*xs* @ [*ys* ! *length xs*]) *ys*
⟨*proof*⟩

theorem *prefix-length-le*: *prefix xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *prefix-same-cases*:
prefix (*xs*$_1$::'*a list*) *ys* ⟹ *prefix xs*$_2$ *ys* ⟹ *prefix xs*$_1$ *xs*$_2$ ∨ *prefix xs*$_2$ *xs*$_1$
⟨*proof*⟩

lemma *prefix-length-prefix*:
prefix ps xs ⟹ *prefix qs xs* ⟹ *length ps* ≤ *length qs* ⟹ *prefix ps qs*
⟨*proof*⟩

lemma *set-mono-prefix*: *prefix xs ys* ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *take-is-prefix*: *prefix* (*take n xs*) *xs*
⟨*proof*⟩

lemma *takeWhile-is-prefix*: *prefix* (*takeWhile P xs*) *xs*
⟨*proof*⟩

lemma *prefixeq-butlast*: *prefix* (*butlast xs*) *xs*
⟨*proof*⟩

lemma *prefix-map-rightE*:
assumes *prefix xs* (*map f ys*)
shows ∃ *xs'*. *prefix xs' ys* ∧ *xs* = *map f xs'*

⟨*proof*⟩

lemma *map-mono-prefix*: *prefix xs ys* ⟹ *prefix* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *filter-mono-prefix*: *prefix xs ys* ⟹ *prefix* (*filter P xs*) (*filter P ys*)
⟨*proof*⟩

lemma *sorted-antimono-prefix*: *prefix xs ys* ⟹ *sorted ys* ⟹ *sorted xs*
⟨*proof*⟩

lemma *prefix-length-less*: *strict-prefix xs ys* ⟹ *length xs* < *length ys*
⟨*proof*⟩

lemma *prefix-snocD*: *prefix* (*xs*@[*x*]) *ys* ⟹ *strict-prefix xs ys*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *take-strict-prefix*: *strict-prefix xs ys* ⟹ *strict-prefix* (*take n xs*) *ys*
⟨*proof*⟩

lemma *prefix-takeWhile*:
assumes *prefix xs ys*
shows *prefix* (*takeWhile P xs*) (*takeWhile P ys*)
⟨*proof*⟩

lemma *prefix-dropWhile*:
assumes *prefix xs ys*
shows *prefix* (*dropWhile P xs*) (*dropWhile P ys*)
⟨*proof*⟩

lemma *prefix-remdups-adj*:
assumes *prefix xs ys*
shows *prefix* (*remdups-adj xs*) (*remdups-adj ys*)
⟨*proof*⟩

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*⟩

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* ⟨*proof*⟩

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*⟩

lemma *length-prefixes*[*simp*]: *length* (*prefixes xs*) = *length xs*+*1*
⟨*proof*⟩

lemma *distinct-prefixes* [*intro*]: *distinct* (*prefixes xs*)
⟨*proof*⟩

lemma *prefixes-snoc* [*simp*]: *prefixes* (*xs*@[*x*]) = *prefixes xs* @ [*xs*@[*x*]]
⟨*proof*⟩

lemma *prefixes-not-Nil* [*simp*]: *prefixes xs* ≠ []
⟨*proof*⟩

lemma *hd-prefixes* [*simp*]: *hd* (*prefixes xs*) = []
⟨*proof*⟩

lemma *last-prefixes* [*simp*]: *last* (*prefixes xs*) = *xs*
⟨*proof*⟩

lemma *prefixes-append*:
prefixes (*xs* @ *ys*) = *prefixes xs* @ *map* (λ*ys*′. *xs* @ *ys*′) (*tl* (*prefixes ys*))
⟨*proof*⟩

lemma *prefixes-eq-snoc*:
prefixes ys = *xs* @ [*x*] ⟷
(*ys* = [] ∧ *xs* = [] ∨ (∃ *z zs*. *ys* = *zs*@[*z*] ∧ *xs* = *prefixes zs*)) ∧ *x* = *ys*
⟨*proof*⟩

lemma *prefixes-tailrec* [*code*]:
prefixes xs = *rev* (*snd* (*foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, *rev* (*x*#*acc1*)#*acc2*))
([],[[]]) *xs*))
⟨*proof*⟩

lemma *set-prefixes-eq*: *set* (*prefixes xs*) = {*ys*. *prefix ys xs*}

⟨*proof*⟩

lemma *card-set-prefixes* [*simp*]: *card* (*set* (*prefixes xs*)) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *set-prefixes-append*:
set (*prefixes* (*xs* @ *ys*)) = *set* (*prefixes xs*) ∪ {*xs* @ *ys'* |*ys'*. *ys'* ∈ *set* (*prefixes ys*)}
⟨*proof*⟩

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*⟩

lemma *Longest-common-prefix-unique*:
‹∃! *ps*. (∀ *xs* ∈ *L*. *prefix ps xs*) ∧ (∀ *qs*. (∀ *xs* ∈ *L*. *prefix qs xs*) ⟶ *length qs* ≤ *length ps*)›
if ‹*L* ≠ {}›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Longest-common-prefix-prefix*:
xs ∈ *L* ⟹ *prefix* (*Longest-common-prefix L*) *xs*
⟨*proof*⟩

lemma *Longest-common-prefix-longest*:
L ≠ {} ⟹ ∀ *xs*∈*L*. *prefix ps xs* ⟹ *length ps* ≤ *length*(*Longest-common-prefix L*)
⟨*proof*⟩

lemma *Longest-common-prefix-max-prefix*:
L ≠ {} ⟹ ∀ *xs*∈*L*. *prefix ps xs* ⟹ *prefix ps* (*Longest-common-prefix L*)
⟨*proof*⟩

lemma *Longest-common-prefix-Nil*: [] ∈ *L* ⟹ *Longest-common-prefix L* = []
⟨*proof*⟩

lemma *Longest-common-prefix-image-Cons*:
assumes $L \neq \{\}$
shows *Longest-common-prefix* $((\#)\ x\ `\ L) = x\ \#$ *Longest-common-prefix* L
⟨*proof*⟩

lemma *Longest-common-prefix-eq-Cons*: **assumes** $L \neq \{\}$ $[] \notin L$ $\forall xs{\in}L.\ hd\ xs = x$
shows *Longest-common-prefix* $L = x\ \#$ *Longest-common-prefix* $\{ys.\ x\#ys \in L\}$
⟨*proof*⟩

lemma *Longest-common-prefix-eq-Nil*:
$[\![x\#ys \in L;\ y\#zs \in L;\ x \neq y\ ]\!] \Longrightarrow$ *Longest-common-prefix* $L = []$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *longest-common-prefix-prefix2*:
prefix (*longest-common-prefix xs ys*) *ys*
⟨*proof*⟩

lemma *longest-common-prefix-max-prefix*:
[[*prefix ps xs*; *prefix ps ys*]]
⟹ *prefix ps* (*longest-common-prefix xs ys*)
⟨*proof*⟩

58.5 Parallel lists

definition *parallel* :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** ‹‖› *50*)
where $(xs \parallel ys) = (\neg$ *prefix xs ys* $\wedge \neg$ *prefix ys xs*$)$

lemma *parallelI* [*intro*]: $\neg$ *prefix xs ys* $\Longrightarrow \neg$ *prefix ys xs* $\Longrightarrow xs \parallel ys$
⟨*proof*⟩

lemma *parallelE* [*elim*]:
assumes $xs \parallel ys$
obtains $\neg$ *prefix xs ys* $\wedge \neg$ *prefix ys xs*
⟨*proof*⟩

theorem *prefix-cases*:
obtains *prefix xs ys* | *strict-prefix ys xs* | $xs \parallel ys$
⟨*proof*⟩

lemma *parallel-cancel*: $a\#xs \parallel a\#ys \Longrightarrow xs \parallel ys$
⟨*proof*⟩

theorem *parallel-decomp*:
$xs \parallel ys \Longrightarrow \exists\, as\ b\ bs\ c\ cs.\ b \neq c \wedge xs = as\ @\ b\ \#\ bs \wedge ys = as\ @\ c\ \#\ cs$
⟨*proof*⟩

lemma *parallel-append*: $a \parallel b \Longrightarrow a\ @\ c \parallel b\ @\ d$
⟨*proof*⟩

lemma *parallel-appendI*: $xs \parallel ys \Longrightarrow x = xs\ @\ xs' \Longrightarrow y = ys\ @\ ys' \Longrightarrow x \parallel y$
⟨*proof*⟩

lemma *parallel-commute*: $a \parallel b \longleftrightarrow b \parallel a$
⟨*proof*⟩

58.6 Suffix order on lists

definition *suffix* :: *'a list ⇒ 'a list ⇒ bool*
where $suffix\ xs\ ys = (\exists\, zs.\ ys = zs\ @\ xs)$

definition *strict-suffix* :: *'a list ⇒ 'a list ⇒ bool*
where $strict\text{-}suffix\ xs\ ys \longleftrightarrow suffix\ xs\ ys \wedge xs \neq ys$

global-interpretation *suffix-order*: *ordering suffix strict-suffix*
⟨*proof*⟩

interpretation *suffix-order*: *order suffix strict-suffix*
⟨*proof*⟩

global-interpretation *suffix-bot*: *ordering-top ‹λxs ys. suffix ys xs› ‹λxs ys. strict-suffix ys xs› ‹[]›*
⟨*proof*⟩

interpretation *suffix-bot*: *order-bot Nil suffix strict-suffix*
⟨*proof*⟩

lemma *suffixI* [*intro?*]: $ys = zs\ @\ xs \Longrightarrow suffix\ xs\ ys$
⟨*proof*⟩

lemma *suffixE* [*elim?*]:
assumes *suffix xs ys*
obtains *zs* **where** $ys = zs\ @\ xs$
⟨*proof*⟩

lemma *suffix-tl* [*simp*]: $suffix\ (tl\ xs)\ xs$
⟨*proof*⟩

lemma *strict-suffix-tl* [*simp*]: $xs \neq [] \Longrightarrow strict\text{-}suffix\ (tl\ xs)\ xs$

⟨*proof*⟩

lemma *Nil-suffix* [*simp*]: *suffix* [] *xs*
⟨*proof*⟩

lemma *suffix-Nil* [*simp*]: (*suffix xs* []) = (*xs* = [])
⟨*proof*⟩

lemma *suffix-ConsI*: *suffix xs ys* ⟹ *suffix xs* (*y* # *ys*)
⟨*proof*⟩

lemma *suffix-ConsD*: *suffix* (*x* # *xs*) *ys* ⟹ *suffix xs ys*
⟨*proof*⟩

lemma *suffix-appendI*: *suffix xs ys* ⟹ *suffix xs* (*zs* @ *ys*)
⟨*proof*⟩

lemma *suffix-appendD*: *suffix* (*zs* @ *xs*) *ys* ⟹ *suffix xs ys*
⟨*proof*⟩

lemma *strict-suffix-set-subset*: *strict-suffix xs ys* ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *set-mono-suffix*: *suffix xs ys* ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *sorted-antimono-suffix*: *suffix xs ys* ⟹ *sorted ys* ⟹ *sorted xs*
⟨*proof*⟩

lemma *suffix-ConsD2*: *suffix* (*x* # *xs*) (*y* # *ys*) ⟹ *suffix xs ys*
⟨*proof*⟩

lemma *suffix-to-prefix* [*code*]: *suffix xs ys* ⟷ *prefix* (*rev xs*) (*rev ys*)
⟨*proof*⟩

lemma *strict-suffix-to-prefix* [*code*]: *strict-suffix xs ys* ⟷ *strict-prefix* (*rev xs*) (*rev ys*)
⟨*proof*⟩

lemma *distinct-suffix*: *distinct ys* ⟹ *suffix xs ys* ⟹ *distinct xs*
⟨*proof*⟩

lemma *map-mono-suffix*: *suffix xs ys* ⟹ *suffix* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *map-mono-strict-suffix*: *strict-suffix xs ys* ⟹ *strict-suffix* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *filter-mono-suffix*: *suffix xs ys* $\Longrightarrow$ *suffix* (*filter P xs*) (*filter P ys*)
⟨*proof*⟩

lemma *suffix-drop*: *suffix* (*drop n as*) *as*
⟨*proof*⟩

lemma *suffix-dropWhile*: *suffix* (*dropWhile P xs*) *xs*
⟨*proof*⟩

lemma *suffix-take*: *suffix xs ys* $\Longrightarrow$ *ys* = *take* (*length ys* − *length xs*) *ys* @ *xs*
⟨*proof*⟩

lemma *strict-suffix-reflclp-conv*: *strict-suffix*$^{==}$ = *suffix*
⟨*proof*⟩

lemma *suffix-lists*: *suffix xs ys* $\Longrightarrow$ *ys* ∈ *lists A* $\Longrightarrow$ *xs* ∈ *lists A*
⟨*proof*⟩

lemma *suffix-snoc* [*simp*]: *suffix xs* (*ys* @ [*y*]) $\longleftrightarrow$ *xs* = [] ∨ (∃ *zs*. *xs* = *zs* @ [*y*] ∧ *suffix zs ys*)
⟨*proof*⟩

lemma *snoc-suffix-snoc* [*simp*]: *suffix* (*xs* @ [*x*]) (*ys* @ [*y*]) = (*x* = *y* ∧ *suffix xs ys*)
⟨*proof*⟩

lemma *same-suffix-suffix* [*simp*]: *suffix* (*ys* @ *xs*) (*zs* @ *xs*) = *suffix ys zs*
⟨*proof*⟩

lemma *same-suffix-nil* [*simp*]: *suffix* (*ys* @ *xs*) *xs* = (*ys* = [])
⟨*proof*⟩

theorem *suffix-Cons*: *suffix xs* (*y* # *ys*) $\longleftrightarrow$ *xs* = *y* # *ys* ∨ *suffix xs ys*
⟨*proof*⟩

theorem *suffix-append*:
suffix xs (*ys* @ *zs*) $\longleftrightarrow$ *suffix xs zs* ∨ (∃ *xs'*. *xs* = *xs'* @ *zs* ∧ *suffix xs' ys*)
⟨*proof*⟩

theorem *suffix-length-le*: *suffix xs ys* $\Longrightarrow$ *length xs* ≤ *length ys*
⟨*proof*⟩

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*$_1$
⟨*proof*⟩

lemma *suffix-length-suffix*:
suffix ps xs $\Longrightarrow$ *suffix qs xs* $\Longrightarrow$ *length ps* ≤ *length qs* $\Longrightarrow$ *suffix ps qs*
⟨*proof*⟩

lemma *suffix-length-less*: *strict-suffix xs ys* $\Longrightarrow$ *length xs* $<$ *length ys*
⟨*proof*⟩

lemma *suffix-ConsD'*: *suffix* (*x*#*xs*) *ys* $\Longrightarrow$ *strict-suffix xs ys*
⟨*proof*⟩

lemma *drop-strict-suffix*: *strict-suffix xs ys* $\Longrightarrow$ *strict-suffix* (*drop n xs*) *ys*
⟨*proof*⟩

lemma *suffix-map-rightE*:
assumes *suffix xs* (*map f ys*)
shows $\exists$ *xs'*. *suffix xs' ys* $\wedge$ *xs* = *map f xs'*
⟨*proof*⟩

lemma *suffix-remdups-adj*: *suffix xs ys* $\Longrightarrow$ *suffix* (*remdups-adj xs*) (*remdups-adj ys*)
⟨*proof*⟩

lemma *not-suffix-cases*:
assumes *pfx*: $\neg$ *suffix ps ls*
obtains
(*c1*) *ps* $\neq$ [] **and** *ls* = []
| (*c2*) *a as x xs* **where** *ps* = *as*@[*a*] **and** *ls* = *xs*@[*x*] **and** *x* = *a* **and** $\neg$ *suffix as xs*
| (*c3*) *a as x xs* **where** *ps* = *as*@[*a*] **and** *ls* = *xs*@[*x*] **and** *x* $\neq$ *a*
⟨*proof*⟩

lemma *not-suffix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
assumes *np*: $\neg$ *suffix ps ls*
and *base*: $\bigwedge$*x xs*. *P* (*xs*@[*x*]) []
and *r1*: $\bigwedge$*x xs y ys*. *x* $\neq$ *y* $\Longrightarrow$ *P* (*xs*@[*x*]) (*ys*@[*y*])
and *r2*: $\bigwedge$*x xs y ys*. ⟦ *x* = *y*; $\neg$ *suffix xs ys*; *P xs ys* ⟧ $\Longrightarrow$ *P* (*xs*@[*x*]) (*ys*@[*y*])
shows *P ps ls* ⟨*proof*⟩

lemma *parallelD1*: *x* ∥ *y* $\Longrightarrow$ $\neg$ *prefix x y*
⟨*proof*⟩

lemma *parallelD2*: *x* ∥ *y* $\Longrightarrow$ $\neg$ *prefix y x*
⟨*proof*⟩

lemma *parallel-Nil1* [*simp*]: $\neg$ *x* ∥ []
⟨*proof*⟩

lemma *parallel-Nil2* [*simp*]: $\neg$ [] ∥ *x*
⟨*proof*⟩

lemma *Cons-parallelI1*: *a* $\neq$ *b* $\Longrightarrow$ *a* # *as* ∥ *b* # *bs*

⟨*proof*⟩

lemma *Cons-parallelI2*: ⟦ *a* = *b*; *as* ∥ *bs* ⟧ ⟹ *a* # *as* ∥ *b* # *bs*
⟨*proof*⟩

lemma *not-equal-is-parallel*:
assumes *neq*: *xs* ≠ *ys*
and *len*: *length xs* = *length ys*
shows *xs* ∥ *ys*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *distinct-suffixes* [*intro*]: *distinct* (*suffixes xs*)
⟨*proof*⟩

lemma *length-suffixes* [*simp*]: *length* (*suffixes xs*) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *suffixes-snoc* [*simp*]: *suffixes* (*xs* @ [*x*]) = [] # *map* (λ*ys*. *ys* @ [*x*]) (*suffixes xs*)
⟨*proof*⟩

lemma *suffixes-not-Nil* [*simp*]: *suffixes xs* ≠ []
⟨*proof*⟩

lemma *hd-suffixes* [*simp*]: *hd* (*suffixes xs*) = []
⟨*proof*⟩

lemma *last-suffixes* [*simp*]: *last* (*suffixes xs*) = *xs*
⟨*proof*⟩

lemma *suffixes-append*:
suffixes (*xs* @ *ys*) = *suffixes ys* @ *map* (λ*xs*′. *xs*′ @ *ys*) (*tl* (*suffixes xs*))
⟨*proof*⟩

lemma *suffixes-eq-snoc*:
suffixes ys = *xs* @ [*x*] ⟷
(*ys* = [] ∧ *xs* = [] ∨ (∃ *z zs*. *ys* = *z*#*zs* ∧ *xs* = *suffixes zs*)) ∧ *x* = *ys*
⟨*proof*⟩

lemma *suffixes-tailrec* [*code*]:

suffixes xs = *rev* (*snd* (*foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, (*x*#*acc1*)#*acc2*)) ([],[[]]) (*rev xs*)))
⟨*proof*⟩

lemma *set-suffixes-eq*: *set* (*suffixes xs*) = {*ys*. *suffix ys xs*}
⟨*proof*⟩

lemma *card-set-suffixes* [*simp*]: *card* (*set* (*suffixes xs*)) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *set-suffixes-append*:
set (*suffixes* (*xs* @ *ys*)) = *set* (*suffixes ys*) ∪ {*xs'* @ *ys* |*xs'*. *xs'* ∈ *set* (*suffixes xs*)}
⟨*proof*⟩

lemma *suffixes-conv-prefixes*: *suffixes xs* = *map rev* (*prefixes* (*rev xs*))
⟨*proof*⟩

lemma *prefixes-conv-suffixes*: *prefixes xs* = *map rev* (*suffixes* (*rev xs*))
⟨*proof*⟩

lemma *prefixes-rev*: *prefixes* (*rev xs*) = *map rev* (*suffixes xs*)
⟨*proof*⟩

lemma *suffixes-rev*: *suffixes* (*rev xs*) = *map rev* (*prefixes xs*)
⟨*proof*⟩

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*⟩

lemma *list-emb-Nil2* [*simp*]:
assumes *list-emb P xs* [] **shows** *xs* = []
⟨*proof*⟩

lemma *list-emb-refl*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *P x x*
shows *list-emb P xs xs*

⟨*proof*⟩

lemma *list-emb-Cons-Nil* [*simp*]: *list-emb P* (*x*#*xs*) [] = *False*
⟨*proof*⟩

lemma *list-emb-append2* [*intro*]: *list-emb P xs ys* ⟹ *list-emb P xs* (*zs* @ *ys*)
⟨*proof*⟩

lemma *list-emb-prefix* [*intro*]:
assumes *list-emb P xs ys* **shows** *list-emb P xs* (*ys* @ *zs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-emb-strict-suffix*:
assumes *list-emb P xs ys* **and** *strict-suffix ys zs*
shows *list-emb P xs zs*
⟨*proof*⟩

lemma *list-emb-suffix*:
assumes *list-emb P xs ys* **and** *suffix ys zs*
shows *list-emb P xs zs*
⟨*proof*⟩

lemma *list-emb-length*: *list-emb P xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

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*⟩

lemma *list-emb-set*:
assumes *list-emb P xs ys* **and** *x* ∈ *set xs*
obtains *y* **where** *y* ∈ *set ys* **and** *P x y*
⟨*proof*⟩

lemma *list-emb-Cons-iff1* [*simp*]:
assumes *P x y*
shows *list-emb P* (*x*#*xs*) (*y*#*ys*) ⟷ *list-emb P xs ys*
⟨*proof*⟩

lemma *list-emb-Cons-iff2* [*simp*]:
assumes $\neg P\ x\ y$
shows *list-emb* $P\ (x\#xs)\ (y\#ys) \longleftrightarrow$ *list-emb* $P\ (x\#xs)\ ys$
⟨*proof*⟩

lemma *list-emb-code* [*code*]:
list-emb $P\ []\ ys \longleftrightarrow$ *True*
list-emb $P\ (x\#xs)\ [] \longleftrightarrow$ *False*
list-emb $P\ (x\#xs)\ (y\#ys) \longleftrightarrow$ (*if* $P\ x\ y$ *then list-emb* $P\ xs\ ys$ *else list-emb* $P\ (x\#xs)\ ys$)
⟨*proof*⟩

58.9 Subsequences (special case of homeomorphic embedding)

abbreviation *subseq* :: $'a$ *list* ⇒ $'a$ *list* ⇒ *bool*
where *subseq* $xs\ ys \equiv$ *list-emb* $(=)\ xs\ ys$

definition *strict-subseq* **where** *strict-subseq* $xs\ ys \longleftrightarrow xs \neq ys \wedge$ *subseq* $xs\ ys$

lemma *subseq-Cons2*: *subseq* $xs\ ys \Longrightarrow$ *subseq* $(x\#xs)\ (x\#ys)$ ⟨*proof*⟩

lemma *subseq-same-length*:
assumes *subseq* $xs\ ys$ **and** *length* $xs =$ *length* ys **shows** $xs = ys$
⟨*proof*⟩

lemma *not-subseq-length* [*simp*]: *length* $ys <$ *length* $xs \Longrightarrow \neg$ *subseq* $xs\ ys$
⟨*proof*⟩

lemma *subseq-Cons'*: *subseq* $(x\#xs)\ ys \Longrightarrow$ *subseq* $xs\ ys$
⟨*proof*⟩

lemma *subseq-Cons2'*:
assumes *subseq* $(x\#xs)\ (x\#ys)$ **shows** *subseq* $xs\ ys$
⟨*proof*⟩

lemma *subseq-Cons2-neq*:
assumes *subseq* $(x\#xs)\ (y\#ys)$
shows $x \neq y \Longrightarrow$ *subseq* $(x\#xs)\ ys$
⟨*proof*⟩

lemma *subseq-Cons2-iff* [*simp*]:
subseq $(x\#xs)\ (y\#ys) =$ (*if* $x = y$ *then subseq* $xs\ ys$ *else subseq* $(x\#xs)\ ys$)
⟨*proof*⟩

lemma *subseq-append'*: *subseq* $(zs\ @\ xs)\ (zs\ @\ ys) \longleftrightarrow$ *subseq* $xs\ ys$
⟨*proof*⟩

global-interpretation *subseq-order*: *ordering subseq strict-subseq*
⟨*proof*⟩

interpretation *subseq-order*: *order subseq strict-subseq*
⟨*proof*⟩

lemma *in-set-subseqs* [*simp*]: *xs* ∈ *set* (*subseqs ys*) ⟷ *subseq xs ys*
⟨*proof*⟩

lemma *set-subseqs-eq*: *set* (*subseqs ys*) = {*xs*. *subseq xs ys*}
⟨*proof*⟩

lemma *subseq-append-le-same-iff*: *subseq* (*xs* @ *ys*) *ys* ⟷ *xs* = []
⟨*proof*⟩

lemma *subseq-singleton-left*: *subseq* [*x*] *ys* ⟷ *x* ∈ *set ys*
⟨*proof*⟩

lemma *list-emb-append-mono*:
⟦ *list-emb P xs xs'*; *list-emb P ys ys'* ⟧ ⟹ *list-emb P* (*xs*@*ys*) (*xs'*@*ys'*)
⟨*proof*⟩

lemma *prefix-imp-subseq* [*intro*]: *prefix xs ys* ⟹ *subseq xs ys*
⟨*proof*⟩

lemma *suffix-imp-subseq* [*intro*]: *suffix xs ys* ⟹ *subseq xs ys*
⟨*proof*⟩

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*
⟨*proof*⟩

58.10 Appending elements

lemma *subseq-append* [*simp*]:
subseq (*xs* @ *zs*) (*ys* @ *zs*) ⟷ *subseq xs ys* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *subseq-append-iff*:
subseq xs (*ys* @ *zs*) ⟷ (∃ *xs1 xs2*. *xs* = *xs1* @ *xs2* ∧ *subseq xs1 ys* ∧ *subseq xs2 zs*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *subseq-appendE* [*case-names append*]:
assumes *subseq xs* (*ys* @ *zs*)
obtains *xs1 xs2* **where** *xs* = *xs1* @ *xs2 subseq xs1 ys subseq xs2 zs*

⟨*proof*⟩

lemma *subseq-drop-many*: *subseq xs ys* ⟹ *subseq xs* (*zs* @ *ys*)
⟨*proof*⟩

lemma *subseq-rev-drop-many*: *subseq xs ys* ⟹ *subseq xs* (*ys* @ *zs*)
⟨*proof*⟩

58.11 Relation to standard list operations

lemma *subseq-map*:
assumes *subseq xs ys* **shows** *subseq* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *subseq-filter-left* [*simp*]: *subseq* (*filter P xs*) *xs*
⟨*proof*⟩

lemma *subseq-filter* [*simp*]:
assumes *subseq xs ys* **shows** *subseq* (*filter P xs*) (*filter P ys*)
⟨*proof*⟩

lemma *subseq-conv-nths*: *subseq xs ys* ⟷ (∃ *N*. *xs* = *nths ys N*)
(**is** *?L* = *?R*)
⟨*proof*⟩

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*⟩

lemma *sublist-Nil-left* [*simp*, *intro*]: *sublist* [] *ys*
⟨*proof*⟩

lemma *sublist-Cons-Nil* [*simp*]: ¬*sublist* (*x*#*xs*) []
⟨*proof*⟩

lemma *sublist-Nil-right* [*simp*]: *sublist xs* [] ⟷ *xs* = []
⟨*proof*⟩

lemma *sublist-appendI* [*simp*, *intro*]: *sublist xs* (*ps* @ *xs* @ *ss*)
⟨*proof*⟩

lemma *sublist-append-leftI* [*simp*, *intro*]: *sublist xs* (*ps* @ *xs*)
⟨*proof*⟩

lemma *sublist-append-rightI* [*simp*, *intro*]: *sublist xs* (*xs* @ *ss*)
⟨*proof*⟩

lemma *sublist-altdef*: *sublist xs ys* ⟷ (∃ *ys'*. *prefix ys' ys* ∧ *suffix xs ys'*)
⟨*proof*⟩

lemma *sublist-altdef'*: *sublist xs ys* ⟷ (∃ *ys'*. *suffix ys' ys* ∧ *prefix xs ys'*)
⟨*proof*⟩

lemma *sublist-Cons-right*: *sublist xs* (*y* # *ys*) ⟷ *prefix xs* (*y* # *ys*) ∨ *sublist xs ys*
⟨*proof*⟩

lemma *sublist-code* [*code*]:
sublist [] *ys* ⟷ *True*
sublist (*x* # *xs*) [] ⟷ *False*
sublist (*x* # *xs*) (*y* # *ys*) ⟷ *prefix* (*x* # *xs*) (*y* # *ys*) ∨ *sublist* (*x* # *xs*) *ys*
⟨*proof*⟩

lemma *sublist-append*:
sublist xs (*ys* @ *zs*) ⟷
sublist xs ys ∨ *sublist xs zs* ∨ (∃ *xs1 xs2*. *xs* = *xs1* @ *xs2* ∧ *suffix xs1 ys* ∧ *prefix xs2 zs*)
⟨*proof*⟩

lemma *map-mono-sublist*:
assumes *sublist xs ys*
shows *sublist* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *sublist-length-le*: *sublist xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *set-mono-sublist*: *sublist xs ys* ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *prefix-imp-sublist* [*simp*, *intro*]: *prefix xs ys* ⟹ *sublist xs ys*
⟨*proof*⟩

lemma *suffix-imp-sublist* [*simp*, *intro*]: *suffix xs ys* ⟹ *sublist xs ys*
⟨*proof*⟩

lemma *sublist-take* [*simp*, *intro*]: *sublist* (*take n xs*) *xs*
⟨*proof*⟩

lemma *sublist-takeWhile* [*simp*, *intro*]: *sublist* (*takeWhile P xs*) *xs*

⟨*proof*⟩

lemma *sublist-drop* [*simp*, *intro*]: *sublist* (*drop n xs*) *xs*
⟨*proof*⟩

lemma *sublist-dropWhile* [*simp*, *intro*]: *sublist* (*dropWhile P xs*) *xs*
⟨*proof*⟩

lemma *sublist-tl* [*simp*, *intro*]: *sublist* (*tl xs*) *xs*
⟨*proof*⟩

lemma *sublist-butlast* [*simp*, *intro*]: *sublist* (*butlast xs*) *xs*
⟨*proof*⟩

lemma *sublist-rev* [*simp*]: *sublist* (*rev xs*) (*rev ys*) = *sublist xs ys*
⟨*proof*⟩

lemma *sublist-rev-left*: *sublist* (*rev xs*) *ys* = *sublist xs* (*rev ys*)
⟨*proof*⟩

lemma *sublist-rev-right*: *sublist xs* (*rev ys*) = *sublist* (*rev xs*) *ys*
⟨*proof*⟩

lemma *snoc-sublist-snoc*:
sublist (*xs* @ [*x*]) (*ys* @ [*y*]) ⟷
(*x* = *y* ∧ *suffix xs ys* ∨ *sublist* (*xs* @ [*x*]) *ys*)
⟨*proof*⟩

lemma *sublist-snoc*:
sublist xs (*ys* @ [*y*]) ⟷ *suffix xs* (*ys* @ [*y*]) ∨ *sublist xs ys*
⟨*proof*⟩

lemma *sublist-imp-subseq* [*intro*]: *sublist xs ys* ⟹ *subseq xs ys*
⟨*proof*⟩

lemma *sublist-map-rightE*:
assumes *sublist xs* (*map f ys*)
shows ∃ *xs′*. *sublist xs′ ys* ∧ *xs* = *map f xs′*
⟨*proof*⟩

lemma *sublist-remdups-adj*:
assumes *sublist xs ys*
shows *sublist* (*remdups-adj xs*) (*remdups-adj ys*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-sublists-eq*: *set* (*sublists xs*) = {*ys*. *sublist ys xs*}
⟨*proof*⟩

lemma *length-sublists* [*simp*]: *length* (*sublists xs*) = *Suc* (*length xs* ∗ *Suc* (*length xs*) *div 2*)
⟨*proof*⟩

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*⟩ **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*⟩ **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*⟩

lemma *prefix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *prefix prefix*
⟨*proof*⟩

lemma *suffix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *suffix suffix*
⟨*proof*⟩

lemma *sublist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *sublist sublist*
⟨*proof*⟩

lemma *parallel-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *parallel parallel*
⟨*proof*⟩

lemma *list-emb-transfer* [*transfer-rule*]:
((*A* ===> *A* ===> (=)) ===> *list-all2 A* ===> *list-all2 A* ===> (=))
list-emb list-emb
⟨*proof*⟩

lemma *strict-prefix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-prefix strict-prefix*
⟨*proof*⟩

lemma *strict-suffix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-suffix strict-suffix*
⟨*proof*⟩

lemma *strict-subseq-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-subseq strict-subseq*
⟨*proof*⟩

lemma *strict-sublist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-sublist strict-sublist*
⟨*proof*⟩

lemma *prefixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *prefixes prefixes*
⟨*proof*⟩

lemma *suffixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *suffixes suffixes*
⟨*proof*⟩

lemma *sublists-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *sublists sublists*
⟨*proof*⟩

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* ≤ *length yl*
shows *prefix xl yl*
⟨*proof*⟩

lemma *shift-prefix-cases*:
assumes *xl* @− *xs* = *yl* @− *ys*
shows *prefix xl yl* ∨ *prefix yl xl*
⟨*proof*⟩

61 Linear temporal logic

Propositional connectives:

abbreviation (*input*) *IMPL* (**infix** ‹*impl*› *60*)
where *φ impl ψ* ≡ λ *xs*. *φ xs* ⟶ *ψ xs*

abbreviation (*input*) *OR* (**infix** ‹*or*› *60*)
where *φ or ψ* ≡ λ *xs*. *φ xs* ∨ *ψ xs*

abbreviation (*input*) *AND* (**infix** ‹*aand*› *60*)
where *φ aand ψ* ≡ λ *xs*. *φ xs* ∧ *ψ xs*

abbreviation (*input*) *not* **where** *not φ* ≡ λ *xs*. ¬ *φ xs*

abbreviation (*input*) *true* ≡ λ *xs*. *True*

abbreviation (*input*) *false* ≡ λ *xs*. *False*

lemma *impl-not-or*: *φ impl ψ* = (*not φ*) *or ψ*
⟨*proof*⟩

lemma *not-or*: *not* (*φ or ψ*) = (*not φ*) *aand* (*not ψ*)
⟨*proof*⟩

lemma *not-aand*: *not* (*φ aand ψ*) = (*not φ*) *or* (*not ψ*)
⟨*proof*⟩

lemma *non-not*[*simp*]: *not* (*not φ*) = *φ* ⟨*proof*⟩

Temporal (LTL) connectives:

fun *holds* **where** *holds P xs* ⟷ *P* (*shd xs*)
fun *nxt* **where** *nxt φ xs* = *φ* (*stl xs*)

definition *HLD s* = *holds* (λ*x*. *x* ∈ *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* ⟹ *ev* φ *xs*
|
step: *ev* φ (*stl xs*) ⟹ *ev* φ *xs*

coinductive *alw* **for** φ **where**
alw: ⟦φ *xs*; *alw* φ (*stl xs*)⟧ ⟹ *alw* φ *xs*

— weak until:
coinductive *UNTIL* (**infix** ‹*until*› *60*) **for** φ ψ **where**
base: ψ *xs* ⟹ (φ *until* ψ) *xs*
|
step: ⟦φ *xs*; (φ *until* ψ) (*stl xs*)⟧ ⟹ (φ *until* ψ) *xs*

end

lemma *holds-mono*:
assumes *holds*: *holds P xs* **and** *0*: ⋀ *x*. *P x* ⟹ *Q x*
shows *holds Q xs*
⟨*proof*⟩

lemma *holds-aand*:
(*holds P aand holds Q*) *steps* ⟷ *holds* (λ *step*. *P step* ∧ *Q step*) *steps* ⟨*proof*⟩

lemma *HLD-iff*: *HLD s* ω ⟷ *shd* ω ∈ *s*
⟨*proof*⟩

lemma *HLD-Stream*[*simp*]: *HLD X* (*x* ## ω) ⟷ *x* ∈ *X*
⟨*proof*⟩

lemma *nxt-mono*:
assumes *nxt*: *nxt* φ *xs* **and** *0*: ⋀ *xs*. φ *xs* ⟹ ψ *xs*
shows *nxt* ψ *xs*
⟨*proof*⟩

declare *ev.intros*[*intro*]
declare *alw.cases*[*elim*]

lemma *ev-induct-strong*[*consumes 1*, *case-names base step*]:
ev φ *x* ⟹ (⋀*xs*. φ *xs* ⟹ *P xs*) ⟹ (⋀*xs*. *ev* φ (*stl xs*) ⟹ ¬ φ *xs* ⟹ *P* (*stl*

$xs) \Longrightarrow P\ xs) \Longrightarrow P\ x$
⟨*proof*⟩

lemma *alw-coinduct*[*consumes 1*, *case-names alw stl*]:
$X\ x \Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \varphi\ x) \Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \neg\ alw\ \varphi\ (stl\ x) \Longrightarrow X\ (stl\ x))$
$\Longrightarrow alw\ \varphi\ x$
⟨*proof*⟩

lemma *ev-mono*:
assumes *ev*: $ev\ \varphi\ xs$ **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows $ev\ \psi\ xs$
⟨*proof*⟩

lemma *alw-mono*:
assumes *alw*: $alw\ \varphi\ xs$ **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows $alw\ \psi\ xs$
⟨*proof*⟩

lemma *until-monoL*:
assumes *until*: $(\varphi 1\ until\ \psi)\ xs$ **and** *0*: $\bigwedge xs.\ \varphi 1\ xs \Longrightarrow \varphi 2\ xs$
shows $(\varphi 2\ until\ \psi)\ xs$
⟨*proof*⟩

lemma *until-monoR*:
assumes *until*: $(\varphi\ until\ \psi 1)\ xs$ **and** *0*: $\bigwedge xs.\ \psi 1\ xs \Longrightarrow \psi 2\ xs$
shows $(\varphi\ until\ \psi 2)\ xs$
⟨*proof*⟩

lemma *until-mono*:
assumes *until*: $(\varphi 1\ until\ \psi 1)\ xs$ **and**
0: $\bigwedge xs.\ \varphi 1\ xs \Longrightarrow \varphi 2\ xs\ \bigwedge xs.\ \psi 1\ xs \Longrightarrow \psi 2\ xs$
shows $(\varphi 2\ until\ \psi 2)\ xs$
⟨*proof*⟩

lemma *until-false*: $\varphi\ until\ false = alw\ \varphi$
⟨*proof*⟩

lemma *ev-nxt*: $ev\ \varphi = (\varphi\ or\ nxt\ (ev\ \varphi))$
⟨*proof*⟩

lemma *alw-nxt*: $alw\ \varphi = (\varphi\ aand\ nxt\ (alw\ \varphi))$
⟨*proof*⟩

lemma *ev-ev*[*simp*]: $ev\ (ev\ \varphi) = ev\ \varphi$
⟨*proof*⟩

lemma *alw-alw*[*simp*]: $alw\ (alw\ \varphi) = alw\ \varphi$
⟨*proof*⟩

lemma *ev-shift*:
assumes *ev* φ *xs*
shows *ev* φ (*xl* @− *xs*)
⟨*proof*⟩

lemma *ev-imp-shift*:
assumes *ev* φ *xs* **shows** $\exists$ *xl xs2*. *xs* = *xl* @− *xs2* $\wedge$ φ *xs2*
⟨*proof*⟩

lemma *alw-ev-shift*: *alw* φ *xs1* $\Longrightarrow$ *ev* (*alw* φ) (*xl* @− *xs1*)
⟨*proof*⟩

lemma *alw-shift*:
assumes *alw* φ (*xl* @− *xs*)
shows *alw* φ *xs*
⟨*proof*⟩

lemma *ev-ex-nxt*:
assumes *ev* φ *xs*
shows $\exists$ *n*. (*nxt* ^^ *n*) φ *xs*
⟨*proof*⟩

lemma *alw-sdrop*:
assumes *alw* φ *xs* **shows** *alw* φ (*sdrop n xs*)
⟨*proof*⟩

lemma *nxt-sdrop*: (*nxt* ^^ *n*) φ *xs* $\longleftrightarrow$ φ (*sdrop n xs*)
⟨*proof*⟩

definition *wait* φ *xs* ≡ *LEAST n*. (*nxt* ^^ *n*) φ *xs*

lemma *nxt-wait*:
assumes *ev* φ *xs* **shows** (*nxt* ^^ (*wait* φ *xs*)) φ *xs*
⟨*proof*⟩

lemma *nxt-wait-least*:
assumes *ev*: *ev* φ *xs* **and** *nxt*: (*nxt* ^^ *n*) φ *xs* **shows** *wait* φ *xs* ≤ *n*
⟨*proof*⟩

lemma *sdrop-wait*:
assumes *ev* φ *xs* **shows** φ (*sdrop* (*wait* φ *xs*) *xs*)
⟨*proof*⟩

lemma *sdrop-wait-least*:
assumes *ev*: *ev* φ *xs* **and** *nxt*: φ (*sdrop n xs*) **shows** *wait* φ *xs* ≤ *n*
⟨*proof*⟩

lemma *nxt-ev*: (*nxt* ^^ *n*) φ *xs* $\Longrightarrow$ *ev* φ *xs*
⟨*proof*⟩

lemma *not-ev*: *not* (*ev* φ) = *alw* (*not* φ)
⟨*proof*⟩

lemma *not-alw*: *not* (*alw* φ) = *ev* (*not* φ)
⟨*proof*⟩

lemma *not-ev-not*[*simp*]: *not* (*ev* (*not* φ)) = *alw* φ
⟨*proof*⟩

lemma *not-alw-not*[*simp*]: *not* (*alw* (*not* φ)) = *ev* φ
⟨*proof*⟩

lemma *alw-ev-sdrop*:
assumes *alw* (*ev* φ) (*sdrop m xs*)
shows *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *ev-alw-imp-alw-ev*:
assumes *ev* (*alw* φ) *xs* **shows** *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *alw-aand*: *alw* (φ *aand* ψ) = *alw* φ *aand alw* ψ
⟨*proof*⟩

lemma *ev-or*: *ev* (φ *or* ψ) = *ev* φ *or ev* ψ
⟨*proof*⟩

lemma *ev-alw-aand*:
assumes φ: *ev* (*alw* φ) *xs* **and** ψ: *ev* (*alw* ψ) *xs*
shows *ev* (*alw* (φ *aand* ψ)) *xs*
⟨*proof*⟩

lemma *ev-alw-alw-impl*:
assumes *ev* (*alw* φ) *xs* **and** *alw* (*alw* φ *impl ev* ψ) *xs*
shows *ev* ψ *xs*
⟨*proof*⟩

lemma *ev-alw-stl*[*simp*]: *ev* (*alw* φ) (*stl x*) ⟷ *ev* (*alw* φ) *x*
⟨*proof*⟩

lemma *alw-alw-impl-ev*:
alw (*alw* φ *impl ev* ψ) = (*ev* (*alw* φ) *impl alw* (*ev* ψ)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *ev-alw-impl*:
assumes *ev* φ *xs* **and** *alw* (φ *impl* ψ) *xs* **shows** *ev* ψ *xs*
⟨*proof*⟩

lemma *ev-alw-impl-ev*:
assumes *ev* φ *xs* **and** *alw* (φ *impl ev* ψ) *xs* **shows** *ev* ψ *xs*
⟨*proof*⟩

lemma *alw-mp*:
assumes *alw* φ *xs* **and** *alw* (φ *impl* ψ) *xs*
shows *alw* ψ *xs*
⟨*proof*⟩

lemma *all-imp-alw*:
assumes $\bigwedge$ *xs*. φ *xs* **shows** *alw* φ *xs*
⟨*proof*⟩

lemma *alw-impl-ev-alw*:
assumes *alw* (φ *impl ev* ψ) *xs*
shows *alw* (*ev* φ *impl ev* ψ) *xs*
⟨*proof*⟩

lemma *ev-holds-sset*:
ev (*holds P*) *xs* ⟷ (∃ *x* ∈ *sset xs*. *P x*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

LTL as a program logic:

lemma *alw-invar*:
assumes φ *xs* **and** *alw* (φ *impl nxt* φ) *xs*
shows *alw* φ *xs*
⟨*proof*⟩

lemma *variance*:
assumes *1*: φ *xs* **and** *2*: *alw* (φ *impl* (ψ *or nxt* φ)) *xs*
shows (*alw* φ *or ev* ψ) *xs*
⟨*proof*⟩

lemma *ev-alw-imp-nxt*:
assumes *e*: *ev* φ *xs* **and** *a*: *alw* (φ *impl* (*nxt* φ)) *xs*
shows *ev* (*alw* φ) *xs*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *ev-at-HLD-imp-snth*: *ev-at* (*HLD X*) *n* $\omega \Longrightarrow \omega$!! *n* $\in$ *X*
⟨*proof*⟩

lemma *ev-at-HLD-single-imp-snth*: *ev-at* (*HLD* {*x*}) *n* $\omega \Longrightarrow \omega$!! *n* = *x*
⟨*proof*⟩

lemma *ev-at-unique*: *ev-at P n* $\omega \Longrightarrow$ *ev-at P m* $\omega \Longrightarrow$ *n* = *m*
⟨*proof*⟩

lemma *ev-iff-ev-at*: *ev P* $\omega \longleftrightarrow$ ($\exists$ *n*. *ev-at P n* ω)
⟨*proof*⟩

lemma *ev-at-shift*: *ev-at* (*HLD X*) *i* (*stake* (*Suc i*) ω @− ω' :: *'s stream*) $\longleftrightarrow$ *ev-at* (*HLD X*) *i* ω
⟨*proof*⟩

lemma *ev-iff-ev-at-unique*: *ev P* $\omega \longleftrightarrow$ ($\exists!$*n*. *ev-at P n* ω)
⟨*proof*⟩

lemma *alw-HLD-iff-streams*: *alw* (*HLD X*) $\omega \longleftrightarrow \omega \in$ *streams X*
⟨*proof*⟩

lemma *not-HLD*: *not* (*HLD X*) = *HLD* (− *X*)
⟨*proof*⟩

lemma *not-alw-iff*: ¬ (*alw P* ω) $\longleftrightarrow$ *ev* (*not P*) ω
⟨*proof*⟩

lemma *not-ev-iff*: ¬ (*ev P* ω) $\longleftrightarrow$ *alw* (*not P*) ω
⟨*proof*⟩

lemma *ev-Stream*: *ev P* (*x* ## *s*) $\longleftrightarrow$ *P* (*x* ## *s*) ∨ *ev P s*
⟨*proof*⟩

lemma *alw-ev-imp-ev-alw*:
assumes *alw* (*ev P*) ω **shows** *ev* (*P aand alw* (*ev P*)) ω
⟨*proof*⟩

lemma *ev-False*: *ev* (λx. *False*) $\omega \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *alw-False*: *alw* (λx. *False*) $\omega \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *ev-iff-sdrop*: *ev P* $\omega \longleftrightarrow$ ($\exists$ *m*. *P* (*sdrop m* ω))
⟨*proof*⟩

lemma *alw-iff-sdrop*: *alw P* $\omega \longleftrightarrow$ ($\forall$ *m*. *P* (*sdrop m* ω))

⟨*proof*⟩

lemma *infinite-iff-alw-ev*: *infinite* {*m*. *P* (*sdrop m ω*)} ⟷ *alw* (*ev P*) *ω*
⟨*proof*⟩

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*⟩

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*⟩

lemma *alw-smap*: *alw P* (*smap f s*) ⟷ *alw* (λ*x*. *P* (*smap f x*)) *s*
⟨*proof*⟩

lemma *ev-smap*: *ev P* (*smap f s*) ⟷ *ev* (λ*x*. *P* (*smap f x*)) *s*
⟨*proof*⟩

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*⟩

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*⟩

lemma *alwD*: *alw P x* ⟹ *P x*
⟨*proof*⟩

lemma *alw-alwD*: *alw P ω* ⟹ *alw* (*alw P*) *ω*
⟨*proof*⟩

lemma *alw-ev-stl*: *alw* (*ev P*) (*stl ω*) ⟷ *alw* (*ev P*) *ω*
⟨*proof*⟩

lemma *holds-Stream*: *holds P* (*x* ## *s*) ⟷ *P x*
⟨*proof*⟩

lemma *holds-eq1*[*simp*]: *holds* ((=) *x*) = *HLD* {*x*}
⟨*proof*⟩

lemma *holds-eq2*[*simp*]: *holds* (λ*y*. *y* = *x*) = *HLD* {*x*}
⟨*proof*⟩

lemma *not-holds-eq*[*simp*]: *holds* (− (=) *x*) = *not* (*HLD* {*x*})
⟨*proof*⟩

Strong until

context
notes [[*inductive-internals*]]
begin

inductive *suntil* (**infix** ‹*suntil*› *60*) **for** φ ψ **where**
base: $\psi\ \omega \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ \omega$
| *step*: $\varphi\ \omega \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ (\mathit{stl}\ \omega) \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ \omega$

inductive-simps *suntil-Stream*: $(\varphi\ \mathit{suntil}\ \psi)\ (x\ \#\#\ s)$

end

lemma *suntil-induct-strong*[*consumes 1*, *case-names base step*]:
$(\varphi\ \mathit{suntil}\ \psi)\ x \Longrightarrow$
$(\bigwedge\omega.\ \psi\ \omega \Longrightarrow P\ \omega) \Longrightarrow$
$(\bigwedge\omega.\ \varphi\ \omega \Longrightarrow \neg\ \psi\ \omega \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ (\mathit{stl}\ \omega) \Longrightarrow P\ (\mathit{stl}\ \omega) \Longrightarrow P\ \omega) \Longrightarrow P\ x$
⟨*proof*⟩

lemma *ev-suntil*: $(\varphi\ \mathit{suntil}\ \psi)\ \omega \Longrightarrow \mathit{ev}\ \psi\ \omega$
⟨*proof*⟩

lemma *suntil-inv*:
assumes *stl*: $\bigwedge s.\ f\ (\mathit{stl}\ s) = \mathit{stl}\ (f\ s)$
shows $(P\ \mathit{suntil}\ Q)\ (f\ s) \longleftrightarrow ((\lambda x.\ P\ (f\ x))\ \mathit{suntil}\ (\lambda x.\ Q\ (f\ x)))\ s$
⟨*proof*⟩

lemma *suntil-smap*: $(P\ \mathit{suntil}\ Q)\ (\mathit{smap}\ f\ s) \longleftrightarrow ((\lambda x.\ P\ (\mathit{smap}\ f\ x))\ \mathit{suntil}\ (\lambda x.\ Q\ (\mathit{smap}\ f\ x)))\ s$
⟨*proof*⟩

lemma *hld-smap*: $\mathit{HLD}\ x\ (\mathit{smap}\ f\ s) = \mathit{holds}\ (\lambda y.\ f\ y \in x)\ s$
⟨*proof*⟩

lemma *suntil-mono*:
assumes *eq*: $\bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{Q1}\ \omega \Longrightarrow \mathit{Q2}\ \omega\ \bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{R1}\ \omega \Longrightarrow \mathit{R2}\ \omega$
assumes ∗: $(\mathit{Q1}\ \mathit{suntil}\ \mathit{R1})\ \omega\ \mathit{alw}\ P\ \omega$ **shows** $(\mathit{Q2}\ \mathit{suntil}\ \mathit{R2})\ \omega$
⟨*proof*⟩

lemma *suntil-cong*:
$\mathit{alw}\ P\ \omega \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{Q1}\ \omega \longleftrightarrow \mathit{Q2}\ \omega) \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{R1}\ \omega \longleftrightarrow \mathit{R2}\ \omega) \Longrightarrow$
$(\mathit{Q1}\ \mathit{suntil}\ \mathit{R1})\ \omega \longleftrightarrow (\mathit{Q2}\ \mathit{suntil}\ \mathit{R2})\ \omega$
⟨*proof*⟩

lemma *ev-suntil-iff*: $\mathit{ev}\ (P\ \mathit{suntil}\ Q)\ \omega \longleftrightarrow \mathit{ev}\ Q\ \omega$

⟨*proof*⟩

lemma *true-suntil*: $((\lambda\text{-}.\ True)\ suntil\ P) = ev\ P$
⟨*proof*⟩

lemma *suntil-lfp*: $(\varphi\ suntil\ \psi) = lfp\ (\lambda P\ s.\ \psi\ s \vee (\varphi\ s \wedge P\ (stl\ s)))$
⟨*proof*⟩

lemma *sfilter-P*[*simp*]: $P\ (shd\ s) \Longrightarrow sfilter\ P\ s = shd\ s\ \#\#\ sfilter\ P\ (stl\ s)$
⟨*proof*⟩

lemma *sfilter-not-P*[*simp*]: $\neg\ P\ (shd\ s) \Longrightarrow sfilter\ P\ s = sfilter\ P\ (stl\ s)$
⟨*proof*⟩

lemma *sfilter-eq*:
assumes $ev\ (holds\ P)\ s$
shows $sfilter\ P\ s = x\ \#\#\ s' \longleftrightarrow$
$P\ x \wedge (not\ (holds\ P)\ suntil\ (HLD\ \{x\}\ aand\ nxt\ (\lambda s.\ sfilter\ P\ s = s')))\ s$
⟨*proof*⟩

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*⟩

lemma *alw-sfilter*:
assumes $*$: $alw\ (ev\ (holds\ P))\ s$
shows $alw\ Q\ (sfilter\ P\ s) \longleftrightarrow alw\ (\lambda x.\ Q\ (sfilter\ P\ x))\ s$
⟨*proof*⟩

lemma *ev-sfilter*:
assumes $*$: $alw\ (ev\ (holds\ P))\ s$
shows $ev\ Q\ (sfilter\ P\ s) \longleftrightarrow ev\ (\lambda x.\ Q\ (sfilter\ P\ x))\ s$
⟨*proof*⟩

lemma *holds-sfilter*:
assumes $ev\ (holds\ Q)\ s$ **shows** $holds\ P\ (sfilter\ Q\ s) \longleftrightarrow (not\ (holds\ Q)\ suntil\ (holds\ (Q\ aand\ P)))\ s$
⟨*proof*⟩

lemma *suntil-aand-nxt*:
$(\varphi\ suntil\ (\varphi\ aand\ nxt\ \psi))\ \omega \longleftrightarrow (\varphi\ aand\ nxt\ (\varphi\ suntil\ \psi))\ \omega$
⟨*proof*⟩

lemma *alw-sconst*: $alw\ P\ (sconst\ x) \longleftrightarrow P\ (sconst\ x)$
⟨*proof*⟩

lemma *ev-sconst*: $ev\ P\ (sconst\ x) \longleftrightarrow P\ (sconst\ x)$
⟨*proof*⟩

lemma *suntil-sconst*: $(\varphi\ suntil\ \psi)\ (sconst\ x) \longleftrightarrow \psi\ (sconst\ x)$
⟨*proof*⟩

lemma *hld-smap'*: $HLD\ x\ (smap\ f\ s) = HLD\ (f\ -`\ x)\ s$
⟨*proof*⟩

lemma *pigeonhole-stream*:
assumes $alw\ (HLD\ s)\ \omega$
assumes *finite s*
shows $\exists x{\in}s.\ alw\ (ev\ (HLD\ \{x\}))\ \omega$
⟨*proof*⟩

lemma *ev-eq-suntil*: $ev\ P\ \omega \longleftrightarrow (not\ P\ suntil\ P)\ \omega$
⟨*proof*⟩

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

lemma *suntil-implies-until*: $(\varphi\ suntil\ \psi)\ \omega \Longrightarrow (\varphi\ until\ \psi)\ \omega$
⟨*proof*⟩

lemma *alw-implies-until*: $alw\ \varphi\ \omega \Longrightarrow (\varphi\ until\ \psi)\ \omega$
⟨*proof*⟩

lemma *until-ev-suntil*: $(\varphi\ until\ \psi)\ \omega \Longrightarrow ev\ \psi\ \omega \Longrightarrow (\varphi\ suntil\ \psi)\ \omega$
⟨*proof*⟩

lemma *suntil-as-until*: $(\varphi\ suntil\ \psi)\ \omega = ((\varphi\ until\ \psi)\ \omega \wedge ev\ \psi\ \omega)$
⟨*proof*⟩

lemma *until-not-relesased-now*: $(\varphi\ until\ \psi)\ \omega \Longrightarrow \neg\ \psi\ \omega \Longrightarrow \varphi\ \omega$
⟨*proof*⟩

lemma *until-must-release-ev*: $(\varphi\ until\ \psi)\ \omega \Longrightarrow ev\ (not\ \varphi)\ \omega \Longrightarrow ev\ \psi\ \omega$
⟨*proof*⟩

lemma *until-as-suntil*: $(\varphi\ until\ \psi)\ \omega = ((\varphi\ suntil\ \psi)\ or\ (alw\ \varphi))\ \omega$
⟨*proof*⟩

lemma *alw-holds*: $alw\ (holds\ P)\ (h\#\#t) = (P\ h \wedge alw\ (holds\ P)\ t)$
⟨*proof*⟩

lemma *alw-holds2*: $alw\ (holds\ P)\ ss = (P\ (shd\ ss) \wedge alw\ (holds\ P)\ (stl\ ss))$
⟨*proof*⟩

lemma *alw-eq-sconst*: $(alw\ (HLD\ \{h\})\ t) = (t = sconst\ h)$
⟨*proof*⟩

lemma *sdrop-if-suntil*: $(p\ \mathit{suntil}\ q)\ \omega \Longrightarrow \exists j.\ q\ (\mathit{sdrop}\ j\ \omega) \wedge (\forall k < j.\ p\ (\mathit{sdrop}\ k\ \omega))$
⟨*proof*⟩

lemma *not-suntil*: $(\neg\ (p\ \mathit{suntil}\ q)\ \omega) = (\neg\ (p\ \mathit{until}\ q)\ \omega \vee \mathit{alw}\ (\mathit{not}\ q)\ \omega)$
⟨*proof*⟩

lemma *sdrop-until*: $q\ (\mathit{sdrop}\ j\ \omega) \Longrightarrow \forall k{<}j.\ p\ (\mathit{sdrop}\ k\ \omega) \Longrightarrow (p\ \mathit{until}\ q)\ \omega$
⟨*proof*⟩

lemma *sdrop-suntil*: $q\ (\mathit{sdrop}\ j\ \omega) \Longrightarrow (\forall k < j.\ p\ (\mathit{sdrop}\ k\ \omega)) \Longrightarrow (p\ \mathit{suntil}\ q)\ \omega$
⟨*proof*⟩

lemma *suntil-iff-sdrop*: $(p\ \mathit{suntil}\ q)\ \omega = (\exists j.\ q\ (\mathit{sdrop}\ j\ \omega) \wedge (\forall k < j.\ p\ (\mathit{sdrop}\ k\ \omega)))$
⟨*proof*⟩

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* :: $('a{::}\mathit{times}) \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$ (**infix** $\langle *_s \rangle$ *70*)
where $x *_s xs \equiv \mathit{map}\ ((*)\ x)\ xs$

lemma *scale1*[*simp*]: $(1{::}'a{::}\mathit{monoid\text{-}mult}) *_s xs = xs$
⟨*proof*⟩

63.1 + and −

fun *zipwith0* :: $('a{::}\mathit{zero} \Rightarrow 'b{::}\mathit{zero} \Rightarrow 'c) \Rightarrow 'a\ \mathit{list} \Rightarrow 'b\ \mathit{list} \Rightarrow 'c\ \mathit{list}$
where
$\mathit{zipwith0}\ f\ []\ [] = []$ |
$\mathit{zipwith0}\ f\ (x\#xs)\ (y\#ys) = f\ x\ y\ \#\ \mathit{zipwith0}\ f\ xs\ ys$ |
$\mathit{zipwith0}\ f\ (x\#xs)\ [] = f\ x\ 0\ \#\ \mathit{zipwith0}\ f\ xs\ []$ |
$\mathit{zipwith0}\ f\ []\ (y\#ys) = f\ 0\ y\ \#\ \mathit{zipwith0}\ f\ []\ ys$

instantiation *list* :: $(\{\mathit{zero}, \mathit{plus}\})\ \mathit{plus}$
begin

definition
list-add-def: $(+) = \mathit{zipwith0}\ (+)$

instance ⟨*proof*⟩

end

instantiation *list* :: ({*zero*, *uminus*}) *uminus*
begin

definition
list-uminus-def: *uminus* = *map uminus*

instance ⟨*proof*⟩

end

instantiation *list* :: ({*zero*,*minus*}) *minus*
begin

definition
list-diff-def: (−) = *zipwith0* (−)

instance ⟨*proof*⟩

end

lemma *zipwith0-Nil*[*simp*]: *zipwith0 f* [] *ys* = *map* (*f 0*) *ys*
⟨*proof*⟩

lemma *list-add-Nil*[*simp*]: [] + *xs* = (*xs*::'*a*::*monoid-add list*)
⟨*proof*⟩

lemma *list-add-Nil2*[*simp*]: *xs* + [] = (*xs*::'*a*::*monoid-add list*)
⟨*proof*⟩

lemma *list-add-Cons*[*simp*]: (*x*#*xs*) + (*y*#*ys*) = (*x*+*y*)#(*xs*+*ys*)
⟨*proof*⟩

lemma *list-diff-Nil*[*simp*]: [] − *xs* = −(*xs*::'*a*::*group-add list*)
⟨*proof*⟩

lemma *list-diff-Nil2*[*simp*]: *xs* − [] = (*xs*::'*a*::*group-add list*)
⟨*proof*⟩

lemma *list-diff-Cons-Cons*[*simp*]: (*x*#*xs*) − (*y*#*ys*) = (*x*−*y*)#(*xs*−*ys*)
⟨*proof*⟩

lemma *list-uminus-Cons*[*simp*]: −(*x*#*xs*) = (−*x*)#(−*xs*)
⟨*proof*⟩

lemma *self-list-diff*:
xs − xs = replicate (length(xs::'a::group-add list)) 0
⟨*proof*⟩

lemma *list-add-assoc*:
fixes *xs* :: *'a::monoid-add list*
shows *(xs+ys)+zs = xs+(ys+zs)*
⟨*proof*⟩

63.2 Inner product

definition *iprod* :: *'a::ring list ⇒ 'a list ⇒ 'a* (‹(‹*open-block notation=‹mixfix iprod›*›⟨-,-⟩)›)
where ⟨*xs,ys*⟩ = (∑ (*x,y*) ← *zip xs ys. x∗y*)

lemma *iprod-Nil*[*simp*]: ⟨[],*ys*⟩ = *0*
⟨*proof*⟩

lemma *iprod-Nil2*[*simp*]: ⟨*xs*,[]⟩ = *0*
⟨*proof*⟩

lemma *iprod-Cons*[*simp*]: ⟨*x#xs,y#ys*⟩ = *x∗y* + ⟨*xs,ys*⟩
⟨*proof*⟩

lemma *iprod0-if-coeffs0*: ∀ *c∈set cs. c = 0 ⟹* ⟨*cs,xs*⟩ = *0*
⟨*proof*⟩

lemma *iprod-uminus*[*simp*]: ⟨*−xs,ys*⟩ = *−*⟨*xs,ys*⟩
⟨*proof*⟩

lemma *iprod-left-add-distrib*: ⟨*xs* + *ys,zs*⟩ = ⟨*xs,zs*⟩ + ⟨*ys,zs*⟩
⟨*proof*⟩

lemma *iprod-left-diff-distrib*: ⟨*xs* − *ys*, *zs*⟩ = ⟨*xs,zs*⟩ − ⟨*ys,zs*⟩
⟨*proof*⟩

lemma *iprod-assoc*: ⟨*x* ∗*s xs*, *ys*⟩ = *x* ∗ ⟨*xs,ys*⟩
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *setleD*: *S *<= x ⟹ y∈S ⟹ y ≤ x*
⟨*proof*⟩

lemma *setgeI*: *∀ y∈S. x ≤ y ⟹ x <=* S*
⟨*proof*⟩

lemma *setgeD*: *x <=* S ⟹ y∈S ⟹ x ≤ y*
⟨*proof*⟩

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 ⟹ P x*
⟨*proof*⟩

lemma *leastPD2*: *leastP P x ⟹ x <=* Collect P*
⟨*proof*⟩

lemma *leastPD3*: *leastP P x ⟹ y ∈ Collect P ⟹ x ≤ y*
⟨*proof*⟩

lemma *isLubD1*: *isLub R S x ⟹ S *<= x*
⟨*proof*⟩

lemma *isLubD1a*: *isLub R S x ⟹ x ∈ R*
⟨*proof*⟩

lemma *isLub-isUb*: $isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ x$
⟨*proof*⟩

lemma *isLubD2*: $isLub\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \leq x$
⟨*proof*⟩

lemma *isLubD3*: $isLub\ R\ S\ x \Longrightarrow leastP\ (isUb\ R\ S)\ x$
⟨*proof*⟩

lemma *isLubI1*: $leastP(isUb\ R\ S)\ x \Longrightarrow isLub\ R\ S\ x$
⟨*proof*⟩

lemma *isLubI2*: $isUb\ R\ S\ x \Longrightarrow x <=* Collect\ (isUb\ R\ S) \Longrightarrow isLub\ R\ S\ x$
⟨*proof*⟩

lemma *isUbD*: $isUb\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \leq x$
⟨*proof*⟩

lemma *isUbD2*: $isUb\ R\ S\ x \Longrightarrow S\ *<=\ x$
⟨*proof*⟩

lemma *isUbD2a*: $isUb\ R\ S\ x \Longrightarrow x \in R$
⟨*proof*⟩

lemma *isUbI*: $S\ *<=\ x \Longrightarrow x \in R \Longrightarrow isUb\ R\ S\ x$
⟨*proof*⟩

lemma *isLub-le-isUb*: $isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ y \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *isLub-ubs*: $isLub\ R\ S\ x \Longrightarrow x <=* ubs\ R\ S$
⟨*proof*⟩

lemma *isLub-unique*: $[|\ isLub\ R\ S\ x;\ isLub\ R\ S\ y\ |] ==> x = (y::'a::linorder)$
⟨*proof*⟩

lemma *isUb-UNIV-I*: $(\bigwedge y.\ y \in S \Longrightarrow y \leq u) \Longrightarrow isUb\ UNIV\ S\ u$
⟨*proof*⟩

definition *greatestP* :: $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$
where $greatestP\ P\ x = (P\ x \wedge Collect\ P\ *<=\ x)$

definition *isLb* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where $isLb\ R\ S\ x = (x <=*\ S \wedge x \in R)$

definition *isGlb* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
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*
⟨*proof*⟩

lemma *greatestPD2*: *greatestP P x* ⟹ *Collect P* *<= *x*
⟨*proof*⟩

lemma *greatestPD3*: *greatestP P x* ⟹ *y* ∈ *Collect P* ⟹ *x* ≥ *y*
⟨*proof*⟩

lemma *isGlbD1*: *isGlb R S x* ⟹ *x* <=* *S*
⟨*proof*⟩

lemma *isGlbD1a*: *isGlb R S x* ⟹ *x* ∈ *R*
⟨*proof*⟩

lemma *isGlb-isLb*: *isGlb R S x* ⟹ *isLb R S x*
⟨*proof*⟩

lemma *isGlbD2*: *isGlb R S x* ⟹ *y* ∈ *S* ⟹ *y* ≥ *x*
⟨*proof*⟩

lemma *isGlbD3*: *isGlb R S x* ⟹ *greatestP* (*isLb R S*) *x*
⟨*proof*⟩

lemma *isGlbI1*: *greatestP* (*isLb R S*) *x* ⟹ *isGlb R S x*
⟨*proof*⟩

lemma *isGlbI2*: *isLb R S x* ⟹ *Collect* (*isLb R S*) *<= *x* ⟹ *isGlb R S x*
⟨*proof*⟩

lemma *isLbD*: *isLb R S x* ⟹ *y* ∈ *S* ⟹ *y* ≥ *x*
⟨*proof*⟩

lemma *isLbD2*: *isLb R S x* ⟹ *x* <=* *S*
⟨*proof*⟩

lemma *isLbD2a*: *isLb R S x* ⟹ *x* ∈ *R*
⟨*proof*⟩

lemma *isLbI*: *x* <=* *S* ⟹ *x* ∈ *R* ⟹ *isLb R S x*
⟨*proof*⟩

lemma *isGlb-le-isLb*: *isGlb R S x* ⟹ *isLb R S y* ⟹ *x* ≥ *y*
⟨*proof*⟩

lemma *isGlb-ubs*: *isGlb R S x* $\Longrightarrow$ *lbs R S* $*<=$ *x*
⟨*proof*⟩

lemma *isGlb-unique*: [| *isGlb R S x*; *isGlb R S y* |] ==> *x* = (*y*::'*a*::*linorder*)
⟨*proof*⟩

lemma *bdd-above-setle*: *bdd-above A* $\longleftrightarrow$ ($\exists$ *a*. *A* $*<=$ *a*)
⟨*proof*⟩

lemma *bdd-below-setge*: *bdd-below A* $\longleftrightarrow$ ($\exists$ *a*. *a* $<=*$ *A*)
⟨*proof*⟩

lemma *isLub-cSup*:
(*S*::'*a* :: *conditionally-complete-lattice set*) $\neq$ {} $\Longrightarrow$ ($\exists$ *b*. *S* $*<=$ *b*) $\Longrightarrow$ *isLub UNIV S* (*Sup S*)
⟨*proof*⟩

lemma *isGlb-cInf*:
(*S*::'*a* :: *conditionally-complete-lattice set*) $\neq$ {} $\Longrightarrow$ ($\exists$ *b*. *b* $<=*$ *S*) $\Longrightarrow$ *isGlb UNIV S* (*Inf S*)
⟨*proof*⟩

lemma *cSup-le*: (*S*::'*a*::*conditionally-complete-lattice set*) $\neq$ {} $\Longrightarrow$ *S* $*<=$ *b* $\Longrightarrow$ *Sup S* $\leq$ *b*
⟨*proof*⟩

lemma *cInf-ge*: (*S*::'*a* :: *conditionally-complete-lattice set*) $\neq$ {} $\Longrightarrow$ *b* $<=*$ *S* $\Longrightarrow$ *Inf S* $\geq$ *b*
⟨*proof*⟩

lemma *cSup-bounds*:
fixes *S* :: '*a* :: *conditionally-complete-lattice set*
shows *S* $\neq$ {} $\Longrightarrow$ *a* $<=*$ *S* $\Longrightarrow$ *S* $*<=$ *b* $\Longrightarrow$ *a* $\leq$ *Sup S* $\wedge$ *Sup S* $\leq$ *b*
⟨*proof*⟩

lemma *cSup-unique*: (*S*::'*a* :: {*conditionally-complete-linorder*, *no-bot*} *set*) $*<=$ *b* $\Longrightarrow$ ($\forall$ *b'*<*b*. $\exists$ *x*∈*S*. *b'* < *x*) $\Longrightarrow$ *Sup S* = *b*
⟨*proof*⟩

lemma *cInf-unique*: *b* $<=*$ (*S*::'*a* :: {*conditionally-complete-linorder*, *no-top*} *set*) $\Longrightarrow$ ($\forall$ *b'*>*b*. $\exists$ *x*∈*S*. *b'* > *x*) $\Longrightarrow$ *Inf S* = *b*
⟨*proof*⟩

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: $\exists$ *X*. *X* $\in$ *S* $\Longrightarrow$ $\exists$ *Y*. *isUb* (*UNIV*::*real set*) *S Y* $\Longrightarrow$ $\exists$ *t*. *isLub* (*UNIV* :: *real set*) *S t*
⟨*proof*⟩

lemma *Bseq-isUb*: $\bigwedge X :: nat \Rightarrow real.\ Bseq\ X \Longrightarrow \exists\, U.\ isUb\ (UNIV::real\ set)\ \{x.\ \exists\, n.\ X\ n = x\}\ U$
⟨*proof*⟩

lemma *Bseq-isLub*: $\bigwedge X :: nat \Rightarrow real.\ Bseq\ X \Longrightarrow \exists\, U.\ isLub\ (UNIV::real\ set)\ \{x.\ \exists\, n.\ X\ n = x\}\ U$
⟨*proof*⟩

lemma *isLub-mono-imp-LIMSEQ*:
fixes $X :: nat \Rightarrow real$
assumes *u*: *isLub UNIV* $\{x.\ \exists\, n.\ X\ n = x\}\ u$
assumes *X*: $\forall\, m\ n.\ m \leq n \longrightarrow X\ m \leq X\ n$
shows $X \longrightarrow u$
⟨*proof*⟩

lemmas *real-isGlb-unique* = *isGlb-unique*[**where** *'a=real*]

lemma *real-le-inf-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists\, b.\ b <=* s \Longrightarrow Inf\ s \leq Inf\ (t::real\ set)$
⟨*proof*⟩

lemma *real-ge-sup-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists\, b.\ s *<= b \Longrightarrow Sup\ s \geq Sup\ (t::real\ set)$
⟨*proof*⟩

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*
⟨*proof*⟩

context includes *lifting-syntax*
begin

lemma *empty-parametric*: (*A* ===> *rel-option B*) *Map.empty Map.empty*
⟨*proof*⟩

lemma *lookup-parametric*: $((A ===> B) ===> A ===> B)\ (\lambda m\ k.\ m\ k)\ (\lambda m\ k.\ m\ k)$
⟨*proof*⟩

lemma *update-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *B* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k v m*. *m*(*k* ↦ *v*)) (λ*k v m*. *m*(*k* ↦ *v*))
⟨*proof*⟩

lemma *delete-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k m*. *m*(*k* := *None*)) (λ*k m*. *m*(*k* := *None*))
⟨*proof*⟩

lemma *is-none-parametric* [*transfer-rule*]:
(*rel-option A* ===> *HOL.eq*) *Option.is-none Option.is-none*
⟨*proof*⟩

lemma *dom-parametric*:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set A*) *dom dom*
⟨*proof*⟩

lemma *graph-parametric*:
assumes *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set* (*rel-prod A B*)) *Map.graph Map.graph*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

end

65.2 Type definition and primitive operations

typedef ('*a*, '*b*) *mapping* = *UNIV* :: ('*a* ⇀ '*b*) *set*
morphisms *rep Mapping* ⟨*proof*⟩

setup-lifting *type-definition-mapping*

lift-definition *empty* :: ('*a*, '*b*) *mapping*
is *Map.empty* **parametric** *empty-parametric* ⟨*proof*⟩

lift-definition *lookup* :: ('*a*, '*b*) *mapping* ⇒ '*a* ⇒ '*b option*
is λ*m k. m k* **parametric** *lookup-parametric* ⟨*proof*⟩

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* ⟨*proof*⟩

lift-definition *delete* :: '*a* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*k m. m*(*k* := *None*) **parametric** *delete-parametric* ⟨*proof*⟩

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* | *Some* $v \Rightarrow$ *if* $P\ k\ v$ *then Some* v *else None*
⟨*proof*⟩

lift-definition *keys* :: $('a, 'b)$ *mapping* $\Rightarrow 'a$ *set*
is *dom* **parametric** *dom-parametric* ⟨*proof*⟩

lift-definition *entries* :: $('a, 'b)$ *mapping* $\Rightarrow ('a \times 'b)$ *set*
is *Map.graph* **parametric** *graph-parametric* ⟨*proof*⟩

lift-definition *tabulate* :: $'a$ *list* $\Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a, 'b)$ *mapping*
is $\lambda ks\ f.$ (*map-of* (*List.map* $(\lambda k.\ (k, f\ k))\ ks$)) **parametric** *tabulate-parametric*
⟨*proof*⟩

lift-definition *bulkload* :: $'a$ *list* $\Rightarrow$ (*nat*, $'a$) *mapping*
is $\lambda xs\ k.$ *if* $k <$ *length xs then Some* $(xs\ !\ k)$ *else None* **parametric** *bulk-load-parametric* ⟨*proof*⟩

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* ⟨*proof*⟩

lift-definition *map-values* :: $('c \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('c, 'a)$ *mapping* $\Rightarrow ('c, 'b)$ *mapping*
is $\lambda f\ m\ x.$ *map-option* $(f\ x)\ (m\ x)$ ⟨*proof*⟩

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-options* $(f\ x)\ (m1\ x)\ (m2\ x)$ **parametric** *combine-with-key-parametric*
⟨*proof*⟩

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-options* $f\ (m1\ x)\ (m2\ x)$ **parametric** *combine-parametric*
⟨*proof*⟩

definition *All-mapping* $m\ P \longleftrightarrow$
$(\forall x.$ *case Mapping.lookup* $m\ x$ *of None* $\Rightarrow$ *True* | *Some* $y \Rightarrow P\ x\ y)$

declare [[*code drop*: *map*]]

65.3 Functorial structure

functor *map*: *map*
⟨*proof*⟩

65.4 Derived operations

definition *ordered-keys* :: $('a$::*linorder*, $'b)$ *mapping* $\Rightarrow 'a$ *list*
where *ordered-keys* $m =$ (*if finite* (*keys* m) *then sorted-list-of-set* (*keys* m) *else* []$)$

definition *ordered-entries* :: (*'a*::*linorder*, *'b*) *mapping* ⇒ (*'a* × *'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* ⟨*proof*⟩

lemma *map-entry-code* [*code*]:
  map-entry k f m =
    (*case lookup m k of*
      None ⇒ *m*
    | *Some v* ⇒ *update k* (*f v*) *m*)
  ⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

end

65.5 Properties

lemma *mapping-eqI*: $(\bigwedge x.\ lookup\ m\ x = lookup\ m'\ x) \Longrightarrow m = m'$
⟨*proof*⟩

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*⟩

lemma *lookup-update*[*simp*]: *lookup* (*update k v m*) *k* = *Some v*
⟨*proof*⟩

lemma *lookup-update-neq*[*simp*]: $k \neq k' \Longrightarrow lookup\ (update\ k\ v\ m)\ k' = lookup\ m\ k'$
⟨*proof*⟩

lemma *lookup-update'*: *lookup* (*update k v m*) *k'* = (*if k* = *k' then Some v else lookup m k'*)
⟨*proof*⟩

lemma *lookup-empty*[*simp*]: *lookup empty k* = *None*
⟨*proof*⟩

lemma *lookup-delete*[*simp*]: *lookup* (*delete k m*) *k* = *None*
⟨*proof*⟩

lemma *lookup-delete-neq*[*simp*]: $k \neq k' \Longrightarrow$ *lookup* (*delete k m*) $k' =$ *lookup m k'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lookup-map-values*: *lookup* (*map-values f m*) *k* = *map-option* (*f k*) (*lookup m k*)
⟨*proof*⟩

lemma *lookup-default-empty*: *lookup-default d empty k* = *d*
⟨*proof*⟩

lemma *lookup-default-update*: *lookup-default d* (*update k v m*) *k* = *v*
⟨*proof*⟩

lemma *lookup-default-update-neq*:
$k \neq k' \Longrightarrow$ *lookup-default d* (*update k v m*) $k' =$ *lookup-default d m k'*
⟨*proof*⟩

lemma *lookup-default-update'*:
lookup-default d (*update k v m*) $k' =$ (*if k* = k' *then v else lookup-default d m k'*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lookup-default-map-values*:
lookup-default (*f k d*) (*map-values f m*) *k* = *f k* (*lookup-default d m k*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *combine-altdef*: *combine f m1 m2* = *combine-with-key* (λ-. *f*) *m1 m2*
⟨*proof*⟩

lemma *lookup-combine*:
Mapping.lookup (*combine f m1 m2*) *x* =
combine-options f (*Mapping.lookup m1 x*) (*Mapping.lookup m2 x*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lookup-map-entry*: *lookup* (*map-entry x f m*) *x* = *map-option f* (*lookup m x*)
⟨*proof*⟩

lemma *lookup-map-entry-neq*: $x \neq y \Longrightarrow$ *lookup* (*map-entry x f m*) *y* = *lookup m y*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lookup-default*: *lookup* (*default x d m*) *x* = *Some* (*lookup-default d m x*)
⟨*proof*⟩

lemma *lookup-default-neq*: $x \neq y \Longrightarrow$ *lookup* (*default x d m*) *y* = *lookup m y*
⟨*proof*⟩

lemma *lookup-default'*:
lookup (*default x d m*) *y* =
(*if x* = *y then Some* (*lookup-default d m x*) *else lookup m y*)
⟨*proof*⟩

lemma *lookup-map-default*: *lookup* (*map-default x d f m*) *x* = *Some* (*f* (*lookup-default d m x*))
⟨*proof*⟩

lemma *lookup-map-default-neq*: $x \neq y \Longrightarrow$ *lookup* (*map-default x d f m*) *y* = *lookup m y*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lookup-tabulate*:
assumes *distinct xs*
shows *Mapping.lookup* (*Mapping.tabulate xs f*) *x* = (*if x* ∈ *set xs then Some* (*f x*) *else None*)
⟨*proof*⟩

lemma *lookup-of-alist*: *lookup* (*of-alist xs*) *k* = *map-of xs k*
⟨*proof*⟩

lemma *keys-is-none-rep* [*code-unfold*]: *k* ∈ *keys m* ⟷ ¬ (*Option.is-none* (*lookup m k*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *update-delete* [*simp*]: *update k v* (*delete k m*) = *update k v m*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *delete-empty* [*simp*]: *delete k empty* = *empty*
⟨*proof*⟩

lemma *Mapping-delete-if-notin-keys*[*simp*]:
k ∉ *keys m* ⟹ *delete k m* = *m*
⟨*proof*⟩

lemma *replace-update*:
k ∉ *keys m* ⟹ *replace k v m* = *m*
k ∈ *keys m* ⟹ *replace k v m* = *update k v m*
⟨*proof*⟩

lemma *map-values-update*: *map-values f* (*update k v m*) = *update k* (*f k v*) (*map-values f m*)
⟨*proof*⟩

lemma *size-mono*: *finite* (*keys m'*) ⟹ *keys m* ⊆ *keys m'* ⟹ *size m* ≤ *size m'*
⟨*proof*⟩

lemma *size-empty* [*simp*]: *size empty* = *0*
⟨*proof*⟩

lemma *size-update*:
finite (*keys m*) ⟹ *size* (*update k v m*) =
(*if k* ∈ *keys m then size m else Suc* (*size m*))
⟨*proof*⟩

lemma *size-delete*: *size* (*delete k m*) = (*if k* ∈ *keys m then size m* − *1 else size m*)
⟨*proof*⟩

lemma *size-tabulate* [*simp*]: *size* (*tabulate ks f*) = *length* (*remdups ks*)
⟨*proof*⟩

lemma *keys-filter*: *keys* (*filter P m*) ⊆ *keys m*
⟨*proof*⟩

lemma *size-filter*: *finite* (*keys m*) ⟹ *size* (*filter P m*) ≤ *size m*
⟨*proof*⟩

lemma *bulkload-tabulate*: *bulkload xs* = *tabulate* [*0..*<*length xs*] (*nth xs*)
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]: *is-empty empty*
⟨*proof*⟩

lemma *is-empty-update* [*simp*]: ¬ *is-empty* (*update k v m*)
⟨*proof*⟩

lemma *is-empty-delete*: *is-empty* (*delete k m*) ⟷ *is-empty m* ∨ *keys m* = {*k*}
⟨*proof*⟩

lemma *is-empty-replace* [*simp*]: *is-empty* (*replace k v m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-default* [*simp*]: ¬ *is-empty* (*default k v m*)
⟨*proof*⟩

lemma *is-empty-map-entry* [*simp*]: *is-empty* (*map-entry k f m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-map-values* [*simp*]: *is-empty* (*map-values f m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-map-default* [*simp*]: ¬ *is-empty* (*map-default k v f m*)
⟨*proof*⟩

lemma *keys-dom-lookup*: *keys m* = *dom* (*Mapping.lookup m*)
⟨*proof*⟩

lemma *keys-empty* [*simp*]: *keys empty* = {}
⟨*proof*⟩

lemma *in-keysD*: $k \in keys\ m \Longrightarrow \exists v.\ lookup\ m\ k = Some\ v$
⟨*proof*⟩

lemma *keys-update* [*simp*]: $keys\ (update\ k\ v\ m) = insert\ k\ (keys\ m)$
⟨*proof*⟩

lemma *keys-delete* [*simp*]: $keys\ (delete\ k\ m) = keys\ m - \{k\}$
⟨*proof*⟩

lemma *keys-replace* [*simp*]: $keys\ (replace\ k\ v\ m) = keys\ m$
⟨*proof*⟩

lemma *keys-default* [*simp*]: $keys\ (default\ k\ v\ m) = insert\ k\ (keys\ m)$
⟨*proof*⟩

lemma *keys-map-entry* [*simp*]: $keys\ (map\text{-}entry\ k\ f\ m) = keys\ m$
⟨*proof*⟩

lemma *keys-map-default* [*simp*]: $keys\ (map\text{-}default\ k\ v\ f\ m) = insert\ k\ (keys\ m)$
⟨*proof*⟩

lemma *keys-map-values* [*simp*]: $keys\ (map\text{-}values\ f\ m) = keys\ m$
⟨*proof*⟩

lemma *keys-combine-with-key* [*simp*]:
$Mapping.keys\ (combine\text{-}with\text{-}key\ f\ m1\ m2) = Mapping.keys\ m1 \cup Mapping.keys\ m2$
⟨*proof*⟩

lemma *keys-combine* [*simp*]: $Mapping.keys\ (combine\ f\ m1\ m2) = Mapping.keys\ m1 \cup Mapping.keys\ m2$
⟨*proof*⟩

lemma *keys-tabulate* [*simp*]: $keys\ (tabulate\ ks\ f) = set\ ks$
⟨*proof*⟩

lemma *keys-of-alist* [*simp*]: $keys\ (of\text{-}alist\ xs) = set\ (List.map\ fst\ xs)$
⟨*proof*⟩

lemma *keys-bulkload* [*simp*]: $keys\ (bulkload\ xs) = \{0..<length\ xs\}$
⟨*proof*⟩

lemma *finite-keys-update*[*simp*]:
$finite\ (keys\ (update\ k\ v\ m)) = finite\ (keys\ m)$
⟨*proof*⟩

lemma *set-ordered-keys*[*simp*]:
$finite\ (Mapping.keys\ m) \Longrightarrow set\ (Mapping.ordered\text{-}keys\ m) = Mapping.keys\ m$

⟨*proof*⟩

lemma *distinct-ordered-keys* [*simp*]: *distinct* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-infinite* [*simp*]: ¬ *finite* (*keys m*) ⟹ *ordered-keys m* = []
⟨*proof*⟩

lemma *ordered-keys-empty* [*simp*]: *ordered-keys empty* = []
⟨*proof*⟩

lemma *sorted-ordered-keys*[*simp*]: *sorted* (*ordered-keys m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ordered-keys-delete* [*simp*]: *ordered-keys* (*delete k m*) = *remove1 k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-replace* [*simp*]: *ordered-keys* (*replace k v m*) = *ordered-keys m*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ordered-keys-map-entry* [*simp*]: *ordered-keys* (*map-entry k f m*) = *ordered-keys m*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ordered-keys-tabulate* [*simp*]: *ordered-keys* (*tabulate ks f*) = *sort* (*remdups ks*)
⟨*proof*⟩

lemma *ordered-keys-bulkload* [*simp*]: *ordered-keys* (*bulkload ks*) = [*0*..<*length ks*]
⟨*proof*⟩

lemma *tabulate-fold*: *tabulate xs f = List.fold* (λ*k m. update k* (*f k*) *m*) *xs empty*
⟨*proof*⟩

lemma *All-mapping-mono*:
(⋀*k v. k* ∈ *keys m* ⟹ *P k v* ⟹ *Q k v*) ⟹ *All-mapping m P* ⟹ *All-mapping m Q*
⟨*proof*⟩

lemma *All-mapping-empty* [*simp*]: *All-mapping Mapping.empty P*
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *All-mapping-filter-iff*: *All-mapping* (*filter P m*) *Q* ⟷ *All-mapping m* (λ*k v. P k v* ⟶ *Q k v*)
⟨*proof*⟩

lemma *All-mapping-filter*: *All-mapping m Q* ⟹ *All-mapping* (*filter P m*) *Q*
⟨*proof*⟩

lemma *All-mapping-map-values*: *All-mapping* (*map-values f m*) *P* ⟷ *All-mapping m* (λ*k v. P k* (*f k v*))
⟨*proof*⟩

lemma *All-mapping-tabulate*: (∀ *x*∈*set xs. P x* (*f x*)) ⟹ *All-mapping* (*Mapping.tabulate xs f*) *P*
⟨*proof*⟩

lemma *All-mapping-alist*:
(⋀*k v*. (*k*, *v*) ∈ *set xs* ⟹ *P k v*) ⟹ *All-mapping* (*Mapping.of-alist xs*) *P*
⟨*proof*⟩

lemma *combine-empty* [*simp*]: *combine f Mapping.empty y = y combine f y Mapping.empty = y*
⟨*proof*⟩

lemma (**in** *abel-semigroup*) *comm-monoid-set-combine*: *comm-monoid-set* (*combine f*) *Mapping.empty*
⟨*proof*⟩

locale *combine-mapping-abel-semigroup* = *abel-semigroup*
begin

sublocale *combine*: *comm-monoid-set combine f Mapping.empty*
⟨*proof*⟩

lemma *fold-combine-code*:
combine.F g (*set xs*) = *foldr* (λ*x*. *combine f* (*g x*)) (*remdups xs*) *Mapping.empty*
⟨*proof*⟩

lemma *keys-fold-combine*: *finite A* ⟹ *Mapping.keys* (*combine.F g A*) = (⋃ *x*∈*A*. *Mapping.keys* (*g x*))
⟨*proof*⟩

end

65.5.1 *entries*, *ordered-entries*, and *fold*

context *linorder*
begin

sublocale *folding-Map-graph*: *folding-insort-key* (≤) (<) *Map.graph m fst* **for** *m*
⟨*proof*⟩

end

lemma *sorted-fst-list-of-set-insort-Map-graph*[*simp*]:
assumes *finite* (*dom m*) *fst x* ∉ *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*⟩

lemma *sorted-fst-list-of-set-insort-insert-Map-graph*[*simp*]:
assumes *finite* (*dom m*) *fst x* ∉ *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*⟩

lemma *linorder-finite-Map-induct*[*consumes 1*, *case-names empty update*]:
fixes *m* :: *'a*::*linorder* ⇀ *'b*
assumes *finite* (*dom m*)
assumes *P Map.empty*
assumes ⋀*k v m*. ⟦ *finite* (*dom m*); *k* ∉ *dom m*; (⋀*k'*. *k'* ∈ *dom m* ⟹ *k'* ≤ *k*); *P m* ⟧
⟹ *P* (*m*(*k* ↦ *v*))
shows *P m*
⟨*proof*⟩

lemma *delete-insort-fst*[*simp*]: *AList.delete k* (*insort-key fst* (*k*, *v*) *xs*) = *AL-*

ist.delete k xs
⟨*proof*⟩

lemma *insort-fst-delete*: ⟦ *fst x* ≠ *k2*; *sorted* (*List.map fst xs*) ⟧
⟹ *insort-key fst x* (*AList.delete k2 xs*) = *AList.delete k2* (*insort-key fst x xs*)
⟨*proof*⟩

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*⟩

lemma *entries-empty*[*simp*]: *entries empty* = {}
⟨*proof*⟩

lemma *entries-lookup*: *entries m* = *Map.graph* (*lookup m*)
⟨*proof*⟩

lemma *in-entriesI*: *lookup m k* = *Some v* ⟹ (*k*, *v*) ∈ *entries m*
⟨*proof*⟩

lemma *in-entriesD*: (*k*, *v*) ∈ *entries m* ⟹ *lookup m k* = *Some v*
⟨*proof*⟩

lemma *fst-image-entries-eq-keys*[*simp*]: *fst* ‘ *Mapping.entries m* = *Mapping.keys m*
⟨*proof*⟩

lemma *finite-entries-iff-finite-keys*[*simp*]:
finite (*entries m*) = *finite* (*keys m*)
⟨*proof*⟩

lemma *entries-update*:
entries (*update k v m*) = *insert* (*k*, *v*) (*entries* (*delete k m*))
⟨*proof*⟩

lemma *entries-delete*:
entries (*delete k m*) = {*e* ∈ *entries m*. *fst e* ≠ *k*}
⟨*proof*⟩

lemma *entries-of-alist*[*simp*]:
distinct (*List.map fst xs*) ⟹ *entries* (*of-alist xs*) = *set xs*
⟨*proof*⟩

lemma *entries-keysD*:
x ∈ *entries m* ⟹ *fst x* ∈ *keys m*
⟨*proof*⟩

lemma *set-ordered-entries*[*simp*]:

finite (*keys m*) ⟹ *set* (*ordered-entries m*) = *entries m*
⟨*proof*⟩

lemma *distinct-ordered-entries*[*simp*]: *distinct* (*List.map fst* (*ordered-entries m*))
⟨*proof*⟩

lemma *sorted-ordered-entries*[*simp*]: *sorted* (*List.map fst* (*ordered-entries m*))
⟨*proof*⟩

lemma *ordered-entries-infinite*[*simp*]:
¬ *finite* (*Mapping.keys m*) ⟹ *ordered-entries m* = []
⟨*proof*⟩

lemma *ordered-entries-empty*[*simp*]: *ordered-entries empty* = []
⟨*proof*⟩

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*⟩

lemma *ordered-entries-delete*[*simp*]:
ordered-entries (*delete k m*) = *AList.delete k* (*ordered-entries m*)
⟨*proof*⟩

lemma *map-fst-ordered-entries*[*simp*]:
List.map fst (*ordered-entries m*) = *ordered-keys m*
⟨*proof*⟩

lemma *fold-empty*[*simp*]: *fold f empty a* = *a*
⟨*proof*⟩

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*]
⟨*proof*⟩

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*⟩

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*); $k \notin$ *keys m*; $(\bigwedge k'.\ k' \in$ *keys m* $\Longrightarrow k' \leq k)$; *P m* ⟧
$\Longrightarrow$ *P* (*update k v m*)
shows *P m*
⟨*proof*⟩

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* ⇒ (*'b* ⇒ *'c*) ⇒ *'d* (**infixl** ‹≫=› *54*)

notation (*ASCII*)
bind (**infixl** ‹>>=› *54*)

abbreviation (*do-notation*)
bind-do :: *'a* ⇒ (*'b* ⇒ *'c*) ⇒ *'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*)
⟨*proof*⟩

lemma *strip-while-Nil* [*simp*]:

strip-while P [] = []
⟨*proof*⟩

lemma *strip-while-append* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*xs* @ [*x*]) = *xs* @ [*x*]
⟨*proof*⟩

lemma *strip-while-append-rec* [*simp*]:
P x ⟹ *strip-while P* (*xs* @ [*x*]) = *strip-while P xs*
⟨*proof*⟩

lemma *strip-while-Cons* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*x* # *xs*) = *x* # *strip-while P xs*
⟨*proof*⟩

lemma *strip-while-eq-Nil* [*simp*]:
strip-while P xs = [] ⟷ (∀ *x*∈*set xs*. *P x*)
⟨*proof*⟩

lemma *strip-while-eq-Cons-rec*:
strip-while P (*x* # *xs*) = *x* # *strip-while P xs* ⟷ ¬ (*P x* ∧ (∀ *x*∈*set xs*. *P x*))
⟨*proof*⟩

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*⟩

lemma *strip-while-snoc* [*simp*]:
strip-while P (*xs* @ [*x*]) = (*if P x then strip-while P xs else xs* @ [*x*])
⟨*proof*⟩

lemma *strip-while-map*:
strip-while P (*map f xs*) = *map f* (*strip-while* (*P* ∘ *f*) *xs*)
⟨*proof*⟩

lemma *strip-while-dropWhile-commute*:
strip-while P (*dropWhile Q xs*) = *dropWhile Q* (*strip-while P xs*)
⟨*proof*⟩

lemma *dropWhile-strip-while-commute*:
dropWhile P (*strip-while Q xs*) = *strip-while Q* (*dropWhile P xs*)
⟨*proof*⟩

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 []
⟨*proof*⟩

lemma *no-leading-Cons* [*iff*]:
no-leading P (*x* # *xs*) ⟷ ¬ *P x*
⟨*proof*⟩

lemma *no-leading-append* [*simp*]:
no-leading P (*xs* @ *ys*) ⟷ *no-leading P xs* ∧ (*xs* = [] ⟶ *no-leading P ys*)
⟨*proof*⟩

lemma *no-leading-dropWhile* [*simp*]:
no-leading P (*dropWhile P xs*)
⟨*proof*⟩

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*⟩

lemma *dropWhile-idem-iff*:
dropWhile P xs = *xs* ⟷ *no-leading P xs*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *no-trailing-Nil* [*iff*]:
no-trailing P []
⟨*proof*⟩

lemma *no-trailing-Cons* [*simp*]:
no-trailing P (*x* # *xs*) ⟷ *no-trailing P xs* ∧ (*xs* = [] ⟶ ¬ *P x*)
⟨*proof*⟩

lemma *no-trailing-append*:
no-trailing P (*xs* @ *ys*) ⟷ *no-trailing P ys* ∧ (*ys* = [] ⟶ *no-trailing P xs*)
⟨*proof*⟩

lemma *no-trailing-append-Cons* [*simp*]:

no-trailing P (xs @ y # ys) ⟷ no-trailing P (y # ys)
⟨*proof*⟩

lemma *no-trailing-strip-while* [*simp*]:
no-trailing P (strip-while P xs)
⟨*proof*⟩

lemma *strip-while-idem* [*simp*]:
no-trailing P xs ⟹ strip-while P xs = xs
⟨*proof*⟩

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*⟩

lemma *strip-while-idem-iff*:
strip-while P xs = xs ⟷ no-trailing P xs
⟨*proof*⟩

lemma *no-trailing-map*:
no-trailing P (map f xs) ⟷ no-trailing (P ∘ f) xs
⟨*proof*⟩

lemma *no-trailing-drop* [*simp*]:
no-trailing P (drop n xs) **if** *no-trailing P xs*
⟨*proof*⟩

lemma *no-trailing-upt* [*simp*]:
no-trailing P [n..<m] ⟷ (n < m ⟶ ¬ P (m − 1))
⟨*proof*⟩

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
⟨*proof*⟩

lemma *nth-default-beyond*:
length xs ≤ n ⟹ nth-default dflt xs n = dflt
⟨*proof*⟩

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] n = dflt
⟨*proof*⟩

lemma *nth-default-Cons*:
nth-default dflt (x # xs) n = (case n of 0 ⇒ x | Suc n′ ⇒ nth-default dflt xs n′)
⟨*proof*⟩

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (x # xs) 0 = x
⟨*proof*⟩

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (x # xs) (Suc n) = nth-default dflt xs n
⟨*proof*⟩

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (replicate n dflt) m = dflt
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (xs @ replicate n dflt) = nth-default dflt xs
⟨*proof*⟩

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (xs @ [dflt]) = nth-default dflt xs
⟨*proof*⟩

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = dflt ⟷ (k < length xs ⟶ xs ! k = dflt)
⟨*proof*⟩

lemma *nth-default-take-eq*:
nth-default dflt (take m xs) n =
(if n < m then nth-default dflt xs n else dflt)
⟨*proof*⟩

lemma *in-indexed-from-iff-nth-default-eq*:
x ≠ dflt ⟹ (n, x) ∈ set (indexed-from 0 xs) ⟷ nth-default dflt xs n = x
⟨*proof*⟩

lemma *last-conv-nth-default*:
assumes *xs ≠ []*
shows *last xs = nth-default dflt xs (length xs − 1)*
⟨*proof*⟩

lemma *nth-default-map-eq*:

f dflt' = dflt ⟹ nth-default dflt (map f xs) n = f (nth-default dflt' xs n)
⟨*proof*⟩

lemma *finite-nth-default-neq-default* [*simp*]:
finite {k. nth-default dflt xs k ≠ dflt}
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *map-nth-default*:
map (nth-default x xs) [0..<length xs] = xs
⟨*proof*⟩

lemma *range-nth-default* [*simp*]:
range (nth-default dflt xs) = insert dflt (set xs)
⟨*proof*⟩

lemma *nth-strip-while*:
assumes *n < length (strip-while P xs)*
shows *strip-while P xs ! n = xs ! n*
⟨*proof*⟩

lemma *length-strip-while-le*:
length (strip-while P xs) ≤ length xs
⟨*proof*⟩

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (strip-while ((=) dflt) xs) = nth-default dflt xs
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *iterate-add-empty*[*simp*]: ‹*iterate-add n 0* = *0*›
⟨*proof*⟩

lemma *iterate-add-distrib*[*simp*]: ‹*iterate-add* (*m*+*n*) *a* = *iterate-add m a* + *iterate-add n a*›
⟨*proof*⟩

lemma *iterate-add-Numeral1*: ‹*iterate-add n Numeral1* = *of-nat n*›
⟨*proof*⟩

lemma *iterate-add-1*: ‹*iterate-add n 1* = *of-nat n*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *iterate-add-less-iff1*:
$j \leq (i::nat) \Longrightarrow (iterate\text{-}add\ i\ (u::'a::\{cancel\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le\}) + m < iterate\text{-}add\ j\ u + n) = (iterate\text{-}add\ (i-j)\ u + m < n)$
⟨*proof*⟩

lemma *iterate-add-less-iff2*:
$i \leq (j::nat) \Longrightarrow (iterate\text{-}add\ i\ (u::'a::\{cancel\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le\}) + m < iterate\text{-}add\ j\ u + n) = (m < iterate\text{-}add\ (j - i)\ u + n)$
⟨*proof*⟩

lemma *iterate-add-less-eq-iff1*:
$j \leq (i::nat) \Longrightarrow (iterate\text{-}add\ i\ (u::'a::\{cancel\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le\}) + m \leq iterate\text{-}add\ j\ u + n) = (iterate\text{-}add\ (i-j)\ u + m \leq n)$
⟨*proof*⟩

lemma *iterate-add-less-eq-iff2*:
$i \leq (j::nat) \Longrightarrow (iterate\text{-}add\ i\ (u::'a::\{cancel\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le\}) + m \leq iterate\text{-}add\ j\ u + n) = (m \leq iterate\text{-}add\ (j - i)\ u + n)$
⟨*proof*⟩

lemma *iterate-add-add-eq1*:
$j \leq (i::nat) \Longrightarrow ((iterate\text{-}add\ i\ u + m) - (iterate\text{-}add\ j\ u + n)) = ((iterate\text{-}add\ (i-j)\ u + m) - n)$
⟨*proof*⟩

lemma *iterate-add-diff-add-eq2*:
$i \leq (j::nat) \Longrightarrow ((iterate\text{-}add\ i\ u + m) - (iterate\text{-}add\ j\ u + n)) = (m - (iterate\text{-}add\ (j-i)\ u + n))$
⟨*proof*⟩

Simproc Set-Up

⟨*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*⟩

setup-lifting *type-definition-multiset*

lemma *count-Abs-multiset*:
‹*count* (*Abs-multiset f*) = *f*› **if** ‹*finite* {*x*. *f x* > *0*}›
⟨*proof*⟩

lemma *multiset-eq-iff*: *M* = *N* ⟷ (∀ *a*. *count M a* = *count N a*)
⟨*proof*⟩

lemma *multiset-eqI*: (⋀*x*. *count A x* = *count B x*) ⟹ *A* = *B*
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lift-definition *minus-multiset* :: ‹*'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*›
is ‹λ*M N a*. *M a* − *N a*›
⟨*proof*⟩

instance
⟨*proof*⟩

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*}›
⟨*proof*⟩

lift-definition *add-mset* :: *'a* ⇒ *'a multiset* ⇒ *'a multiset* **is**
λ*a M b*. *if b* = *a then Suc* (*M b*) *else M b*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *count-add-mset* [*simp*]:
count (*add-mset b A*) *a* = (*if b* = *a then Suc* (*count A a*) *else count A a*)
⟨*proof*⟩

lemma *count-single*: *count* {#*b*#} *a* = (*if b* = *a then 1 else 0*)
⟨*proof*⟩

lemma
add-mset-not-empty [*simp*]: ‹*add-mset a A* ≠ {#}› **and**
empty-not-add-mset [*simp*]: {#} ≠ *add-mset a A*
⟨*proof*⟩

lemma *add-mset-add-mset-same-iff* [*simp*]:
add-mset a A = *add-mset a B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *add-mset-commute*:
add-mset x (*add-mset y M*) = *add-mset y* (*add-mset x M*)
⟨*proof*⟩

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 $\rightleftharpoons$ *Multiset.Ball* **and**
-MBex $\rightleftharpoons$ *Multiset.Bex*
translations
$\forall x{\in}\#A.\ P \rightleftharpoons$ *CONST Multiset.Ball A* $(\lambda x.\ P)$
$\exists x{\in}\#A.\ P \rightleftharpoons$ *CONST Multiset.Bex A* $(\lambda x.\ P)$

⟨*ML*⟩

lemma *count-eq-zero-iff*:
count M x $= 0 \longleftrightarrow x \notin\# M$
⟨*proof*⟩

lemma *not-in-iff*:
$x \notin\# M \longleftrightarrow$ *count M x* $= 0$
⟨*proof*⟩

lemma *count-greater-zero-iff* [*simp*]:
count M x $> 0 \longleftrightarrow x \in\# M$
⟨*proof*⟩

lemma *count-inI*:
assumes *count M x* $= 0 \Longrightarrow$ *False*
shows $x \in\# M$
⟨*proof*⟩

lemma *in-countE*:
assumes $x \in\# M$
obtains n **where** *count M x* $=$ *Suc n*
⟨*proof*⟩

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:
count M x $\geq$ *Suc 0* $\longleftrightarrow x \in\# M$
⟨*proof*⟩

lemma *count-greater-eq-one-iff* [*simp*]:
count M x $\geq 1 \longleftrightarrow x \in\# M$
⟨*proof*⟩

lemma *set-mset-empty* [*simp*]:
set-mset $\{\#\} = \{\}$
⟨*proof*⟩

lemma *set-mset-single*:
set-mset $\{\#b\#\} = \{b\}$
⟨*proof*⟩

lemma *set-mset-eq-empty-iff* [*simp*]:
set-mset M $= \{\} \longleftrightarrow M = \{\#\}$

⟨*proof*⟩

lemma *finite-set-mset* [*iff*]:
finite (*set-mset M*)
⟨*proof*⟩

lemma *set-mset-add-mset-insert* [*simp*]: ‹*set-mset* (*add-mset a A*) = *insert a* (*set-mset A*)›
⟨*proof*⟩

lemma *multiset-nonemptyE* [*elim*]:
assumes $A \neq \{\#\}$
obtains *x* **where** $x \in\# A$
⟨*proof*⟩

lemma *count-gt-imp-in-mset*: *count M x* > *n* ⟹ $x \in\# M$
⟨*proof*⟩

68.3.2 Union

lemma *count-union* [*simp*]:
count (*M* + *N*) *a* = *count M a* + *count N a*
⟨*proof*⟩

lemma *set-mset-union* [*simp*]:
set-mset (*M* + *N*) = *set-mset M* ∪ *set-mset N*
⟨*proof*⟩

lemma *union-mset-add-mset-left* [*simp*]:
add-mset a A + *B* = *add-mset a* (*A* + *B*)
⟨*proof*⟩

lemma *union-mset-add-mset-right* [*simp*]:
A + *add-mset a B* = *add-mset a* (*A* + *B*)
⟨*proof*⟩

lemma *add-single-right-eq-add-mset*: ‹*A* + {#*a*#} = *add-mset a A*›
⟨*proof*⟩

lemma *add-single-left-eq-add-mset*: ‹{#*a*#} + *A* = *add-mset a A*›
⟨*proof*⟩

68.3.3 Difference

instance *multiset* :: (*type*) *comm-monoid-diff*
⟨*proof*⟩

lemma *count-diff* [*simp*]:
count (*M* − *N*) *a* = *count M a* − *count N a*

⟨*proof*⟩

lemma *add-mset-diff-bothsides*:
‹*add-mset a M − add-mset a A = M − A*›
⟨*proof*⟩

lemma *in-diff-count*:
a ∈# *M − N* ⟷ *count N a < count M a*
⟨*proof*⟩

lemma *count-in-diffI*:
assumes ⋀*n*. *count N x = n + count M x* ⟹ *False*
shows *x* ∈# *M − N*
⟨*proof*⟩

lemma *in-diff-countE*:
assumes *x* ∈# *M − N*
obtains *n* **where** *count M x = Suc n + count N x*
⟨*proof*⟩

lemma *in-diffD*:
assumes *a* ∈# *M − N*
shows *a* ∈# *M*
⟨*proof*⟩

lemma *set-mset-diff*:
set-mset (*M − N*) = {*a*. *count N a < count M a*}
⟨*proof*⟩

lemma *diff-empty* [*simp*]: *M − {#} = M* ∧ {#} *− M* = {#}
⟨*proof*⟩

lemma *diff-cancel*: *A − A* = {#}
⟨*proof*⟩

lemma *diff-union-cancelR*: *M + N − N* = (*M*::'*a multiset*)
⟨*proof*⟩

lemma *diff-union-cancelL*: *N + M − N* = (*M*::'*a multiset*)
⟨*proof*⟩

lemma *diff-right-commute*:
fixes *M N Q* :: '*a multiset*
shows *M − N − Q = M − Q − N*
⟨*proof*⟩

lemma *diff-add*:
fixes *M N Q* :: '*a multiset*
shows *M − (N + Q) = M − N − Q*

⟨*proof*⟩

lemma *insert-DiffM* [*simp*]: $x \in\# M \Longrightarrow$ *add-mset* x $(M - \{\#x\#\}) = M$
⟨*proof*⟩

lemma *insert-DiffM2*: $x \in\# M \Longrightarrow (M - \{\#x\#\}) + \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-union-swap*: $a \neq b \Longrightarrow$ *add-mset* b $(M - \{\#a\#\}) =$ *add-mset* b $M - \{\#a\#\}$
⟨*proof*⟩

lemma *diff-add-mset-swap* [*simp*]: $b \notin\# A \Longrightarrow$ *add-mset* b $M - A =$ *add-mset* b $(M - A)$
⟨*proof*⟩

lemma *diff-union-swap2* [*simp*]: $y \in\# M \Longrightarrow$ *add-mset* x $M - \{\#y\#\} =$ *add-mset* x $(M - \{\#y\#\})$
⟨*proof*⟩

lemma *diff-diff-add-mset* [*simp*]: $(M::'a\ multiset) - N - P = M - (N + P)$
⟨*proof*⟩

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

lemma *mset-add* [*elim?*]:
assumes $a \in\# A$
obtains B **where** $A =$ *add-mset* a B
⟨*proof*⟩

lemma *union-iff*:
$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
⟨*proof*⟩

lemma *count-minus-inter-lt-count-minus-inter-iff*:
count $(M2 - M1)$ $y <$ *count* $(M1 - M2)$ $y \longleftrightarrow y \in\# M1 - M2$
⟨*proof*⟩

lemma *minus-inter-eq-minus-inter-iff*:
$(M1 - M2) = (M2 - M1) \longleftrightarrow$ *set-mset* $(M1 - M2) =$ *set-mset* $(M2 - M1)$
⟨*proof*⟩

68.3.4 Min and Max

abbreviation *Min-mset* :: $'a$::*linorder multiset* ⇒ $'a$ **where**
Min-mset $m \equiv$ *Min* (*set-mset* m)

abbreviation *Max-mset* :: *'a::linorder multiset* ⇒ *'a* **where**
Max-mset m ≡ *Max* (*set-mset m*)

lemma
Min-in-mset: *M* ≠ {#} ⟹ *Min-mset M* ∈# *M* **and**
Max-in-mset: *M* ≠ {#} ⟹ *Max-mset M* ∈# *M*
⟨*proof*⟩

68.3.5 Equality of multisets

lemma *single-eq-single* [*simp*]: {#*a*#} = {#*b*#} ⟷ *a* = *b*
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: *M* + *N* = {#} ⟷ *M* = {#} ∧ *N* = {#}
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: {#} = *M* + *N* ⟷ *M* = {#} ∧ *N* = {#}
⟨*proof*⟩

lemma *multi-self-add-other-not-self* [*simp*]: *M* = *add-mset x M* ⟷ *False*
⟨*proof*⟩

lemma *add-mset-remove-trivial* [*simp*]: ‹*add-mset x M* − {#*x*#} = *M*›
⟨*proof*⟩

lemma *diff-single-trivial*: ¬ *x* ∈# *M* ⟹ *M* − {#*x*#} = *M*
⟨*proof*⟩

lemma *diff-single-eq-union*: *x* ∈# *M* ⟹ *M* − {#*x*#} = *N* ⟷ *M* = *add-mset x N*
⟨*proof*⟩

lemma *union-single-eq-diff*: *add-mset x M* = *N* ⟹ *M* = *N* − {#*x*#}
⟨*proof*⟩

lemma *union-single-eq-member*: *add-mset x M* = *N* ⟹ *x* ∈# *N*
⟨*proof*⟩

lemma *add-mset-remove-trivial-If*:
add-mset a (*N* − {#*a*#}) = (*if a* ∈# *N then N else add-mset a N*)
⟨*proof*⟩

lemma *add-mset-remove-trivial-eq*: ‹*N* = *add-mset a* (*N* − {#*a*#}) ⟷ *a* ∈# *N*›
⟨*proof*⟩

lemma *union-is-single*:
M + *N* = {#*a*#} ⟷ *M* = {#*a*#} ∧ *N* = {#} ∨ *M* = {#} ∧ *N* = {#*a*#}
(**is** *?lhs* = *?rhs*)

⟨*proof*⟩

lemma *single-is-union*: $\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
⟨*proof*⟩

lemma *add-eq-conv-diff*:
$add\text{-}mset\ a\ M = add\text{-}mset\ b\ N \longleftrightarrow M = N \wedge a = b \vee M = add\text{-}mset\ b\ (N - \{\#a\#\}) \wedge N = add\text{-}mset\ a\ (M - \{\#b\#\})$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)

⟨*proof*⟩

lemma *add-mset-eq-single* [*iff*]: $add\text{-}mset\ b\ M = \{\#a\#\} \longleftrightarrow b = a \wedge M = \{\#\}$
⟨*proof*⟩

lemma *single-eq-add-mset* [*iff*]: $\{\#a\#\} = add\text{-}mset\ b\ M \longleftrightarrow b = a \wedge M = \{\#\}$
⟨*proof*⟩

lemma *insert-noteq-member*:
assumes *BC*: $add\text{-}mset\ b\ B = add\text{-}mset\ c\ C$
and *bnotc*: $b \neq c$
shows $c \in\# B$
⟨*proof*⟩

lemma *add-eq-conv-ex*:
$(add\text{-}mset\ a\ M = add\text{-}mset\ b\ N) =$
$(M = N \wedge a = b \vee (\exists K.\ M = add\text{-}mset\ b\ K \wedge N = add\text{-}mset\ a\ K))$
⟨*proof*⟩

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = add\text{-}mset\ x\ A$
⟨*proof*⟩

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$
and $b \neq c$
shows $add\text{-}mset\ b\ (B - \{\#c\#\}) = add\text{-}mset\ b\ B - \{\#c\#\}$
⟨*proof*⟩

lemma *add-mset-eq-singleton-iff*[*iff*]:
$add\text{-}mset\ x\ M = \{\#y\#\} \longleftrightarrow M = \{\#\} \wedge x = y$
⟨*proof*⟩

68.3.6 Pointwise ordering induced by count

definition *subseteq-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ‹⊆#› *50*)
where $A \subseteq\# B \longleftrightarrow (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition *subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ‹⊂#› *50*)

where $A \subset\# B \longleftrightarrow A \subseteq\# B \wedge A \neq B$

abbreviation (*input*) *supseteq-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ‹⊇#› *50*)
where *supseteq-mset A B* ≡ *B* ⊆# *A*

abbreviation (*input*) *supset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ‹⊃#› *50*)
where *supset-mset A B* ≡ *B* ⊂# *A*

notation (*input*)
subseteq-mset (**infix** ‹≤#› *50*) **and**
supseteq-mset (**infix** ‹≥#› *50*)

notation (*ASCII*)
subseteq-mset (**infix** ‹<=#› *50*) **and**
subset-mset (**infix** ‹<#› *50*) **and**
supseteq-mset (**infix** ‹>=#› *50*) **and**
supset-mset (**infix** ‹>#› *50*)

global-interpretation *subset-mset*: *ordering* ‹(⊆#)› ‹(⊂#)›
⟨*proof*⟩

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le* ‹(+)› ‹(−)› ‹(⊆#)› ‹(⊂#)›
⟨*proof*⟩

interpretation *subset-mset*: *ordered-ab-semigroup-monoid-add-imp-le* (+) *0* (−) (⊆#) (⊂#)
⟨*proof*⟩

lemma *mset-subset-eqI*:
(⋀*a*. *count A a* ≤ *count B a*) ⟹ *A* ⊆# *B*
⟨*proof*⟩

lemma *mset-subset-eq-count*:
A ⊆# *B* ⟹ *count A a* ≤ *count B a*
⟨*proof*⟩

lemma *mset-subset-eq-exists-conv*: (*A*::*'a multiset*) ⊆# *B* ⟷ (∃ *C*. *B* = *A* + *C*)
⟨*proof*⟩

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff* (+) *0* (⊆#) (⊂#) (−)
⟨*proof*⟩

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*

$\longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-mono-add-left-cancel*: $C + (A::'a\ multiset) \subseteq\# C + B \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-mono-add*: $(A::'a\ multiset) \subseteq\# B \Longrightarrow C \subseteq\# D \Longrightarrow A + C \subseteq\# B + D$
⟨*proof*⟩

lemma *mset-subset-eq-add-left*: $(A::'a\ multiset) \subseteq\# A + B$
⟨*proof*⟩

lemma *mset-subset-eq-add-right*: $B \subseteq\# (A::'a\ multiset) + B$
⟨*proof*⟩

lemma *single-subset-iff* [*simp*]:
$\{\#a\#\} \subseteq\# M \longleftrightarrow a \in\# M$
⟨*proof*⟩

lemma *mset-subset-eq-single*: $a \in\# B \Longrightarrow \{\#a\#\} \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-add-mset-cancel*: ‹*add-mset a A* $\subseteq\#$ *add-mset a B* $\longleftrightarrow A \subseteq\# B$›
⟨*proof*⟩

lemma *multiset-diff-union-assoc*:
fixes $A\ B\ C\ D :: 'a\ multiset$
shows $C \subseteq\# B \Longrightarrow A + B - C = A + (B - C)$
⟨*proof*⟩

lemma *mset-subset-eq-multiset-union-diff-commute*:
fixes $A\ B\ C\ D :: 'a\ multiset$
shows $B \subseteq\# A \Longrightarrow A - B + C = A + C - B$
⟨*proof*⟩

lemma *diff-subset-eq-self*[*simp*]:
$(M::'a\ multiset) - N \subseteq\# M$
⟨*proof*⟩

lemma *mset-subset-eqD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
⟨*proof*⟩

lemma *mset-subsetD*:
$A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$

⟨*proof*⟩

lemma *set-mset-mono*:
A ⊆# *B* ⟹ *set-mset A* ⊆ *set-mset B*
⟨*proof*⟩

lemma *mset-subset-eq-insertD*:
assumes *add-mset x A* ⊆# *B*
shows *x* ∈# *B* ∧ *A* ⊂# *B*
⟨*proof*⟩

lemma *mset-subset-insertD*:
add-mset x A ⊂# *B* ⟹ *x* ∈# *B* ∧ *A* ⊂# *B*
⟨*proof*⟩

lemma *mset-subset-of-empty*[*simp*]: *A* ⊂# {#} ⟷ *False*
⟨*proof*⟩

lemma *empty-subset-add-mset*[*simp*]: {#} ⊂# *add-mset x M*
⟨*proof*⟩

lemma *empty-le*: {#} ⊆# *A*
⟨*proof*⟩

lemma *insert-subset-eq-iff*:
add-mset a A ⊆# *B* ⟷ *a* ∈# *B* ∧ *A* ⊆# *B* − {#*a*#}
⟨*proof*⟩

lemma *insert-union-subset-iff*:
add-mset a A ⊂# *B* ⟷ *a* ∈# *B* ∧ *A* ⊂# *B* − {#*a*#}
⟨*proof*⟩

lemma *subset-eq-diff-conv*:
A − *C* ⊆# *B* ⟷ *A* ⊆# *B* + *C*
⟨*proof*⟩

lemma *multi-psub-of-add-self* [*simp*]: *A* ⊂# *add-mset x A*
⟨*proof*⟩

lemma *multi-psub-self*: *A* ⊂# *A* = *False*
⟨*proof*⟩

lemma *mset-subset-add-mset* [*simp*]: *add-mset x N* ⊂# *add-mset x M* ⟷ *N* ⊂# *M*
⟨*proof*⟩

lemma *mset-subset-diff-self*: *c* ∈# *B* ⟹ *B* − {#*c*#} ⊂# *B*
⟨*proof*⟩

lemma *Diff-eq-empty-iff-mset*: $A - B = \{\#\} \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *add-mset-subseteq-single-iff*[*iff*]: *add-mset a M* $\subseteq\#$ $\{\#b\#\} \longleftrightarrow M = \{\#\} \wedge a = b$
⟨*proof*⟩

lemma *nonempty-subseteq-mset-eq-single*: $M \neq \{\#\} \Longrightarrow M \subseteq\# \{\#x\#\} \Longrightarrow M = \{\#x\#\}$
⟨*proof*⟩

lemma *nonempty-subseteq-mset-iff-single*: $(M \neq \{\#\} \wedge M \subseteq\# \{\#x\#\} \wedge P) \longleftrightarrow M = \{\#x\#\} \wedge P$
⟨*proof*⟩

68.3.7 Intersection and bounded union

definition *inter-mset* :: ‹*'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*› (**infixl** ‹∩#› *70*)
where ‹$A \cap\# B = A - (A - B)$›

lemma *count-inter-mset* [*simp*]:
‹*count* $(A \cap\# B)$ x = *min* (*count A x*) (*count B x*)›
⟨*proof*⟩

interpretation *subset-mset*: *semilattice-inf* ‹(∩#)› ‹(⊆#)› ‹(⊂#)›
⟨*proof*⟩

definition *union-mset* :: ‹*'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*› (**infixl** ‹∪#› *70*)
where ‹$A \cup\# B = A + (B - A)$›

lemma *count-union-mset* [*simp*]:
‹*count* $(A \cup\# B)$ x = *max* (*count A x*) (*count B x*)›
⟨*proof*⟩

global-interpretation *subset-mset*: *semilattice-neutr-order* ‹(∪#)› ‹{#}› ‹(⊇#)› ‹(⊃#)›
⟨*proof*⟩

interpretation *subset-mset*: *semilattice-sup* ‹(∪#)› ‹(⊆#)› ‹(⊂#)›
⟨*proof*⟩

interpretation *subset-mset*: *bounded-lattice-bot* (∩#) (⊆#) (⊂#) (∪#) {#}
⟨*proof*⟩

68.3.8 Additional intersection facts

lemma *set-mset-inter* [*simp*]:
set-mset $(A \cap\# B) = \textit{set-mset}\ A \cap \textit{set-mset}\ B$
⟨*proof*⟩

lemma *diff-intersect-left-idem* [*simp*]:
$M - M \cap\# N = M - N$
⟨*proof*⟩

lemma *diff-intersect-right-idem* [*simp*]:
$M - N \cap\# M = M - N$
⟨*proof*⟩

lemma *multiset-inter-single*[*simp*]: $a \neq b \Longrightarrow \{\#a\#\} \cap\# \{\#b\#\} = \{\#\}$
⟨*proof*⟩

lemma *multiset-union-diff-commute*:
assumes $B \cap\# C = \{\#\}$
shows $A + B - C = A - C + B$
⟨*proof*⟩

lemma *disjunct-not-in*:
$A \cap\# B = \{\#\} \longleftrightarrow (\forall a.\ a \notin\# A \vee a \notin\# B)$
⟨*proof*⟩

lemma *inter-mset-empty-distrib-right*: $A \cap\# (B + C) = \{\#\} \longleftrightarrow A \cap\# B = \{\#\} \wedge A \cap\# C = \{\#\}$
⟨*proof*⟩

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$
⟨*proof*⟩

lemma *add-mset-inter-add-mset* [*simp*]:
add-mset a A $\cap\#$ *add-mset a B* = *add-mset a* $(A \cap\# B)$
⟨*proof*⟩

lemma *add-mset-disjoint* [*simp*]:
add-mset a A $\cap\# B = \{\#\} \longleftrightarrow a \notin\# B \wedge A \cap\# B = \{\#\}$
$\{\#\} =$ *add-mset a A* $\cap\# B \longleftrightarrow a \notin\# B \wedge \{\#\} = A \cap\# B$
⟨*proof*⟩

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* $\longleftrightarrow b \notin\# A \wedge \{\#\} = A \cap\# B$
⟨*proof*⟩

lemma *inter-add-left1*: $\neg\ x \in\# N \Longrightarrow (\textit{add-mset}\ x\ M) \cap\# N = M \cap\# N$
⟨*proof*⟩

lemma *inter-add-left2*: $x \in\# N \Longrightarrow (\textit{add-mset}\ x\ M) \cap\# N = \textit{add-mset}\ x\ (M \cap\# (N - \{\#x\#\}))$
⟨*proof*⟩

lemma *inter-add-right1*: $\neg\ x \in\# N \Longrightarrow N \cap\# (\textit{add-mset}\ x\ M) = N \cap\# M$
⟨*proof*⟩

lemma *inter-add-right2*: $x \in\# N \Longrightarrow N \cap\# (\textit{add-mset}\ x\ M) = \textit{add-mset}\ x\ ((N - \{\#x\#\}) \cap\# M)$
⟨*proof*⟩

lemma *disjunct-set-mset-diff*:
assumes $M \cap\# N = \{\#\}$
shows $\textit{set-mset}\ (M - N) = \textit{set-mset}\ M$
⟨*proof*⟩

lemma *at-most-one-mset-mset-diff*:
assumes $a \notin\# M - \{\#a\#\}$
shows $\textit{set-mset}\ (M - \{\#a\#\}) = \textit{set-mset}\ M - \{a\}$
⟨*proof*⟩

lemma *more-than-one-mset-mset-diff*:
assumes $a \in\# M - \{\#a\#\}$
shows $\textit{set-mset}\ (M - \{\#a\#\}) = \textit{set-mset}\ M$
⟨*proof*⟩

lemma *inter-iff*:
$a \in\# A \cap\# B \longleftrightarrow a \in\# A \wedge a \in\# B$
⟨*proof*⟩

lemma *inter-union-distrib-left*:
$A \cap\# B + C = (A + C) \cap\# (B + C)$
⟨*proof*⟩

lemma *inter-union-distrib-right*:
$C + A \cap\# B = (C + A) \cap\# (C + B)$
⟨*proof*⟩

lemma *inter-subset-eq-union*:
$A \cap\# B \subseteq\# A + B$
⟨*proof*⟩

68.3.9 Additional bounded union facts

lemma *set-mset-sup* [*simp*]:
‹$\textit{set-mset}\ (A \cup\# B) = \textit{set-mset}\ A \cup \textit{set-mset}\ B$›
⟨*proof*⟩

lemma *sup-union-left1* [*simp*]: $\neg\ x \in\# N \Longrightarrow (\mathit{add\text{-}mset}\ x\ M) \cup\# N = \mathit{add\text{-}mset}\ x\ (M \cup\# N)$
⟨*proof*⟩

lemma *sup-union-left2*: $x \in\# N \Longrightarrow (\mathit{add\text{-}mset}\ x\ M) \cup\# N = \mathit{add\text{-}mset}\ x\ (M \cup\# (N - \{\#x\#\}))$
⟨*proof*⟩

lemma *sup-union-right1* [*simp*]: $\neg\ x \in\# N \Longrightarrow N \cup\# (\mathit{add\text{-}mset}\ x\ M) = \mathit{add\text{-}mset}\ x\ (N \cup\# M)$
⟨*proof*⟩

lemma *sup-union-right2*: $x \in\# N \Longrightarrow N \cup\# (\mathit{add\text{-}mset}\ x\ M) = \mathit{add\text{-}mset}\ x\ ((N - \{\#x\#\}) \cup\# M)$
⟨*proof*⟩

lemma *sup-union-distrib-left*:
$A \cup\# B + C = (A + C) \cup\# (B + C)$
⟨*proof*⟩

lemma *union-sup-distrib-right*:
$C + A \cup\# B = (C + A) \cup\# (C + B)$
⟨*proof*⟩

lemma *union-diff-inter-eq-sup*:
$A + B - A \cap\# B = A \cup\# B$
⟨*proof*⟩

lemma *union-diff-sup-eq-inter*:
$A + B - A \cup\# B = A \cap\# B$
⟨*proof*⟩

lemma *add-mset-union*:
‹*add-mset a A* ∪# *add-mset a B* = *add-mset a* (*A* ∪# *B*)›
⟨*proof*⟩

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* = {#}
⟨*proof*⟩

lemma *replicate-mset-Suc* [*simp*]: *replicate-mset* (*Suc n*) *x* = *add-mset x* (*replicate-mset n x*)
⟨*proof*⟩

lemma *count-replicate-mset*[*simp*]: *count* (*replicate-mset n x*) *y* = (*if y* = *x then*

n else 0)
⟨*proof*⟩

lift-definition *repeat-mset* :: ‹*nat* ⇒ *'a multiset* ⇒ *'a multiset*›
is ‹λ*n M a*. *n* ∗ *M a*› ⟨*proof*⟩

lemma *count-repeat-mset* [*simp*]: *count* (*repeat-mset i A*) *a* = *i* ∗ *count A a*
⟨*proof*⟩

lemma *repeat-mset-0* [*simp*]:
‹*repeat-mset 0 M* = {#}›
⟨*proof*⟩

lemma *repeat-mset-Suc* [*simp*]:
‹*repeat-mset* (*Suc n*) *M* = *M* + *repeat-mset n M*›
⟨*proof*⟩

lemma *repeat-mset-right* [*simp*]: *repeat-mset a* (*repeat-mset b A*) = *repeat-mset* (*a* ∗ *b*) *A*
⟨*proof*⟩

lemma *left-diff-repeat-mset-distrib'*: ‹*repeat-mset* (*i* − *j*) *u* = *repeat-mset i u* − *repeat-mset j u*›
⟨*proof*⟩

lemma *left-add-mult-distrib-mset*:
repeat-mset i u + (*repeat-mset j u* + *k*) = *repeat-mset* (*i*+*j*) *u* + *k*
⟨*proof*⟩

lemma *repeat-mset-distrib*:
repeat-mset (*m* + *n*) *A* = *repeat-mset m A* + *repeat-mset n A*
⟨*proof*⟩

lemma *repeat-mset-distrib2*[*simp*]:
repeat-mset n (*A* + *B*) = *repeat-mset n A* + *repeat-mset n B*
⟨*proof*⟩

lemma *repeat-mset-replicate-mset*[*simp*]:
repeat-mset n {#*a*#} = *replicate-mset n a*
⟨*proof*⟩

lemma *repeat-mset-distrib-add-mset*[*simp*]:
repeat-mset n (*add-mset a A*) = *replicate-mset n a* + *repeat-mset n A*
⟨*proof*⟩

lemma *repeat-mset-empty*[*simp*]: *repeat-mset n* {#} = {#}
⟨*proof*⟩

lemma *set-mset-sum*: *finite A* ⟹ *set-mset* (∑ *x*∈*A*. *f x*) = (⋃ *x*∈*A*. *set-mset* (*f*

x))
⟨*proof*⟩

68.4.1 Simprocs

lemma *repeat-mset-iterate-add*: ‹*repeat-mset n M* = *iterate-add n M*›
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mset-subset-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊂# *repeat-mset j u* + *n*) = (*repeat-mset* (*i*−*j*) *u* + *m* ⊂# *n*)
⟨*proof*⟩

lemma *mset-subset-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊂# *repeat-mset j u* + *n*) = (*m* ⊂# *repeat-mset* (*j*−*i*) *u* + *n*)
⟨*proof*⟩

⟨*ML*⟩

lemma *add-mset-replicate-mset-safe*[*cancelation-simproc-pre*]: ‹*NO-MATCH* {#} *M* ⟹ *add-mset a M* = {#*a*#} + *M*›
⟨*proof*⟩

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*]

⟨*ML*⟩

68.4.2 Conditionally complete lattice

instantiation *multiset* :: (*type*) *Inf*
begin

lift-definition *Inf-multiset* :: *'a multiset set* ⇒ *'a multiset* **is**
  λ*A i*. *if A* = {} *then 0 else Inf* ((λ*f*. *f i*) ‘ *A*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *Inf-multiset-empty*: *Inf* {} = {#}
  ⟨*proof*⟩

lemma *count-Inf-multiset-nonempty*: *A* ≠ {} ⟹ *count* (*Inf A*) *x* = *Inf* ((λ*X*. *count X x*) ‘ *A*)
  ⟨*proof*⟩

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* {} = {#}
  ⟨*proof*⟩

lemma *Sup-multiset-unbounded*: ¬ *subset-mset.bdd-above A* ⟹ *Sup A* = {#}
  ⟨*proof*⟩

instance ⟨*proof*⟩

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*⟩

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*⟩

lemma *Sup-multiset-in-multiset*:
  ‹*finite* {*i*. *0* < (*SUP M*∈*A*. *count M i*)}›

if ‹$A \neq \{\}$› ‹*subset-mset.bdd-above* A›
⟨*proof*⟩

lemma *count-Sup-multiset-nonempty*:
‹*count* (*Sup* A) $x = (SUP\ X{\in}A.\ count\ X\ x)$›
if ‹$A \neq \{\}$› ‹*subset-mset.bdd-above* A›
⟨*proof*⟩

interpretation *subset-mset*: *conditionally-complete-lattice Inf Sup* (∩#) (⊆#) (⊂#) (∪#)
⟨*proof*⟩

lemma *set-mset-Inf*:
assumes $A \neq \{\}$
shows *set-mset* (*Inf* A) = $(\bigcap X{\in}A.\ set\text{-}mset\ X)$
⟨*proof*⟩

lemma *in-Inf-multiset-iff*:
assumes $A \neq \{\}$
shows $x \in\# Inf\ A \longleftrightarrow (\forall\, X{\in}A.\ x \in\# X)$
⟨*proof*⟩

lemma *in-Inf-multisetD*: $x \in\# Inf\ A \Longrightarrow X \in A \Longrightarrow x \in\# X$
⟨*proof*⟩

lemma *set-mset-Sup*:
assumes *subset-mset.bdd-above* A
shows *set-mset* (*Sup* A) = $(\bigcup X{\in}A.\ set\text{-}mset\ X)$
⟨*proof*⟩

lemma *in-Sup-multiset-iff*:
assumes *subset-mset.bdd-above* A
shows $x \in\# Sup\ A \longleftrightarrow (\exists\, X{\in}A.\ x \in\# X)$
⟨*proof*⟩

lemma *in-Sup-multisetD*:
assumes $x \in\# Sup\ A$
shows $\exists\, X{\in}A.\ x \in\# X$
⟨*proof*⟩

interpretation *subset-mset*: *distrib-lattice* (∩#) (⊆#) (⊂#) (∪#)
⟨*proof*⟩

68.4.3 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: $('a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$
is $\lambda P\ M.\ \lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *set-mset-filter* [*simp*]:
set-mset (*filter-mset P M*) = {*a* ∈ *set-mset M*. *P a*}
⟨*proof*⟩

lemma *filter-empty-mset* [*simp*]: *filter-mset P* {#} = {#}
⟨*proof*⟩

lemma *filter-single-mset*: *filter-mset P* {#*x*#} = (*if P x then* {#*x*#} *else* {#})
⟨*proof*⟩

lemma *filter-union-mset* [*simp*]: *filter-mset P* (*M* + *N*) = *filter-mset P M* + *filter-mset P N*
⟨*proof*⟩

lemma *filter-diff-mset* [*simp*]: *filter-mset P* (*M* − *N*) = *filter-mset P M* − *filter-mset P N*
⟨*proof*⟩

lemma *filter-inter-mset* [*simp*]: *filter-mset P* (*M* ∩# *N*) = *filter-mset P M* ∩# *filter-mset P N*
⟨*proof*⟩

lemma *filter-sup-mset*[*simp*]: *filter-mset P* (*A* ∪# *B*) = *filter-mset P A* ∪# *filter-mset P B*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *multiset-filter-subset*[*simp*]: *filter-mset f M* ⊆# *M*
⟨*proof*⟩

lemma *filter-mset-mono-strong*:
assumes $A \subseteq\# B$ $\bigwedge x.\ x \in\# A \Longrightarrow P\ x \Longrightarrow Q\ x$
shows *filter-mset* $P\ A \subseteq\#$ *filter-mset* $Q\ B$
⟨*proof*⟩

lemma *multiset-filter-mono*:
assumes $A \subseteq\# B$
shows *filter-mset* $f\ A \subseteq\#$ *filter-mset* $f\ B$
⟨*proof*⟩

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*⟩

lemma *filter-mset-eq-mempty-iff*[*simp*]: *filter-mset* $P\ A = \{\#\} \longleftrightarrow (\forall x.\ x \in\# A \longrightarrow \neg\ P\ x)$
⟨*proof*⟩

lemma *filter-filter-mset*: *filter-mset* P (*filter-mset* $Q\ M$) $= \{\#x \in\# M.\ Q\ x \wedge P\ x\#\}$
⟨*proof*⟩

lemma
filter-mset-True[*simp*]: $\{\#y \in\# M.\ True\#\} = M$ **and**
filter-mset-False[*simp*]: $\{\#y \in\# M.\ False\#\} = \{\#\}$
⟨*proof*⟩

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*⟩

lemma *filter-mset-cong*:
assumes $M = M'$ **and** $\bigwedge x.\ x \in\# M' \Longrightarrow f\ x \longleftrightarrow g\ x$
shows *filter-mset* $f\ M =$ *filter-mset* $g\ M'$
⟨*proof*⟩

lemma *filter-eq-replicate-mset*: $\{\#y \in\# D.\ y = x\#\} =$ *replicate-mset* (*count* $D\ x$) x
⟨*proof*⟩

68.4.4 Size

definition *wcount* **where** *wcount* $f\ M = (\lambda x.$ *count* $M\ x * Suc\ (f\ x))$

lemma *wcount-union*: *wcount* $f\ (M + N)\ a =$ *wcount* $f\ M\ a +$ *wcount* $f\ N\ a$

⟨*proof*⟩

lemma *wcount-add-mset*:
wcount f (*add-mset x M*) *a* = (*if x* = *a then Suc* (*f a*) *else 0*) + *wcount f M a*
⟨*proof*⟩

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 ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-multiset-single* : *size-multiset f* {#*b*#} = *Suc* (*f b*)
⟨*proof*⟩

lemma *size-single*: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *sum-wcount-Int*:
finite A ⟹ *sum* (*wcount f N*) (*A* ∩ *set-mset N*) = *sum* (*wcount f N*) *A*
⟨*proof*⟩

lemma *size-multiset-union* [*simp*]:
size-multiset f (*M* + *N*::′*a multiset*) = *size-multiset f M* + *size-multiset f N*
⟨*proof*⟩

lemma *size-multiset-add-mset* [*simp*]:
size-multiset f (*add-mset a M*) = *Suc* (*f a*) + *size-multiset f M*
⟨*proof*⟩

lemma *size-add-mset* [*simp*]: *size* (*add-mset a A*) = *Suc* (*size A*)
⟨*proof*⟩

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
⟨*proof*⟩

lemma *size-multiset-eq-0-iff-empty* [*iff*]:
size-multiset f M = *0* ⟷ *M* = {#}
⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
⟨*proof*⟩

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ⟹ ∃ *a*. *a* ∈# *M*
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*:
assumes *size M* = *Suc n*
shows ∃ *a N*. *M* = *add-mset a N*
⟨*proof*⟩

lemma *size-mset-mono*:
fixes *A B* :: '*a multiset*
assumes *A* ⊆# *B*
shows *size A* ≤ *size B*
⟨*proof*⟩

lemma *size-filter-mset-lesseq*[*simp*]: *size* (*filter-mset f M*) ≤ *size M*
⟨*proof*⟩

lemma *size-Diff-submset*:
M ⊆# *M'* ⟹ *size* (*M'* − *M*) = *size M'* − *size*(*M*::'*a multiset*)
⟨*proof*⟩

lemma *size-lt-imp-ex-count-lt*: *size M* < *size N* ⟹ ∃ *x* ∈# *N*. *count M x* < *count N x*
⟨*proof*⟩

68.5 Induction and case splits

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: *P* {#}
assumes *add*: ⋀*x M*. *P M* ⟹ *P* (*add-mset x M*)
shows *P M*
⟨*proof*⟩

lemma *multiset-induct-min*[*case-names empty add*]:
fixes *M* :: '*a*::*linorder multiset*

assumes
empty: $P\ \{\#\}$ **and**
add: $\bigwedge x\ M.\ P\ M \Longrightarrow (\forall\, y \in\# M.\ y \geq x) \Longrightarrow P\ (\mathit{add\text{-}mset}\ x\ M)$
shows $P\ M$
⟨*proof*⟩

lemma *multiset-induct-max*[*case-names empty add*]:
fixes $M :: 'a::\mathit{linorder}\ \mathit{multiset}$
assumes
empty: $P\ \{\#\}$ **and**
add: $\bigwedge x\ M.\ P\ M \Longrightarrow (\forall\, y \in\# M.\ y \leq x) \Longrightarrow P\ (\mathit{add\text{-}mset}\ x\ M)$
shows $P\ M$
⟨*proof*⟩

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists\, A\ a.\ M = \mathit{add\text{-}mset}\ a\ A$
⟨*proof*⟩

lemma *multiset-cases* [*cases type*]:
obtains (*empty*) $M = \{\#\}$ | (*add*) $x\ N$ **where** $M = \mathit{add\text{-}mset}\ x\ N$
⟨*proof*⟩

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
⟨*proof*⟩

lemma *union-filter-mset-complement*[*simp*]:
$\forall\, x.\ P\ x = (\neg\ Q\ x) \Longrightarrow \mathit{filter\text{-}mset}\ P\ M + \mathit{filter\text{-}mset}\ Q\ M = M$
⟨*proof*⟩

lemma *multiset-partition*: $M = \{\#x \in\# M.\ P\ x\#\} + \{\#x \in\# M.\ \neg\ P\ x\#\}$
⟨*proof*⟩

lemma *mset-subset-size*: $A \subset\# B \Longrightarrow \mathit{size}\ A < \mathit{size}\ B$
⟨*proof*⟩

lemma *size-1-singleton-mset*: $\mathit{size}\ M = 1 \Longrightarrow \exists\, a.\ M = \{\#a\#\}$
⟨*proof*⟩

lemma *set-mset-subset-singletonD*:
assumes $\mathit{set\text{-}mset}\ A \subseteq \{x\}$
shows $A = \mathit{replicate\text{-}mset}\ (\mathit{size}\ A)\ x$
⟨*proof*⟩

lemma *count-conv-size-mset*: $\mathit{count}\ A\ x = \mathit{size}\ (\mathit{filter\text{-}mset}\ (\lambda y.\ y = x)\ A)$
⟨*proof*⟩

lemma *size-conv-count-bool-mset*: $\mathit{size}\ A = \mathit{count}\ A\ \mathit{True} + \mathit{count}\ A\ \mathit{False}$
⟨*proof*⟩

68.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-subset-mset-rel*: *wf* $\{(M, N :: 'a\ multiset).\ M \subset\# N\}$
⟨*proof*⟩

lemma *wfp-subset-mset*[*simp*]: *wfp* $(\subset\#)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (add\text{-}mset\ a\ F)$
shows *P F*
⟨*proof*⟩

68.6 Least and greatest elements

context begin

qualified lemma
assumes
$M \neq \{\#\}$ **and**
transp-on $(set\text{-}mset\ M)\ R$ **and**
totalp-on $(set\text{-}mset\ M)\ R$
shows
bex-least-element: $(\exists\, l \in\# M.\ \forall\, x \in\# M.\ x \neq l \longrightarrow R\ l\ x)$ **and**
bex-greatest-element: $(\exists\, g \in\# M.\ \forall\, x \in\# M.\ x \neq g \longrightarrow R\ x\ g)$
⟨*proof*⟩

end

68.7 The fold combinator

definition *fold-mset* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$
where
$fold\text{-}mset\ f\ s\ M = Finite\text{-}Set.fold\ (\lambda x.\ f\ x\ \widehat{\ }\widehat{\ }\ count\ M\ x)\ s\ (set\text{-}mset\ M)$

lemma *fold-mset-empty* [*simp*]: $fold\text{-}mset\ f\ s\ \{\#\} = s$
⟨*proof*⟩

lemma *fold-mset-single* [*simp*]: $fold\text{-}mset\ f\ s\ \{\#x\#\} = f\ x\ s$
⟨*proof*⟩

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*⟩

lemma *fold-mset-fun-left-comm*: *f x* (*fold-mset f s M*) = *fold-mset f* (*f x s*) *M*
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]: *fold-mset f s* (*M* + *N*) = *fold-mset f* (*fold-mset f s M*) *N*
⟨*proof*⟩

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*⟩

end

lemma *union-fold-mset-add-mset*: *A* + *B* = *fold-mset add-mset A B*
⟨*proof*⟩

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* ⇒ '*b*) ⇒ '*a multiset* ⇒ '*b multiset* **where**
image-mset f = *fold-mset* (*add-mset* ∘ *f*) {#}

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* (*add-mset* ∘ *f*)
⟨*proof*⟩

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
⟨*proof*⟩

lemma *image-mset-single*: *image-mset f* {#*x*#} = {#*f x*#}
⟨*proof*⟩

lemma *image-mset-union* [*simp*]: *image-mset f* (*M* + *N*) = *image-mset f M* + *image-mset f N*
⟨*proof*⟩

corollary *image-mset-add-mset* [*simp*]:
image-mset f (*add-mset a M*) = *add-mset* (*f a*) (*image-mset f M*)
⟨*proof*⟩

lemma *image-mset-sum*: *image-mset f* (∑ *x*∈*A*. *g x*) = (∑ *x*∈*A*. *image-mset f* (*g x*))
⟨*proof*⟩

lemma *set-image-mset* [*simp*]: *set-mset* (*image-mset f M*) = *image f* (*set-mset M*)
⟨*proof*⟩

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
⟨*proof*⟩

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *filter-mset-image-mset*:
filter-mset P (*image-mset f A*) = *image-mset f* (*filter-mset* (λ*x*. *P* (*f x*)) *A*)
⟨*proof*⟩

lemma *image-mset-Diff*:
assumes *B* ⊆# *A*
shows *image-mset f* (*A* − *B*) = *image-mset f A* − *image-mset f B*
⟨*proof*⟩

lemma *minus-add-mset-if-not-in-lhs*[*simp*]: *x* ∉# *A* ⟹ *A* − *add-mset x B* = *A* − *B*
⟨*proof*⟩

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*⟩

lemma *count-image-mset*:
‹*count* (*image-mset f A*) *x* = (∑ *y*∈*f* −‘ {*x*} ∩ *set-mset A*. *count A y*)›
⟨*proof*⟩

lemma *count-image-mset′*:
‹*count* (*image-mset f X*) *y* = (∑ *x* | *x* ∈# *X* ∧ *y* = *f x*. *count X x*)›
⟨*proof*⟩

lemma *image-mset-subseteq-mono*: $A \subseteq\# B \Longrightarrow$ *image-mset f A* $\subseteq\#$ *image-mset f B*
⟨*proof*⟩

lemma *image-mset-subset-mono*: $M \subset\# N \Longrightarrow$ *image-mset f M* $\subset\#$ *image-mset f N*
⟨*proof*⟩

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 $\{\#x \in\# M.\ x < c\#\}$ but also images like $\{\#x + x.\ x \in\# M\#\}$ and $\{\#x+x|x\in\#M.\ x<c\#\}$, where the latter is currently displayed as $\{\#x + x.\ x \in\# \{\#x \in\# M.\ x < c\#\}\#\}$.

lemma *in-image-mset*: *y* ∈# {#*f x*. *x* ∈# *M*#} ⟷ *y* ∈ *f* ‘ *set-mset M*
⟨*proof*⟩

functor *image-mset*: *image-mset*
⟨*proof*⟩

declare
image-mset.id [*simp*]
image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id x* = *x*
⟨*proof*⟩

lemma *image-mset-cong*: (⋀*x*. *x* ∈# *M* ⟹ *f x* = *g x*) ⟹ {#*f x*. *x* ∈# *M*#} = {#*g x*. *x* ∈# *M*#}
⟨*proof*⟩

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*#}
⟨*proof*⟩

lemma *image-mset-const-eq*:
{#*c*. *a* ∈# *M*#} = *replicate-mset* (*size M*) *c*
⟨*proof*⟩

lemma *image-mset-filter-mset-swap*:
image-mset f (*filter-mset* (λ*x*. *P* (*f x*)) *M*) = *filter-mset P* (*image-mset f M*)
⟨*proof*⟩

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*⟩

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*′
⟨*proof*⟩

lemma *image-mset-eq-plus-image-msetD*:
image-mset f A = *B* + *image-mset f C* ⟹ *inj-on f* (*set-mset A* ∪ *set-mset C*) ⟹
∃ *B*′. *A* = *B*′ + *C* ∧ *B* = *image-mset f B*′
⟨*proof*⟩

68.9 Further conversions

primrec *mset* :: ′*a list* ⇒ ′*a multiset* **where**
mset [] = {#} |
mset (*a* # *x*) = *add-mset a* (*mset x*)

lemma *in-multiset-in-set*:
x ∈# *mset xs* ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemma *count-mset*:
count (*mset xs*) *x* = *count-list xs x*
⟨*proof*⟩

lemma *mset-zero-iff*[*simp*]: (*mset x* = {#}) = (*x* = [])
⟨*proof*⟩

lemma *mset-zero-iff-right*[*simp*]: ({#} = *mset x*) = (*x* = [])

⟨*proof*⟩

lemma *mset-replicate* [*simp*]: *mset* (*replicate n x*) = *replicate-mset n x*
⟨*proof*⟩

lemma *count-mset-gt-0*: *x* ∈ *set xs* ⟹ *count* (*mset xs*) *x* > *0*
⟨*proof*⟩

lemma *count-mset-0-iff* [*simp*]: *count* (*mset xs*) *x* = *0* ⟷ *x* ∉ *set xs*
⟨*proof*⟩

lemma *mset-single-iff*[*iff*]: *mset xs* = {#*x*#} ⟷ *xs* = [*x*]
⟨*proof*⟩

lemma *mset-single-iff-right*[*iff*]: {#*x*#} = *mset xs* ⟷ *xs* = [*x*]
⟨*proof*⟩

lemma *set-mset-mset*[*simp*]: *set-mset* (*mset xs*) = *set xs*
⟨*proof*⟩

lemma *set-mset-comp-mset* [*simp*]: *set-mset* ∘ *mset* = *set*
⟨*proof*⟩

lemma *size-mset* [*simp*]: *size* (*mset xs*) = *length xs*
⟨*proof*⟩

lemma *mset-append* [*simp*]: *mset* (*xs* @ *ys*) = *mset xs* + *mset ys*
⟨*proof*⟩

lemma *mset-filter*[*simp*]: *mset* (*filter P xs*) = {#*x* ∈# *mset xs*. *P x* #}
⟨*proof*⟩

lemma *mset-rev* [*simp*]:
mset (*rev xs*) = *mset xs*
⟨*proof*⟩

lemma *surj-mset*: *surj mset*
⟨*proof*⟩

lemma *distinct-count-atmost-1*:
distinct x = (∀ *a*. *count* (*mset x*) *a* = (*if a* ∈ *set x then 1 else 0*))
⟨*proof*⟩

lemma *mset-eq-setD*:
assumes *mset xs* = *mset ys*
shows *set xs* = *set ys*
⟨*proof*⟩

lemma *set-eq-iff-mset-eq-distinct*:

‹*distinct x* ⟹ *distinct y* ⟹ *set x* = *set y* ⟷ *mset x* = *mset y*›
⟨*proof*⟩

lemma *set-eq-iff-mset-remdups-eq*:
‹*set x* = *set y* ⟷ *mset* (*remdups x*) = *mset* (*remdups y*)›
⟨*proof*⟩

lemma *mset-eq-imp-distinct-iff*:
‹*distinct xs* ⟷ *distinct ys*› **if** ‹*mset xs* = *mset ys*›
⟨*proof*⟩

lemma *nth-mem-mset*: *i* < *length ls* ⟹ (*ls* ! *i*) ∈# *mset ls*
⟨*proof*⟩

lemma *mset-remove1*[*simp*]: *mset* (*remove1 a xs*) = *mset xs* − {#*a*#}
⟨*proof*⟩

lemma *mset-eq-length*:
assumes *mset xs* = *mset ys*
shows *length xs* = *length ys*
⟨*proof*⟩

lemma *mset-eq-length-filter*:
assumes *mset xs* = *mset ys*
shows *count-list xs z* = *count-list ys z*
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mset-shuffles*: *zs* ∈ *shuffles xs ys* ⟹ *mset zs* = *mset xs* + *mset ys*
⟨*proof*⟩

lemma *mset-insort* [*simp*]: *mset* (*insort x xs*) = *add-mset x* (*mset xs*)
⟨*proof*⟩

lemma *mset-map*[*simp*]: *mset* (*map f xs*) = *image-mset f* (*mset xs*)

⟨*proof*⟩

lemma *mset-removeAll-eq*:
‹*mset* (*removeAll x xs*) = *filter-mset* ((≠) *x*) (*mset xs*)›
⟨*proof*⟩

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*⟩

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*⟩

global-interpretation *mset-set*: *folding add-mset* {#}
defines *mset-set* = *folding-on.F add-mset* {#}
⟨*proof*⟩

lemma *sum-multiset-singleton* [*simp*]: *sum* (λ*n*. {#*n*#}) *A* = *mset-set A*
⟨*proof*⟩

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*⟩

lemma *elem-mset-set*[*simp*, *intro*]: *finite A* ⟹ *x* ∈# *mset-set A* ⟷ *x* ∈ *A*
⟨*proof*⟩

lemma *mset-set-Union*:
finite A ⟹ *finite B* ⟹ *A* ∩ *B* = {} ⟹ *mset-set* (*A* ∪ *B*) = *mset-set A* + *mset-set B*
⟨*proof*⟩

lemma *filter-mset-mset-set* [*simp*]:
finite A ⟹ *filter-mset P* (*mset-set A*) = *mset-set* {*x*∈*A*. *P x*}
⟨*proof*⟩

lemma *mset-set-Diff*:
assumes *finite A B* ⊆ *A*
shows *mset-set* (*A* − *B*) = *mset-set A* − *mset-set B*
⟨*proof*⟩

lemma *mset-minus-list-mset*[*simp*]: *mset*(*minus-list-mset xs ys*) = *mset xs* − *mset ys*
⟨*proof*⟩

lemma *mset-set-set*: *distinct xs* $\Longrightarrow$ *mset-set* (*set xs*) = *mset xs*
⟨*proof*⟩

lemma *count-mset-set'*: *count* (*mset-set A*) *x* = (*if finite A* ∧ *x* ∈ *A then 1 else 0*)
⟨*proof*⟩

lemma *subset-imp-msubset-mset-set*:
assumes *A* ⊆ *B finite B*
shows *mset-set A* ⊆# *mset-set B*
⟨*proof*⟩

lemma *mset-set-set-mset-msubset*: *mset-set* (*set-mset A*) ⊆# *A*
⟨*proof*⟩

lemma *mset-set-upto-eq-mset-upto*:
‹*mset-set* {..<*n*} = *mset* [*0*..<*n*]›
⟨*proof*⟩

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 {#} = []
⟨*proof*⟩

lemma *sorted-list-of-multiset-singleton* [*simp*]:
sorted-list-of-multiset {#*x*#} = [*x*]
⟨*proof*⟩

lemma *sorted-list-of-multiset-insert* [*simp*]:
sorted-list-of-multiset (*add-mset x M*) = *List.insort x* (*sorted-list-of-multiset M*)
⟨*proof*⟩

end

lemma *mset-sorted-list-of-multiset*[*simp*]: *mset* (*sorted-list-of-multiset M*) = *M*
⟨*proof*⟩

lemma *sorted-list-of-multiset-mset*[*simp*]: *sorted-list-of-multiset* (*mset xs*) = *sort xs*
⟨*proof*⟩

lemma *finite-set-mset-mset-set*[*simp*]: *finite A* $\Longrightarrow$ *set-mset* (*mset-set A*) = *A*

⟨*proof*⟩

lemma *mset-set-empty-iff*: *mset-set A* = {#} ⟷ *A* = {} ∨ *infinite A*
⟨*proof*⟩

lemma *infinite-set-mset-mset-set*: ¬ *finite A* ⟹ *set-mset* (*mset-set A*) = {}
⟨*proof*⟩

lemma *set-sorted-list-of-multiset* [*simp*]:
set (*sorted-list-of-multiset M*) = *set-mset M*
⟨*proof*⟩

lemma *sorted-sorted-list-of-multiset* [*iff*]:
‹*sorted* (*sorted-list-of-multiset M*)›
⟨*proof*⟩

lemma *sorted-list-of-mset-set* [*simp*]:
sorted-list-of-multiset (*mset-set A*) = *sorted-list-of-set A*
⟨*proof*⟩

lemma *mset-upt* [*simp*]: *mset* [*m*..<*n*] = *mset-set* {*m*..<*n*}
⟨*proof*⟩

lemma *image-mset-map-of*:
distinct (*map fst xs*) ⟹ {#*the* (*map-of xs i*). *i* ∈# *mset* (*map fst xs*)#} = *mset* (*map snd xs*)
⟨*proof*⟩

lemma *msubset-mset-set-iff*[*simp*]:
assumes *finite A finite B*
shows *mset-set A* ⊆# *mset-set B* ⟷ *A* ⊆ *B*
⟨*proof*⟩

lemma *mset-set-eq-iff*[*simp*]:
assumes *finite A finite B*
shows *mset-set A* = *mset-set B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *image-mset-mset-set*:
assumes *inj-on f A*
shows *image-mset f* (*mset-set A*) = *mset-set* (*f* ‘ *A*)
⟨*proof*⟩

68.10 More properties of the replicate, repeat, and image operations

lemma *in-replicate-mset*[*simp*]: *x* ∈# *replicate-mset n y* ⟷ *n* > *0* ∧ *x* = *y*
⟨*proof*⟩

lemma *set-mset-replicate-mset-subset*[*simp*]: *set-mset* (*replicate-mset n x*) = (*if n = 0 then* {} *else* {*x*})
⟨*proof*⟩

lemma *size-replicate-mset*[*simp*]: *size* (*replicate-mset n M*) = *n*
⟨*proof*⟩

lemma *size-repeat-mset* [*simp*]: *size* (*repeat-mset n A*) = *n* ∗ *size A*
⟨*proof*⟩

lemma *size-multiset-sum* [*simp*]: *size* (∑ *x*∈*A*. *f x* :: *'a multiset*) = (∑ *x*∈*A*. *size* (*f x*))
⟨*proof*⟩

lemma *size-multiset-sum-list* [*simp*]: *size* (∑ *X*←*Xs*. *X* :: *'a multiset*) = (∑ *X*←*Xs*. *size X*)
⟨*proof*⟩

lemma *count-le-replicate-mset-subset-eq*: *n* ≤ *count M x* ⟷ *replicate-mset n x* ⊆# *M*
⟨*proof*⟩

lemma *replicate-count-mset-eq-filter-eq*: *replicate* (*count* (*mset xs*) *k*) *k* = *filter* (*HOL.eq k*) *xs*
⟨*proof*⟩

lemma *replicate-mset-eq-empty-iff* [*simp*]: *replicate-mset n a* = {#} ⟷ *n* = *0*
⟨*proof*⟩

lemma *replicate-mset-eq-iff*:
replicate-mset m a = *replicate-mset n b* ⟷ *m* = *0* ∧ *n* = *0* ∨ *m* = *n* ∧ *a* = *b*
⟨*proof*⟩

lemma *repeat-mset-cancel1*: *repeat-mset a A* = *repeat-mset a B* ⟷ *A* = *B* ∨ *a* = *0*
⟨*proof*⟩

lemma *repeat-mset-cancel2*: *repeat-mset a A* = *repeat-mset b A* ⟷ *a* = *b* ∨ *A* = {#}
⟨*proof*⟩

lemma *repeat-mset-eq-empty-iff*: *repeat-mset n A* = {#} ⟷ *n* = *0* ∨ *A* = {#}
⟨*proof*⟩

lemma *image-replicate-mset* [*simp*]:
image-mset f (*replicate-mset n a*) = *replicate-mset n* (*f a*)
⟨*proof*⟩

lemma *replicate-mset-msubseteq-iff*:

replicate-mset m a $\subseteq\#$ *replicate-mset n b* $\longleftrightarrow$ $m = 0 \vee a = b \wedge m \leq n$
⟨*proof*⟩

lemma *msubseteq-replicate-msetE*:
assumes $A \subseteq\#$ *replicate-mset n a*
obtains m **where** $m \leq n$ **and** $A =$ *replicate-mset m a*
⟨*proof*⟩

lemma *count-image-mset-lt-imp-lt-raw*:
assumes
finite A **and**
$A =$ *set-mset M* $\cup$ *set-mset N* **and**
count (*image-mset f M*) $b <$ *count* (*image-mset f N*) b
shows $\exists x.\ f\ x = b \wedge$ *count M x* $<$ *count N x*
⟨*proof*⟩

lemma *count-image-mset-lt-imp-lt*:
assumes *cnt-b*: *count* (*image-mset f M*) $b <$ *count* (*image-mset f N*) b
shows $\exists x.\ f\ x = b \wedge$ *count M x* $<$ *count N x*
⟨*proof*⟩

lemma *count-image-mset-le-imp-lt-raw*:
assumes
finite A **and**
$A =$ *set-mset M* $\cup$ *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 $\exists b.\ f\ b = f\ a \wedge$ *count M b* $<$ *count N b*
⟨*proof*⟩

lemma *count-image-mset-le-imp-lt*:
assumes
count (*image-mset f M*) (*f a*) $\leq$ *count* (*image-mset f N*) (*f a*) **and**
count M a $>$ *count N a*
shows $\exists b.\ f\ b = f\ a \wedge$ *count M b* $<$ *count N b*
⟨*proof*⟩

lemma *size-filter-unsat-elem*:
assumes $x \in\#\ M$ **and** $\neg\ P\ x$
shows *size* $\{\#x \in\#\ M.\ P\ x\#\} <$ *size M*
⟨*proof*⟩

lemma *size-filter-ne-elem*: $x \in\#\ M \Longrightarrow$ *size* $\{\#y \in\#\ M.\ y \neq x\#\} <$ *size M*
⟨*proof*⟩

lemma *size-eq-ex-count-lt*:
assumes *size M* = *size N* **and** $M \neq N$
shows $\exists x.$ *count M x* $<$ *count N x*
⟨*proof*⟩

68.11 Big operators

locale *comm-monoid-mset* = *comm-monoid*
begin

interpretation *comp-fun-commute f*
⟨*proof*⟩

interpretation *comp?*: *comp-fun-commute f* ∘ *g*
⟨*proof*⟩

context
begin

definition *F* :: *'a multiset* ⇒ *'a*
where *eq-fold*: *F M* = *fold-mset f* **1** *M*

lemma *empty* [*simp*]: *F* {#} = **1**
⟨*proof*⟩

lemma *singleton* [*simp*]: *F* {#*x*#} = *x*
⟨*proof*⟩

lemma *union* [*simp*]: *F* (*M* + *N*) = *F M* ∗ *F N*
⟨*proof*⟩

lemma *add-mset* [*simp*]: *F* (*add-mset x N*) = *x* ∗ *F N*
⟨*proof*⟩

lemma *insert* [*simp*]:
shows *F* (*image-mset g* (*add-mset x A*)) = *g x* ∗ *F* (*image-mset g A*)
⟨*proof*⟩

lemma *remove*:
assumes *x* ∈# *A*
shows *F A* = *x* ∗ *F* (*A* − {#*x*#})
⟨*proof*⟩

lemma *neutral*:
∀ *x*∈#*A*. *x* = **1** ⟹ *F A* = **1**
⟨*proof*⟩

lemma *neutral-const* [*simp*]:
F (*image-mset* (λ-. **1**) *A*) = **1**
⟨*proof*⟩ **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*#}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *distrib*: *F* (*image-mset* (λ*x*. *g x* ∗ *h x*) *A*) = *F* (*image-mset g A*) ∗ *F* (*image-mset h A*)
⟨*proof*⟩

lemma *union-disjoint*:
A ∩# *B* = {#} ⟹ *F* (*image-mset g* (*A* ∪# *B*)) = *F* (*image-mset g A*) ∗ *F* (*image-mset g B*)
⟨*proof*⟩

end
end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* ((+) :: 'a *multiset* ⇒ - ⇒ -)
⟨*proof*⟩

declare *comp-fun-commute.fold-mset-add-mset*[*OF comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: *x* ∈# *fold-mset* (+) *M NN* ⟷ *x* ∈# *M* ∨ (∃ *N*. *N* ∈# *NN* ∧ *x* ∈# *N*)
⟨*proof*⟩

context *comm-monoid-add*
begin

sublocale *sum-mset*: *comm-monoid-mset plus 0*
defines *sum-mset* = *sum-mset.F* ⟨*proof*⟩

lemma *sum-unfold-sum-mset*:
sum f A = *sum-mset* (*image-mset f* (*mset-set A*))
⟨*proof*⟩

end

notation *sum-mset* (‹∑ #›)

syntax (*ASCII*)
-sum-mset-image :: *pttrn* ⇒ 'b *set* ⇒ 'a ⇒ 'a::*comm-monoid-add*
(‹(‹*indent=3 notation=‹binder SUM››SUM* -:#-. -)› [0, 51, 10] 10)
syntax
-sum-mset-image :: *pttrn* ⇒ 'b *set* ⇒ 'a ⇒ 'a::*comm-monoid-add*
(‹(‹*indent=3 notation=‹binder* ∑››∑ -∈#-. -)› [0, 51, 10] 10)
syntax-consts
-sum-mset-image ⇌ *sum-mset*

translations
$\sum i \in\# A.\ b \rightleftharpoons CONST\ sum\text{-}mset\ (CONST\ image\text{-}mset\ (\lambda i.\ b)\ A)$

context *comm-monoid-add*
begin

lemma *sum-mset-sum-list*:
$sum\text{-}mset\ (mset\ xs) = sum\text{-}list\ xs$
⟨*proof*⟩

lemma *sum-mset-sum*: $sum\text{-}mset\ (\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ sum\text{-}mset\ (f\ x))$
⟨*proof*⟩

end

context *canonically-ordered-monoid-add*
begin

lemma *sum-mset-0-iff* [*simp*]:
$sum\text{-}mset\ M = 0 \longleftrightarrow (\forall x \in set\text{-}mset\ M.\ x = 0)$
⟨*proof*⟩

end

context *ordered-comm-monoid-add*
begin

lemma *sum-mset-mono*:
$sum\text{-}mset\ (image\text{-}mset\ f\ K) \leq sum\text{-}mset\ (image\text{-}mset\ g\ K)$
if $\bigwedge i.\ i \in\# K \Longrightarrow f\ i \leq g\ i$
⟨*proof*⟩

end

context *cancel-comm-monoid-add*
begin

lemma *sum-mset-diff*:
$sum\text{-}mset\ (M - N) = sum\text{-}mset\ M - sum\text{-}mset\ N$ **if** $N \subseteq\# M$ **for** $M\ N :: 'a$ *multiset*
⟨*proof*⟩

end

context *ab-group-add*
begin

lemma *sum-mset-negf*: $(\sum x{\in}\#M.\ -f\ x :: 'a) = -(\sum x{\in}\#M.\ f\ x)$
⟨*proof*⟩

lemma *sum-mset-subtractf*: $(\sum x{\in}\#M.\ f\ x - g\ x) = (\sum x{\in}\#M.\ f\ x) - (\sum x{\in}\#M.\ g\ x)$
⟨*proof*⟩

end

context *semiring-0*
begin

lemma *sum-mset-distrib-left*:
$c * (\sum x \in\#\ M.\ f\ x) = (\sum x \in\#\ M.\ c * f(x))$
⟨*proof*⟩

lemma *sum-mset-distrib-right*:
$(\sum x \in\#\ M.\ f\ x) * c = (\sum x \in\#\ M.\ f\ x * c)$
⟨*proof*⟩

end

lemma *sum-mset-product*:
fixes $f :: 'a{::}\{comm\text{-}monoid\text{-}add,times\} \Rightarrow 'b{::}semiring\text{-}0$
shows $(\sum i \in\#\ A.\ f\ i) * (\sum i \in\#\ B.\ g\ i) = (\sum i{\in}\#A.\ \sum j{\in}\#B.\ f\ i * g\ j)$
⟨*proof*⟩

context *semiring-1*
begin

lemma *sum-mset-replicate-mset* [*simp*]:
sum-mset (*replicate-mset n a*) = *of-nat n* * *a*
⟨*proof*⟩

lemma *sum-mset-delta*:
sum-mset (*image-mset* (λ*x*. *if x* = *y then c else 0*) *A*) = *c* * *of-nat* (*count A y*)
⟨*proof*⟩

lemma *sum-mset-delta'*:
sum-mset (*image-mset* (λ*x*. *if y* = *x then c else 0*) *A*) = *c* * *of-nat* (*count A y*)
⟨*proof*⟩

end

lemma *of-nat-sum-mset* [*simp*]:
of-nat (*sum-mset A*) = *sum-mset* (*image-mset of-nat A*)
⟨*proof*⟩

lemma *size-eq-sum-mset*:
size M = $(\sum a{\in}\#M.\ 1)$
⟨*proof*⟩

lemma *size-mset-set* [*simp*]:
size (*mset-set A*) = *card A*
⟨*proof*⟩

lemma *sum-mset-constant* [*simp*]:
fixes *y* :: *'b*::*semiring-1*
shows ‹(∑ *x*∈#*A*. *y*) = *of-nat* (*size A*) * *y*›
⟨*proof*⟩

lemma *set-mset-Union-mset*[*simp*]: *set-mset* (∑# *MM*) = (⋃ *M* ∈ *set-mset MM*. *set-mset M*)
⟨*proof*⟩

lemma *in-Union-mset-iff*[*iff*]: *x* ∈# ∑# *MM* ⟷ (∃ *M*. *M* ∈# *MM* ∧ *x* ∈# *M*)
⟨*proof*⟩

lemma *count-sum*:
count (*sum f A*) *x* = *sum* (λ*a*. *count* (*f a*) *x*) *A*
⟨*proof*⟩

lemma *sum-eq-empty-iff*:
assumes *finite A*
shows *sum f A* = {#} ⟷ (∀ *a*∈*A*. *f a* = {#})
⟨*proof*⟩

lemma *mset-concat*: *mset* (*concat xss*) = (∑ *xs*←*xss*. *mset xs*)
⟨*proof*⟩

lemma *set-mset-sum-list* [*simp*]: *set-mset* (*sum-list xs*) = (⋃ *x*∈*set xs*. *set-mset x*)
⟨*proof*⟩

lemma *filter-mset-sum-list*: *filter-mset P* (*sum-list xs*) = *sum-list* (*map* (*filter-mset P*) *xs*)
⟨*proof*⟩

lemma *sum-mset-singleton-mset* [*simp*]: (∑ *x*∈#*A*. {#*f x*#}) = *image-mset f A*
⟨*proof*⟩

lemma *sum-list-singleton-mset* [*simp*]: (∑ *x*←*xs*. {#*f x*#}) = *image-mset f* (*mset xs*)
⟨*proof*⟩

lemma *Union-mset-empty-conv*[*simp*]: ∑# *M* = {#} ⟷ (∀ *i*∈#*M*. *i* = {#})
⟨*proof*⟩

lemma *Union-image-single-mset*[*simp*]: ∑# (*image-mset* (λ*x*. {#*x*#}) *m*) = *m*
⟨*proof*⟩

lemma *size-mset-sum-mset-conv* [*simp*]: *size* ($\sum_{\#}$ *A* :: *'a multiset*) = ($\sum$ *X*∈#*A*. *size X*)
⟨*proof*⟩

lemma *sum-mset-image-mset-mono-strong*:
assumes *A* ⊆# *B* **and** *f-subeq-g*: ⋀*x*. *x* ∈# *A* ⟹ *f x* ⊆# *g x*
shows ($\sum$ *x*∈#*A*. *f x*) ⊆# ($\sum$ *x*∈#*B*. *g x*)
⟨*proof*⟩

context *comm-monoid-mult*
begin

sublocale *prod-mset*: *comm-monoid-mset times 1*
defines *prod-mset* = *prod-mset.F* ⟨*proof*⟩

lemma *prod-mset-empty*:
prod-mset {#} = *1*
⟨*proof*⟩

lemma *prod-mset-singleton*:
prod-mset {#*x*#} = *x*
⟨*proof*⟩

lemma *prod-mset-Un*:
prod-mset (*A* + *B*) = *prod-mset A* ∗ *prod-mset B*
⟨*proof*⟩

lemma *prod-mset-prod-list*:
prod-mset (*mset xs*) = *prod-list xs*
⟨*proof*⟩

lemma *prod-mset-replicate-mset* [*simp*]:
prod-mset (*replicate-mset n a*) = *a* ^ *n*
⟨*proof*⟩

lemma *prod-unfold-prod-mset*:
prod f A = *prod-mset* (*image-mset f* (*mset-set A*))
⟨*proof*⟩

lemma *prod-mset-multiplicity*:
prod-mset M = *prod* (λ*x*. *x* ^ *count M x*) (*set-mset M*)
⟨*proof*⟩

lemma *prod-mset-delta*: *prod-mset* (*image-mset* (λ*x*. *if x* = *y then c else 1*) *A*) = *c* ^ *count A y*
⟨*proof*⟩

lemma *prod-mset-delta'*: *prod-mset* (*image-mset* (λ*x*. *if y* = *x then c else 1*) *A*) = *c* ^ *count A y*

⟨*proof*⟩

lemma *prod-mset-subset-imp-dvd*:
assumes $A \subseteq\# B$
shows *prod-mset A dvd prod-mset B*
⟨*proof*⟩

lemma *dvd-prod-mset*:
assumes $x \in\# A$
shows *x dvd prod-mset A*
⟨*proof*⟩

end

notation *prod-mset* (‹$\prod_{\#}$›)

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* $\prod$ *››*$\prod$ *-∈#-. -*)›) [*0*, *51*, *10*] *10*)
syntax-consts
-prod-mset-image ⇌ *prod-mset*
translations
$\prod i \in\# A.\ b$ ⇌ *CONST prod-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma *prod-mset-constant* [*simp*]: ($\prod$ *-∈#A. c*) = *c* ^ *size A*
⟨*proof*⟩

lemma (**in** *semidom*) *prod-mset-zero-iff* [*iff*]:
prod-mset A = *0* ⟷ *0* ∈# *A*
⟨*proof*⟩

lemma (**in** *semidom-divide*) *prod-mset-diff*:
assumes $B \subseteq\# A$ **and** $0 \notin\# B$
shows *prod-mset* (*A* − *B*) = *prod-mset A div prod-mset B*
⟨*proof*⟩

lemma (**in** *semidom-divide*) *prod-mset-minus*:
assumes $a \in\# A$ **and** $a \neq 0$
shows *prod-mset* (*A* − {#*a*#}) = *prod-mset A div a*
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *normalize-prod-mset-normalize*:
normalize (*prod-mset* (*image-mset normalize A*)) = *normalize* (*prod-mset A*)
⟨*proof*⟩

lemma (**in** *algebraic-semidom*) *is-unit-prod-mset-iff*:

is-unit (*prod-mset A*) ⟷ (∀ *x* ∈# *A*. *is-unit x*)
⟨*proof*⟩

lemma (**in** *normalization-semidom-multiplicative*) *normalize-prod-mset*:
normalize (*prod-mset A*) = *prod-mset* (*image-mset normalize A*)
⟨*proof*⟩

lemma (**in** *normalization-semidom-multiplicative*) *normalized-prod-msetI*:
assumes ⋀*a*. *a* ∈# *A* ⟹ *normalize a* = *a*
shows *normalize* (*prod-mset A*) = *prod-mset A*
⟨*proof*⟩

lemma *image-prod-mset-multiplicity*:
prod-mset (*image-mset f M*) = *prod* (λ*x*. *f x* ^ *count M x*) (*set-mset M*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mset-sort* [*simp*]:
mset (*sort-key k xs*) = *mset xs*
⟨*proof*⟩

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* (λ*x*. *f k* = *f x*) *ys* = *filter* (λ*x*. *f k* = *f x*) *xs*
and *sorted* (*map f ys*)
shows *sort-key f xs* = *ys*
⟨*proof*⟩

lemma *properties-for-sort*:
assumes *multiset*: *mset ys* = *mset xs*
and *sorted ys*
shows *sort xs* = *ys*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *sort-append*:
assumes ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ *x* ≤ *y*
shows *sort* (*xs* @ *ys*) = *sort xs* @ *sort ys*
⟨*proof*⟩

lemma *sort-append-replicate-left*:
(⋀*y*. *y* ∈ *set xs* ⟹ *x* ≤ *y*) ⟹ *sort* (*replicate n x* @ *xs*) = *replicate n x* @ *sort xs*
⟨*proof*⟩

lemma *sort-append-replicate-right*:
(⋀*y*. *y* ∈ *set xs* ⟹ *x* ≥ *y*) ⟹ *sort* (*xs* @ *replicate n x*) = *sort xs* @ *replicate n x*
⟨*proof*⟩

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*))
⟨*proof*⟩

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⟩

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›
  ⟨proof⟩

lemma mset-remdups-subset-eq: mset (remdups xs) ⊆# mset xs
  ⟨proof⟩

lemma mset-update:
  i < length ls ⟹ mset (ls[i := v]) = add-mset v (mset ls − {#ls ! i#})
⟨proof⟩

lemma mset-swap:
  i < length ls ⟹ j < length ls ⟹
    mset (ls[j := ls ! i, i := ls ! j]) = mset ls
  ⟨proof⟩

lemma mset-eq-finite:
  ‹finite {ys. mset ys = mset xs}›
⟨proof⟩
```

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 $M = add\text{-}mset\ a\ M0$ **and** $N = M0 + K$ **and** $\bigwedge b.\ b \in\# K \Longrightarrow (b, a) \in r$
shows $(N, M) \in mult1\ r$
⟨*proof*⟩

lemma *mult1E*:
assumes $(N, M) \in mult1\ r$
obtains $a\ M0\ K$ **where** $M = add\text{-}mset\ a\ M0\ N = M0 + K \bigwedge b.\ b \in\# K \Longrightarrow (b, a) \in r$
⟨*proof*⟩

lemma *mono-mult1*:
assumes $r \subseteq r'$ **shows** $mult1\ r \subseteq mult1\ r'$
⟨*proof*⟩

lemma *mono-mult*:
assumes $r \subseteq r'$ **shows** $mult\ r \subseteq mult\ r'$
⟨*proof*⟩

lemma *mono-multp*[*mono*]: $r \leq r' \Longrightarrow multp\ r \leq multp\ r'$
⟨*proof*⟩

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
⟨*proof*⟩

68.13.1 Well-foundedness

lemma *less-add*:
assumes *mult1*: $(N, add\text{-}mset\ a\ M0) \in mult1\ r$
shows
$(\exists M.\ (M, M0) \in mult1\ r \wedge N = add\text{-}mset\ a\ M) \vee$
$(\exists K.\ (\forall b.\ b \in\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
⟨*proof*⟩

lemma *all-accessible*:
assumes *wf r*
shows $\forall M.\ M \in Wellfounded.acc\ (mult1\ r)$
⟨*proof*⟩

lemma *wf-mult1*: $wf\ r \Longrightarrow wf\ (mult1\ r)$
⟨*proof*⟩

lemma *wf-mult*: $wf\ r \Longrightarrow wf\ (mult\ r)$
⟨*proof*⟩

lemma *wfp-multp*: $wfp\ r \Longrightarrow wfp\ (multp\ r)$
⟨*proof*⟩

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-mset* $K.\ \exists j \in$ *set-mset* $J.\ (k, j) \in r)$
⟨*proof*⟩

lemma *multp-implies-one-step*:
transp $R \Longrightarrow$ *multp* $R\ M\ N \Longrightarrow \exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k{\in}\#K.\ \exists x{\in}\#J.\ R\ k\ x)$
⟨*proof*⟩

lemma *one-step-implies-mult*:
assumes
$J \neq \{\#\}$ **and**
$\forall k \in$ *set-mset* $K.\ \exists j \in$ *set-mset* $J.\ (k, j) \in r$
shows $(I + K, I + J) \in$ *mult r*
⟨*proof*⟩

lemma *one-step-implies-multp*:
$J \neq \{\#\} \Longrightarrow \forall k{\in}\#K.\ \exists j{\in}\#J.\ R\ k\ j \Longrightarrow$ *multp* $R\ (I + K)\ (I + J)$
⟨*proof*⟩

lemma *subset-implies-mult*:
assumes *sub*: $A \subset\# B$
shows $(A, B) \in$ *mult r*
⟨*proof*⟩

lemma *subset-implies-multp*: $A \subset\# B \Longrightarrow$ *multp r A B*
⟨*proof*⟩

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*⟩

68.13.3 Monotonicity

lemma *multp-mono-strong*:
assumes *multp R M1 M2* **and** *transp R* **and**
S-if-R: $\bigwedge x\ y.\ x \in$ *set-mset M1* $\Longrightarrow y \in$ *set-mset M2* $\Longrightarrow R\ x\ y \Longrightarrow S\ x\ y$
shows *multp S M1 M2*
⟨*proof*⟩

lemma *mult-mono-strong*:

assumes $(M1,\ M2) \in mult\ r$ **and** *trans r* **and**
S-if-R: $\bigwedge x\ y.\ x \in set\text{-}mset\ M1 \Longrightarrow y \in set\text{-}mset\ M2 \Longrightarrow (x,\ y) \in r \Longrightarrow (x,\ y) \in s$
shows $(M1,\ M2) \in mult\ s$
⟨*proof*⟩

lemma *monotone-on-multp-multp-image-mset*:
assumes *monotone-on A orda ordb f* **and** *transp orda*
shows *monotone-on* $\{M.\ set\text{-}mset\ M \subseteq A\}$ (*multp orda*) (*multp ordb*) (*image-mset f*)
⟨*proof*⟩

lemma *monotone-multp-multp-image-mset*:
assumes *monotone orda ordb f* **and** *transp orda*
shows *monotone* (*multp orda*) (*multp ordb*) (*image-mset f*)
⟨*proof*⟩

lemma *multp-image-mset-image-msetI*:
assumes *multp* $(\lambda x\ y.\ R\ (f\ x)\ (f\ y))$ *M1 M2* **and** *transp R*
shows *multp R* (*image-mset f M1*) (*image-mset f M2*)
⟨*proof*⟩

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* $(\lambda x\ y.\ R\ (f\ x)\ (f\ y))\ A\ B$
⟨*proof*⟩

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 $(X + Z,\ Y + Z) \in mult\ s \longleftrightarrow (X,\ Y) \in mult\ s$ (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *multp-cancel*:
transp R ⟹ *irreflp-on* (*set-mset Z*) *R* ⟹ *multp R* $(X + Z)$ $(Y + Z)$ ⟷ *multp R X Y*
⟨*proof*⟩

lemma *mult-cancel-add-mset*:
trans r ⟹ *irrefl-on* $\{z\}$ *r* ⟹
$((add\text{-}mset\ z\ X,\ add\text{-}mset\ z\ Y) \in mult\ r) = ((X,\ Y) \in mult\ r)$
⟨*proof*⟩

lemma *multp-cancel-add-mset*:
transp R ⟹ *irreflp-on* $\{z\}$ *R* ⟹ *multp R* (*add-mset z X*) (*add-mset z Y*) =

multp R X Y
⟨*proof*⟩

lemma *mult-cancel-max0*:
assumes *trans s* **and** *irrefl-on* (*set-mset X* ∩ *set-mset Y*) *s*
shows (*X*, *Y*) ∈ *mult s* ⟷ (*X* − *X* ∩# *Y*, *Y* − *X* ∩# *Y*) ∈ *mult s* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *mult-cancel-max*:
trans r ⟹ *irrefl-on* (*set-mset X* ∩ *set-mset Y*) *r* ⟹
(*X*, *Y*) ∈ *mult r* ⟷ (*X* − *Y*, *Y* − *X*) ∈ *mult r*
⟨*proof*⟩

lemma *multp-cancel-max*:
transp R ⟹ *irreflp-on* (*set-mset X* ∩ *set-mset Y*) *R* ⟹ *multp R X Y* ⟷ *multp R* (*X* − *Y*) (*Y* − *X*)
⟨*proof*⟩

68.13.5 Strict partial-order properties

lemma *mult1-lessE*:
assumes (*N*, *M*) ∈ *mult1* {(*a*, *b*). *r a b*} **and** *asymp r*
obtains *a M0 K* **where** *M* = *add-mset a M0 N* = *M0* + *K*
a ∉# *K* ⋀*b*. *b* ∈# *K* ⟹ *r b a*
⟨*proof*⟩

lemma *trans-on-mult*:
assumes *trans-on A r* **and** ⋀*M*. *M* ∈ *B* ⟹ *set-mset M* ⊆ *A*
shows *trans-on B* (*mult r*)
⟨*proof*⟩

lemma *trans-mult*: *trans r* ⟹ *trans* (*mult r*)
⟨*proof*⟩

lemma *transp-on-multp*:
assumes *transp-on A r* **and** ⋀*M*. *M* ∈ *B* ⟹ *set-mset M* ⊆ *A*
shows *transp-on B* (*multp r*)
⟨*proof*⟩

lemma *transp-multp*: *transp r* ⟹ *transp* (*multp r*)
⟨*proof*⟩

lemma *irrefl-mult*:
assumes *trans r irrefl r*
shows *irrefl* (*mult r*)
⟨*proof*⟩

lemma *irreflp-multp*: *transp R* ⟹ *irreflp R* ⟹ *irreflp* (*multp R*)

⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *order* **begin**

definition *less-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool*
where $M < N \longleftrightarrow$ *multp* (<) M N

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool*
where *less-eq-multiset* M $N \longleftrightarrow M < N \vee M = N$

instance
⟨*proof*⟩

end

lemma *mset-le-irrefl* [*elim*!]:
fixes M :: *'a*::*preorder multiset*
shows $M < M \Longrightarrow R$
⟨*proof*⟩

lemma *wfp-less-multiset*[*simp*]:
assumes *wf*: *wfp* ((<) :: (*'a* :: *preorder*) ⇒ *'a* ⇒ *bool*)
shows *wfp* ((<) :: *'a multiset* ⇒ *'a multiset* ⇒ *bool*)
⟨*proof*⟩

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*⟩

lemma *total-mult*: *total* $r \Longrightarrow$ *trans* $r \Longrightarrow$ *total* (*mult* r)
⟨*proof*⟩

lemma *totalp-on-multp*:
totalp-on A $R \Longrightarrow$ *transp* $R \Longrightarrow (\bigwedge M.\ M \in B \Longrightarrow$ *set-mset* $M \subseteq A) \Longrightarrow$ *totalp-on* B (*multp* R)
⟨*proof*⟩

lemma *totalp-multp*: *totalp* $R \Longrightarrow$ *transp* $R \Longrightarrow$ *totalp* (*multp* R)
⟨*proof*⟩

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* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$
where
multp-code P N M =
(*let* $Z = M \cap\# N$; $X = M - Z$ *in*
$X \neq \{\#\} \wedge$ (*let* $Y = N - Z$ *in* $(\forall y \in$ *set-mset* $Y.\ \exists x \in$ *set-mset* $X.\ P\ y\ x)$))

definition *multeqp-code* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$
where
multeqp-code P N M =
(*let* $Z = M \cap\# N$; $X = M - Z$; $Y = N - Z$ *in*
$(\forall y \in$ *set-mset* $Y.\ \exists x \in$ *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*]: $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x, y) \in R$
shows *multp-code P N M* $\longleftrightarrow (N, M) \in$ *mult R* (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *multp-code-iff-multp*:
irreflp-on (*set-mset M* ∩ *set-mset N*) *R* $\Longrightarrow$ *transp R* $\Longrightarrow$ *multp-code R M N* $\longleftrightarrow$ *multp R M N*
⟨*proof*⟩

lemma *multp-code-eq-multp*:
assumes *irreflp R* **and** *transp R*
shows *multp-code R* = *multp R*
⟨*proof*⟩

lemma *multeqp-code-iff-reflcl-mult*:
assumes *irrefl-on* (*set-mset N* ∩ *set-mset M*) *R* **and** *trans R* **and** $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x, y) \in R$
shows *multeqp-code P N M* $\longleftrightarrow (N, M) \in (mult\ R)^=$
⟨*proof*⟩

lemma *multeqp-code-iff-reflclp-multp*:
irreflp-on (*set-mset M* ∩ *set-mset N*) *R* $\Longrightarrow$ *transp R* $\Longrightarrow$ *multeqp-code R M N* $\longleftrightarrow (multp\ R)^{==}$ *M N*
⟨*proof*⟩

lemma *multeqp-code-eq-reflclp-multp*:
assumes *irreflp R* **and** *transp R*
shows *multeqp-code R* = $(multp\ R)^{==}$
⟨*proof*⟩

68.14.1 Monotonicity of multiset union

lemma *mult1-union*: $(B, D) \in mult1\ r \Longrightarrow (C + B, C + D) \in mult1\ r$
⟨*proof*⟩

lemma *union-le-mono2*: $B < D \Longrightarrow C + B < C + (D::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *union-le-mono1*: $B < D \Longrightarrow B + C < D + (C::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *union-less-mono*:
fixes $A\ B\ C\ D :: 'a::preorder\ multiset$
shows $A < C \Longrightarrow B < D \Longrightarrow A + B < C + D$
⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-add*
begin
instance
⟨*proof*⟩
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*: $x \in\# \{\#\ x\ \#\}$
⟨*proof*⟩

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
⟨*proof*⟩

lemma *smsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
⟨*proof*⟩

lemma *wmsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
⟨*proof*⟩

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*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
⟨*proof*⟩

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows $\exists A\ B\ Z.\ X = A + Z \wedge Y = B + Z \wedge ((\textit{set-mset}\ A, \textit{set-mset}\ B) \in \textit{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$
⟨*proof*⟩

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*⟩

lemma *empty-neutral*: $\{\#\} + x = x\ x + \{\#\} = x$
and *nonempty-plus*: $\{\#\ x\ \#\} + rs \neq \{\#\}$
and *nonempty-single*: $\{\#\ x\ \#\} \neq \{\#\}$
⟨*proof*⟩

⟨*ML*⟩

68.15 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ \textit{multiset})$
⟨*proof*⟩

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ \textit{multiset}))$
⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ \textit{multiset}))$
⟨*proof*⟩

lemmas *union-ac* = *union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ \textit{multiset})$
⟨*proof*⟩

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ \textit{multiset})$
⟨*proof*⟩

lemma *multi-union-self-other-eq*: $(A::'a\ \textit{multiset}) + X = A + Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *mset-subset-trans*: $(M::'a\ \textit{multiset}) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
⟨*proof*⟩

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
⟨*proof*⟩

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
⟨*proof*⟩

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
⟨*proof*⟩

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg M < (M::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-not-sym*: $M < N \Longrightarrow \neg N < (M::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-asym*: $M < N \Longrightarrow (\neg P \Longrightarrow N < (M::'a::preorder\ multiset)) \Longrightarrow P$
⟨*proof*⟩

⟨*ML*⟩

68.16 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: $\{\#\} = mset\ []$
⟨*proof*⟩

lemma [*code*]: *add-mset x* (*mset xs*) = *mset* (*x # xs*)
⟨*proof*⟩

lemma [*code*]: *Multiset.is-empty* (*mset xs*) ⟷ *List.null xs*
⟨*proof*⟩

lemma *union-code* [*code*]: *mset xs* + *mset ys* = *mset* (*xs* @ *ys*)
⟨*proof*⟩

lemma [*code*]: *image-mset f* (*mset xs*) = *mset* (*map f xs*)
⟨*proof*⟩

lemma [*code*]: *filter-mset f* (*mset xs*) = *mset* (*filter f xs*)

⟨*proof*⟩

lemma [*code*]: *mset xs − mset ys = mset* (*minus-list-mset xs ys*)
⟨*proof*⟩

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*⟩

lemma [*code*]:
mset xs ∪# *mset ys* =
mset (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, [])))
⟨*proof*⟩

declare *in-multiset-in-set* [*code-unfold*]

lemma [*code*]: *count* (*mset xs*) *x* = *fold* (λ*y*. *if x* = *y then Suc else id*) *xs 0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma [*code*]: *mset xs* ⊆# *mset ys* ⟷ *subset-eq-mset-impl xs ys* ≠ *None*
⟨*proof*⟩

lemma [*code*]: *mset xs* ⊂# *mset ys* ⟷ *subset-eq-mset-impl xs ys* = *Some True*
⟨*proof*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

declare *sum-mset-sum-list* [*code*]

lemma [*code*]: *prod-mset* (*mset xs*) = *fold times xs 1*
⟨*proof*⟩

Exercise for the casual reader: add implementations for (≤) 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 ⟨*proof*⟩

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 ⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *ex-mset*: ∃ *xs*. *mset xs* = *X*
⟨*proof*⟩

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*⟩

bnf *'a multiset*
map: *image-mset*
sets: *set-mset*
bd: *natLeq*
wits: {#}
rel: *rel-mset*
pred: *pred-mset*
⟨*proof*⟩

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* {#} {#}
⟨*proof*⟩

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*⟩

lemma *rel-mset'-imp-rel-mset*: *rel-mset' R M N* ⟹ *rel-mset R M N*
⟨*proof*⟩

lemma *rel-mset-size*: *rel-mset R M N* ⟹ *size M* = *size N*
⟨*proof*⟩

lemma *rel-mset-Zero-iff* [*simp*]:
shows *rel-mset rel* {#} *Y* ⟷ *Y* = {#} **and** *rel-mset rel X* {#} ⟷ *X* = {#}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *multiset-induct2-size*[*consumes 1*, *case-names empty add*]:
assumes *c*: *size M* = *size N*
and *empty*: *P* {#} {#}

and *add*: $\bigwedge a\ b\ M\ N\ a\ b.\ P\ M\ N \Longrightarrow P\ (add\text{-}mset\ a\ M)\ (add\text{-}mset\ b\ N)$
shows $P\ M\ N$
⟨*proof*⟩

lemma *msed-map-invL*:
assumes $image\text{-}mset\ f\ (add\text{-}mset\ a\ M) = N$
shows $\exists N1.\ N = add\text{-}mset\ (f\ a)\ N1 \wedge image\text{-}mset\ f\ M = N1$
⟨*proof*⟩

lemma *msed-map-invR*:
assumes $image\text{-}mset\ f\ M = add\text{-}mset\ b\ N$
shows $\exists M1\ a.\ M = add\text{-}mset\ a\ M1 \wedge f\ a = b \wedge image\text{-}mset\ f\ M1 = N$
⟨*proof*⟩

lemma *msed-rel-invL*:
assumes $rel\text{-}mset\ R\ (add\text{-}mset\ a\ M)\ N$
shows $\exists N1\ b.\ N = add\text{-}mset\ b\ N1 \wedge R\ a\ b \wedge rel\text{-}mset\ R\ M\ N1$
⟨*proof*⟩

lemma *msed-rel-invR*:
assumes $rel\text{-}mset\ R\ M\ (add\text{-}mset\ b\ N)$
shows $\exists M1\ a.\ M = add\text{-}mset\ a\ M1 \wedge R\ a\ b \wedge rel\text{-}mset\ R\ M1\ N$
⟨*proof*⟩

lemma *rel-mset-imp-rel-mset'*:
assumes $rel\text{-}mset\ R\ M\ N$
shows $rel\text{-}mset'\ R\ M\ N$
⟨*proof*⟩

lemma *rel-mset-rel-mset'*: $rel\text{-}mset\ R\ M\ N = rel\text{-}mset'\ R\ M\ N$
⟨*proof*⟩

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*: $size\text{-}multiset\ g \circ image\text{-}mset\ f = size\text{-}multiset\ (g \circ f)$
⟨*proof*⟩

⟨*ML*⟩

68.19 Lemmas about Size

lemma *size-mset-SucE*: $size\ A = Suc\ n \Longrightarrow (\bigwedge a\ B.\ A = \{\#a\#\} + B \Longrightarrow size\ B = n \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *size-Suc-Diff1*: $x \in\# M \Longrightarrow Suc\ (size\ (M - \{\#x\#\})) = size\ M$

⟨*proof*⟩

lemma *size-Diff-singleton*: $x \in\# M \Longrightarrow size\ (M - \{\#x\#\}) = size\ M - 1$
⟨*proof*⟩

lemma *size-Diff-singleton-if*: $size\ (A - \{\#x\#\}) = (if\ x \in\# A\ then\ size\ A - 1\ else\ size\ A)$
⟨*proof*⟩

lemma *size-Un-Int*: $size\ A + size\ B = size\ (A \cup\# B) + size\ (A \cap\# B)$
⟨*proof*⟩

lemma *size-Un-disjoint*: $A \cap\# B = \{\#\} \Longrightarrow size\ (A \cup\# B) = size\ A + size\ B$
⟨*proof*⟩

lemma *size-Diff-subset-Int*: $size\ (M - M') = size\ M - size\ (M \cap\# M')$
⟨*proof*⟩

lemma *diff-size-le-size-Diff*: $size\ (M :: - multiset) - size\ M' \leq size\ (M - M')$
⟨*proof*⟩

lemma *size-Diff1-less*: $x \in\# M \Longrightarrow size\ (M - \{\#x\#\}) < size\ M$
⟨*proof*⟩

lemma *size-Diff2-less*: $x \in\# M \Longrightarrow y \in\# M \Longrightarrow size\ (M - \{\#x\#\} - \{\#y\#\}) < size\ M$
⟨*proof*⟩

lemma *size-Diff1-le*: $size\ (M - \{\#x\#\}) \leq size\ M$
⟨*proof*⟩

lemma *size-psubset*: $M \subseteq\# M' \Longrightarrow size\ M < size\ M' \Longrightarrow M \subset\# M'$
⟨*proof*⟩

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* $X \subseteq A$
and *multisets-of-size-size*: *size* $X = n$
⟨*proof*⟩

lemma *multisets-of-size-mono*:
assumes $A \subseteq B$
shows *multisets-of-size* A $n \subseteq$ *multisets-of-size* B n
⟨*proof*⟩

lemma *multisets-of-size-0* [*simp*]: *multisets-of-size* A $0 = \{\{\#\}\}$
⟨*proof*⟩

lemma *multisets-of-size-empty* [*simp*]: $n > 0 \Longrightarrow$ *multisets-of-size* $\{\}$ $n = \{\}$
⟨*proof*⟩

lemma *count-le-size*: *count* X $x \leq$ *size* X
⟨*proof*⟩

lemma *bij-betw-multisets-of-size-insert*:
assumes $a \notin A$
shows *bij-betw* ($\lambda(m,X)$. X + *replicate-mset* m a)
(*SIGMA* m:{0..n}. *multisets-of-size* A ($n - m$)) (*multisets-of-size* (*insert* a A) n)
⟨*proof*⟩

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*⟩

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* . $m \leftarrow$ [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
⟨*proof*⟩

lemma *multisets-of-size-code* [*code*]:
multisets-of-size (*set xs*) n = *set* (*map mset* (*multisets-of-size-list* (*remdups xs*) n))
⟨*proof*⟩

lemma *finite-multisets-of-size* [*intro*]:
assumes *finite* A
shows *finite* (*multisets-of-size* A n)

⟨*proof*⟩

lemma *card-multisets-of-size*:
assumes *finite A*
shows *card* (*multisets-of-size A n*) = (*card A* + *n* − *1*) *choose n*
⟨*proof*⟩

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**
$multp_{DM}\ r\ M\ N \longleftrightarrow$
$(\exists\, X\ Y.\ X \neq \{\#\} \wedge X \subseteq\# N \wedge M = (N - X) + Y \wedge (\forall\, k.\ k \in\# Y \longrightarrow (\exists\, a.\ a \in\# X \wedge r\ k\ a)))$

lemma $multp_{DM}$*-imp-multp*:
$multp_{DM}\ r\ M\ N \Longrightarrow multp\ r\ M\ N$
⟨*proof*⟩

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* $\Longrightarrow$ $multp_{HO}\ r\ M\ N$
⟨*proof*⟩

lemma $multp_{HO}$*-imp-*$multp_{DM}$: $multp_{HO}\ r\ M\ N \Longrightarrow multp_{DM}\ r\ M\ N$
⟨*proof*⟩

lemma *multp-eq-*$multp_{DM}$: *asymp r* $\Longrightarrow$ *transp r* $\Longrightarrow$ *multp r* = $multp_{DM}\ r$
⟨*proof*⟩

lemma *multp-eq-*$multp_{HO}$: *asymp r* $\Longrightarrow$ *transp r* $\Longrightarrow$ *multp r* = $multp_{HO}\ r$
⟨*proof*⟩

lemma $multp_{DM}$*-plus-plusI*[*simp*]:

assumes $mult_{DM}\ R\ M1\ M2$
shows $mult_{DM}\ R\ (M + M1)\ (M + M2)$
⟨*proof*⟩

lemma $mult_{HO}$*-plus-plus*[*simp*]: $mult_{HO}\ R\ (M + M1)\ (M + M2) \longleftrightarrow mult_{HO}\ R\ M1\ M2$
⟨*proof*⟩

lemma *strict-subset-implies-*$mult_{DM}$: $A \subset\# B \Longrightarrow mult_{DM}\ r\ A\ B$
⟨*proof*⟩

lemma *strict-subset-implies-*$mult_{HO}$: $A \subset\# B \Longrightarrow mult_{HO}\ r\ A\ B$
⟨*proof*⟩

lemma $mult_{HO}$*-implies-one-step-strong*:
assumes $mult_{HO}\ R\ A\ B$
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*⟩

lemma $mult_{HO}$*-minus-inter-minus-inter-iff*:
fixes *M1 M2* :: - *multiset*
shows $mult_{HO}\ R\ (M1 - M2)\ (M2 - M1) \longleftrightarrow mult_{HO}\ R\ M1\ M2$
⟨*proof*⟩

lemma $mult_{HO}$*-iff-set-mset-less*$_{HO}$*-set-mset*:
$mult_{HO}\ R\ M1\ M2 \longleftrightarrow (set\text{-}mset\ (M1 - M2) \neq set\text{-}mset\ (M2 - M1) \wedge$
$(\forall y \in\# M1 - M2.\ (\exists x \in\# M2 - M1.\ R\ y\ x)))$
⟨*proof*⟩

69.1.3 Monotonicity

lemma $mult_{DM}$*-mono-strong*:
$mult_{DM}\ R\ M1\ M2 \Longrightarrow (\bigwedge x\ y.\ x \in\# M1 \Longrightarrow y \in\# M2 \Longrightarrow R\ x\ y \Longrightarrow S\ x\ y) \Longrightarrow mult_{DM}\ S\ M1\ M2$
⟨*proof*⟩

lemma $mult_{HO}$*-mono-strong*:
$mult_{HO}\ R\ M1\ M2 \Longrightarrow (\bigwedge x\ y.\ x \in\# M1 \Longrightarrow y \in\# M2 \Longrightarrow R\ x\ y \Longrightarrow S\ x\ y) \Longrightarrow mult_{HO}\ S\ M1\ M2$
⟨*proof*⟩

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 $mult_{HO}$. It suffices to have four distinct values to build a counterexample.

lemma *asymp-not-liftable-to-*$mult_{HO}$:
fixes *a b c d* :: *'a*

assumes *distinct* [*a*, *b*, *c*, *d*]
shows ¬ (∀(*R* :: *'a* ⇒ *'a* ⇒ *bool*). *asymp R* ⟶ *asymp* (*multp*$_{HO}$ *R*))
⟨*proof*⟩

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*⟩

lemma *asymp-multp*$_{HO}$:
assumes *asymp R* **and** *transp R*
shows *asymp* (*multp*$_{HO}$ *R*)
⟨*proof*⟩

Irreflexivity **lemma** *irreflp-on-multp*$_{HO}$[*simp*]: *irreflp-on B* (*multp*$_{HO}$ *R*)
⟨*proof*⟩

Transitivity **lemma** *transp-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 *transp-on B* (*multp*$_{HO}$ *R*)
⟨*proof*⟩

lemma *transp-multp*$_{HO}$:
assumes *asymp R* **and** *transp R*
shows *transp* (*multp*$_{HO}$ *R*)
⟨*proof*⟩

Totality **lemma** *totalp-on-multp*$_{DM}$:
totalp-on A R ⟹ (⋀*M*. *M* ∈ *B* ⟹ *set-mset M* ⊆ *A*) ⟹ *totalp-on B* (*multp*$_{DM}$ *R*)
⟨*proof*⟩

lemma *totalp-multp*$_{DM}$: *totalp R* ⟹ *totalp* (*multp*$_{DM}$ *R*)
⟨*proof*⟩

lemma *totalp-on-multp*$_{HO}$:
totalp-on A R ⟹ (⋀*M*. *M* ∈ *B* ⟹ *set-mset M* ⊆ *A*) ⟹ *totalp-on B* (*multp*$_{HO}$ *R*)
⟨*proof*⟩

lemma *totalp-multp*$_{HO}$: *totalp R* ⟹ *totalp* (*multp*$_{HO}$ *R*)
⟨*proof*⟩

Type Classes **context** *preorder*
begin

lemma *order-mult*: *class.order*
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ∨ *M* = *N*)
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*})
(**is** *class.order ?le ?less*)
⟨*proof*⟩

The Dershowitz–Manna ordering:

definition *less-multiset*$_{DM}$ **where**
less-multiset$_{DM}$ *M N* ⟷
(∃ *X Y*. *X* ≠ {#} ∧ *X* ⊆# *N* ∧ *M* = (*N* − *X*) + *Y* ∧ (∀ *k*. *k* ∈# *Y* ⟶ (∃ *a*. *a* ∈# *X* ∧ *k* < *a*)))

The Huet–Oppen ordering:

definition *less-multiset*$_{HO}$ **where**
less-multiset$_{HO}$ *M N* ⟷ *M* ≠ *N* ∧ (∀ *y*. *count N y* < *count M y* ⟶ (∃ *x*. *y* < *x* ∧ *count M x* < *count N x*))

lemma *mult-imp-less-multiset*$_{HO}$:
(*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟹ *less-multiset*$_{HO}$ *M N*
⟨*proof*⟩

lemma *less-multiset*$_{DM}$*-imp-mult*:
less-multiset$_{DM}$ *M N* ⟹ (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*}
⟨*proof*⟩

lemma *less-multiset*$_{HO}$*-imp-less-multiset*$_{DM}$: *less-multiset*$_{HO}$ *M N* ⟹ *less-multiset*$_{DM}$ *M N*
⟨*proof*⟩

lemma *mult-less-multiset*$_{DM}$: (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟷ *less-multiset*$_{DM}$ *M N*
⟨*proof*⟩

lemma *mult-less-multiset*$_{HO}$: (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟷ *less-multiset*$_{HO}$ *M N*
⟨*proof*⟩

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* ⟷ *less-multiset*$_{HO}$ *M N*
⟨*proof*⟩

lemma *less-multiset*$_{DM}$:
M < *N* ⟷ (∃ *X Y*. *X* ≠ {#} ∧ *X* ⊆# *N* ∧ *M* = *N* − *X* + *Y* ∧ (∀ *k*. *k* ∈# *Y* ⟶ (∃ *a*. *a* ∈# *X* ∧ *k* < *a*)))

⟨*proof*⟩

lemma *less-multiset$_{HO}$*:
$M < N \longleftrightarrow M \neq N \wedge (\forall y.\ count\ N\ y < count\ M\ y \longrightarrow (\exists x{>}y.\ count\ M\ x < count\ N\ x))$
⟨*proof*⟩

lemma *subset-eq-imp-le-multiset*:
shows $M \subseteq\# N \Longrightarrow M \leq N$
⟨*proof*⟩

lemma *le-multiset-right-total*: $M < add\text{-}mset\ x\ M$
⟨*proof*⟩

lemma *less-eq-multiset-empty-left*[*simp*]: $\{\#\} \leq M$
⟨*proof*⟩

lemma *ex-gt-imp-less-multiset*: $(\exists y.\ y \in\# N \wedge (\forall x.\ x \in\# M \longrightarrow x < y)) \Longrightarrow M < N$
⟨*proof*⟩

lemma *less-eq-multiset-empty-right*[*simp*]: $M \neq \{\#\} \Longrightarrow \neg\ M \leq \{\#\}$
⟨*proof*⟩

lemma *le-multiset-empty-left*[*simp*]: $M \neq \{\#\} \Longrightarrow \{\#\} < M$
⟨*proof*⟩

lemma *le-multiset-empty-right*[*simp*]: $\neg\ M < \{\#\}$
⟨*proof*⟩

lemma *union-le-diff-plus*: $P \subseteq\# M \Longrightarrow N < P \Longrightarrow M - P + N < M$
⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-monoid-add-imp-le*
begin

lemma *less-eq-multiset$_{HO}$*:
$M \leq N \longleftrightarrow (\forall y.\ count\ N\ y < count\ M\ y \longrightarrow (\exists x.\ y < x \wedge count\ M\ x < count\ N\ x))$
⟨*proof*⟩

instance ⟨*proof*⟩

lemma
fixes $M\ N$ :: *'a multiset*

shows *less-eq-multiset-plus-left*: $N \leq (M + N)$
and *less-eq-multiset-plus-right*: $M \leq (M + N)$
⟨*proof*⟩

lemma
fixes $M\ N :: 'a\ multiset$
shows *le-multiset-plus-left-nonempty*: $M \neq \{\#\} \Longrightarrow N < M + N$
and *le-multiset-plus-right-nonempty*: $N \neq \{\#\} \Longrightarrow M < M + N$
⟨*proof*⟩

end

lemma *all-lt-Max-imp-lt-mset*: $N \neq \{\#\} \Longrightarrow (\forall\, a \in\# M.\ a < Max\ (set\text{-}mset\ N)) \Longrightarrow M < N$
⟨*proof*⟩

lemma *lt-imp-ex-count-lt*: $M < N \Longrightarrow \exists\, y.\ count\ M\ y < count\ N\ y$
⟨*proof*⟩

lemma *subset-imp-less-mset*: $A \subset\# B \Longrightarrow A < B$
⟨*proof*⟩

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*⟩

lemma *image-mset-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*: $M \leq N$
shows *image-mset f M* $\leq$ *image-mset f N*
⟨*proof*⟩

lemma *mset-lt-single-right-iff* [*simp*]: $M < \{\#y\#\} \longleftrightarrow (\forall\, x \in\# M.\ x < y)$ **for** $y :: 'a::linorder$
⟨*proof*⟩

lemma *mset-le-single-right-iff* [*simp*]:
$M \leq \{\#y\#\} \longleftrightarrow M = \{\#y\#\} \vee (\forall\, x \in\# M.\ x < y)$ **for** $y :: 'a::linorder$
⟨*proof*⟩

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*⟩

lemma *multp$_{HO}$-double-double*[*simp*]: *multp$_{HO}$ R* (*A* + *A*) (*B* + *B*) ⟷ *multp$_{HO}$ R A B*
⟨*proof*⟩

69.2 Simprocs

lemma *mset-le-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* ≤ *repeat-mset j u* + *n*) = (*repeat-mset* (*i*−*j*) *u* + *m* ≤ *n*)
⟨*proof*⟩

lemma *mset-le-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* ≤ *repeat-mset j u* + *n*) = (*m* ≤ *repeat-mset* (*j*−*i*) *u* + *n*)
⟨*proof*⟩

⟨*ML*⟩

69.3 Additional facts and instantiations

lemma *ex-gt-count-imp-le-multiset*:
(∀ *y* :: *'a* :: *order*. *y* ∈# *M* + *N* ⟶ *y* ≤ *x*) ⟹ *count M x* < *count N x* ⟹ *M* < *N*
⟨*proof*⟩

lemma *mset-lt-single-iff*[*iff*]: {#*x*#} < {#*y*#} ⟷ *x* < *y*
⟨*proof*⟩

lemma *mset-le-single-iff*[*iff*]: {#*x*#} ≤ {#*y*#} ⟷ *x* ≤ *y* **for** *x y* :: *'a*::*order*
⟨*proof*⟩

instance *multiset* :: (*linorder*) *linordered-cancel-ab-semigroup-add*
⟨*proof*⟩

lemma *less-eq-multiset-total*: ¬ *M* ≤ *N* ⟹ *N* ≤ *M* **for** *M N* :: *'a* :: *linorder multiset*
⟨*proof*⟩

instantiation *multiset* :: (*wellorder*) *wellorder*
begin

lemma *wf-less-multiset*: *wf* {(*M* :: *'a multiset*, *N*). *M* < *N*}
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *multiset* :: (*preorder*) *order-bot*

begin

definition *bot-multiset* :: *'a multiset* **where** *bot-multiset* = {#}

instance ⟨*proof*⟩

end

instance *multiset* :: (*preorder*) *no-top*
⟨*proof*⟩

instance *multiset* :: (*preorder*) *ordered-cancel-comm-monoid-add*
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *add-mset-lt-left-lt*: *a* < *b* ⟹ *add-mset a A* < *add-mset b A*
⟨*proof*⟩

lemma *add-mset-le-left-le*: *a* ≤ *b* ⟹ *add-mset a A* ≤ *add-mset b A* **for** *a* :: *'a* ::
linorder
⟨*proof*⟩

lemma *add-mset-lt-right-lt*: *A* < *B* ⟹ *add-mset a A* < *add-mset a B*
⟨*proof*⟩

lemma *add-mset-le-right-le*: *A* ≤ *B* ⟹ *add-mset a A* ≤ *add-mset a B*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-mset-lt-lt-le*: *a* < *b* ⟹ *A* ≤ *B* ⟹ *add-mset a A* < *add-mset b B*
⟨*proof*⟩

lemma *add-mset-lt-le-lt*: $a \leq b \Longrightarrow A < B \Longrightarrow$ *add-mset a A* $<$ *add-mset b B* **for** $a :: 'a ::$ *linorder*
⟨*proof*⟩

lemma *add-mset-le-le-le*:
fixes $a :: 'a ::$ *linorder*
assumes *a-le-b*: $a \leq b$ **and** *A-le-B*: $A \leq B$
shows *add-mset a A* $\leq$ *add-mset b B*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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* $= n$
⟨*proof*⟩

lemma *in-nlists-UNIV*: *xs* $\in$ *nlists k UNIV* $\longleftrightarrow$ *length xs* $= k$
⟨*proof*⟩

lemma *less-lengthI*: $[\![$ *xs* $\in$ *nlists n A*; $p < n$ $]\!] \Longrightarrow p <$ *size xs*
⟨*proof*⟩

lemma *nlistsE-set*[*simp*]: *xs* $\in$ *nlists n A* $\Longrightarrow$ *set xs* $\subseteq A$
⟨*proof*⟩

lemma *nlists-mono*:
assumes $A \subseteq B$ **shows** *nlists n A* $\subseteq$ *nlists n B*
⟨*proof*⟩

lemma *nlists-singleton*: *nlists n* $\{a\}$ = $\{$*replicate n a*$\}$

⟨*proof*⟩

lemma *nlists-n-0* [*simp*]: *nlists 0 A* = {[]}
⟨*proof*⟩

lemma *in-nlists-Suc-iff*: (*xs* ∈ *nlists* (*Suc n*) *A*) = (∃ *y*∈*A*. ∃ *ys* ∈ *nlists n A*. *xs* = *y*#*ys*)
⟨*proof*⟩

lemma *Cons-in-nlists-Suc* [*iff*]: (*x*#*xs* ∈ *nlists* (*Suc n*) *A*) ⟷ (*x*∈*A* ∧ *xs* ∈ *nlists n A*)
⟨*proof*⟩

lemma *nlists-Suc*: *nlists* (*Suc n*) *A* = (⋃ *a*∈*A*. (#) *a* ‘ *nlists n A*)
⟨*proof*⟩

lemma *nlists-not-empty*: *A*≠{} ⟹ ∃ *xs*. *xs* ∈ *nlists n A*
⟨*proof*⟩

lemma *nlistsE-nth-in*: ⟦ *xs* ∈ *nlists n A*; *i* < *n* ⟧ ⟹ *xs*!*i* ∈ *A*
⟨*proof*⟩

lemma *nlists-Cons-Suc* [*elim*!]:
l#*xs* ∈ *nlists n A* ⟹ (⋀*n*′. *n* = *Suc n*′ ⟹ *l* ∈ *A* ⟹ *xs* ∈ *nlists n*′ *A* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

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*⟩

lemma *nlists-update-in-list* [*simp*, *intro*!]:
⟦ *xs* ∈ *nlists n A*; *x*∈*A* ⟧ ⟹ *xs*[*i* := *x*] ∈ *nlists n A*
⟨*proof*⟩

lemma *nlists-appendI* [*intro?*]:
⟦ *a* ∈ *nlists n A*; *b* ∈ *nlists m A* ⟧ ⟹ *a* @ *b* ∈ *nlists* (*n*+*m*) *A*
⟨*proof*⟩

lemma *nlists-append*:
xs @ *ys* ∈ *nlists k A* ⟷
k = *length*(*xs* @ *ys*) ∧ *xs* ∈ *nlists* (*length xs*) *A* ∧ *ys* ∈ *nlists* (*length ys*) *A*
⟨*proof*⟩

lemma *nlists-map* [*simp*]: (*map f xs* ∈ *nlists* (*size xs*) *A*) = (*f* ‘ *set xs* ⊆ *A*)
⟨*proof*⟩

lemma *nlists-replicateI* [*intro*]: $x \in A \Longrightarrow$ *replicate n x* $\in$ *nlists n A*
⟨*proof*⟩

Link to an executable version on lists in List.

lemma *nlists-set*[*code*]: *nlists n* (*set xs*) = *set*(*List.n-lists n xs*)
⟨*proof*⟩

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* $\leq$ *0*$\}$

lemma *zero-in-nonpos-Ints* [*simp*,*intro*]: *0* $\in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *neg-one-in-nonpos-Ints* [*simp*,*intro*]: $-1 \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *neg-numeral-in-nonpos-Ints* [*simp*,*intro*]: $-$*numeral n* $\in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *one-notin-nonpos-Ints* [*simp*]: (*1* :: *'a* :: *ring-char-0*) $\notin \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *numeral-notin-nonpos-Ints* [*simp*]: (*numeral n* :: *'a* :: *ring-char-0*) $\notin \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *minus-of-nat-in-nonpos-Ints* [*simp*, *intro*]: $-$ *of-nat n* $\in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *of-nat-in-nonpos-Ints-iff*: (*of-nat n* :: *'a* :: {*ring-1*,*ring-char-0*}) $\in \mathbb{Z}_{\leq 0}$
$\longleftrightarrow$ *n* = *0*
⟨*proof*⟩

lemma *nonpos-Ints-of-int*: *n* $\leq$ *0* $\Longrightarrow$ *of-int n* $\in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-IntsI*:
x $\in \mathbb{Z} \Longrightarrow$ *x* $\leq$ *0* $\Longrightarrow$ (*x* :: *'a* :: *linordered-idom*) $\in \mathbb{Z}_{\leq 0}$

⟨*proof*⟩

lemma *nonpos-Ints-subset-Ints*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{Z}$
⟨*proof*⟩

lemma *nonpos-Ints-nonpos* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow x \leq (0 :: 'a :: \textit{linordered-idom})$
⟨*proof*⟩

lemma *nonpos-Ints-Int* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow x \in \mathbb{Z}$
⟨*proof*⟩

lemma *nonpos-Ints-cases*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = \textit{of-int}\ n\ n \leq 0$
⟨*proof*⟩

lemma *nonpos-Ints-cases'*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = -\textit{of-nat}\ n$
⟨*proof*⟩

lemma *of-real-in-nonpos-Ints-iff*: $(\textit{of-real}\ x :: 'a :: \textit{real-algebra-1}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-altdef*: $\mathbb{Z}_{\leq 0} = \{n \in \mathbb{Z}.\ (n :: 'a :: \textit{linordered-idom}) \leq 0\}$
⟨*proof*⟩

lemma *uminus-in-Nats-iff*: $-x \in \mathbb{N} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *uminus-in-nonpos-Ints-iff*: $-x \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{N}$
⟨*proof*⟩

lemma *nonpos-Ints-mult*: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x * y \in \mathbb{N}$
⟨*proof*⟩

lemma *Nats-mult-nonpos-Ints*: $x \in \mathbb{N} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x * y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-mult-Nats*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{N} \Longrightarrow x * y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-add*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x + y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-diff-Nats*:

$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{N} \Longrightarrow x - y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *Nats-diff-nonpos-Ints*:
$x \in \mathbb{N} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x - y \in \mathbb{N}$
⟨*proof*⟩

lemma *plus-of-nat-eq-0-imp*: $z + \textit{of-nat}\ n = 0 \Longrightarrow z \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

71.2 Non-negative reals

definition *nonneg-Reals* :: *'a*::*real-algebra-1 set* (‹$\mathbb{R}_{\geq 0}$›)
where $\mathbb{R}_{\geq 0} = \{\textit{of-real}\ r \mid r.\ r \geq 0\}$

lemma *nonneg-Reals-of-real-iff* [*simp*]: $\textit{of-real}\ r \in \mathbb{R}_{\geq 0} \longleftrightarrow r \geq 0$
⟨*proof*⟩

lemma *nonneg-Reals-subset-Reals*: $\mathbb{R}_{\geq 0} \subseteq \mathbb{R}$
⟨*proof*⟩

lemma *nonneg-Reals-Real* [*dest*]: $x \in \mathbb{R}_{\geq 0} \Longrightarrow x \in \mathbb{R}$
⟨*proof*⟩

lemma *nonneg-Reals-of-nat-I* [*simp*]: $\textit{of-nat}\ n \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-cases*:
assumes $x \in \mathbb{R}_{\geq 0}$
obtains r **where** $x = \textit{of-real}\ r\ r \geq 0$
⟨*proof*⟩

lemma *nonneg-Reals-zero-I* [*simp*]: $0 \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-one-I* [*simp*]: $1 \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-minus-one-I* [*simp*]: $-1 \notin \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-numeral-I* [*simp*]: $\textit{numeral}\ w \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-minus-numeral-I* [*simp*]: $-\ \textit{numeral}\ w \notin \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-add-I* [*simp*]: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a + b \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-mult-I* [*simp*]: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-inverse-I* [*simp*]:
fixes a :: *'a*::*real-div-algebra*
shows $a \in \mathbb{R}_{\geq 0} \Longrightarrow$ *inverse* $a \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-divide-I* [*simp*]:
fixes a :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-pow-I* [*simp*]: $a \in \mathbb{R}_{\geq 0} \Longrightarrow a\hat{}n \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *complex-nonneg-Reals-iff*: $z \in \mathbb{R}_{\geq 0} \longleftrightarrow$ *Re* $z \geq 0 \wedge$ *Im* $z = 0$
⟨*proof*⟩

lemma *ii-not-nonneg-Reals* [*iff*]: $\mathrm{i} \notin \mathbb{R}_{\geq 0}$
⟨*proof*⟩

71.3 Non-positive reals

definition *nonpos-Reals* :: *'a*::*real-algebra-1 set* (‹$\mathbb{R}_{\leq 0}$›)
where $\mathbb{R}_{\leq 0} = \{$*of-real* $r \mid r.\ r \leq 0\}$

lemma *nonpos-Reals-of-real-iff* [*simp*]: *of-real* $r \in \mathbb{R}_{\leq 0} \longleftrightarrow r \leq 0$
⟨*proof*⟩

lemma *nonpos-Reals-subset-Reals*: $\mathbb{R}_{\leq 0} \subseteq \mathbb{R}$
⟨*proof*⟩

lemma *nonpos-Ints-subset-nonpos-Reals*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-of-nat-iff* [*simp*]: *of-nat* $n \in \mathbb{R}_{\leq 0} \longleftrightarrow n{=}0$
⟨*proof*⟩

lemma *nonpos-Reals-Real* [*dest*]: $x \in \mathbb{R}_{\leq 0} \Longrightarrow x \in \mathbb{R}$
⟨*proof*⟩

lemma *nonpos-Reals-cases*:
assumes $x \in \mathbb{R}_{\leq 0}$
obtains r **where** $x =$ *of-real* r $r \leq 0$
⟨*proof*⟩

lemma *uminus-nonneg-Reals-iff* [*simp*]: $-x \in \mathbb{R}_{\geq 0} \longleftrightarrow x \in \mathbb{R}_{\leq 0}$

⟨*proof*⟩

lemma *uminus-nonpos-Reals-iff* [*simp*]: $-x \in \mathbb{R}_{\leq 0} \longleftrightarrow x \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-zero-I* [*simp*]: $0 \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-one-I* [*simp*]: $1 \notin \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-numeral-I* [*simp*]: *numeral* $w \notin \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-add-I* [*simp*]: $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a + b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-mult-I1*: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-mult-I2*: $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-mult-of-nat-iff*:
fixes *a*:: *'a* :: *real-div-algebra* **shows** $a * \mathit{of\text{-}nat}\ n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n{=}0$
⟨*proof*⟩

lemma *nonpos-Reals-inverse-I*:
fixes *a* :: *'a*::*real-div-algebra*
shows $a \in \mathbb{R}_{\leq 0} \Longrightarrow \mathit{inverse}\ a \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-divide-I1*:
fixes *a* :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-divide-I2*:
fixes *a* :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-divide-of-nat-iff*:
fixes *a*:: *'a* :: *real-div-algebra* **shows** $a\ /\ \mathit{of\text{-}nat}\ n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n{=}0$
⟨*proof*⟩

lemma *nonpos-Reals-inverse-iff* [*simp*]:
fixes *a* :: *'a*::*real-div-algebra*
shows $\mathit{inverse}\ a \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0}$

⟨*proof*⟩

lemma *nonpos-Reals-pow-I*: ⟦$a \in \mathbb{R}_{\leq 0}$; *odd n*⟧ ⟹ $a \hat{} n \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *complex-nonpos-Reals-iff*: $z \in \mathbb{R}_{\leq 0} \longleftrightarrow Re\ z \leq 0 \wedge Im\ z = 0$
⟨*proof*⟩

lemma *ii-not-nonpos-Reals* [*iff*]: $\mathrm{i} \notin \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *plus-one-in-nonpos-Ints-imp*: $z + 1 \in \mathbb{Z}_{\leq 0} \Longrightarrow z \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *of-int-in-nonpos-Ints-iff*:
(*of-int n* :: *'a* :: *ring-char-0*) $\in \mathbb{Z}_{\leq 0} \longleftrightarrow n \leq 0$
⟨*proof*⟩

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*⟩

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*⟩

lemma *fraction-not-in-Nats*:
assumes ¬*n dvd m n* ≠ *0*
shows *of-int m / of-int n* ∉ (**ℕ** :: *'a* :: {*division-ring*,*ring-char-0*} *set*)
⟨*proof*⟩

lemma *not-in-Ints-imp-not-in-nonpos-Ints*: $z \notin \mathbb{Z} \Longrightarrow z \notin \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *double-in-nonpos-Ints-imp*:
assumes *2* ∗ (*z* :: *'a* :: *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*⟩

lemma *fraction-numeral-Ints-iff* [*simp*]:
numeral a / numeral b ∈ (**ℤ** :: *'a* :: {*division-ring*, *ring-char-0*} *set*)
⟷ (*numeral b* :: *int*) *dvd numeral a* (**is** *?L=?R*)
⟨*proof*⟩

lemma *fraction-numeral-Ints-iff1* [*simp*]:
1 / numeral b ∈ (**ℤ** :: *'a* :: {*division-ring*, *ring-char-0*} *set*)
⟷ *b* = *Num.One* (**is** *?L=?R*)
⟨*proof*⟩

lemma *fraction-numeral-Nats-iff* [*simp*]:
numeral a / numeral b ∈ (ℕ :: *'a* :: {*division-ring*, *ring-char-0*} *set*)
⟷ (*numeral b* :: *int*) *dvd numeral a* (**is** *?L=?R*)
⟨*proof*⟩

lemma *fraction-numeral-Nats-iff1* [*simp*]:
1 / numeral b ∈ (ℕ :: *'a* :: {*division-ring*, *ring-char-0*} *set*)
⟷ *b* = *Num.One* (**is** *?L=?R*)
⟨*proof*⟩

end

72 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

72.1 Numeral Types

typedef *num0* = *UNIV* :: *nat set* ⟨*proof*⟩
typedef *num1* = *UNIV* :: *unit set* ⟨*proof*⟩

typedef *'a bit0* = {*0* ..< *2* ∗ *int CARD*(*'a*::*finite*)}
⟨*proof*⟩

typedef *'a bit1* = {*0* ..< *1* + *2* ∗ *int CARD*(*'a*::*finite*)}
⟨*proof*⟩

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
⟨*proof*⟩

lemma *infinite-num0*: ¬ *finite* (*UNIV* :: *num0 set*)
⟨*proof*⟩

lemma *card-num1* [*simp*]: *CARD*(*num1*) = *1*
⟨*proof*⟩

lemma *card-bit0* [*simp*]: *CARD*(*'a bit0*) = *2* ∗ *CARD*(*'a*::*finite*)
⟨*proof*⟩

lemma *card-bit1* [*simp*]: *CARD*(*'a bit1*) = *Suc* (*2* ∗ *CARD*(*'a*::*finite*))
⟨*proof*⟩

72.2 *num1*

instance *num1* :: *finite*
⟨*proof*⟩

instantiation *num1* :: *CARD-1*
begin

instance
⟨*proof*⟩

end

lemma *num1-eq-iff*: (*x*::*num1*) = (*y*::*num1*) ⟷ *True*
⟨*proof*⟩

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*numeral*}
begin

instance
⟨*proof*⟩

end

lemma *num1-eqI*:
fixes *a*::*num1* **shows** *a* = *b*
⟨*proof*⟩

lemma *num1-eq1* [*simp*]:
fixes *a*::*num1* **shows** *a* = *1*
⟨*proof*⟩

lemma *forall-1*[*simp*]: (∀ *i*::*num1*. *P i*) ⟷ *P 1*
⟨*proof*⟩

lemma *ex-1*[*simp*]: (∃ *x*::*num1*. *P x*) ⟷ *P 1*
⟨*proof*⟩

instantiation *num1* :: *linorder* **begin**
definition *a* < *b* ⟷ *Rep-num1 a* < *Rep-num1 b*
definition *a* ≤ *b* ⟷ *Rep-num1 a* ≤ *Rep-num1 b*
instance
⟨*proof*⟩
end

instance *num1* :: *wellorder*
⟨*proof*⟩

instance *bit0* :: (*finite*) *card2*
⟨*proof*⟩

instance *bit1* :: (*finite*) *card2*
⟨*proof*⟩

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*: $0 < n$
  ⟨*proof*⟩

lemmas *definitions* =
  zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: $Rep\ x < n$
  ⟨*proof*⟩

lemma *Rep-le-n*: $Rep\ x \leq n$
  ⟨*proof*⟩

lemma *Rep-inject-sym*: $x = y \longleftrightarrow Rep\ x = Rep\ y$
  ⟨*proof*⟩

lemma *Rep-inverse*: $Abs\ (Rep\ x) = x$
  ⟨*proof*⟩

lemma *Abs-inverse*: $m \in \{0..<n\} \Longrightarrow Rep\ (Abs\ m) = m$
  ⟨*proof*⟩

lemma *Rep-Abs-mod*: $Rep\ (Abs\ (m\ mod\ n)) = m\ mod\ n$
  ⟨*proof*⟩

lemma *Rep-Abs-0*: $Rep\ (Abs\ 0) = 0$
  ⟨*proof*⟩

lemma *Rep-0*: $Rep\ 0 = 0$
  ⟨*proof*⟩

lemma *Rep-Abs-1*: $Rep\ (Abs\ 1) = 1$
  ⟨*proof*⟩

lemma *Rep-1*: $Rep\ 1 = 1$

⟨*proof*⟩

lemma *Rep-mod*: *Rep x mod n = Rep x*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *of-int-eq*: *of-int z = Abs (z mod n)*
⟨*proof*⟩

lemma *Rep-numeral*: *Rep (numeral w) = numeral w mod n*
⟨*proof*⟩

lemma *iszero-numeral*:
iszero (numeral w::'a) ⟷ numeral w mod n = 0
⟨*proof*⟩

lemma *cases*:
assumes *1*: *⋀z. ⟦(x::'a) = of-int z; 0 ≤ z; z < n⟧ ⟹ P*
shows *P*
⟨*proof*⟩

lemma *induct*:
(⋀z. ⟦0 ≤ z; z < n⟧ ⟹ P (of-int z)) ⟹ P (x::'a)
⟨*proof*⟩

lemma *UNIV-eq*: *(UNIV :: 'a set) = Abs ' {0..<n}*
⟨*proof*⟩

lemma *CARD-eq*: *CARD('a) = nat n*
⟨*proof*⟩

lemma *CHAR-eq* [*simp*]: *CHAR('a) = CARD('a)*
⟨*proof*⟩

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 ⟨*proof*⟩

end

interpretation *bit0*:
mod-type int CARD(*'a*::*finite bit0*)
Rep-bit0 :: *'a*::*finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ *'a*::*finite bit0*
⟨*proof*⟩

interpretation *bit1*:
mod-type int CARD(*'a*::*finite bit1*)
Rep-bit1 :: *'a*::*finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a*::*finite bit1*
⟨*proof*⟩

instance *bit0* :: (*finite*) *comm-ring-1*
⟨*proof*⟩

instance *bit1* :: (*finite*) *comm-ring-1*
⟨*proof*⟩

interpretation *bit0*:
mod-ring int CARD('a::finite bit0)
Rep-bit0 :: *'a::finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ *'a::finite bit0*
⟨*proof*⟩

interpretation *bit1*:
mod-ring int CARD('a::finite bit1)
Rep-bit1 :: *'a::finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a::finite bit1*
⟨*proof*⟩

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* ⟷ *Rep-bit0 a* < *Rep-bit0 b*
definition *a* ≤ *b* ⟷ *Rep-bit0 a* ≤ *Rep-bit0 b*
definition *a* < *b* ⟷ *Rep-bit1 a* < *Rep-bit1 b*
definition *a* ≤ *b* ⟷ *Rep-bit1 a* ≤ *Rep-bit1 b*

instance
⟨*proof*⟩
end

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
⟨*proof*⟩

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 ⟨*proof*⟩
end

lemma *equal-num0-code* [*code*]:
  equal-class.equal Num0 Num0 = *True*
⟨*proof*⟩

code-datatype *1* :: *num1*

instantiation *num1* :: *equal* **begin**
definition *equal-num1* :: *num1* ⇒ *num1* ⇒ *bool*
  where *equal-num1* = (=)
instance ⟨*proof*⟩
end

lemma *equal-num1-code* [*code*]:
  equal-class.equal (*1* :: *num1*) *1* = *True*
⟨*proof*⟩

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
  ⟨*proof*⟩
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
  ⟨*proof*⟩
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*))
⟨*proof*⟩

lemma *inj-on-Abs-bit0*:
inj-on (*Abs-bit0* :: *int* ⇒ *'a bit0*) {*0..<2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *inj-on-Abs-bit1*:
inj-on (*Abs-bit1* :: *int* ⇒ *'a bit1*) {*0..<1* + *2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*⟩

end

instantiation *bit0* **and** *bit1* :: (*finite*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a bit0*) *True*
definition *finite-UNIV* = *Phantom*('*a bit1*) *True*
instance ⟨*proof*⟩
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 ⟨*proof*⟩
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*

⟨*ML*⟩

72.8 Examples

lemma *CARD*(*0*) = *0* ⟨*proof*⟩
lemma *CARD*(*17*) = *17* ⟨*proof*⟩
lemma *CHAR*(*23*) = *23* ⟨*proof*⟩
lemma *8* ∗ *11* ^ *3* − *6* = (*2*::*5*) ⟨*proof*⟩

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* ⇒ *'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*] ⇒ *'a word* (**infixr** ‹⌢› *65*)
where *w* ⌢ *x* == λ*n*. *if n* < *length w then w*!*n else x* (*n* − *length w*)

definition
iter :: *'a list* ⇒ *'a word* (‹(‹*notation*=‹*postfix* ω››-^ω)› [*1000*])
where *iter w* == *if w* = [] *then undefined else* (λ*n*. *w*!(*n mod* (*length w*)))

lemma *conc-empty*[*simp*]: [] ⌢ *w* = *w*
⟨*proof*⟩

lemma *conc-fst*[*simp*]: *n* < *length w* ⟹ (*w* ⌢ *x*) *n* = *w*!*n*
⟨*proof*⟩

lemma *conc-snd*[*simp*]: ¬(*n* < *length w*) ⟹ (*w* ⌢ *x*) *n* = *x* (*n* − *length w*)
⟨*proof*⟩

lemma *iter-nth* [*simp*]: *0* < *length w* ⟹ *w*^ω *n* = *w*!(*n mod* (*length w*))
⟨*proof*⟩

lemma *conc-conc*[*simp*]: *u* ⌢ *v* ⌢ *w* = (*u* @ *v*) ⌢ *w* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *range-conc*[*simp*]: *range* (*w*$_1$ ⌢ *w*$_2$) = *set w*$_1$ ∪ *range w*$_2$
⟨*proof*⟩

lemma *iter-unroll*: *0* < *length w* ⟹ *w*^ω = *w* ⌢ *w*^ω
⟨*proof*⟩

73.2 Subsequence, Prefix, and Suffix

definition *suffix* :: [*nat*, *'a word*] ⇒ *'a word*
where *suffix k x* ≡ λ*n*. *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)*
⟨*proof*⟩

lemma *suffix-0* [*simp*]: *suffix 0 x = x*
⟨*proof*⟩

lemma *suffix-suffix* [*simp*]: *suffix m (suffix k x) = suffix (k+m) x*
⟨*proof*⟩

lemma *subsequence-append*: *prefix (i + j) w = prefix i w @ (w [i → i + j])*
⟨*proof*⟩

lemma *subsequence-drop*[*simp*]: *drop i (w [j → k]) = w [j + i → k]*
⟨*proof*⟩

lemma *subsequence-empty*[*simp*]: *w [i → j] = [] ⟷ j ≤ i*
⟨*proof*⟩

lemma *subsequence-length*[*simp*]: *length (subsequence w i j) = j − i*
⟨*proof*⟩

lemma *subsequence-nth*[*simp*]: *k < j − i ⟹ (w [i → j]) ! k = w (i + k)*
⟨*proof*⟩

lemma *subseq-to-zero*[*simp*]: *w[i→0] = []*
⟨*proof*⟩

lemma *subseq-to-smaller*[*simp*]: *i≥j ⟹ w[i→j] = []*
⟨*proof*⟩

lemma *subseq-to-Suc*[*simp*]: *i≤j ⟹ w [i → Suc j] = w [i → j] @ [w j]*
⟨*proof*⟩

lemma *subsequence-singleton*[*simp*]: *w [i → Suc i] = [w i]*
⟨*proof*⟩

lemma *subsequence-prefix-suffix*: *prefix (j − i) (suffix i w) = w [i → j]*
⟨*proof*⟩

lemma *prefix-suffix*: *x = prefix n x ⌢ (suffix n x)*
⟨*proof*⟩

declare *prefix-suffix*[*symmetric*, *simp*]

lemma *word-split*: **obtains** v_1 v_2 **where** $v = v_1 \frown v_2$ *length* $v_1 = k$
⟨*proof*⟩

lemma *set-subsequence*[*simp*]: *set* $(w[i \rightarrow j]) = w`\{i..<j\}$
⟨*proof*⟩

lemma *subsequence-take*[*simp*]: *take* i $(w\ [j \rightarrow k]) = w\ [j \rightarrow min\ (j + i)\ k]$
⟨*proof*⟩

lemma *subsequence-shift*[*simp*]: $(suffix\ i\ w)\ [j \rightarrow k] = w\ [i + j \rightarrow i + k]$
⟨*proof*⟩

lemma *suffix-subseq-join*[*simp*]: $i \leq j \Longrightarrow v\ [i \rightarrow j] \frown suffix\ j\ v = suffix\ i\ v$
⟨*proof*⟩

lemma *prefix-conc-fst*[*simp*]:
assumes $j \leq length\ w$
shows $prefix\ j\ (w \frown w') = take\ j\ w$
⟨*proof*⟩

lemma *prefix-conc-snd*[*simp*]:
assumes $n \geq length\ u$
shows $prefix\ n\ (u \frown v) = u\ @\ prefix\ (n - length\ u)\ v$
⟨*proof*⟩

lemma *prefix-conc-length*[*simp*]: $prefix\ (length\ w)\ (w \frown w') = w$
⟨*proof*⟩

lemma *suffix-conc-fst*[*simp*]:
assumes $n \leq length\ u$
shows $suffix\ n\ (u \frown v) = drop\ n\ u \frown v$
⟨*proof*⟩

lemma *suffix-conc-snd*[*simp*]:
assumes $n \geq length\ u$
shows $suffix\ n\ (u \frown v) = suffix\ (n - length\ u)\ v$
⟨*proof*⟩

lemma *suffix-conc-length*[*simp*]: $suffix\ (length\ w)\ (w \frown w') = w'$
⟨*proof*⟩

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*⟩

lemma *same-concat-eq*[*iff*]: $u \frown v = u \frown w \longleftrightarrow v = w$
⟨*proof*⟩

lemma *comp-concat*[*simp*]: $f \circ u \frown v = map\ f\ u \frown (f \circ v)$
⟨*proof*⟩

73.3 Prepending

primrec *build* :: $'a \Rightarrow 'a\ word \Rightarrow 'a\ word$ (**infixr** ‹##› 65)
where $(a \#\# w)\ 0 = a \mid (a \#\# w)\ (Suc\ i) = w\ i$

lemma *build-eq*[*iff*]: $a_1 \#\# w_1 = a_2 \#\# w_2 \longleftrightarrow a_1 = a_2 \wedge w_1 = w_2$
⟨*proof*⟩

lemma *build-cons*[*simp*]: $(a \# u) \frown v = a \#\# u \frown v$
⟨*proof*⟩

lemma *build-append*[*simp*]: $(w @ a \# u) \frown v = w \frown a \#\# u \frown v$
⟨*proof*⟩

lemma *build-first*[*simp*]: $w\ 0 \#\# suffix\ (Suc\ 0)\ w = w$
⟨*proof*⟩

lemma *build-split*[*intro*]: $w = w\ 0 \#\# suffix\ 1\ w$
⟨*proof*⟩

lemma *build-range*[*simp*]: $range\ (a \#\# w) = insert\ a\ (range\ w)$
⟨*proof*⟩

lemma *suffix-singleton-suffix*[*simp*]: $w\ i \#\# suffix\ (Suc\ i)\ w = suffix\ i\ w$
⟨*proof*⟩

Find the first occurrence of a letter from a given set

lemma *word-first-split-set*:
assumes $A \cap range\ w \neq \{\}$
obtains $u\ a\ v$ **where** $w = u \frown [a] \frown v\ A \cap set\ u = \{\}\ a \in A$
⟨*proof*⟩

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* :: $'a\ word \Rightarrow 'a\ set$
where $limit\ x \equiv \{a\ .\ \exists_\infty n\ .\ x\ n = a\}$

lemma *limit-iff-frequent*: $a \in \mathit{limit}\ x \longleftrightarrow (\exists_\infty n\ .\ x\ n = a)$
⟨*proof*⟩

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*: $(a \in \mathit{limit}\ x) = \mathit{infinite}\ (x -` \{a\})$
⟨*proof*⟩

lemma *two-in-limit-iff*:
$(\{a,\ b\} \subseteq \mathit{limit}\ x) =$
$((\exists n.\ x\ n = a\ ) \wedge (\forall n.\ x\ n = a \longrightarrow (\exists m{>}n.\ x\ m = b)) \wedge (\forall m.\ x\ m = b \longrightarrow (\exists n{>}m.\ x\ n = a)))$
(**is** *?lhs* = (*?r1* ∧ *?r2* ∧ *?r3*))
⟨*proof*⟩

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 $\exists a.\ a \in \mathit{limit}\ x$
⟨*proof*⟩

lemmas *limit-nonemptyE* = *limit-nonempty*[*THEN exE*]

lemma *limit-inter-INF*:
assumes *hyp*: *limit w* ∩ *S* ≠ {}
shows $\exists_\infty n.\ w\ n \in S$
⟨*proof*⟩

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* ∩ *range w*)
shows $\exists a.\ a \in \mathit{limit}\ w \cap S$
⟨*proof*⟩

lemma *fin-ex-inf-eq-limit*: *finite A* ⟹ $(\exists_\infty i.\ w\ i \in A) \longleftrightarrow \mathit{limit}\ w \cap A \neq \{\}$
⟨*proof*⟩

lemma *limit-in-range-suffix*: *limit x* ⊆ *range* (*suffix k x*)
⟨*proof*⟩

lemma *limit-in-range*: *limit r* ⊆ *range r*
⟨*proof*⟩

lemmas *limit-in-range-suffixD* = *limit-in-range-suffix*[*THEN subsetD*]

lemma *limit-subset*: *limit* $f \subseteq f$ ‘ $\{n..\}$
⟨*proof*⟩

theorem *limit-is-suffix*:
assumes *fin*: *finite* (*range* x)
shows $\exists\, k.$ *limit* $x =$ *range* (*suffix* k x)
⟨*proof*⟩

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*⟩

theorem *limit-suffix* [*simp*]: *limit* (*suffix* n x) = *limit* x
⟨*proof*⟩

theorem *limit-iter* [*simp*]:
assumes *nempty*: $0 <$ *length* w
shows *limit* $w^\omega =$ *set* w
⟨*proof*⟩

lemma *limit-o* [*simp*]:
assumes a: $a \in$ *limit* w
shows $f\ a \in$ *limit* $(f \circ w)$
⟨*proof*⟩

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: $x \in$ *limit* $(f \circ w)$
shows $\exists\, a \in (f -‘ \{x\}).\ a \in$ *limit* w
⟨*proof*⟩

theorem *limit-inj* [*simp*]:
assumes *inj*: *inj* f
shows *limit* $(f \circ w) = f$ ‘ (*limit* w)
⟨*proof*⟩

lemma *limit-inter-empty*:
assumes *fin*: *finite* (*range* w)
assumes *hyp*: *limit* $w \cap S = \{\}$
shows $\forall_\infty n.\ w\ n \notin S$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *idx-sequence-inj*:
  assumes *iseq*: *idx-sequence idx*
    and *eq*: *idx m = idx n*
  shows *m = n*
⟨*proof*⟩

lemma *idx-sequence-mono*:
  assumes *iseq*: *idx-sequence idx*
    and *m*: *m ≤ n*
  shows *idx m ≤ idx n*
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*⟩

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*⟩

lemma *idx-sequence-unique-interval*:
assumes *iseq*: *idx-sequence idx*
shows ∃! *k*. *n* ∈ {*idx k* ..< *idx* (*Suc k*) }
⟨*proof*⟩

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* ≡ λ*n*. *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*⟩

lemma *merge0*:
assumes *idx*: *idx-sequence idx*
shows *merge ws idx 0* = *ws 0 0*
⟨*proof*⟩

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*⟩

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* ⇒ *True* | *Some x* ⇒ (*case y of None* ⇒ *False* | *Some y* ⇒ $x \leq y$))

definition *less-option* **where**
$x < y \longleftrightarrow$ (*case y of None* ⇒ *False* | *Some y* ⇒ (*case x of None* ⇒ *True* | *Some x* ⇒ $x < y$))

lemma *less-eq-option-None* [*simp*]: $None \leq x$
⟨*proof*⟩

lemma *less-eq-option-None-code* [*code*]: $None \leq x \longleftrightarrow True$
⟨*proof*⟩

lemma *less-eq-option-None-is-None*: $x \leq None \Longrightarrow x = None$
⟨*proof*⟩

lemma *less-eq-option-Some-None* [*simp*, *code*]: $Some\ x \leq None \longleftrightarrow False$
⟨*proof*⟩

lemma *less-eq-option-Some* [*simp*, *code*]: $Some\ x \leq Some\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *less-option-None* [*simp*, *code*]: $x < None \longleftrightarrow False$
⟨*proof*⟩

lemma *less-option-None-is-Some*: $None < x \Longrightarrow \exists\, z.\ x = Some\ z$
⟨*proof*⟩

lemma *less-option-None-Some* [*simp*]: $None < Some\ x$
⟨*proof*⟩

lemma *less-option-None-Some-code* [*code*]: $None < Some\ x \longleftrightarrow True$
⟨*proof*⟩

lemma *less-option-Some* [*simp*, *code*]: $Some\ x < Some\ y \longleftrightarrow x < y$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *option* :: (*order*) *order*
⟨*proof*⟩

instance *option* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *option* :: (*order*) *order-bot*
begin

definition *bot-option* **where** $\bot = None$

instance
⟨*proof*⟩

end

instantiation *option* :: (*order-top*) *order-top*
begin

definition *top-option* **where** $\top = \mathit{Some}\ \top$

instance
⟨*proof*⟩

end

instance *option* :: (*wellorder*) *wellorder*
⟨*proof*⟩

instantiation *option* :: (*inf*) *inf*
begin

definition *inf-option* **where**
$x \sqcap y = (\mathit{case}\ x\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ x \Rightarrow (\mathit{case}\ y\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ y \Rightarrow \mathit{Some}\ (x \sqcap y)))$

lemma *inf-None-1* [*simp*, *code*]: $\mathit{None} \sqcap y = \mathit{None}$
⟨*proof*⟩

lemma *inf-None-2* [*simp*, *code*]: $x \sqcap \mathit{None} = \mathit{None}$
⟨*proof*⟩

lemma *inf-Some* [*simp*, *code*]: $\mathit{Some}\ x \sqcap \mathit{Some}\ y = \mathit{Some}\ (x \sqcap y)$
⟨*proof*⟩

instance ⟨*proof*⟩

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*]: $\mathit{None} \sqcup y = y$
⟨*proof*⟩

lemma *sup-None-2* [*simp*, *code*]: $x \sqcup \mathit{None} = x$
⟨*proof*⟩

lemma *sup-Some* [*simp*, *code*]: $\mathit{Some}\ x \sqcup \mathit{Some}\ y = \mathit{Some}\ (x \sqcup y)$

⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *option* :: (*semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

instance *option* :: (*semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *option* :: (*lattice*) *lattice* ⟨*proof*⟩

instance *option* :: (*lattice*) *bounded-lattice-bot* ⟨*proof*⟩

instance *option* :: (*bounded-lattice-top*) *bounded-lattice-top* ⟨*proof*⟩

instance *option* :: (*bounded-lattice-top*) *bounded-lattice* ⟨*proof*⟩

instance *option* :: (*distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

instantiation *option* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-option* :: *'a option set* ⇒ *'a option* **where**
⨅ *A* = (*if None* ∈ *A then None else Some* (⨅ *Option.these A*))

lemma *None-in-Inf* [*simp*]: *None* ∈ *A* ⟹ ⨅ *A* = *None*
⟨*proof*⟩

definition *Sup-option* :: *'a option set* ⇒ *'a option* **where**
⨆ *A* = (*if A* = {} ∨ *A* = {*None*} *then None else Some* (⨆ *Option.these A*))

lemma *empty-Sup* [*simp*]: ⨆ {} = *None*
⟨*proof*⟩

lemma *singleton-None-Sup* [*simp*]: ⨆ {*None*} = *None*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *Some-Inf*:
Some (⨅ *A*) = ⨅ (*Some* ‘ *A*)
⟨*proof*⟩

lemma *Some-Sup*:
$A \neq \{\} \Longrightarrow Some\ (\bigsqcup A) = \bigsqcup (Some\ `\ A)$
⟨*proof*⟩

lemma *Some-INF*:
$Some\ (\bigsqcap x{\in}A.\ f\ x) = (\bigsqcap x{\in}A.\ Some\ (f\ x))$
⟨*proof*⟩

lemma *Some-SUP*:
$A \neq \{\} \Longrightarrow Some\ (\bigsqcup x{\in}A.\ f\ x) = (\bigsqcup x{\in}A.\ Some\ (f\ x))$
⟨*proof*⟩

lemma *option-Inf-Sup*: $\bigsqcap (Sup\ `\ A) \leq \bigsqcup (Inf\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y\ \in\ Y\})$
for A :: ('a::*complete-distrib-lattice option*) *set set*
⟨*proof*⟩

instance *option* :: (*complete-distrib-lattice*) *complete-distrib-lattice*
⟨*proof*⟩

instance *option* :: (*complete-linorder*) *complete-linorder* ⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

definition *exists* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where**
exists = *list-ex*

lemma *exists-ex* [*simp*]:
exists P xs ⟷ (∃ *x*∈*set xs*. *P x*)
⟨*proof*⟩

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
  $as :: 'a \Rightarrow 'a \Rightarrow 'a$
  $fake\text{-}quant :: ('a \Rightarrow prop) \Rightarrow prop$

lemma *let-cong-unfolding*: $M = N \Longrightarrow f\ N = g\ N \Longrightarrow Let\ M\ f = Let\ N\ g$
⟨*proof*⟩

translations $P <= CONST\ fake\text{-}quant\ (\lambda x.\ P)$

⟨*ML*⟩

bundle *pattern-aliases* **begin**

  notation *as* (**infixr** ‹=:› *1*)

⟨*ML*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

⟨*ML*⟩

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-1*}) ⇒ '*b* **and** *g gm* :: '*a* ⇒ '*a* ⇒ '*a* **and** *g1 gn1* :: '*a* ⇒ '*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*: $f\ (g\ x\ (of\text{-}nat\ n)) = f\ x$
⟨*proof*⟩

lemma *minus-of-nat*: $f\ (gm\ x\ (of\text{-}nat\ n)) = f\ x$
⟨*proof*⟩

lemma *plus-of-int*: $f\ (g\ x\ (of\text{-}int\ n)) = f\ x$
⟨*proof*⟩

lemma *minus-of-int*: $f\ (gm\ x\ (of\text{-}int\ n)) = f\ x$
⟨*proof*⟩

lemma *plus-numeral*: $f\ (g\ x\ (numeral\ n)) = f\ x$
⟨*proof*⟩

lemma *minus-numeral*: $f\ (gm\ x\ (numeral\ n)) = f\ x$
⟨*proof*⟩

lemma *minus-1*: $f\ (gn1\ x) = f\ x$
⟨*proof*⟩

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* $f\ \lambda z\ x.\ z + x * period\ \lambda z\ x.\ z - x * period$
$\lambda z.\ z + period\ \lambda z.\ z - period$
⟨*proof*⟩
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* $\lambda z\ x.\ z + x\ \lambda z\ x.\ z - x\ \lambda z.\ z + 1\ \lambda z.\ z - 1$
⟨*proof*⟩

lemma *of-nat*: $f\ (of\text{-}nat\ n) = f\ 0$ ⟨*proof*⟩
lemma *uminus-of-nat*: $f\ (-of\text{-}nat\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-int*: $f\ (of\text{-}int\ n) = f\ 0$ ⟨*proof*⟩
lemma *uminus-of-int*: $f\ (-of\text{-}int\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-numeral*: $f\ (numeral\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-neg-numeral*: $f\ (-numeral\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-1*: $f\ 1 = f\ 0$ ⟨*proof*⟩
lemma *of-neg-1*: $f\ (-1) = f\ 0$ ⟨*proof*⟩

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*: $sin\ ((z :: 'a :: \{real\text{-}normed\text{-}field, banach\}) + of\text{-}real\ pi) = -\ sin\ z$
⟨*proof*⟩

lemma *cos-plus-pi*: $cos\ ((z :: 'a :: \{real\text{-}normed\text{-}field, banach\}) + of\text{-}real\ pi) = -\ cos\ z$
⟨*proof*⟩

interpretation *sin*: *periodic-fun-simple sin* $2 * of\text{-}real\ pi :: 'a :: \{real\text{-}normed\text{-}field, banach\}$
⟨*proof*⟩

interpretation *cos*: *periodic-fun-simple cos* $2 * of\text{-}real\ pi :: 'a :: \{real\text{-}normed\text{-}field, banach\}$
⟨*proof*⟩

interpretation *tan*: *periodic-fun-simple tan* $2 * of\text{-}real\ pi :: 'a :: \{real\text{-}normed\text{-}field, banach\}$
⟨*proof*⟩

interpretation *cot*: *periodic-fun-simple cot* $2 * of\text{-}real\ pi :: 'a :: \{real\text{-}normed\text{-}field, banach\}$
⟨*proof*⟩

lemma *cos-eq-neg-periodic-intro*:
assumes $x - y = 2*(of\text{-}int\ k)*pi + pi \vee x + y = 2*(of\text{-}int\ k)*pi + pi$
shows $cos\ x = -\ cos\ y$ ⟨*proof*⟩

lemma *cos-eq-periodic-intro*:
assumes $x - y = 2*(of\text{-}int\ k)*pi \vee x + y = 2*(of\text{-}int\ k)*pi$
shows $cos\ x = cos\ y$
⟨*proof*⟩

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*⟩

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*}
⟨*proof*⟩

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*}
⟨*proof*⟩

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*⟩

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*⟩

lemma *finite-mult-not-eq-zero-sumI*:
fixes *f g* :: *'a*::*monoid-add* ⇒ *'b*::*semiring-0*
assumes *finite* {*a*. *f a* ≠ *0*}

assumes *finite* $\{b.\ g\ b \neq 0\}$
shows *finite* $\{a + b \mid a\ b.\ f\ a * g\ b \neq 0\}$
⟨*proof*⟩

lemma *finite-Sum-any-not-eq-zero-weakenI*:
assumes *finite* $\{a.\ \exists b.\ f\ a\ b \neq 0\}$
shows *finite* $\{a.\ Sum\text{-}any\ (f\ a) \neq 0\}$
⟨*proof*⟩

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*]:
$P \Longrightarrow (a\ when\ P) = a$
$\neg\ P \Longrightarrow (a\ when\ P) = 0$
⟨*proof*⟩

lemma *when-simps* [*simp*]:
$(a\ when\ True) = a$
$(a\ when\ False) = 0$
⟨*proof*⟩

lemma *when-cong*:
assumes $P \longleftrightarrow Q$
and $Q \Longrightarrow a = b$
shows $(a\ when\ P) = (b\ when\ Q)$
⟨*proof*⟩

lemma *zero-when* [*simp*]:
$(0\ when\ P) = 0$
⟨*proof*⟩

lemma *when-when*:
$(a\ when\ P\ when\ Q) = (a\ when\ P \wedge Q)$
⟨*proof*⟩

lemma *when-commute*:
$(a\ when\ Q\ when\ P) = (a\ when\ P\ when\ Q)$
⟨*proof*⟩

lemma *when-neq-zero* [*simp*]:
$(a\ when\ P) \neq 0 \longleftrightarrow P \wedge a \neq 0$

⟨*proof*⟩

end

context *monoid-add*
begin

lemma *when-add-distrib*:
(*a* + *b when P*) = (*a when P*) + (*b when P*)
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *zero-power-eq*:
0 ^ *n* = (*1 when n* = *0*)
⟨*proof*⟩

end

context *comm-monoid-add*
begin

lemma *Sum-any-when-equal* [*simp*]:
(∑ *a*. (*f a when a* = *b*)) = *f b*
⟨*proof*⟩

lemma *Sum-any-when-equal'* [*simp*]:
(∑ *a*. (*f a when b* = *a*)) = *f b*
⟨*proof*⟩

lemma *Sum-any-when-independent*:
(∑ *a*. *g a when P*) = ((∑ *a*. *g a*) *when P*)
⟨*proof*⟩

lemma *Sum-any-when-dependent-prod-right*:
(∑ (*a*, *b*). *g a when b* = *h a*) = (∑ *a*. *g a*)
⟨*proof*⟩

lemma *Sum-any-when-dependent-prod-left*:
(∑ (*a*, *b*). *g b when a* = *h b*) = (∑ *b*. *g b*)
⟨*proof*⟩

end

context *cancel-comm-monoid-add*
begin

lemma *when-diff-distrib*:
$(a - b \ when \ P) = (a \ when \ P) - (b \ when \ P)$
⟨*proof*⟩

end

context *group-add*
begin

lemma *when-uminus-distrib*:
$(- a \ when \ P) = - (a \ when \ P)$
⟨*proof*⟩

end

context *mult-zero*
begin

lemma *mult-when*:
$a * (b \ when \ P) = (a * b \ when \ P)$
⟨*proof*⟩

lemma *when-mult*:
$(a \ when \ P) * b = (a * b \ when \ P)$
⟨*proof*⟩

end

79.2 Type definition

The following type is of central importance:

typedef (**overloaded**) $('a, 'b)$ *poly-mapping* (‹(- ⇒$_0$ /-)› [*1*, *0*] *0*) =
$\{f :: 'a \Rightarrow 'b::zero.$ *finite* $\{x.\ f\ x \neq 0\}\}$
morphisms *lookup Abs-poly-mapping*
⟨*proof*⟩

declare *lookup-inverse* [*simp*]
declare *lookup-inject* [*simp*]

lemma *lookup-Abs-poly-mapping* [*simp*]:
finite $\{x.\ f\ x \neq 0\} \Longrightarrow$ *lookup* (*Abs-poly-mapping* f) $= f$
⟨*proof*⟩

lemma *finite-lookup* [*simp*]:
finite $\{k.$ *lookup* $f\ k \neq 0\}$
⟨*proof*⟩

lemma *finite-lookup-nat* [*simp*]:

fixes $f :: 'a \Rightarrow_0 nat$
shows *finite* $\{k.\ 0 < lookup\ f\ k\}$
⟨*proof*⟩

lemma *poly-mapping-eqI*:
assumes $\bigwedge k.\ lookup\ f\ k = lookup\ g\ k$
shows $f = g$
⟨*proof*⟩

lemma *poly-mapping-eq-iff*: $a = b \longleftrightarrow lookup\ a = lookup\ b$
⟨*proof*⟩

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 $\lambda k.\ 0$
  ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *Abs-poly-mapping* [*simp*]: *Abs-poly-mapping* $(\lambda k.\ 0) = 0$
  ⟨*proof*⟩

lemma *lookup-zero* [*simp*]: *lookup 0 k* = *0*
  ⟨*proof*⟩

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*⟩

instance
  ⟨*proof*⟩

end

lemma *lookup-add*: *lookup* $(f + g)\ k$ = *lookup f k* + *lookup g k*
  ⟨*proof*⟩

instance *poly-mapping* :: (*type*, *comm-monoid-add*) *comm-monoid-add*

⟨*proof*⟩

lemma *lookup-sum*: *lookup* (*sum pp X*) *i* = *sum* (λ*x*. *lookup* (*pp x*) *i*) *X*
⟨*proof*⟩

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*⟩

instance
⟨*proof*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *lookup-uminus* [*simp*]:
lookup (− *f*) *k* = − *lookup f k*
⟨*proof*⟩

lemma *lookup-minus*:
lookup (*f* − *g*) *k* = *lookup f k* − *lookup g k*
⟨*proof*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *lookup-one*: *lookup 1 k* = (*1 when k* = *0*)
⟨*proof*⟩

lemma *lookup-one-zero* [*simp*]:
lookup 1 0 = *1*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *lookup-mult*:
lookup (*f* ∗ *g*) *k* = (∑ *l*. *lookup f l* ∗ (∑ *q*. *lookup g q when k* = *l* + *q*))
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0*) *comm-semiring-0*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-0-cancel*) *semiring-0-cancel*
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-1*) *semiring-1*
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-1*) *comm-semiring-1*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-1-cancel*) *semiring-1-cancel*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *ring*) *ring*
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring*) *comm-ring*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *ring-1*) *ring-1*
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring-1*) *comm-ring-1*
⟨*proof*⟩

79.5 Single-point mappings

lift-definition *single* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_0 'b$::*zero*
is $\lambda k\ v\ k'.\ (v\ when\ k = k')$
⟨*proof*⟩

lemma *inj-single* [*iff*]:
inj (*single k*)
⟨*proof*⟩

lemma *lookup-single*:
lookup (*single k v*) $k' = (v\ when\ k = k')$
⟨*proof*⟩

lemma *lookup-single-eq* [*simp*]:
lookup (*single k v*) *k* = *v*
⟨*proof*⟩

lemma *lookup-single-not-eq*:
$k \neq k' \Longrightarrow$ *lookup* (*single k v*) $k' = 0$
⟨*proof*⟩

lemma *single-zero* [*simp*]:
single k 0 = 0
⟨*proof*⟩

lemma *single-one* [*simp*]:
single 0 1 = 1
⟨*proof*⟩

lemma *single-add*:
single k (a + b) = single k a + single k b
⟨*proof*⟩

lemma *single-uminus*:
single k (− a) = − single k a
⟨*proof*⟩

lemma *single-diff*:
single k (a − b) = single k a − single k b
⟨*proof*⟩

lemma *single-numeral* [*simp*]:
single 0 (numeral n) = numeral n
⟨*proof*⟩

lemma *lookup-numeral*:
lookup (numeral n) k = (numeral n when k = 0)
⟨*proof*⟩

lemma *single-of-nat* [*simp*]:
single 0 (of-nat n) = of-nat n
⟨*proof*⟩

lemma *lookup-of-nat*:
lookup (of-nat n) k = (of-nat n when k = 0)
⟨*proof*⟩

lemma *of-nat-single*:
of-nat = single 0 ∘ of-nat
⟨*proof*⟩

lemma *mult-single*:
single k a ∗ single l b = single (k + l) (a ∗ b)
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-char-0*) *semiring-char-0*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *ring-char-0*) *ring-char-0*
⟨*proof*⟩

lemma *single-of-int* [*simp*]:
single 0 (*of-int k*) = *of-int k*
⟨*proof*⟩

lemma *lookup-of-int*:
lookup (*of-int l*) *k* = (*of-int l when k* = *0*)
⟨*proof*⟩

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*⟩

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *ring-no-zero-divisors*) *ring-no-zero-divisors*
⟨*proof*⟩

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *ring-1-no-zero-divisors*) *ring-1-no-zero-divisors*
⟨*proof*⟩

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *idom*) *idom*
⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *less-eq-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) ⇒ *bool*
is λ*f g*. *less-fun f g* ∨ *f* = *g*
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *linorder*}) *ordered-ab-semigroup-add*
⟨*proof*⟩

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *linordered-cancel-ab-semigroup-add*
⟨*proof*⟩

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *ordered-comm-monoid-add*
⟨*proof*⟩

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *ordered-cancel-comm-monoid-add*
⟨*proof*⟩

instance *poly-mapping* :: (*linorder*, *linordered-ab-group-add*) *linordered-ab-group-add*
⟨*proof*⟩

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*} ⟨*proof*⟩

lift-definition *range* :: ('*a* ⇒$_0$ '*b*::*zero*) ⇒ '*b set*
is λ*f* :: '*a* ⇒ '*b*. *Set.range f* − {*0*} ⟨*proof*⟩

lemma *finite-keys* [*simp*]:
finite (*keys f*)
⟨*proof*⟩

lemma *not-in-keys-iff-lookup-eq-zero*:
k ∉ *keys f* ⟷ *lookup f k* = *0*
⟨*proof*⟩

lemma *lookup-not-eq-zero-eq-in-keys*:
lookup f k ≠ *0* ⟷ *k* ∈ *keys f*
⟨*proof*⟩

lemma *lookup-eq-zero-in-keys-contradict* [*dest*]:
lookup f k = *0* ⟹ ¬ *k* ∈ *keys f*
⟨*proof*⟩

lemma *finite-range* [*simp*]: *finite* (*Poly-Mapping.range p*)
⟨*proof*⟩

lemma *in-keys-lookup-in-range* [*simp*]:
k ∈ *keys f* ⟹ *lookup f k* ∈ *range f*
⟨*proof*⟩

lemma *in-keys-iff*: $x \in (keys\ s) = (lookup\ s\ x \neq 0)$
⟨*proof*⟩

lemma *keys-zero* [*simp*]:
$keys\ 0 = \{\}$
⟨*proof*⟩

lemma *range-zero* [*simp*]:
$range\ 0 = \{\}$
⟨*proof*⟩

lemma *keys-add*:
$keys\ (f + g) \subseteq keys\ f \cup keys\ g$
⟨*proof*⟩

lemma *keys-one* [*simp*]:
$keys\ 1 = \{0\}$
⟨*proof*⟩

lemma *range-one* [*simp*]:
$range\ 1 = \{1\}$
⟨*proof*⟩

lemma *keys-single* [*simp*]:
$keys\ (single\ k\ v) = (if\ v = 0\ then\ \{\}\ else\ \{k\})$
⟨*proof*⟩

lemma *range-single* [*simp*]:
$range\ (single\ k\ v) = (if\ v = 0\ then\ \{\}\ else\ \{v\})$
⟨*proof*⟩

lemma *keys-mult*:
$keys\ (f * g) \subseteq \{a + b \mid a\ b.\ a \in keys\ f \wedge b \in keys\ g\}$
⟨*proof*⟩

lemma *setsum-keys-plus-distrib*:
assumes *hom-0*: $\bigwedge k.\ f\ k\ 0 = 0$
and *hom-plus*: $\bigwedge k.\ k \in Poly\text{-}Mapping.keys\ p \cup Poly\text{-}Mapping.keys\ q \Longrightarrow f\ k\ (Poly\text{-}Mapping.lookup\ p\ k + Poly\text{-}Mapping.lookup\ q\ k) = f\ k\ (Poly\text{-}Mapping.lookup\ p\ k) + f\ k\ (Poly\text{-}Mapping.lookup\ q\ k)$
shows
$(\sum k \in Poly\text{-}Mapping.keys\ (p + q).\ f\ k\ (Poly\text{-}Mapping.lookup\ (p + q)\ k)) =$
$(\sum k \in Poly\text{-}Mapping.keys\ p.\ f\ k\ (Poly\text{-}Mapping.lookup\ p\ k)) +$
$(\sum k \in Poly\text{-}Mapping.keys\ q.\ f\ k\ (Poly\text{-}Mapping.lookup\ q\ k))$
(**is** *?lhs* = *?p* + *?q*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *degree-one* [*simp*]:
degree 1 = *1*
⟨*proof*⟩

lemma *degree-single-zero* [*simp*]:
degree (*single k 0*) = *0*
⟨*proof*⟩

lemma *degree-single-not-zero* [*simp*]:
$v \neq 0 \Longrightarrow$ *degree* (*single k v*) = *Suc k*
⟨*proof*⟩

lemma *degree-zero-iff* [*simp*]:
degree f = *0* $\longleftrightarrow$ *f* = *0*
⟨*proof*⟩

lemma *degree-greater-zero-in-keys*:
assumes *0* < *degree f*
shows *degree f* − *1* ∈ *keys f*
⟨*proof*⟩

lemma *in-keys-less-degree*:
n ∈ *keys f* $\Longrightarrow$ *n* < *degree f*
⟨*proof*⟩

lemma *beyond-degree-lookup-zero*:
degree f ≤ *n* $\Longrightarrow$ *lookup f n* = *0*
⟨*proof*⟩

lemma *degree-add*:
degree (*f* + *g*) ≤ *max* (*degree f*) (*Poly-Mapping.degree g*)
⟨*proof*⟩

lemma *sorted-list-of-set-keys*:
sorted-list-of-set (*keys f*) = *filter* (λ*k*. *k* ∈ *keys f*) [*0*..<*degree f*] (**is** - = *?r*)
⟨*proof*⟩

79.10 Inductive structure

lift-definition *update* :: $'a \Rightarrow 'b \Rightarrow ('a \Rightarrow_0 'b::zero) \Rightarrow 'a \Rightarrow_0 'b$

is $\lambda k\ v\ f.\ f(k := v)$
⟨*proof*⟩

lemma *update-induct* [*case-names const update*]:
assumes *const'*: $P\ 0$
assumes *update'*: $\bigwedge f\ a\ b.\ a \notin keys\ f \Longrightarrow b \neq 0 \Longrightarrow P\ f \Longrightarrow P\ (update\ a\ b\ f)$
shows $P\ f$
⟨*proof*⟩

lemma *lookup-update*:
$lookup\ (update\ k\ v\ f)\ k' = (if\ k = k'\ then\ v\ else\ lookup\ f\ k')$
⟨*proof*⟩

lemma *keys-update*:
$keys\ (update\ k\ v\ f) = (if\ v = 0\ then\ keys\ f - \{k\}\ else\ insert\ k\ (keys\ f))$
⟨*proof*⟩

79.11 Quasi-functorial structure

lift-definition *map* :: $('b::zero \Rightarrow 'c::zero) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c::zero)$
is $\lambda g\ f\ k.\ g\ (f\ k)\ when\ f\ k \neq 0$
⟨*proof*⟩

context
fixes $f :: 'b \Rightarrow 'a$
assumes *inj-f*: *inj f*
begin

lift-definition *map-key* :: $('a \Rightarrow_0 'c::zero) \Rightarrow 'b \Rightarrow_0 'c$
is $\lambda p.\ p \circ f$
⟨*proof*⟩

end

lemma *map-key-compose*:
assumes [*transfer-rule*]: *inj f inj g*
shows *map-key f* (*map-key g p*) = *map-key* $(g \circ f)$ *p*
⟨*proof*⟩

lemma *map-key-id*:
map-key $(\lambda x.\ x)$ *p* = *p*
⟨*proof*⟩

context
fixes $f :: 'a \Rightarrow 'b$
assumes *inj-f* [*transfer-rule*]: *inj f*
begin

lemma *map-key-map*:
map-key f (map g p) = map g (map-key f p)
⟨*proof*⟩

lemma *map-key-plus*:
map-key f (p + q) = map-key f p + map-key f q
⟨*proof*⟩

lemma *keys-map-key*:
keys (map-key f p) = f −‘ keys p
⟨*proof*⟩

lemma *map-key-zero* [*simp*]:
map-key f 0 = 0
⟨*proof*⟩

lemma *map-key-single* [*simp*]:
map-key f (single (f k) v) = single k v
⟨*proof*⟩

end

lemma *mult-map-scale-conv-mult*: *map ((∗) s) p = single 0 s ∗ p*
⟨*proof*⟩

lemma *map-single* [*simp*]:
(c = 0 ⟹ f 0 = 0) ⟹ map f (single x c) = single x (f c)
⟨*proof*⟩

lemma *map-eq-zero-iff*: *map f p = 0 ⟷ (∀ k ∈ keys p. f (lookup p k) = 0)*
⟨*proof*⟩

79.12 Canonical dense representation of *nat* ⇒$_0$ *'a*

abbreviation *no-trailing-zeros* :: *'a* :: *zero list* ⇒ *bool*
where
no-trailing-zeros ≡ *no-trailing ((=) 0)*

lift-definition *nth* :: *'a list* ⇒ (*nat* ⇒$_0$ *'a*::*zero*)
is *nth-default 0*
⟨*proof*⟩

The opposite direction is directly specified on (later) type *nat-mapping*.

lemma *nth-Nil* [*simp*]:
nth [] = 0
⟨*proof*⟩

lemma *nth-singleton* [*simp*]:
nth [v] = single 0 v
⟨*proof*⟩

lemma *nth-replicate* [*simp*]:
nth (*replicate n 0* @ [*v*]) = *single n v*
⟨*proof*⟩

lemma *nth-strip-while* [*simp*]:
nth (*strip-while* ((=) *0*) *xs*) = *nth xs*
⟨*proof*⟩

lemma *nth-strip-while'* [*simp*]:
nth (*strip-while* (λ*k*. *k* = *0*) *xs*) = *nth xs*
⟨*proof*⟩

lemma *nth-eq-iff*:
nth xs = *nth ys* ⟷ *strip-while* (*HOL.eq 0*) *xs* = *strip-while* (*HOL.eq 0*) *ys*
⟨*proof*⟩

lemma *lookup-nth* [*simp*]:
lookup (*nth xs*) = *nth-default 0 xs*
⟨*proof*⟩

lemma *keys-nth* [*simp*]:
keys (*nth xs*) = *fst* ‘ {(*n*, *v*) ∈ *set* (*indexed-from 0 xs*). *v* ≠ *0*}
⟨*proof*⟩

lemma *range-nth* [*simp*]:
range (*nth xs*) = *set xs* − {*0*}
⟨*proof*⟩

lemma *degree-nth*:
no-trailing-zeros xs ⟹ *degree* (*nth xs*) = *length xs*
⟨*proof*⟩

lemma *nth-trailing-zeros* [*simp*]:
nth (*xs* @ *replicate n 0*) = *nth xs*
⟨*proof*⟩

lemma *nth-idem*:
nth (*List.map* (*lookup f*) [*0*..<*degree f*]) = *f*
⟨*proof*⟩

lemma *nth-idem-bound*:
assumes *degree f* ≤ *n*
shows *nth* (*List.map* (*lookup f*) [*0*..<*n*]) = *f*
⟨*proof*⟩

79.13 Canonical sparse representation of *'a* ⇒$_0$ *'b*

lift-definition *the-value* :: (*'a* × *'b*) *list* ⇒ *'a* ⇒$_0$ *'b*::*zero*

is $\lambda xs\ k.\ case\ map\text{-}of\ xs\ k\ of\ None \Rightarrow 0 \mid Some\ v \Rightarrow v$
⟨*proof*⟩

definition $items :: ('a::linorder \Rightarrow_0 'b::zero) \Rightarrow ('a \times 'b)\ list$
where
$items\ f = List.map\ (\lambda k.\ (k,\ lookup\ f\ k))\ (sorted\text{-}list\text{-}of\text{-}set\ (keys\ f))$

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*]:
$the\text{-}value\ (items\ f) = f$
⟨*proof*⟩

lemma *lookup-the-value*:
$lookup\ (the\text{-}value\ xs)\ k = (case\ map\text{-}of\ xs\ k\ of\ None \Rightarrow 0 \mid Some\ v \Rightarrow v)$
⟨*proof*⟩

lemma *items-the-value*:
assumes $sorted\ (List.map\ fst\ xs)$ **and** $distinct\ (List.map\ fst\ xs)$ **and** $0 \notin snd\ `\ set\ xs$
shows $items\ (the\text{-}value\ xs) = xs$
⟨*proof*⟩

lemma *the-value-Nil* [*simp*]:
$the\text{-}value\ [] = 0$
⟨*proof*⟩

lemma *the-value-Cons* [*simp*]:
$the\text{-}value\ (x \#\ xs) = update\ (fst\ x)\ (snd\ x)\ (the\text{-}value\ xs)$
⟨*proof*⟩

lemma *items-zero* [*simp*]:
$items\ 0 = []$
⟨*proof*⟩

lemma *items-one* [*simp*]:
$items\ 1 = [(0,\ 1)]$
⟨*proof*⟩

lemma *items-single* [*simp*]:
$items\ (single\ k\ v) = (if\ v = 0\ then\ []\ else\ [(k,\ v)])$
⟨*proof*⟩

lemma *in-set-items-iff* [*simp*]:
$(k,\ v) \in set\ (items\ f) \longleftrightarrow k \in keys\ f \wedge lookup\ f\ k = v$
⟨*proof*⟩

79.14 Size estimation

context
fixes $f :: 'a \Rightarrow nat$
and $g :: 'b :: zero \Rightarrow nat$
begin

definition *poly-mapping-size* :: $('a \Rightarrow_0 'b) \Rightarrow nat$
where
$poly\text{-}mapping\text{-}size\ m = g\ 0 + (\sum k \in keys\ m.\ Suc\ (f\ k + g\ (lookup\ m\ k)))$

lemma *poly-mapping-size-0* [*simp*]:
$poly\text{-}mapping\text{-}size\ 0 = g\ 0$
⟨*proof*⟩

lemma *poly-mapping-size-single* [*simp*]:
$poly\text{-}mapping\text{-}size\ (single\ k\ v) = (if\ v = 0\ then\ g\ 0\ else\ g\ 0 + f\ k + g\ v + 1)$
⟨*proof*⟩

lemma *keys-less-poly-mapping-size*:
$k \in keys\ m \Longrightarrow f\ k + g\ (lookup\ m\ k) < poly\text{-}mapping\text{-}size\ m$
⟨*proof*⟩

lemma *lookup-le-poly-mapping-size*:
$g\ (lookup\ m\ k) \leq poly\text{-}mapping\text{-}size\ m$
⟨*proof*⟩

lemma *poly-mapping-size-estimation*:
$k \in keys\ m \Longrightarrow y \leq f\ k + g\ (lookup\ m\ k) \Longrightarrow y < poly\text{-}mapping\text{-}size\ m$
⟨*proof*⟩

lemma *poly-mapping-size-estimation2*:
assumes $v \in range\ m$ **and** $y \leq g\ v$
shows $y < poly\text{-}mapping\text{-}size\ m$
⟨*proof*⟩

end

lemma *poly-mapping-size-one* [*simp*]:
$poly\text{-}mapping\text{-}size\ f\ g\ 1 = g\ 0 + f\ 0 + g\ 1 + 1$
⟨*proof*⟩

lemma *poly-mapping-size-cong* [*fundef-cong*]:
$m = m' \Longrightarrow g\ 0 = g'\ 0 \Longrightarrow (\bigwedge k.\ k \in keys\ m' \Longrightarrow f\ k = f'\ k)$
$\Longrightarrow (\bigwedge v.\ v \in range\ m' \Longrightarrow g\ v = g'\ v)$
$\Longrightarrow poly\text{-}mapping\text{-}size\ f\ g\ m = poly\text{-}mapping\text{-}size\ f'\ g'\ m'$
⟨*proof*⟩

instantiation *poly-mapping* :: (*type*, *zero*) *size*
begin

definition *size* = *poly-mapping-size* (λ-. *0*) (λ-. *0*)

instance ⟨*proof*⟩

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*)
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *lookup-mapp*:
lookup (*mapp f p*) *k* = (*f k* (*lookup p k*) *when k* ∈ *keys p*)
⟨*proof*⟩

lemma *keys-mapp-subset*: *keys* (*mapp f p*) ⊆ *keys p*
⟨*proof*⟩

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
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *frag-of-nonzero* [*simp*]: *frag-of a* $\neq$ *0*
⟨*proof*⟩

definition *frag-cmul* :: *int* $\Rightarrow$ (*'a* $\Rightarrow_0$ *int*) $\Rightarrow$ (*'a* $\Rightarrow_0$ *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*
⟨*proof*⟩

lemma *frag-cmul-zero2* [*simp*]: *frag-cmul c 0* = *0*
⟨*proof*⟩

lemma *frag-cmul-one* [*simp*]: *frag-cmul 1 x* = *x*
⟨*proof*⟩

lemma *frag-cmul-minus-one* [*simp*]: *frag-cmul* (*−1*) *x* = *−x*
⟨*proof*⟩

lemma *frag-cmul-cmul* [*simp*]: *frag-cmul c* (*frag-cmul d x*) = *frag-cmul* (*c*$*$*d*) *x*
⟨*proof*⟩

lemma *lookup-frag-cmul* [*simp*]: *poly-mapping.lookup* (*frag-cmul c x*) *i* = *c* $*$ *poly-mapping.lookup x i*
⟨*proof*⟩

lemma *minus-frag-cmul* [*simp*]: *−* *frag-cmul k x* = *frag-cmul* (*−k*) *x*
⟨*proof*⟩

lemma *keys-frag-of*: *Poly-Mapping.keys*(*frag-of a*) = {*a*}
⟨*proof*⟩

lemma *finite-cmul-nonzero*: *finite* {*x*. *c* $*$ *Poly-Mapping.lookup a x* $\neq$ (*0*::*int*)}
⟨*proof*⟩

lemma *keys-cmul*: *Poly-Mapping.keys*(*frag-cmul c a*) $\subseteq$ *Poly-Mapping.keys a*
⟨*proof*⟩

lemma *keys-cmul-iff* [*iff*]: *i* $\in$ *Poly-Mapping.keys* (*frag-cmul c x*) $\longleftrightarrow$ *i* $\in$ *Poly-Mapping.keys x* $\wedge$ *c* $\neq$ *0*
⟨*proof*⟩

lemma *keys-minus* [*simp*]: *Poly-Mapping.keys*(*−a*) = *Poly-Mapping.keys a*
⟨*proof*⟩

lemma *keys-diff*:
Poly-Mapping.keys(*a* *−* *b*) $\subseteq$ *Poly-Mapping.keys a* $\cup$ *Poly-Mapping.keys b*
⟨*proof*⟩

lemma *keys-eq-empty* [*simp*]: *Poly-Mapping.keys c = {} ⟷ c = 0*
⟨*proof*⟩

lemma *frag-cmul-eq-0-iff* [*simp*]: *frag-cmul k c = 0 ⟷ k=0 ∨ c=0*
⟨*proof*⟩

lemma *frag-of-eq*: *frag-of x = frag-of y ⟷ x = y*
⟨*proof*⟩

lemma *frag-cmul-distrib*: *frag-cmul (c+d) a = frag-cmul c a + frag-cmul d a*
⟨*proof*⟩

lemma *frag-cmul-distrib2*: *frag-cmul c (a+b) = frag-cmul c a + frag-cmul c b*
⟨*proof*⟩

lemma *frag-cmul-diff-distrib*: *frag-cmul (a − b) c = frag-cmul a c − frag-cmul b c*
⟨*proof*⟩

lemma *frag-cmul-sum*:
frag-cmul a (sum b I) = (∑ i∈I. frag-cmul a (b i))
⟨*proof*⟩

lemma *keys-sum*: *Poly-Mapping.keys(sum b I) ⊆ (⋃ i ∈I. Poly-Mapping.keys(b i))*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *frag-extend-of* [*simp*]: *frag-extend f (frag-of a) = f a*
⟨*proof*⟩

lemma *frag-extend-cmul*:
frag-extend f (frag-cmul c x) = frag-cmul c (frag-extend f x)
⟨*proof*⟩

lemma *frag-extend-minus*:
frag-extend f (− x) = − (frag-extend f x)
⟨*proof*⟩

lemma *frag-extend-add*:
frag-extend f (a+b) = (frag-extend f a) + (frag-extend f b)
⟨*proof*⟩

lemma *frag-extend-diff*:
frag-extend f (a−b) = (frag-extend f a) − (frag-extend f b)
⟨*proof*⟩

lemma *frag-extend-sum*:
finite I ⟹ frag-extend f (∑ i∈I. g i) = sum (frag-extend f o g) I
⟨*proof*⟩

lemma *frag-extend-eq*:
(⋀f. f ∈ Poly-Mapping.keys c ⟹ g f = h f) ⟹ frag-extend g c = frag-extend h c
⟨*proof*⟩

lemma *frag-extend-eq-0*:
(⋀x. x ∈ Poly-Mapping.keys c ⟹ f x = 0) ⟹ frag-extend f c = 0
⟨*proof*⟩

lemma *keys-frag-extend*: *Poly-Mapping.keys(frag-extend f c) ⊆ (⋃ x ∈ Poly-Mapping.keys c. Poly-Mapping.keys(f x))*
⟨*proof*⟩

lemma *frag-expansion*: *a = frag-extend frag-of a*
⟨*proof*⟩

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*⟩

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*⟩

lemma *frag-extend-compose*:
frag-extend f (frag-extend (frag-of o g) c) = frag-extend (f o g) c
⟨*proof*⟩

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*⟩

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)*
⟨*proof*⟩
termination ⟨*proof*⟩

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))
⟨*proof*⟩

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* $= (f\ x \mathbin{\frown\!\!\frown} n)\ y$
⟨*proof*⟩

context *monoid-mult*
begin

lemma *power-by-squaring*: *efficient-funpow* $(*)$ $(1 :: 'a) = (\frown)$
⟨*proof*⟩

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* ≈››-/ ≈ -)› [*51*, *51*] *50*)

lemma *equivD1*: $x \leq y$ **if** $x \approx y$
  ⟨*proof*⟩

lemma *equivD2*: $y \leq x$ **if** $x \approx y$
  ⟨*proof*⟩

lemma *equiv-refl* [*iff*]: $x \approx x$
  ⟨*proof*⟩

lemma *equiv-sym*: $x \approx y \longleftrightarrow y \approx x$
  ⟨*proof*⟩

lemma *equiv-trans*: $x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$
  ⟨*proof*⟩

lemma *equiv-antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
  ⟨*proof*⟩

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge \neg\ x \approx y$
  ⟨*proof*⟩

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x \approx y$
  ⟨*proof*⟩

lemma *le-imp-less-or-equiv*: $x \leq y \Longrightarrow x < y \vee x \approx y$
  ⟨*proof*⟩

lemma *less-imp-not-equiv*: $x < y \Longrightarrow \neg\ x \approx y$
  ⟨*proof*⟩

lemma *not-equiv-le-trans*: $\neg\ a \approx b \Longrightarrow a \leq b \Longrightarrow a < b$
  ⟨*proof*⟩

lemma *le-not-equiv-trans*: $a \leq b \Longrightarrow \neg\ a \approx b \Longrightarrow a < b$
⟨*proof*⟩

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x \approx y$
⟨*proof*⟩

end

⟨*ML*⟩

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 ⟨*proof*⟩
end

instantiation *prod* :: (*plus*, *plus*) *plus*
begin

definition *plus-prod-def*:
$x + y = (fst\ x + fst\ y, snd\ x + snd\ y)$

instance ⟨*proof*⟩
end

instantiation *prod* :: (*minus*, *minus*) *minus*
begin

definition *minus-prod-def*:
$x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$

instance ⟨*proof*⟩
end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*
begin

definition *uminus-prod-def*:
$- x = (- fst\ x, - snd\ x)$

instance ⟨*proof*⟩
end

lemma *fst-zero* [*simp*]: $fst\ 0 = 0$
⟨*proof*⟩

lemma *snd-zero* [*simp*]: $snd\ 0 = 0$
⟨*proof*⟩

lemma *fst-add* [*simp*]: $fst\ (x + y) = fst\ x + fst\ y$
⟨*proof*⟩

lemma *snd-add* [*simp*]: $snd\ (x + y) = snd\ x + snd\ y$
⟨*proof*⟩

lemma *fst-diff* [*simp*]: $fst\ (x - y) = fst\ x - fst\ y$
⟨*proof*⟩

lemma *snd-diff* [*simp*]: $snd\ (x - y) = snd\ x - snd\ y$
⟨*proof*⟩

lemma *fst-uminus* [*simp*]: $fst\ (- x) = - fst\ x$
⟨*proof*⟩

lemma *snd-uminus* [*simp*]: $snd\ (- x) = - snd\ x$
⟨*proof*⟩

lemma *add-Pair* [*simp*]: $(a, b) + (c, d) = (a + c, b + d)$
⟨*proof*⟩

lemma *diff-Pair* [*simp*]: $(a, b) - (c, d) = (a - c, b - d)$
⟨*proof*⟩

lemma *uminus-Pair* [*simp*, *code*]: $- (a, b) = (- a, - b)$
⟨*proof*⟩

82.2 Class instances

instance *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *prod* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*monoid-add*, *monoid-add*) *monoid-add*
⟨*proof*⟩

instance *prod* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *prod* :: (*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
⟨*proof*⟩

instance *prod* :: (*group-add*, *group-add*) *group-add*
⟨*proof*⟩

instance *prod* :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
⟨*proof*⟩

lemma *fst-sum*: *fst* $(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ fst\ (f\ x))$
⟨*proof*⟩

lemma *snd-sum*: *snd* $(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ snd\ (f\ x))$
⟨*proof*⟩

lemma *sum-prod*: $(\sum x{\in}A.\ (f\ x,\ g\ x)) = (\sum x{\in}A.\ f\ x,\ \sum x{\in}A.\ g\ x)$
⟨*proof*⟩

end

83 Roots of real quadratics

theory *Quadratic-Discriminant*
imports *Complex-Main*
begin

definition *discrim* :: *real* ⇒ *real* ⇒ *real* ⇒ *real*
where $discrim\ a\ b\ c \equiv b^2 - 4 * a * c$

lemma *complete-square*:
$a \neq 0 \Longrightarrow a * x^2 + b * x + c = 0 \longleftrightarrow (2 * a * x + b)^2 = discrim\ a\ b\ c$
⟨*proof*⟩

lemma *discriminant-negative*:
fixes *a b c x* :: *real*
assumes $a \neq 0$
and $discrim\ a\ b\ c < 0$
shows $a * x^2 + b * x + c \neq 0$
⟨*proof*⟩

lemma *plus-or-minus-sqrt*:
fixes $x\ y :: real$
assumes $y \geq 0$
shows $x^2 = y \longleftrightarrow x = sqrt\ y \vee x = -\ sqrt\ y$
⟨*proof*⟩

lemma *divide-non-zero*:
fixes $x\ y\ z :: real$
assumes $x \neq 0$
shows $x * y = z \longleftrightarrow y = z\ /\ x$
⟨*proof*⟩

lemma *discriminant-nonneg*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c \geq 0$
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*⟩

lemma *discriminant-zero*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c = 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b\ /\ (2 * a)$
⟨*proof*⟩

theorem *discriminant-iff*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
$discrim\ a\ b\ c \geq 0 \wedge$
$(x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a))$
⟨*proof*⟩

lemma *discriminant-nonneg-ex*:
fixes $a\ b\ c :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c \geq 0$
shows $\exists\ x.\ a * x^2 + b * x + c = 0$
⟨*proof*⟩

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*⟩

lemma *discriminant-pos-distinct*:
fixes $a\ b\ c\ x$ :: *real*
assumes $a \neq 0$
and $discrim\ a\ b\ c > 0$
shows $\exists\ y.\ x \neq y \wedge a * y^2 + b * y + c = 0$
⟨*proof*⟩

lemma *Rats-solution-QE*:
assumes $a \in \mathbb{Q}$ $b \in \mathbb{Q}$ $a \neq 0$
and $a*x\hat{}2 + b*x + c = 0$
and $sqrt\ (discrim\ a\ b\ c) \in \mathbb{Q}$
shows $x \in \mathbb{Q}$
⟨*proof*⟩

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*⟩

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 (=) = (=)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *set-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-vset R*) (*vimage Rep*) (*vimage Abs*)
⟨*proof*⟩

declare [[*mapQ3 set* = (*rel-vset*, *set-quotient*)]]

lemma *empty-set-rsp*[*quot-respect*]:
rel-vset R {} {}
⟨*proof*⟩

lemma *collect-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows ((*R* ===> (=)) ===> *rel-vset R*) *Collect Collect*
⟨*proof*⟩

lemma *collect-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((*Abs* ---> *id*) ---> (− ‘) *Rep*) *Collect* = *Collect*
⟨*proof*⟩

lemma *union-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) (∪) (∪)
⟨*proof*⟩

lemma *union-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((− ‘) *Abs* ---> (− ‘) *Abs* ---> (− ‘) *Rep*) (∪) = (∪)
⟨*proof*⟩

lemma *diff-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) (−) (−)
⟨*proof*⟩

lemma *diff-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((−‘) *Abs* −−−> (−‘) *Abs* −−−> (−‘) *Rep*) (−) = (−)
⟨*proof*⟩

lemma *inter-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) (∩) (∩)
⟨*proof*⟩

lemma *inter-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((−‘) *Abs* −−−> (−‘) *Abs* −−−> (−‘) *Rep*) (∩) = (∩)
⟨*proof*⟩

lemma *mem-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*Rep* −−−> (−‘) *Abs* −−−> *id*) (∈) = (∈)
⟨*proof*⟩

lemma *mem-rsp*[*quot-respect*]:
shows (*R* ===> *rel-vset R* ===> (=)) (∈) (∈)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-prod-eq* [*id-simps*]:
shows *rel-prod* (=) (=) = (=)
⟨*proof*⟩

lemma *prod-equivp* [*quot-equiv*]:
assumes *equivp R1*
assumes *equivp R2*
shows *equivp* (*rel-prod R1 R2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fst-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2* ===> *R1*) *fst fst*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *snd-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2* ===> *R2*) *snd snd*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *case-prod-rsp* [*quot-respect*]:
shows ((*R1* ===> *R2* ===> (=)) ===> (*rel-prod R1 R2*) ===> (=))
case-prod case-prod
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*quot-respect*]:
shows ((*R2* ===> *R2* ===> (=)) ===> (*R1* ===> *R1* ===> (=)) ===> *rel-prod R2 R1* ===> *rel-prod R2 R1* ===> (=)) *rel-prod rel-prod*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-option-map2*:
rel-option R x (*map-option f y*) ⟷ *rel-option* (λ*x y*. *R x* (*f y*)) *x y*
⟨*proof*⟩

declare
map-option.id [*id-simps*]
option.rel-eq [*id-simps*]

lemma *reflp-rel-option*:
reflp R $\Longrightarrow$ *reflp* (*rel-option R*)
⟨*proof*⟩

lemma *option-symp*:
symp R $\Longrightarrow$ *symp* (*rel-option R*)
⟨*proof*⟩

lemma *option-transp*:
transp R $\Longrightarrow$ *transp* (*rel-option R*)
⟨*proof*⟩

lemma *option-equivp* [*quot-equiv*]:
equivp R $\Longrightarrow$ *equivp* (*rel-option R*)
⟨*proof*⟩

lemma *option-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-option R*) (*map-option Abs*) (*map-option Rep*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *option-Some-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *rel-option R*) *Some Some*
⟨*proof*⟩

lemma *option-None-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map-option Abs None* = *None*
⟨*proof*⟩

lemma *option-Some-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*Rep* −−−> *map-option Abs*) *Some* = *Some*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *list-all2-eq* [*id-simps*]:
list-all2 (=) = (=)
⟨*proof*⟩

lemma *reflp-list-all2*:
assumes *reflp R*
shows *reflp (list-all2 R)*
⟨*proof*⟩

lemma *list-symp*:
assumes *symp R*
shows *symp (list-all2 R)*
⟨*proof*⟩

lemma *list-transp*:
assumes *transp R*
shows *transp (list-all2 R)*
⟨*proof*⟩

lemma *list-equivp* [*quot-equiv*]:
equivp R ⟹ equivp (list-all2 R)
⟨*proof*⟩

lemma *list-quotient3* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3 (list-all2 R) (map Abs) (map Rep)*
⟨*proof*⟩

declare [[*mapQ3 list = (list-all2, list-quotient3)*]]

lemma *cons-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *(Rep ---> (map Rep) ---> (map Abs)) (#) = (#)*
⟨*proof*⟩

lemma *cons-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *(R ===> list-all2 R ===> list-all2 R) (#) (#)*
⟨*proof*⟩

lemma *nil-prs* [*quot-preserve*]:

assumes *q*: *Quotient3 R Abs Rep*
shows *map Abs* [] = []
⟨*proof*⟩

lemma *nil-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *list-all2 R* [] []
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-all2-rsp*:
assumes *r*: $\forall\, x\ y.\ R\ x\ y \longrightarrow (\forall\, a\ b.\ R\ a\ b \longrightarrow S\ x\ a = T\ y\ b)$
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*
⟨*proof*⟩

lemma [*quot-respect*]:
((R ===> R ===> (=)) ===> list-all2 R ===> list-all2 R ===> (=)) list-all2 list-all2
⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows (*(abs1 ---> abs1 ---> id) ---> map rep1 ---> map rep1 ---> id) list-all2 = list-all2*
⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows (*list-all2 ((rep1 ---> rep1 ---> id) R) l m*) = (*l = m*)
⟨*proof*⟩

lemma *list-all2-find-element*:
assumes *a*: $x \in set\ a$
and *b*: *list-all2 R a b*
shows $\exists\, y.\ (y \in set\ b \wedge R\ x\ y)$

⟨*proof*⟩

lemma *list-all2-refl*:
assumes *a*: $\bigwedge x\ y.\ R\ x\ y = (R\ x = R\ y)$
shows *list-all2 R x x*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-sum-id* [*id-simps*]:
map-sum id id = *id*
⟨*proof*⟩

lemma *rel-sum-eq* [*id-simps*]:
rel-sum (=) (=) = (=)
⟨*proof*⟩

lemma *reflp-rel-sum*:
reflp R1 ⟹ *reflp R2* ⟹ *reflp* (*rel-sum R1 R2*)
⟨*proof*⟩

lemma *sum-symp*:
symp R1 ⟹ *symp R2* ⟹ *symp* (*rel-sum R1 R2*)
⟨*proof*⟩

lemma *sum-transp*:
transp R1 ⟹ *transp R2* ⟹ *transp* (*rel-sum R1 R2*)
⟨*proof*⟩

lemma *sum-equivp* [*quot-equiv*]:
equivp R1 ⟹ *equivp R2* ⟹ *equivp* (*rel-sum R1 R2*)

⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⇒ *'a* ⇒ *bool* (**infixl** ‹∼› *50*)

class *equiv* = *eqv* +
  assumes *equiv-refl* [*intro*]: *x* ∼ *x*
    and *equiv-trans* [*trans*]: *x* ∼ *y* ⟹ *y* ∼ *z* ⟹ *x* ∼ *z*
    and *equiv-sym* [*sym*]: *x* ∼ *y* ⟹ *y* ∼ *x*
begin

lemma *equiv-not-sym* [*sym*]: ¬ *x* ∼ *y* ⟹ ¬ *y* ∼ *x*
⟨*proof*⟩

lemma *not-equiv-trans1* [*trans*]: ¬ *x* ∼ *y* ⟹ *y* ∼ *z* ⟹ ¬ *x* ∼ *z*
⟨*proof*⟩

lemma *not-equiv-trans2* [*trans*]: *x* ∼ *y* ⟹ ¬ *y* ∼ *z* ⟹ ¬ *x* ∼ *z*
⟨*proof*⟩

end

The quotient type *'a quot* consists of all *equivalence classes* over elements of the base type *'a*.

definition (**in** *eqv*) *quot* = {{*x*. *a* ∼ *x*} | *a*. *True*}

typedef (**overloaded**) *'a quot* = *quot* :: *'a*::*eqv set set*
  ⟨*proof*⟩

lemma *quotI* [*intro*]: {*x*. *a* ∼ *x*} ∈ *quot*
  ⟨*proof*⟩

lemma *quotE* [*elim*]:
  assumes *R* ∈ *quot*
  obtains *a* **where** *R* = {*x*. *a* ∼ *x*}
  ⟨*proof*⟩

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 ⌊*a*⌋ = *Abs-quot* {*x*. *a* ∼ *x*}

theorem *quot-exhaust*: ∃ *a*. *A* = ⌊*a*⌋
⟨*proof*⟩

lemma *quot-cases* [*cases type*: *quot*]:
  obtains *a* **where** *A* = ⌊*a*⌋
  ⟨*proof*⟩

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*⟩

90.3 Picking representing elements

definition *pick* :: $'a$::*equiv quot* ⇒ $'a$
  where *pick* $A = (SOME\ a.\ A = \lfloor a \rfloor)$

theorem *pick-equiv* [*intro*]: *pick* $\lfloor a \rfloor \sim a$
⟨*proof*⟩

theorem *pick-inverse* [*intro*]: $\lfloor pick\ A \rfloor = A$
⟨*proof*⟩

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*: $\bigwedge X\ Y.\ P\ X\ Y \Longrightarrow f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)$
    and *cong*: $\bigwedge x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$
      $\Longrightarrow P\ \lfloor x \rfloor\ \lfloor y \rfloor \Longrightarrow P\ \lfloor x' \rfloor\ \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
    and *P*: $P\ \lfloor a \rfloor\ \lfloor b \rfloor$
  shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
⟨*proof*⟩

theorem *quot-function*:
  assumes $\bigwedge X\ Y.\ f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)$
    and $\bigwedge x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
  shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
  ⟨*proof*⟩

theorem *quot-function'*:
  $(\bigwedge X\ Y.\ f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)) \Longrightarrow$
    $(\bigwedge x\ x'\ y\ y'.\ x \sim x' \Longrightarrow y \sim y' \Longrightarrow g\ x\ y = g\ x'\ y') \Longrightarrow$
    $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
  ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nsets-mono*: *A* ⊆ *B* ⟹ *nsets A n* ⊆ *nsets B n*
⟨*proof*⟩

lemma *nsets-Pi-contra*: *A'* ⊆ *A* ⟹ *Pi* ([*A*]^*n*) *B* ⊆ *Pi* ([*A'*]^*n*) *B*
⟨*proof*⟩

lemma *nsets-2-eq*: [*A*]^2 = (⋃*x*∈*A*. ⋃*y*∈*A* − {*x*}. {{*x*, *y*}})
⟨*proof*⟩

lemma *nsets2-E*:
assumes *e* ∈ [*A*]^2
obtains *x y* **where** *e* = {*x*,*y*} *x* ∈ *A* *y* ∈ *A* *x*≠*y*
⟨*proof*⟩

lemma *nsets-doubleton-2-eq* [*simp*]: [{*x*, *y*}]^2 = (*if x*=*y then* {} *else* {{*x*, *y*}})
⟨*proof*⟩

lemma *doubleton-in-nsets-2* [*simp*]: {*x*,*y*} ∈ [*A*]^2 ⟷ *x* ∈ *A* ∧ *y* ∈ *A* ∧ *x* ≠ *y*
⟨*proof*⟩

lemma *nsets-3-eq*: [*A*]^3 = (⋃*x*∈*A*. ⋃*y*∈*A* − {*x*}. ⋃*z*∈*A* − {*x*,*y*}. {{*x*,*y*,*z*}})
⟨*proof*⟩

lemma *nsets-4-eq*: [*A*]^4 = (⋃*u*∈*A*. ⋃*x*∈*A* − {*u*}. ⋃*y*∈*A* − {*u*,*x*}. ⋃*z*∈*A* − {*u*,*x*,*y*}. {{*u*,*x*,*y*,*z*}})
(**is** - = *?rhs*)
⟨*proof*⟩

lemma *nsets-disjoint-2*:
X ∩ *Y* = {} ⟹ [*X* ∪ *Y*]^2 = [*X*]^2 ∪ [*Y*]^2 ∪ (⋃*x*∈*X*. ⋃*y*∈*Y*. {{*x*,*y*}})
⟨*proof*⟩

lemma *ordered-nsets-2-eq*:
fixes *A* :: *'a::linorder set*
shows [*A*]^2 = {{*x*,*y*} | *x y*. *x* ∈ *A* ∧ *y* ∈ *A* ∧ *x*<*y*}
(**is** - = *?rhs*)

⟨*proof*⟩

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*⟩

lemma *ordered-nsets-4-eq*:
fixes *A* :: *'a*::*linorder set*
defines *rhs* ≡ $\lambda U.\ \exists u\ x\ y\ z.\ U = \{u,x,y,z\} \wedge u \in A \wedge x \in A \wedge y \in A \wedge z \in A \wedge u < x \wedge x < y \wedge y < z$
shows $[A]^4$ = *Collect rhs*
⟨*proof*⟩

lemma *ordered-nsets-5-eq*:
fixes *A* :: *'a*::*linorder set*
defines *rhs* ≡ $\lambda U.\ \exists u\ v\ x\ y\ z.\ U = \{u,v,x,y,z\} \wedge u \in A \wedge v \in A \wedge x \in A \wedge y \in A \wedge z \in A \wedge u < v \wedge v < x \wedge x < y \wedge y < z$
shows $[A]^5$ = *Collect rhs*
⟨*proof*⟩

lemma *binomial-eq-nsets*: *n choose k* = *card* (*nsets* {*0*..<*n*} *k*)
⟨*proof*⟩

lemma *nsets-eq-empty-iff*: *nsets A r* = {} ⟷ *finite A* ∧ *card A* < *r*
⟨*proof*⟩

lemma *nsets-eq-empty*: ⟦*finite A*; *card A* < *r*⟧ ⟹ *nsets A r* = {}
⟨*proof*⟩

lemma *nsets-empty-iff*: *nsets* {} *r* = (*if r=0 then* {{}} *else* {})
⟨*proof*⟩

lemma *nsets-singleton-iff*: *nsets* {*a*} *r* = (*if r=0 then* {{}} *else if r=1 then* {{*a*}} *else* {})
⟨*proof*⟩

lemma *nsets-self* [*simp*]: *nsets* {..<*m*} *m* = {{..<*m*}}
⟨*proof*⟩

lemma *nsets-zero* [*simp*]: *nsets A 0* = {{}}
⟨*proof*⟩

lemma *nsets-one*: *nsets A* (*Suc 0*) = (λ*x*. {*x*}) ` *A*
⟨*proof*⟩

lemma *inj-on-nsets*:
assumes *inj-on f A*

shows *inj-on* $(\lambda X.\ f\ `\ X)\ ([A]^n)$
⟨*proof*⟩

lemma *bij-betw-nsets*:
assumes *bij-betw f A B*
shows *bij-betw* $(\lambda X.\ f\ `\ X)\ ([A]^n)\ ([B]^n)$
⟨*proof*⟩

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*⟩

lemma *nsets-image-funcset*:
assumes $g \in S \to T$ **and** *inj-on g S*
shows $(\lambda X.\ g\ `\ X) \in [S]^k \to [T]^k$
⟨*proof*⟩

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*⟩

91.1.2 Further properties, involving equipollence

lemma *nsets-lepoll-cong*:
assumes $A \lesssim B$
shows $[A]^k \lesssim [B]^k$
⟨*proof*⟩

lemma *nsets-eqpoll-cong*:
assumes $A \approx B$
shows $[A]^k \approx [B]^k$
⟨*proof*⟩

lemma *infinite-imp-infinite-nsets*:
assumes *inf*: *infinite A* **and** $k>0$
shows *infinite* $([A]^k)$
⟨*proof*⟩

lemma *finite-nsets-iff*:
assumes $k>0$
shows *finite* $([A]^k) \longleftrightarrow$ *finite A*
⟨*proof*⟩

lemma *card-nsets* [*simp*]: *card* (*nsets A k*) = *card A choose k*
⟨*proof*⟩

91.1.3 Partition predicates

definition *monochromatic* $\equiv \lambda\beta\ \alpha\ \gamma\ f\ i.\ \exists H \in nsets\ \beta\ \alpha.\ f\ `\ (nsets\ H\ \gamma) \subseteq \{i\}$

uniform partition sizes

definition *partn* :: *'a set* ⇒ *nat* ⇒ *nat* ⇒ *'b set* ⇒ *bool*
where *partn* $\beta\ \alpha\ \gamma\ \delta \equiv \forall f \in nsets\ \beta\ \gamma\ \rightarrow\ \delta.\ \exists \xi{\in}\delta.\ monochromatic\ \beta\ \alpha\ \gamma\ f\ \xi$

partition sizes enumerated in a list

definition *partn-lst* :: *'a set* ⇒ *nat list* ⇒ *nat* ⇒ *bool*
where *partn-lst* $\beta\ \alpha\ \gamma \equiv \forall f \in nsets\ \beta\ \gamma\ \rightarrow\ \{..{<}length\ \alpha\}.\ \exists i < length\ \alpha.$ *monochromatic* $\beta\ (\alpha!i)\ \gamma\ f\ i$

There's always a 0-clique

lemma *partn-lst-0*: $\gamma > 0 \Longrightarrow$ *partn-lst* $\beta\ (0\#\alpha)\ \gamma$
⟨*proof*⟩

lemma *partn-lst-0'*: $\gamma > 0 \Longrightarrow$ *partn-lst* $\beta\ (a\#0\#\alpha)\ \gamma$
⟨*proof*⟩

lemma *partn-lst-greater-resource*:
fixes *M*::*nat*
assumes *M*: *partn-lst* $\{..{<}M\}\ \alpha\ \gamma$ **and** $M \le N$
shows *partn-lst* $\{..{<}N\}\ \alpha\ \gamma$
⟨*proof*⟩

lemma *partn-lst-fewer-colours*:
assumes *major*: *partn-lst* $\beta\ (n\#\alpha)\ \gamma$ **and** $n \ge \gamma$
shows *partn-lst* $\beta\ \alpha\ \gamma$
⟨*proof*⟩

lemma *partn-lst-eq-partn*: *partn-lst* $\{..{<}n\}\ [m,m]\ 2 =$ *partn* $\{..{<}n\}\ m\ 2\ \{..{<}2{::}nat\}$
⟨*proof*⟩

lemma *partn-lstE*:
assumes *partn-lst* $\beta\ \alpha\ \gamma\ f \in nsets\ \beta\ \gamma\ \rightarrow\ \{..{<}l\}$ *length* $\alpha = l$
obtains *i H* **where** $i <$ *length* $\alpha\ H \in nsets\ \beta\ (\alpha!i)\ f\ `\ (nsets\ H\ \gamma) \subseteq \{i\}$
⟨*proof*⟩

lemma *partn-lst-less*:
assumes *M*: *partn-lst* $\beta\ \alpha\ n$ **and** *eq*: *length* $\alpha' =$ *length* α
and *le*: $\bigwedge i.\ i <$ *length* $\alpha \Longrightarrow \alpha'!i \le \alpha!i$
shows *partn-lst* $\beta\ \alpha'\ n$
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *ES-1* [*simp*]: *ES 1 k l* = *k*+*l*−*1*
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

The Erds–Szekeres upper bound

lemma *ES2-choose*: *ES 2 k l* = (*k*+*l*) *choose k*
⟨*proof*⟩

91.2.2 Trivial cases

Vacuous, since we are dealing with 0-sets!

lemma *ramsey0*: ∃ *N*::*nat*. *partn-lst* {..<*N*} [*q1*,*q2*] *0*
  ⟨*proof*⟩

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*⟩

lemma *ramsey1*: ∃ *N*::*nat*. *partn-lst* {..<*N*} [*q0*,*q1*] *1*
  ⟨*proof*⟩

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*⟩

proposition *ramsey2-full*: *partn-lst* {..<*ES r q1 q2*} [*q1*,*q2*] *r*
⟨*proof*⟩

91.2.4 Full Ramsey's theorem with multiple colours and arbitrary exponents

theorem *ramsey-full*: ∃ *N*::*nat*. *partn-lst* {..<*N*} *qs r*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *null-clique*[*simp*]: *clique* {} *E* **and** *null-indep*[*simp*]: *indep* {} *E*
⟨*proof*⟩

lemma *smaller-clique*: ⟦*clique R E*; *R*′ ⊆ *R*⟧ ⟹ *clique R*′ *E*
⟨*proof*⟩

lemma *smaller-indep*: ⟦*indep R E*; *R*′ ⊆ *R*⟧ ⟹ *indep R*′ *E*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *part-fn-subset*: *part-fn r s YY f* ⟹ *Y* ⊆ *YY* ⟹ *part-fn r s Y f*
⟨*proof*⟩

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 $\exists\, Y'\ t'.\ Y' \subseteq YY \wedge \mathit{infinite}\ Y' \wedge t' < s \wedge (\forall X.\ X \subseteq Y' \wedge \mathit{finite}\ X \wedge \mathit{card}\ X = r \longrightarrow f\ X = t')$
⟨*proof*⟩

theorem *Ramsey*:
fixes *s r* :: *nat*
and *Z* :: *'a set*
and *f* :: *'a set* ⇒ *nat*
shows
⟦*infinite Z*;
$\forall X.\ X \subseteq Z \wedge \mathit{finite}\ X \wedge \mathit{card}\ X = r \longrightarrow f\ X < s$⟧
⟹ $\exists\, Y\ t.\ Y \subseteq Z \wedge \mathit{infinite}\ Y \wedge t < s$
$\wedge\ (\forall X.\ X \subseteq Y \wedge \mathit{finite}\ X \wedge \mathit{card}\ X = r \longrightarrow f\ X = t)$
⟨*proof*⟩

corollary *Ramsey2*:
fixes *s* :: *nat*

and $Z :: 'a\ set$
and $f :: 'a\ set \Rightarrow nat$
assumes *infZ*: *infinite Z*
and *part*: $\forall x{\in}Z.\ \forall y{\in}Z.\ x \neq y \longrightarrow f\ \{x, y\} < s$
shows $\exists Y\ t.\ Y \subseteq Z \wedge$ *infinite* $Y \wedge t < s \wedge (\forall x{\in}Y.\ \forall y{\in}Y.\ x{\neq}y \longrightarrow f\ \{x, y\} = t)$
⟨*proof*⟩

corollary *Ramsey-nsets*:
fixes $f :: 'a\ set \Rightarrow nat$
assumes *infinite Z f ' nsets Z r* $\subseteq \{..<s\}$
obtains *Y t* **where** $Y \subseteq Z$ *infinite Y t* $< s$ *f ' nsets Y r* $\subseteq \{t\}$
⟨*proof*⟩

Every infinite sequence of linearly ordered elements has an infinite monotone subsequence:

corollary *monotone-subsequence*:
fixes $f :: nat \Rightarrow 'a{::}linorder$
shows $\exists I{::}nat\ set.$ *infinite* $I \wedge$
$((\forall i{\in}I.\ \forall j{\in}I.\ i < j \longrightarrow f\ i \leq f\ j) \vee (\forall i{\in}I.\ \forall j{\in}I.\ i < j \longrightarrow f\ i > f\ j))$
⟨*proof*⟩

Every infinite sequence of *nat*s has an infinite monotonically increasing subsequence:

corollary *mono-incr-subsequence-nat*:
fixes $f :: nat \Rightarrow nat$
shows $\exists I{::}nat\ set.$ *infinite* $I \wedge (\forall i{\in}I.\ \forall j{\in}I.\ i < j \longrightarrow f\ i \leq f\ j)$
⟨*proof*⟩

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*⟩

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* $\{i, j\}$)
⟨*proof*⟩

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*⟩

theorem *trans-disj-wf-implies-wf*:
  assumes *trans r*
    and *disj-wf r*
  shows *wf r*
⟨*proof*⟩

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*|
  ⟨*proof*⟩

lemma *rmod-conv-frac'*: *x rmod y* = (*if y* = *0 then x else frac* (*x* / |*y*|) ∗ |*y*|)
  ⟨*proof*⟩

lemma *rmod-rmod* [*simp*]: (*x rmod y*) *rmod y* = *x rmod y*
  ⟨*proof*⟩

lemma *rmod-0-right* [*simp*]: *x rmod 0* = *x*
  ⟨*proof*⟩

lemma *rmod-less*: *m* > *0* ⟹ *x rmod m* < *m*
  ⟨*proof*⟩

lemma *rmod-less-abs*: *m* ≠ *0* ⟹ *x rmod m* < |*m*|
  ⟨*proof*⟩

lemma *rmod-le*: *m* > *0* ⟹ *x rmod m* ≤ *m*
  ⟨*proof*⟩

lemma *rmod-nonneg*: *m* ≠ *0* ⟹ *x rmod m* ≥ *0*
  ⟨*proof*⟩

lemma *rmod-unique*:
assumes $z \in \{0..<|y|\}$ $x = z + \mathit{of\text{-}int}\ n * y$
shows $x\ \mathit{rmod}\ y = z$
⟨*proof*⟩

lemma *rmod-0* [*simp*]: $0\ \mathit{rmod}\ z = 0$
⟨*proof*⟩

lemma *rmod-add*: $(x\ \mathit{rmod}\ z + y\ \mathit{rmod}\ z)\ \mathit{rmod}\ z = (x + y)\ \mathit{rmod}\ z$
⟨*proof*⟩

lemma *rmod-diff*: $(x\ \mathit{rmod}\ z - y\ \mathit{rmod}\ z)\ \mathit{rmod}\ z = (x - y)\ \mathit{rmod}\ z$
⟨*proof*⟩

lemma *rmod-self* [*simp*]: $x\ \mathit{rmod}\ x = 0$
⟨*proof*⟩

lemma *rmod-self-multiple-int* [*simp*]: $(\mathit{of\text{-}int}\ n * x)\ \mathit{rmod}\ x = 0$
⟨*proof*⟩

lemma *rmod-self-multiple-nat* [*simp*]: $(\mathit{of\text{-}nat}\ n * x)\ \mathit{rmod}\ x = 0$
⟨*proof*⟩

lemma *rmod-self-multiple-numeral* [*simp*]: $(\mathit{numeral}\ n * x)\ \mathit{rmod}\ x = 0$
⟨*proof*⟩

lemma *rmod-self-multiple-int′* [*simp*]: $(x * \mathit{of\text{-}int}\ n)\ \mathit{rmod}\ x = 0$
⟨*proof*⟩

lemma *rmod-self-multiple-nat′* [*simp*]: $(x * \mathit{of\text{-}nat}\ n)\ \mathit{rmod}\ x = 0$
⟨*proof*⟩

lemma *rmod-self-multiple-numeral′* [*simp*]: $(x * \mathit{numeral}\ n)\ \mathit{rmod}\ x = 0$
⟨*proof*⟩

lemma *rmod-idem* [*simp*]: $x \in \{0..<|y|\} \Longrightarrow x\ \mathit{rmod}\ y = x$
⟨*proof*⟩

definition *rcong* :: *real* ⇒ *real* ⇒ *real* ⇒ *bool*
(‹(‹*indent=1 notation=‹mixfix rcong›*›[- = -] ′(′ *rmod* -′))›)
where $[x = y]\ (\mathit{rmod}\ m) \longleftrightarrow x\ \mathit{rmod}\ m = y\ \mathit{rmod}\ m$

named-theorems *rcong-intros*

lemma *rcong-0-right* [*simp*]: $[x = y]\ (\mathit{rmod}\ 0) \longleftrightarrow x = y$
⟨*proof*⟩

lemma *rcong-0-iff*: $[x = 0]\ (rmod\ m) \longleftrightarrow x\ rmod\ m = 0$
and *rcong-0-iff'*: $[0 = x]\ (rmod\ m) \longleftrightarrow x\ rmod\ m = 0$
⟨*proof*⟩

lemma *rcong-refl* [*simp*, *intro*!, *rcong-intros*]: $[x = x]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-sym*: $[y = x]\ (rmod\ m) \Longrightarrow [x = y]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-sym-iff*: $[y = x]\ (rmod\ m) \longleftrightarrow [x = y]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-trans* [*trans*]: $[x = y]\ (rmod\ m) \Longrightarrow [y = z]\ (rmod\ m) \Longrightarrow [x = z]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-add* [*rcong-intros*]:
$[a = b]\ (rmod\ m) \Longrightarrow [c = d]\ (rmod\ m) \Longrightarrow [a + c = b + d]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-diff* [*rcong-intros*]:
$[a = b]\ (rmod\ m) \Longrightarrow [c = d]\ (rmod\ m) \Longrightarrow [a - c = b - d]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-uminus* [*rcong-intros*]:
$[a = b]\ (rmod\ m) \Longrightarrow [-a = -b]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-uminus-uminus-iff* [*simp*]: $[-x = -y]\ (rmod\ m) \longleftrightarrow [x = y]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-uminus-left-iff*: $[-x = y]\ (rmod\ m) \longleftrightarrow [x = -y]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-add-right-cancel* [*simp*]: $[a + c = b + c]\ (rmod\ m) \longleftrightarrow [a = b]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-add-left-cancel* [*simp*]: $[c + a = c + b]\ (rmod\ m) \longleftrightarrow [a = b]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-diff-right-cancel* [*simp*]: $[a - c = b - c]\ (rmod\ m) \longleftrightarrow [a = b]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-diff-left-cancel* [*simp*]: $[c - a = c - b]\ (rmod\ m) \longleftrightarrow [a = b]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-rmod-right-iff* [*simp*]: $[a = (b\ rmod\ m)]\ (rmod\ m) \longleftrightarrow [a = b]\ (rmod\ m)$
and *rcong-rmod-left-iff* [*simp*]: $[(a\ rmod\ m) = b]\ (rmod\ m) \longleftrightarrow [a = b]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-rmod-left* [*rcong-intros*]: $[a = b]\ (rmod\ m) \Longrightarrow [(a\ rmod\ m) = b]\ (rmod\ m)$
and *rcong-rmod-right* [*rcong-intros*]: $[a = b]\ (rmod\ m) \Longrightarrow [a = (b\ rmod\ m)]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-mult-of-int-0-left-left* [*rcong-intros*]: $[0 = of\text{-}int\ n * m]\ (rmod\ m)$
and *rcong-mult-of-int-0-right-left* [*rcong-intros*]: $[0 = m * of\text{-}int\ n]\ (rmod\ m)$
and *rcong-mult-of-int-0-left-right* [*rcong-intros*]: $[of\text{-}int\ n * m = 0]\ (rmod\ m)$
and *rcong-mult-of-int-0-right-right* [*rcong-intros*]: $[m * of\text{-}int\ n = 0]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-altdef*: $[a = b]\ (rmod\ m) \longleftrightarrow (\exists\, n.\ b = a + of\text{-}int\ n * m)$
⟨*proof*⟩

lemma *rcong-conv-diff-rmod-eq-0*: $[x = y]\ (rmod\ m) \longleftrightarrow (x - y)\ rmod\ m = 0$
⟨*proof*⟩

lemma *rcong-imp-eq*:
assumes $[x = y]\ (rmod\ m)\ |x - y| < |m|$
shows $x = y$
⟨*proof*⟩

lemma *rcong-mult-modulus*:
assumes $[a = b]\ (rmod\ (m\ /\ c))\ c \neq 0$
shows $[a * c = b * c]\ (rmod\ m)$
⟨*proof*⟩

lemma *rcong-divide-modulus*:
assumes $[a = b]\ (rmod\ (m * c))\ c \neq 0$
shows $[a\ /\ c = b\ /\ c]\ (rmod\ m)$
⟨*proof*⟩

lemma *sin-rmod* [*simp*]: $sin\ (x\ rmod\ (2{*}pi)) = sin\ x$
and *cos-rmod* [*simp*]: $cos\ (x\ rmod\ (2{*}pi)) = cos\ x$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cis-eq-iff*: *cis a* = *cis b* ⟷ [*a* = *b*] (*rmod* (*2* ∗ *pi*))
⟨*proof*⟩

lemma *cis-eq-1-iff*: *cis a* = *1* ⟷ (∃ *n*. *a* = *of-int n* ∗ (*2* ∗ *pi*))
⟨*proof*⟩

lemma *cis-cong*:
assumes [*a* = *b*] (*rmod 2* ∗ *pi*)
shows *cis a* = *cis b*
⟨*proof*⟩

lemma *rcis-cong*:
assumes [*a* = *b*] (*rmod 2* ∗ *pi*)
shows *rcis r a* = *rcis r b*
⟨*proof*⟩

lemma *sin-rcong*: [*x* = *y*] (*rmod* (*2* ∗ *pi*)) ⟹ *sin x* = *sin y*
and *cos-rcong*: [*x* = *y*] (*rmod* (*2* ∗ *pi*)) ⟹ *cos x* = *cos y*
⟨*proof*⟩

lemma *sin-eq-sin-conv-rmod*:
assumes *sin x* = *sin y*
shows [*x* = *y*] (*rmod 2* ∗ *pi*) ∨ [*x* = *pi* − *y*] (*rmod 2* ∗ *pi*)
⟨*proof*⟩

lemma *cos-eq-cos-conv-rmod*:
assumes *cos x* = *cos y*
shows [*x* = *y*] (*rmod 2* ∗ *pi*) ∨ [*x* = −*y*] (*rmod 2* ∗ *pi*)
⟨*proof*⟩

lemma *sin-eq-sin-conv-rmod-iff*:
sin x = *sin y* ⟷ [*x* = *y*] (*rmod 2* ∗ *pi*) ∨ [*x* = *pi* − *y*] (*rmod 2* ∗ *pi*)
⟨*proof*⟩

lemma *cos-eq-cos-conv-rmod-iff*:
cos x = *cos y* ⟷ [*x* = *y*] (*rmod 2* ∗ *pi*) ∨ [*x* = −*y*] (*rmod 2* ∗ *pi*)
⟨*proof*⟩

end

93 Generic reflection and reification

theory *Reflection*
imports *Main*
begin

⟨*ML*⟩

end

theory *Rewrite*
imports *Main*
begin

consts *rewrite-HOLE* :: *'a*::{} (‹¤›)

lemma *eta-expand*:
fixes *f* :: *'a*::{} ⇒ *'b*::{}
shows *f* ≡ λ*x*. *f x* ⟨*proof*⟩

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'*))
⟨*proof*⟩

⟨*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)*
⟨*ML*⟩

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*
⟨*proof*⟩

end

instantiation *num0* **and** *num1* :: *len0*
begin

definition *len-num0*: *len-of* (- :: *num0 itself*) = *0*
definition *len-num1*: *len-of* (- :: *num1 itself*) = *1*

instance ⟨*proof*⟩

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 ⟨*proof*⟩

end

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len*
⟨*proof*⟩
instance *bit0* :: (*len*) *len*
⟨*proof*⟩
instance *bit1* :: (*len0*) *len*
⟨*proof*⟩

instantiation *Enum.finite-1* :: *len*
begin

definition
len-of-finite-1 (*x* :: *Enum.finite-1 itself*) ≡ (*1* :: *nat*)

instance
⟨*proof*⟩

end

instantiation *Enum.finite-2* :: *len*
begin

definition

len-of-finite-2 (*x* :: *Enum.finite-2 itself*) ≡ (*2* :: *nat*)

instance
⟨*proof*⟩

end

instantiation *Enum.finite-3* :: *len*
begin

definition
len-of-finite-3 (*x* :: *Enum.finite-3 itself*) ≡ (*4* :: *nat*)

instance
⟨*proof*⟩

end

lemma *length-less-eq-Suc-0-iff* [*simp*]:
‹*LENGTH*('*a*::*len*) ≤ *Suc 0* ⟷ *LENGTH*('*a*) = *Suc 0*›
⟨*proof*⟩

lemma *length-not-greater-eq-2-iff* [*simp*]:
‹¬ *2* ≤ *LENGTH*('*a*::*len*) ⟷ *LENGTH*('*a*) = *Suc 0*›
⟨*proof*⟩

lemma *less-eq-decr-length-iff* [*simp*]:
‹*n* ≤ *LENGTH*('*a*::*len*) − *Suc 0* ⟷ *n* < *LENGTH*('*a*)›
⟨*proof*⟩

lemma *decr-length-less-iff* [*simp*]:
‹*LENGTH*('*a*::*len*) − *Suc 0* < *n* ⟷ *LENGTH*('*a*) ≤ *n*›
⟨*proof*⟩

context *linordered-idom*
begin

lemma *two-less-eq-exp-length* [*simp*]:
‹*2* ≤ *2* ^ *LENGTH*('*b*::*len*)›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sat-eqI*:
nat-of m = *nat-of n* ⟹ *m* = *n*
⟨*proof*⟩

lemma *sat-eq-iff*:
m = *n* ⟷ *nat-of m* = *nat-of n*
⟨*proof*⟩

lemma *Abs-sat-nat-of* [*code abstype*]:
Abs-sat (*nat-of n*) = *n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nat-of-le-len-of* [*simp*]:
nat-of (*n* :: ('*a*::*len*) *sat*) ≤ *LENGTH*('*a*)
⟨*proof*⟩

lemma *min-len-of-nat-of* [*simp*]:
min (*LENGTH*('*a*)) (*nat-of* (*n*::('*a*::*len*) *sat*)) = *nat-of n*
⟨*proof*⟩

lemma *min-nat-of-len-of* [*simp*]:
min (*nat-of* (*n*::('*a*::*len*) *sat*)) (*LENGTH*('*a*)) = *nat-of n*
⟨*proof*⟩

lemma *Abs-sat'-nat-of* [*simp*]:
Abs-sat' (*nat-of n*) = *n*
⟨*proof*⟩

instantiation *sat* :: (*len*) *linorder*
begin

definition
less-eq-sat-def: *x* ≤ *y* ⟷ *nat-of x* ≤ *nat-of y*

definition
less-sat-def: $x < y \longleftrightarrow nat\text{-}of\ x < nat\text{-}of\ y$

instance
⟨*proof*⟩

end

instantiation *sat* :: (*len*) {*minus*, *comm-semiring-1*}
begin

definition
$0 = Abs\text{-}sat'\ 0$

definition
$1 = Abs\text{-}sat'\ 1$

lemma *nat-of-zero-sat* [*simp*, *code abstract*]:
$nat\text{-}of\ 0 = 0$
⟨*proof*⟩

lemma *nat-of-one-sat* [*simp*, *code abstract*]:
$nat\text{-}of\ 1 = min\ 1\ (LENGTH('a))$
⟨*proof*⟩

definition
$x + y = Abs\text{-}sat'\ (nat\text{-}of\ x + nat\text{-}of\ y)$

lemma *nat-of-plus-sat* [*simp*, *code abstract*]:
$nat\text{-}of\ (x + y) = min\ (nat\text{-}of\ x + nat\text{-}of\ y)\ (LENGTH('a))$
⟨*proof*⟩

definition
$x - y = Abs\text{-}sat'\ (nat\text{-}of\ x - nat\text{-}of\ y)$

lemma *nat-of-minus-sat* [*simp*, *code abstract*]:
$nat\text{-}of\ (x - y) = nat\text{-}of\ x - nat\text{-}of\ y$
⟨*proof*⟩

definition
$x * y = Abs\text{-}sat'\ (nat\text{-}of\ x * nat\text{-}of\ y)$

lemma *nat-of-times-sat* [*simp*, *code abstract*]:
$nat\text{-}of\ (x * y) = min\ (nat\text{-}of\ x * nat\text{-}of\ y)\ (LENGTH('a))$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *sat* :: (*len*) *ordered-comm-semiring*
begin

instance
⟨*proof*⟩

end

lemma *Abs-sat'-eq-of-nat*: *Abs-sat'* *n* = *of-nat* *n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*code-abbrev*]:
of-nat (*numeral* *k*) = (*numeral* *k* :: *'a*::*len sat*)
⟨*proof*⟩

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*
⟨*proof*⟩

end

instance *sat* :: (*len*) *finite*
⟨*proof*⟩

instantiation *sat* :: (*len*) *equal*
begin

definition *HOL.equal* *A* *B* ⟷ *nat-of* *A* = *nat-of* *B*

instance
⟨*proof*⟩

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
⟨*proof*⟩

end

instantiation *sat* :: (*len*) {*Inf*, *Sup*}
begin

global-interpretation *Inf-sat*: *semilattice-neutr-set min* ‹*top* :: *'a sat*›
defines *Inf-sat* = *Inf-sat.F*
⟨*proof*⟩

global-interpretation *Sup-sat*: *semilattice-neutr-set max* ‹*bot* :: *'a sat*›
defines *Sup-sat* = *Sup-sat.F*
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *sat* :: (*len*) *complete-lattice*
⟨*proof*⟩

95.2 Enumeration

lemma *inj-on-sat-of-nat*:
shows *inj-on* (*of-nat* :: *nat* ⇒ *'a*::*len sat*) {*0*..<*LENGTH*(*'a*)}
⟨*proof*⟩

lemma *UNIV-sat-eq-of-nat*:
shows (*UNIV* :: *'a*::*len sat set*) = *of-nat* ‘ {*0*..*LENGTH*(*'a*)} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *intersection-of-inc*:
⟦*P* {*S*}; *Q S*⟧ ⟹ (*P intersection-of Q*) *S*
⟨*proof*⟩

lemma *union-of-mono*:
⟦(*P union-of Q*) *S*; ⋀*x*. *Q x* ⟹ *Q' x*⟧ ⟹ (*P union-of Q'*) *S*
⟨*proof*⟩

lemma *intersection-of-mono*:

$[\![(P \textit{ intersection-of } Q)\ S;\ \bigwedge x.\ Q\ x \Longrightarrow Q'\ x]\!] \Longrightarrow (P \textit{ intersection-of } Q')\ S$
⟨*proof*⟩

lemma *all-union-of*:
$(\forall S.\ (P \textit{ union-of } Q)\ S \longrightarrow R\ S) \longleftrightarrow (\forall T.\ P\ T \wedge T \subseteq \textit{Collect } Q \longrightarrow R(\bigcup T))$
⟨*proof*⟩

lemma *all-intersection-of*:
$(\forall S.\ (P \textit{ intersection-of } Q)\ S \longrightarrow R\ S) \longleftrightarrow (\forall T.\ P\ T \wedge T \subseteq \textit{Collect } Q \longrightarrow R(\bigcap T))$
⟨*proof*⟩

lemma *intersection-ofE*:
$[\![(P \textit{ intersection-of } Q)\ S;\ \bigwedge T.\ [\![P\ T;\ T \subseteq \textit{Collect } Q]\!] \Longrightarrow R(\bigcap T)]\!] \Longrightarrow R\ S$
⟨*proof*⟩

lemma *union-of-empty*:
$P\ \{\} \Longrightarrow (P \textit{ union-of } Q)\ \{\}$
⟨*proof*⟩

lemma *intersection-of-empty*:
$P\ \{\} \Longrightarrow (P \textit{ intersection-of } Q)\ \textit{UNIV}$
⟨*proof*⟩

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*⟩

lemma *arbitrary-union-of-empty* [*simp*]: $(\textit{arbitrary union-of } P)\ \{\}$
⟨*proof*⟩

lemma *arbitrary-intersection-of-empty* [*simp*]:
$(\textit{arbitrary intersection-of } P)\ \textit{UNIV}$
⟨*proof*⟩

lemma *arbitrary-union-of-inc*:
$P\ S \Longrightarrow (\textit{arbitrary union-of } P)\ S$
⟨*proof*⟩

lemma *arbitrary-intersection-of-inc*:
$P\ S \Longrightarrow (\textit{arbitrary intersection-of } P)\ S$
⟨*proof*⟩

lemma *arbitrary-union-of-complement*:
$(\textit{arbitrary union-of } P)\ S \longleftrightarrow (\textit{arbitrary intersection-of } (\lambda S.\ P(-\ S)))\ (-\ S)$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *arbitrary-intersection-of-complement*:
(*arbitrary intersection-of P*) *S* ⟷ (*arbitrary union-of* (λ*S*. *P*(− *S*))) (− *S*)
⟨*proof*⟩

lemma *arbitrary-union-of-idempot* [*simp*]:
arbitrary union-of arbitrary union-of P = *arbitrary union-of P*
⟨*proof*⟩

lemma *arbitrary-intersection-of-idempot*:
arbitrary intersection-of arbitrary intersection-of P = *arbitrary intersection-of P*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *arbitrary-union-of-Union*:
(⋀*S*. *S* ∈ 𝒰 ⟹ (*arbitrary union-of P*) *S*) ⟹ (*arbitrary union-of P*) (⋃𝒰)
⟨*proof*⟩

lemma *arbitrary-union-of-Un*:
⟦(*arbitrary union-of P*) *S*; (*arbitrary union-of P*) *T*⟧
⟹ (*arbitrary union-of P*) (*S* ∪ *T*)
⟨*proof*⟩

lemma *arbitrary-intersection-of-Inter*:
(⋀*S*. *S* ∈ 𝒰 ⟹ (*arbitrary intersection-of P*) *S*) ⟹ (*arbitrary intersection-of P*) (⋂𝒰)
⟨*proof*⟩

lemma *arbitrary-intersection-of-Int*:
⟦(*arbitrary intersection-of P*) *S*; (*arbitrary intersection-of P*) *T*⟧
⟹ (*arbitrary intersection-of P*) (*S* ∩ *T*)
⟨*proof*⟩

lemma *arbitrary-union-of-Int-eq*:
(∀ *S T*. (*arbitrary union-of P*) *S* ∧ (*arbitrary union-of P*) *T*
⟶ (*arbitrary union-of P*) (*S* ∩ *T*))
⟷ (∀ *S T*. *P S* ∧ *P T* ⟶ (*arbitrary union-of P*) (*S* ∩ *T*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *arbitrary-intersection-of-Un-eq*:
(∀ *S T*. (*arbitrary intersection-of P*) *S* ∧ (*arbitrary intersection-of P*) *T*
⟶ (*arbitrary intersection-of P*) (*S* ∪ *T*)) ⟷
(∀ *S T*. *P S* ∧ *P T* ⟶ (*arbitrary intersection-of P*) (*S* ∪ *T*))
⟨*proof*⟩

lemma *finite-union-of-empty* [*simp*]: (*finite union-of P*) {}
⟨*proof*⟩

lemma *finite-intersection-of-empty* [*simp*]: (*finite intersection-of P*) *UNIV*

⟨*proof*⟩

lemma *finite-union-of-inc*:
P S ⟹ (*finite union-of P*) *S*
⟨*proof*⟩

lemma *finite-intersection-of-inc*:
P S ⟹ (*finite intersection-of P*) *S*
⟨*proof*⟩

lemma *finite-union-of-complement*:
(*finite union-of P*) *S* ⟷ (*finite intersection-of* (λ*S*. *P*(− *S*))) (− *S*)
⟨*proof*⟩

lemma *finite-intersection-of-complement*:
(*finite intersection-of P*) *S* ⟷ (*finite union-of* (λ*S*. *P*(− *S*))) (− *S*)
⟨*proof*⟩

lemma *finite-union-of-idempot* [*simp*]:
finite union-of finite union-of P = *finite union-of P*
⟨*proof*⟩

lemma *finite-intersection-of-idempot* [*simp*]:
finite intersection-of finite intersection-of P = *finite intersection-of P*
⟨*proof*⟩

lemma *finite-union-of-Union*:
⟦*finite* 𝒰; ⋀*S*. *S* ∈ 𝒰 ⟹ (*finite union-of P*) *S*⟧ ⟹ (*finite union-of P*) (⋃𝒰)
⟨*proof*⟩

lemma *finite-union-of-Un*:
⟦(*finite union-of P*) *S*; (*finite union-of P*) *T*⟧ ⟹ (*finite union-of P*) (*S* ∪ *T*)
⟨*proof*⟩

lemma *finite-intersection-of-Inter*:
⟦*finite* 𝒰; ⋀*S*. *S* ∈ 𝒰 ⟹ (*finite intersection-of P*) *S*⟧ ⟹ (*finite intersection-of P*) (⋂𝒰)
⟨*proof*⟩

lemma *finite-intersection-of-Int*:
⟦(*finite intersection-of P*) *S*; (*finite intersection-of P*) *T*⟧
⟹ (*finite intersection-of P*) (*S* ∩ *T*)
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finite′-intersection-of-inc*:
P S ⟹ (*finite′ intersection-of P*) *S*
⟨*proof*⟩

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* ≡ λ*T*. ∃ *U*. *P U* ∧ *S* ∩ *U* = *T*

lemma *relative-to-UNIV* [*simp*]: (*P relative-to UNIV*) *S* ⟷ *P S*
⟨*proof*⟩

lemma *relative-to-imp-subset*:
(*P relative-to S*) *T* ⟹ *T* ⊆ *S*
⟨*proof*⟩

lemma *all-relative-to*: (∀ *S*. (*P relative-to U*) *S* ⟶ *Q S*) ⟷ (∀ *S*. *P S* ⟶ *Q*(*U* ∩ *S*))
⟨*proof*⟩

lemma *relative-toE*: ⟦(*P relative-to U*) *S*; ⋀*S*. *P S* ⟹ *Q*(*U* ∩ *S*)⟧ ⟹ *Q S*
⟨*proof*⟩

lemma *relative-to-inc*:
P S ⟹ (*P relative-to U*) (*U* ∩ *S*)
⟨*proof*⟩

lemma *relative-to-relative-to* [*simp*]:

P relative-to S relative-to T = P relative-to (S ∩ T)
⟨*proof*⟩

lemma *relative-to-compl*:
S ⊆ U ⟹ ((P relative-to U) (U − S) ⟷ ((λc. P(− c)) relative-to U) S)
⟨*proof*⟩

lemma *relative-to-subset-trans*:
⟦(P relative-to U) S; S ⊆ T; T ⊆ U⟧ ⟹ (P relative-to T) S
⟨*proof*⟩

lemma *relative-to-mono*:
⟦(P relative-to U) S; ⋀S. P S ⟹ Q S⟧ ⟹ (Q relative-to U) S
⟨*proof*⟩

lemma *relative-to-subset-inc*: *⟦S ⊆ U; P S⟧ ⟹ (P relative-to U) S*
⟨*proof*⟩

lemma *relative-to-Int*:
⟦(P relative-to S) C; (P relative-to S) D; ⋀X Y. ⟦P X; P Y⟧ ⟹ P(X ∩ Y)⟧
⟹ (P relative-to S) (C ∩ D)
⟨*proof*⟩

lemma *relative-to-Un*:
⟦(P relative-to S) C; (P relative-to S) D; ⋀X Y. ⟦P X; P Y⟧ ⟹ P(X ∪ Y)⟧
⟹ (P relative-to S) (C ∪ D)
⟨*proof*⟩

lemma *arbitrary-union-of-relative-to*:
((arbitrary union-of P) relative-to U) = (arbitrary union-of (P relative-to U))
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *finite-union-of-relative-to*:
((finite union-of P) relative-to U) = (finite union-of (P relative-to U)) (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *countable-union-of-relative-to*:
((countable union-of P) relative-to U) = (countable union-of (P relative-to U))
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *countable-union-of-empty* [*simp*]: (*countable union-of P*) {}
⟨*proof*⟩

lemma *countable-intersection-of-empty* [*simp*]: (*countable intersection-of P*) *UNIV*
⟨*proof*⟩

lemma *countable-union-of-inc*: *P S* ⟹ (*countable union-of P*) *S*
⟨*proof*⟩

lemma *countable-intersection-of-inc*: *P S* ⟹ (*countable intersection-of P*) *S*
⟨*proof*⟩

lemma *countable-union-of-complement*:
(*countable union-of P*) *S* ⟷ (*countable intersection-of* (λ*S*. *P*(−*S*))) (−*S*)
(**is** *?lhs=?rhs*)
⟨*proof*⟩

lemma *countable-intersection-of-complement*:
(*countable intersection-of P*) *S* ⟷ (*countable union-of* (λ*S*. *P*(− *S*))) (− *S*)
⟨*proof*⟩

lemma *countable-union-of-explicit*:
assumes *P* {}
shows (*countable union-of P*) *S* ⟷
(∃ *T*. (∀ *n*::*nat*. *P*(*T n*)) ∧ ⋃(*range T*) = *S*) (**is** *?lhs=?rhs*)
⟨*proof*⟩

lemma *countable-union-of-ascending*:
assumes *empty*: *P* {} **and** *Un*: ⋀*T U*. ⟦*P T*; *P U*⟧ ⟹ *P*(*T* ∪ *U*)
shows (*countable union-of P*) *S* ⟷
(∃ *T*. (∀ *n*. *P*(*T n*)) ∧ (∀ *n*. *T n* ⊆ *T*(*Suc n*)) ∧ ⋃(*range T*) = *S*) (**is** *?lhs=?rhs*)
⟨*proof*⟩

lemma *countable-union-of-idem* [*simp*]:
countable union-of countable union-of P = *countable union-of P* (**is** *?lhs=?rhs*)
⟨*proof*⟩

lemma *countable-intersection-of-idem* [*simp*]:

countable intersection-of countable intersection-of P =
countable intersection-of P
⟨*proof*⟩

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*) $(\bigcup\ \mathcal{U})$
⟨*proof*⟩

lemma *countable-union-of-UN*:
⟦*countable I*; $\bigwedge i.\ i \in I \Longrightarrow$ (*countable union-of P*) (*U i*)⟧
$\Longrightarrow$ (*countable union-of P*) $(\bigcup i{\in}I.\ U\ i)$
⟨*proof*⟩

lemma *countable-union-of-Un*:
⟦(*countable union-of P*) *S*; (*countable union-of P*) *T*⟧
$\Longrightarrow$ (*countable union-of P*) $(S \cup T)$
⟨*proof*⟩

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*) $(\bigcap\ \mathcal{U})$
⟨*proof*⟩

lemma *countable-intersection-of-INT*:
⟦*countable I*; $\bigwedge i.\ i \in I \Longrightarrow$ (*countable intersection-of P*) (*U i*)⟧
$\Longrightarrow$ (*countable intersection-of P*) $(\bigcap i{\in}I.\ U\ i)$
⟨*proof*⟩

lemma *countable-intersection-of-inter*:
⟦(*countable intersection-of P*) *S*; (*countable intersection-of P*) *T*⟧
$\Longrightarrow$ (*countable intersection-of P*) $(S \cap T)$
⟨*proof*⟩

lemma *countable-union-of-Int*:
assumes *S*: (*countable union-of P*) *S* **and** *T*: (*countable union-of P*) *T*
and *Int*: $\bigwedge S\ T.\ P\ S \wedge P\ T \Longrightarrow P(S \cap T)$
shows (*countable union-of P*) $(S \cap T)$
⟨*proof*⟩

lemma *countable-intersection-of-union*:
assumes *S*: (*countable intersection-of P*) *S* **and** *T*: (*countable intersection-of P*) *T*
and *Un*: $\bigwedge S\ T.\ P\ S \wedge P\ T \Longrightarrow P(S \cup T)$
shows (*countable intersection-of P*) $(S \cup T)$
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *smod-sdiv-mult-eq*:
‹*a smod b* + *a sdiv b* ∗ *b* = *a*›
⟨*proof*⟩

lemma *smod-mult-sdiv-eq*:
‹*a smod b* + *b* ∗ (*a sdiv b*) = *a*›
⟨*proof*⟩

lemma *minus-sdiv-mult-eq-smod*:
‹*a* − *a sdiv b* ∗ *b* = *a smod b*›
⟨*proof*⟩

lemma *minus-mult-sdiv-eq-smod*:
‹*a* − *b* ∗ (*a sdiv b*) = *a smod b*›
⟨*proof*⟩

lemma *minus-smod-eq-sdiv-mult*:
‹*a* − *a smod b* = *a sdiv b* ∗ *b*›
⟨*proof*⟩

lemma *minus-smod-eq-mult-sdiv*:
‹*a* − *a smod b* = *b* ∗ (*a sdiv b*)›
⟨*proof*⟩

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 ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *sdiv-int-div-0*:
  (*x* :: *int*) *sdiv 0* = *0*
  ⟨*proof*⟩

lemma *sdiv-int-0-div* [*simp*]:
  0 sdiv (*x* :: *int*) = *0*
  ⟨*proof*⟩

lemma *smod-int-alt-def*:
    (*a*::*int*) *smod b* = *sgn* (*a*) ∗ (*abs a mod abs b*)
  ⟨*proof*⟩

lemma *int-sdiv-simps* [*simp*]:

$(a :: int)\ sdiv\ 1 = a$
$(a :: int)\ sdiv\ 0 = 0$
$(a :: int)\ sdiv\ -1 = -a$
⟨*proof*⟩

lemma *smod-int-mod-0* [*simp*]:
$x\ smod\ (0 :: int) = x$
⟨*proof*⟩

lemma *smod-int-0-mod* [*simp*]:
$0\ smod\ (x :: int) = 0$
⟨*proof*⟩

lemma *sgn-sdiv-eq-sgn-mult*:
$a\ sdiv\ b \neq 0 \Longrightarrow sgn\ ((a :: int)\ sdiv\ b) = sgn\ (a * b)$
⟨*proof*⟩

lemma *int-sdiv-same-is-1* [*simp*]:
assumes $a \neq 0$
shows $((a :: int)\ sdiv\ b = a) = (b = 1)$
⟨*proof*⟩

lemma *int-sdiv-negated-is-minus1* [*simp*]:
$a \neq 0 \Longrightarrow ((a :: int)\ sdiv\ b = -\ a) = (b = -1)$
⟨*proof*⟩

lemma *sdiv-int-range*:
‹$a\ sdiv\ b \in \{-\ |a|..|a|\}$› **for** $a\ b :: int$
⟨*proof*⟩

lemma *smod-int-range*:
‹$a\ smod\ b \in \{-\ |b| + 1..|b| - 1\}$›
if ‹$b \neq 0$› **for** $a\ b :: int$
⟨*proof*⟩

lemma *smod-int-compares*:
$[\![\ 0 \leq a;\ 0 < b\ ]\!] \Longrightarrow (a :: int)\ smod\ b < b$
$[\![\ 0 \leq a;\ 0 < b\ ]\!] \Longrightarrow 0 \leq (a :: int)\ smod\ b$
$[\![\ a \leq 0;\ 0 < b\ ]\!] \Longrightarrow -b < (a :: int)\ smod\ b$
$[\![\ a \leq 0;\ 0 < b\ ]\!] \Longrightarrow (a :: int)\ smod\ b \leq 0$
$[\![\ 0 \leq a;\ b < 0\ ]\!] \Longrightarrow (a :: int)\ smod\ b < -\ b$
$[\![\ 0 \leq a;\ b < 0\ ]\!] \Longrightarrow 0 \leq (a :: int)\ smod\ b$
$[\![\ a \leq 0;\ b < 0\ ]\!] \Longrightarrow (a :: int)\ smod\ b \leq 0$
$[\![\ a \leq 0;\ b < 0\ ]\!] \Longrightarrow b \leq (a :: int)\ smod\ b$
⟨*proof*⟩

lemma *smod-mod-positive*:
$[\![\ 0 \leq (a :: int);\ 0 \leq b\ ]\!] \Longrightarrow a\ smod\ b = a\ mod\ b$
⟨*proof*⟩

lemma *minus-sdiv-eq* [*simp*]:
‹− *k sdiv l* = − (*k sdiv l*)› **for** *k l* :: *int*
⟨*proof*⟩

lemma *sdiv-minus-eq* [*simp*]:
‹*k sdiv* − *l* = − (*k sdiv l*)› **for** *k l* :: *int*
⟨*proof*⟩

lemma *sdiv-int-numeral-numeral* [*simp*]:
‹*numeral m sdiv numeral n* = *numeral m div* (*numeral n* :: *int*)›
⟨*proof*⟩

lemma *minus-smod-eq* [*simp*]:
‹− *k smod l* = − (*k smod l*)› **for** *k l* :: *int*
⟨*proof*⟩

lemma *smod-minus-eq* [*simp*]:
‹*k smod* − *l* = *k smod l*› **for** *k l* :: *int*
⟨*proof*⟩

lemma *smod-int-numeral-numeral* [*simp*]:
‹*numeral m smod numeral n* = *numeral m mod* (*numeral n* :: *int*)›
⟨*proof*⟩

end

98 State monad

theory *State-Monad*
imports *Monad-Syntax*
begin

datatype ('*s*, '*a*) *state* = *State* (*run-state*: '*s* ⇒ ('*a* × '*s*))

lemma *set-state-iff*: *x* ∈ *set-state m* ⟷ (∃ *s s'*. *run-state m s* = (*x*, *s'*))
⟨*proof*⟩

lemma *pred-stateI*[*intro*]:
assumes ⋀*a s s'*. *run-state m s* = (*a*, *s'*) ⟹ *P a*
shows *pred-state P m*
⟨*proof*⟩

lemma *pred-stateD*[*dest*]:
assumes *pred-state P m run-state m s* = (*a*, *s'*)
shows *P a*
⟨*proof*⟩

lemma *pred-state-run-state*: *pred-state P m* ⟹ *P* (*fst* (*run-state m s*))

⟨*proof*⟩

definition *state-io-rel* :: ('*s* ⇒ '*s* ⇒ *bool*) ⇒ ('*s*, '*a*) *state* ⇒ *bool* **where**
state-io-rel P m = (∀ *s*. *P s* (*snd* (*run-state m s*)))

lemma *state-io-relI*[*intro*]:
assumes ⋀*a s s'*. *run-state m s* = (*a*, *s'*) ⟹ *P s s'*
shows *state-io-rel P m*
⟨*proof*⟩

lemma *state-io-relD*[*dest*]:
assumes *state-io-rel P m run-state m s* = (*a*, *s'*)
shows *P s s'*
⟨*proof*⟩

lemma *state-io-rel-mono*[*mono*]: *P* ≤ *Q* ⟹ *state-io-rel P* ≤ *state-io-rel Q*
⟨*proof*⟩

lemma *state-ext*:
assumes ⋀*s*. *run-state m s* = *run-state n s*
shows *m* = *n*
⟨*proof*⟩

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*)
⟨*proof*⟩ **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''*))
⟨*proof*⟩ **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'*)
⟨*proof*⟩

adhoc-overloading *Monad-Syntax.bind* ⇌ *bind*

lemma *bind-left-identity*[*simp*]: *bind* (*return a*) *f* = *f a*
⟨*proof*⟩

lemma *bind-right-identity*[*simp*]: *bind m return = m*
⟨*proof*⟩

lemma *bind-assoc*[*simp*]: *bind* (*bind m f*) *g* = *bind m* (λ*x*. *bind* (*f x*) *g*)
⟨*proof*⟩

lemma *bind-predI*[*intro*]:
assumes *pred-state* (λ*x*. *pred-state P* (*f x*)) *m*
shows *pred-state P* (*bind m f*)
⟨*proof*⟩ **definition** *get* :: ('*s*, '*s*) *state* **where**
get = *State* (λ*s*. (*s*, *s*))

lemma *run-state-get*[*simp*]: *run-state get s* = (*s*, *s*)
⟨*proof*⟩ **definition** *set* :: '*s* ⇒ ('*s*, *unit*) *state* **where**
set s' = *State* (λ-. ((), *s'*))

lemma *run-state-set*[*simp*]: *run-state* (*set s'*) *s* = ((), *s'*)
⟨*proof*⟩

lemma *get-set*[*simp*]: *bind get set* = *return* ()
⟨*proof*⟩

lemma *set-set*[*simp*]: *bind* (*set s*) (λ-. *set s'*) = *set s'*
⟨*proof*⟩

lemma *get-bind-set*[*simp*]: *bind get* (λ*s*. *bind* (*set s*) (*f s*)) = *bind get* (λ*s*. *f s s*)
⟨*proof*⟩

lemma *get-const*[*simp*]: *bind get* (λ-. *m*) = *m*
⟨*proof*⟩

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*)
}
⟨*proof*⟩

lemma *traverse-comp*[*simp*]: *traverse-list* (*g* ∘ *f*) *xs* = *traverse-list g* (*map f xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *return-mono*[*simp*, *intro*]: *mono-state* (*return x*)
⟨*proof*⟩

lemma *get-mono*[*simp*, *intro*]: *mono-state get*
⟨*proof*⟩

lemma *put-mono*:
assumes ⋀*x*. *s'* ≥ *x*
shows *mono-state* (*set s'*)
⟨*proof*⟩

lemma *map-mono*[*intro*]: *mono-state m* ⟹ *mono-state* (*map-state f m*)
⟨*proof*⟩

lemma *map-strict-mono*[*intro*]: *strict-mono-state m* ⟹ *strict-mono-state* (*map-state f m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bind-strict-mono-strong2*:
assumes *strict-mono-state m*
assumes ⋀*x s s'*. *run-state m s* = (*x*, *s'*) ⟹ *mono-state* (*f x*)
shows *strict-mono-state* (*bind m f*)
⟨*proof*⟩

corollary *bind-strict-mono-strong*:
assumes *strict-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*)

⟨*proof*⟩ **definition** *update* :: (*'s* ⇒ *'s*) ⇒ (*'s*, *unit*) *state* **where**
update f = *bind get* (*set* ∘ *f*)

lemma *update-id*[*simp*]: *update* (λ*x*. *x*) = *return* ()
⟨*proof*⟩

lemma *update-comp*[*simp*]: *bind* (*update f*) (λ-. *update g*) = *update* (*g* ∘ *f*)
⟨*proof*⟩

lemma *set-update*[*simp*]: *bind* (*set s*) (λ-. *update f*) = *set* (*f s*)
⟨*proof*⟩

lemma *set-bind-update*[*simp*]: *bind* (*set s*) (λ-. *bind* (*update f*) *g*) = *bind* (*set* (*f s*)) *g*
⟨*proof*⟩

lemma *update-mono*:
assumes ⋀*x*. *x* ≤ *f x*
shows *mono-state* (*update f*)
⟨*proof*⟩

lemma *update-strict-mono*:
assumes ⋀*x*. *x* < *f x*
shows *strict-mono-state* (*update f*)
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *less-iff-sym-greater*:
‹*cmp b a = Less* ⟷ *cmp a b = Greater*›
⟨*proof*⟩

The equivalence part

lemma *sym*:
‹*cmp b a = Equiv* ⟷ *cmp a b = Equiv*›
⟨*proof*⟩

lemma *reflp*:
‹*reflp* (λ*a b*. *cmp a b = Equiv*)›
⟨*proof*⟩

lemma *symp*:
‹*symp* (λ*a b*. *cmp a b = Equiv*)›
⟨*proof*⟩

lemma *transp*:
‹*transp* (λ*a b*. *cmp a b = Equiv*)›
⟨*proof*⟩

lemma *equivp*:
‹*equivp* (λ*a b*. *cmp a b = Equiv*)›
⟨*proof*⟩

The strict part

lemma *irreflp-less*:
‹*irreflp* (λ*a b*. *cmp a b = Less*)›
⟨*proof*⟩

lemma *irreflp-greater*:
‹*irreflp* (λ*a b*. *cmp a b = Greater*)›
⟨*proof*⟩

lemma *asym-less*:
‹*cmp b a* ≠ *Less*› **if** ‹*cmp a b = Less*›
⟨*proof*⟩

lemma *asym-greater*:
‹*cmp b a* ≠ *Greater*› **if** ‹*cmp a b = Greater*›
⟨*proof*⟩

lemma *asymp-less*:
‹*asymp* (λ*a b*. *cmp a b = Less*)›

⟨*proof*⟩

lemma *asymp-greater*:
‹*asymp* (λ*a b*. *cmp a b* = *Greater*)›
⟨*proof*⟩

lemma *trans-equiv-less*:
‹*cmp a c* = *Less*› **if** ‹*cmp a b* = *Equiv*› **and** ‹*cmp b c* = *Less*›
⟨*proof*⟩

lemma *trans-less-equiv*:
‹*cmp a c* = *Less*› **if** ‹*cmp a b* = *Less*› **and** ‹*cmp b c* = *Equiv*›
⟨*proof*⟩

lemma *trans-equiv-greater*:
‹*cmp a c* = *Greater*› **if** ‹*cmp a b* = *Equiv*› **and** ‹*cmp b c* = *Greater*›
⟨*proof*⟩

lemma *trans-greater-equiv*:
‹*cmp a c* = *Greater*› **if** ‹*cmp a b* = *Greater*› **and** ‹*cmp b c* = *Equiv*›
⟨*proof*⟩

lemma *transp-less*:
‹*transp* (λ*a b*. *cmp a b* = *Less*)›
⟨*proof*⟩

lemma *transp-greater*:
‹*transp* (λ*a b*. *cmp a b* = *Greater*)›
⟨*proof*⟩

The reflexive part

lemma *reflp-not-less*:
‹*reflp* (λ*a b*. *cmp a b* ≠ *Less*)›
⟨*proof*⟩

lemma *reflp-not-greater*:
‹*reflp* (λ*a b*. *cmp a b* ≠ *Greater*)›
⟨*proof*⟩

lemma *quasisym-not-less*:
‹*cmp a b* = *Equiv*› **if** ‹*cmp a b* ≠ *Less*› **and** ‹*cmp b a* ≠ *Less*›
⟨*proof*⟩

lemma *quasisym-not-greater*:
‹*cmp a b* = *Equiv*› **if** ‹*cmp a b* ≠ *Greater*› **and** ‹*cmp b a* ≠ *Greater*›
⟨*proof*⟩

lemma *trans-not-less*:
‹*cmp a c* ≠ *Less*› **if** ‹*cmp a b* ≠ *Less*› ‹*cmp b c* ≠ *Less*›

⟨*proof*⟩

lemma *trans-not-greater*:
‹*cmp a c* ≠ *Greater*› **if** ‹*cmp a b* ≠ *Greater*› ‹*cmp b c* ≠ *Greater*›
⟨*proof*⟩

lemma *transp-not-less*:
‹*transp* (λ*a b*. *cmp a b* ≠ *Less*)›
⟨*proof*⟩

lemma *transp-not-greater*:
‹*transp* (λ*a b*. *cmp a b* ≠ *Greater*)›
⟨*proof*⟩

Substitution under equivalences

lemma *equiv-subst-left*:
‹*cmp z y* = *comp* ⟷ *cmp x y* = *comp*› **if** ‹*cmp z x* = *Equiv*› **for** *comp*
⟨*proof*⟩

lemma *equiv-subst-right*:
‹*cmp x z* = *comp* ⟷ *cmp x y* = *comp*› **if** ‹*cmp z y* = *Equiv*› **for** *comp*
⟨*proof*⟩

end

typedef *'a comparator* = ‹{*cmp* :: *'a* ⇒ *'a* ⇒ *comp*. *comparator cmp*}›
morphisms *compare Abs-comparator*
⟨*proof*⟩

setup-lifting *type-definition-comparator*

global-interpretation *compare*: *comparator* ‹*compare cmp*›
⟨*proof*⟩

lift-definition *flat* :: ‹*'a comparator*›
is ‹λ- -. *Equiv*› ⟨*proof*⟩

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*›
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *compare-default-eq-Less-iff* [*simp*]:

‹*compare default x y = Less* ⟷ *x < y*›
⟨*proof*⟩

lemma *compare-default-eq-Equiv-iff* [*simp*]:
‹*compare default x y = Equiv* ⟷ *x = y*›
⟨*proof*⟩

lemma *compare-default-eq-Greater-iff* [*simp*]:
‹*compare default x y = Greater* ⟷ *x > y*›
⟨*proof*⟩

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*› ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma [*code nbe*]:
‹*HOL.equal* (*cmp* :: *'a*::*enum comparator*) *cmp* ⟷ *True*›
⟨*proof*⟩

lemma [*code*]:
‹*HOL.equal cmp1 cmp2* ⟷ *Enum.enum-all* (*λx. compare cmp1 x = compare cmp2 x*)›
⟨*proof*⟩

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 ⟨*proof*⟩

end

99.2 Fundamental comparator combinators

lift-definition *reversed* :: ‹*'a comparator* ⇒ *'a comparator*›
  is ‹λ*cmp a b*. *cmp b a*›
⟨*proof*⟩

lemma *compare-reversed-apply* [*simp*]:
  ‹*compare* (*reversed cmp*) *x y* = *compare cmp y x*›
  ⟨*proof*⟩

lift-definition *key* :: ‹(*'b* ⇒ *'a*) ⇒ *'a comparator* ⇒ *'b comparator*›
  is ‹λ*f cmp a b*. *cmp* (*f a*) (*f b*)›
⟨*proof*⟩

lemma *compare-key-apply* [*simp*]:
  ‹*compare* (*key f cmp*) *x y* = *compare cmp* (*f x*) (*f y*)›
  ⟨*proof*⟩

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*⟩

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*)›
  ⟨*proof*⟩

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)›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

definition *comparator-nat* :: ‹*nat comparator*›
  where [*simp*, *code-abbrev*]: ‹*comparator-nat* = *default*›

lemma *compare-comparator-nat* [*code abstract*]:
  ‹*compare comparator-nat* = *raw-comparator-nat*›
  ⟨*proof*⟩

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*⟩

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*›
  ⟨*proof*⟩

lemma *sorted-Cons-imp-sorted*:
  ‹*sorted cmp xs*› **if** ‹*sorted cmp* (*x* # *xs*)›
  ⟨*proof*⟩

lemma *sorted-Cons-imp-not-less*:
  ‹*compare cmp y x* ≠ *Greater*› **if** ‹*sorted cmp* (*y* # *xs*)›
    and ‹*x* ∈ *set xs*›
  ⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *sorted-remove1*:
‹*sorted cmp* (*remove1 x xs*)› **if** ‹*sorted cmp xs*›
⟨*proof*⟩

lemma *sorted-stable-segment*:
‹*sorted cmp* (*stable-segment cmp x xs*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *length-insort* [*simp*]:
‹*length* (*insort cmp x xs*) = *Suc* (*length xs*)›
⟨*proof*⟩

lemma *sorted-insort*:
‹*sorted cmp* (*insort cmp x xs*)› **if** ‹*sorted cmp xs*›
⟨*proof*⟩

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*⟩

lemma *stable-insort-not-equiv*:
‹*stable-segment cmp y* (*insort cmp x xs*) = *stable-segment cmp y xs*›
if ‹*compare cmp y x* ≠ *Equiv*›
⟨*proof*⟩

lemma *remove1-insort-same-eq* [*simp*]:
‹*remove1 x* (*insort cmp x xs*) = *xs*›
⟨*proof*⟩

lemma *insort-eq-ConsI*:
‹*insort cmp x xs* = *x* # *xs*›
if ‹*sorted cmp xs*› ‹⋀*y*. *y* ∈ *set xs* ⟹ *compare cmp x y* ≠ *Greater*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

lemma *mset-sort* [*simp*]:
‹*mset* (*sort cmp xs*) = *mset xs*›
⟨*proof*⟩

lemma *length-sort* [*simp*]:
‹*length* (*sort cmp xs*) = *length xs*›
⟨*proof*⟩

lemma *sorted-sort* [*simp*]:
‹*sorted cmp* (*sort cmp xs*)›
⟨*proof*⟩

lemma *stable-sort*:
‹*stable-segment cmp x* (*sort cmp xs*) = *stable-segment cmp x xs*›
⟨*proof*⟩

lemma *sort-remove1-eq* [*simp*]:
‹*sort cmp* (*remove1 x xs*) = *remove1 x* (*sort cmp xs*)›
⟨*proof*⟩

lemma *set-insort* [*simp*]:
‹*set* (*insort cmp x xs*) = *insert x* (*set xs*)›
⟨*proof*⟩

lemma *set-sort* [*simp*]:
‹*set* (*sort cmp xs*) = *set xs*›
⟨*proof*⟩

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*⟩

lemma *filter-insort*:
‹*filter P* (*insort cmp x xs*) = *insort cmp x* (*filter P xs*)›
if ‹*sorted cmp xs*› **and** ‹*P x*›
⟨*proof*⟩

lemma *filter-insort-triv*:
‹*filter P* (*insort cmp x xs*) = *filter P xs*›
if ‹¬ *P x*›
⟨*proof*⟩

lemma *filter-sort*:
‹*filter P* (*sort cmp xs*) = *sort cmp* (*filter P xs*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*))›
  ⟨*proof*⟩

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*⟩

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*›
  ⟨*proof*⟩

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))›
⟨*proof*⟩ **termination** ⟨*proof*⟩

lemma *mset-merge*:
‹*mset (merge xs ys) = mset xs + mset ys*›
⟨*proof*⟩

lemma *merge-eq-Cons-imp*:
‹*xs ≠ [] ∧ z = hd xs ∨ ys ≠ [] ∧ z = hd ys*›
if ‹*merge xs ys = z # zs*›
⟨*proof*⟩

lemma *filter-merge*:
‹*filter P (merge xs ys) = merge (filter P xs) (filter P ys)*›
if ‹*sorted cmp xs*› **and** ‹*sorted cmp ys*›
⟨*proof*⟩

lemma *sorted-merge*:
‹*sorted cmp (merge xs ys)*› **if** ‹*sorted cmp xs*› **and** ‹*sorted cmp ys*›
⟨*proof*⟩

lemma *merge-eq-appendI*:
‹*merge xs ys = xs @ ys*›
if ‹*⋀x y. x ∈ set xs ⟹ y ∈ set ys ⟹ compare cmp x y ≠ Greater*›
⟨*proof*⟩

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›
⟨*proof*⟩

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*⟩

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*⟩

end

101.3 Lexicographic products

lemma *sorted-prod-lex-imp-sorted-fst*:
‹*sorted (key fst cmp1) ps*› **if** ‹*sorted (prod-lex cmp1 cmp2) ps*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *T-snd-0*[*simp*]: *T-snd x = 0*
⟨*proof*⟩

time-fun *the*

103.1 List

time-fun *hd*

lemma *T-hd*[*simp*]: *xs ≠ [] ⟹ T-hd xs = 0*
⟨*proof*⟩

time-fun *tl*

lemma *T-tl*[*simp*]: *T-tl xs = 0*
⟨*proof*⟩

declare *T-tl.simps*[*simp del*]

time-fun (@)

lemma *T-append*[*simp*]: *T-append xs ys = length xs + 1*
⟨*proof*⟩

class *T-size* =
fixes *T-size* :: *'a ⇒ nat*

instantiation *list* :: (-) *T-size*
begin

time-fun *length*

instance ⟨*proof*⟩

end

abbreviation *T-length* :: *'a list* ⇒ *nat* **where**
T-length ≡ *T-size*

lemma *T-length*: *T-length xs* = *length xs* + *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *T-map*: *T-map T-f xs* = (∑ *x*←*xs*. *T-f x*) + *length xs* + *1*
⟨*proof*⟩

lemmas [*simp del*] = *T-map-simps*

lemma *T-map-bound*:
∀ *x* ∈ *set xs*. *T-P x* ≤ *k* ⟹ *T-map T-P xs* ≤ *k* ∗ *length xs* + *length xs* + *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *T-filter*: *T-filter T-P xs* = (∑ *x*←*xs*. *T-P x*) + *length xs* + *1*
⟨*proof*⟩

lemma *T-filter-eq-T-map*: *T-filter T-f xs* = *T-map T-f xs*
⟨*proof*⟩

lemma *T-filter-bound*:
∀ *x* ∈ *set xs*. *T-P x* ≤ *k* ⟹ *T-filter T-P xs* ≤ *k* ∗ *length xs* + *length xs* + *1*
⟨*proof*⟩

time-fun *nth*

lemma *T-nth*: *n* < *length xs* ⟹ *T-nth xs n* = *n* + *1*
⟨*proof*⟩

lemmas [*simp del*] = *T-nth.simps*

time-fun *take*
time-fun *drop*

lemma *T-take*: *T-take n xs* = *min n* (*length xs*) + *1*
⟨*proof*⟩

lemma *T-drop*: *T-drop n xs* = *min n* (*length xs*) + *1*
⟨*proof*⟩

time-fun *zip*

lemma *T-zip*: *length xs* = *length ys* ⟹ *T-zip xs ys* = *length ys* + *1*
⟨*proof*⟩

time-fun *rev*

lemma *T-rev*: *T-rev xs* ≤ (*length xs* + *1*)^2
⟨*proof*⟩

fun *itrev* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
itrev [] *ys* = *ys* |
itrev (*x*#*xs*) *ys* = *itrev xs* (*x* # *ys*)

lemma *itrev*: *itrev xs ys* = *rev xs* @ *ys*
⟨*proof*⟩

lemma *itrev-Nil*: *itrev xs* [] = *rev xs*
⟨*proof*⟩

time-fun *itrev*

lemma *T-itrev*: *T-itrev xs ys* = *length xs* + *1*
⟨*proof*⟩

time-fun *list-update*

lemma *T-list-update*[*simp*]: *i* < *length xs* ⟹ *T-list-update xs i x* = *i* + *1*
⟨*proof*⟩

time-fun *last*

lemma *T-last*[*simp*]: *as* ≠ [] ⟹ *T-last as* = *length as*
⟨*proof*⟩

end

104 A table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtrancl-path* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$
  for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where
  base: *rtrancl-path r x* [] *x*
| *step*: $r\ x\ y \Longrightarrow$ *rtrancl-path r y ys z* $\Longrightarrow$ *rtrancl-path r x* $(y \# ys)\ z$

lemma *rtranclp-eq-rtrancl-path*: $r^{**}\ x\ y \longleftrightarrow (\exists xs.$ *rtrancl-path r x xs y*$)$
⟨*proof*⟩

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* $(xs\ @\ ys)\ z$ ⟨*proof*⟩

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*
  ⟨*proof*⟩

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* $xs' \subseteq$ *set xs*
  ⟨*proof*⟩

inductive *rtrancl-tab* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
  for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where
  base: *rtrancl-tab r xs x x*
| *step*: $x \notin$ *set xs* $\Longrightarrow$ *r x y* $\Longrightarrow$ *rtrancl-tab r* $(x \# xs)\ y\ z \Longrightarrow$ *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*: $(\{x\} \cup$ *set xs*$) \cap$ *set ys* $= \{\}$
  shows *rtrancl-tab r ys x y*
  ⟨*proof*⟩

lemma *rtrancl-tab-imp-rtrancl-path*:
  assumes *tab*: *rtrancl-tab r ys x y*
  obtains *xs* **where** *rtrancl-path r x xs y*
  ⟨*proof*⟩

lemma *rtranclp-eq-rtrancl-tab-nil*: $r^{**}\ x\ y \longleftrightarrow$ *rtrancl-tab* $r\ []\ x\ y$
⟨*proof*⟩

declare *rtranclp-eq-rtrancl-tab-nil* [*code*]
declare *rtranclp-eq-rtrancl-tab-nil* [*THEN iffD2*, *code-pred-intro*]

code-pred *rtranclp*
⟨*proof*⟩

lemma *rtrancl-path-Range*: ⟦ *rtrancl-path* $R\ x\ xs\ y$; $z \in$ *set* xs ⟧ ⟹ *Rangep* $R\ z$
⟨*proof*⟩

lemma *rtrancl-path-Range-end*: ⟦ *rtrancl-path* $R\ x\ xs\ y$; $xs \neq []$ ⟧ ⟹ *Rangep* $R\ y$
⟨*proof*⟩

lemma *rtrancl-path-nth*:
⟦ *rtrancl-path* $R\ x\ xs\ y$; $i <$ *length* xs ⟧ ⟹ $R\ ((x \# xs)\ !\ i)\ (xs\ !\ i)$
⟨*proof*⟩

lemma *rtrancl-path-last*: ⟦ *rtrancl-path* $R\ x\ xs\ y$; $xs \neq []$ ⟧ ⟹ *last* $xs = y$
⟨*proof*⟩

lemma *rtrancl-path-mono*:
⟦ *rtrancl-path* $R\ x\ p\ y$; $\bigwedge x\ y.\ R\ x\ y \Longrightarrow S\ x\ y$ ⟧ ⟹ *rtrancl-path* $S\ x\ p\ y$
⟨*proof*⟩

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 ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eq-Leaf-map-tree*[*simp*]: *Leaf* = *map-tree f t* ⟷ *t* = *Leaf*
⟨*proof*⟩

105.2 *size*

lemma *size1-size*: *size1 t* = *size t* + *1*
⟨*proof*⟩

lemma *size1-ge0*[*simp*]: *0* < *size1 t*
⟨*proof*⟩

lemma *eq-size-0*[*simp*]: *size t* = *0* ⟷ *t* = *Leaf*
⟨*proof*⟩

lemma *eq-0-size*[*simp*]: *0* = *size t* ⟷ *t* = *Leaf*

⟨*proof*⟩

lemma *neq-Leaf-iff*: $(t \neq \langle\rangle) = (\exists\, l\ a\ r.\ t = \langle l,\ a,\ r\rangle)$
⟨*proof*⟩

lemma *size-map-tree*[*simp*]: *size* (*map-tree f t*) = *size t*
⟨*proof*⟩

lemma *size1-map-tree*[*simp*]: *size1* (*map-tree f t*) = *size1 t*
⟨*proof*⟩

105.3 *set-tree*

lemma *eq-set-tree-empty*[*simp*]: *set-tree t* = {} ⟷ *t* = *Leaf*
⟨*proof*⟩

lemma *eq-empty-set-tree*[*simp*]: {} = *set-tree t* ⟷ *t* = *Leaf*
⟨*proof*⟩

lemma *finite-set-tree*[*simp*]: *finite*(*set-tree t*)
⟨*proof*⟩

lemma *finite-trees-height-le*:
assumes *finite L*
shows *finite* {*t* :: *'a tree. height t* ≤ *n* ∧ *set-tree t* ⊆ *L*}
⟨*proof*⟩

105.4 *subtrees*

lemma *neq-subtrees-empty*[*simp*]: *subtrees t* ≠ {}
⟨*proof*⟩

lemma *neq-empty-subtrees*[*simp*]: {} ≠ *subtrees t*
⟨*proof*⟩

lemma *size-subtrees*: *s* ∈ *subtrees t* ⟹ *size s* ≤ *size t*
⟨*proof*⟩

lemma *set-treeE*: *a* ∈ *set-tree t* ⟹ ∃ *l r*. ⟨*l*, *a*, *r*⟩ ∈ *subtrees t*
⟨*proof*⟩

lemma *Node-notin-subtrees-if*[*simp*]: *a* ∉ *set-tree t* ⟹ *Node l a r* ∉ *subtrees t*
⟨*proof*⟩

lemma *in-set-tree-if*: ⟨*l*, *a*, *r*⟩ ∈ *subtrees t* ⟹ *a* ∈ *set-tree t*
⟨*proof*⟩

105.5 *height* **and** *min-height*

lemma *eq-height-0*[*simp*]: *height t* = *0* ⟷ *t* = *Leaf*

⟨*proof*⟩

lemma *eq-0-height*[*simp*]: *0 = height t ⟷ t = Leaf*
⟨*proof*⟩

lemma *height-map-tree*[*simp*]: *height* (*map-tree f t*) = *height t*
⟨*proof*⟩

lemma *height-le-size-tree*: *height t* ≤ *size* (*t*::'*a tree*)
⟨*proof*⟩

lemma *size1-height*: *size1 t* ≤ *2* ^ *height* (*t*::'*a tree*)
⟨*proof*⟩

corollary *size-height*: *size t* ≤ *2* ^ *height* (*t*::'*a tree*) − *1*
⟨*proof*⟩

lemma *height-subtrees*: *s* ∈ *subtrees t* ⟹ *height s* ≤ *height t*
⟨*proof*⟩

lemma *min-height-le-height*: *min-height t* ≤ *height t*
⟨*proof*⟩

lemma *min-height-map-tree*[*simp*]: *min-height* (*map-tree f t*) = *min-height t*
⟨*proof*⟩

lemma *min-height-size1*: *2* ^ *min-height t* ≤ *size1 t*
⟨*proof*⟩

105.6 *complete*

lemma *complete-iff-height*: *complete t* ⟷ (*min-height t* = *height t*)
⟨*proof*⟩

lemma *size1-if-complete*: *complete t* ⟹ *size1 t* = *2* ^ *height t*
⟨*proof*⟩

lemma *size-if-complete*: *complete t* ⟹ *size t* = *2* ^ *height t* − *1*
⟨*proof*⟩

lemma *size1-height-if-incomplete*:
¬ *complete t* ⟹ *size1 t* < *2* ^ *height t*
⟨*proof*⟩

lemma *complete-iff-min-height*: *complete t* ⟷ (*height t* = *min-height t*)
⟨*proof*⟩

lemma *min-height-size1-if-incomplete*:

$\neg$ *complete t* $\Longrightarrow$ *2* $\widehat{}$ *min-height t* $<$ *size1 t*
⟨*proof*⟩

lemma *complete-if-size1-height*: *size1 t* $=$ *2* $\widehat{}$ *height t* $\Longrightarrow$ *complete t*
⟨*proof*⟩

lemma *complete-if-size1-min-height*: *size1 t* $=$ *2* $\widehat{}$ *min-height t* $\Longrightarrow$ *complete t*
⟨*proof*⟩

lemma *complete-iff-size1*: *complete t* $\longleftrightarrow$ *size1 t* $=$ *2* $\widehat{}$ *height t*
⟨*proof*⟩

105.7 *acomplete*

lemma *acomplete-subtreeL*: *acomplete* (*Node l x r*) $\Longrightarrow$ *acomplete l*
⟨*proof*⟩

lemma *acomplete-subtreeR*: *acomplete* (*Node l x r*) $\Longrightarrow$ *acomplete r*
⟨*proof*⟩

lemma *acomplete-subtrees*: ⟦ *acomplete t*; *s* $\in$ *subtrees t* ⟧ $\Longrightarrow$ *acomplete s*
⟨*proof*⟩

Balanced trees have optimal height:

lemma *acomplete-optimal*:
fixes *t* :: *'a tree* **and** *t'* :: *'b tree*
assumes *acomplete t size t* $\leq$ *size t'* **shows** *height t* $\leq$ *height t'*
⟨*proof*⟩

105.8 *wbalanced*

lemma *wbalanced-subtrees*: ⟦ *wbalanced t*; *s* $\in$ *subtrees t* ⟧ $\Longrightarrow$ *wbalanced s*
⟨*proof*⟩

105.9 *ipl*

The internal path length of a tree:

lemma *ipl-if-complete-int*:
complete t $\Longrightarrow$ *int*(*ipl t*) $=$ (*int*(*height t*) $-$ *2*) $*$ *2*$\widehat{}$(*height t*) $+$ *2*
⟨*proof*⟩

105.10 List of entries

lemma *eq-inorder-Nil*[*simp*]: *inorder t* $=$ [] $\longleftrightarrow$ *t* $=$ *Leaf*
⟨*proof*⟩

lemmas *eq-Nil-inorder*[*simp*] $=$ *eq-inorder-Nil*[*THEN eq-iff-swap*]

lemma *set-inorder*[*simp*]: *set* (*inorder t*) $=$ *set-tree t*

⟨*proof*⟩

lemma *preorder-eq-Nil-iff*[*simp*]: (*preorder t* = []) = (*t* = ⟨⟩)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemmas *Cons-eq-preorder-iff* = *preorder-eq-Cons-iff*[*THEN eq-iff-swap*]

lemma *set-preorder*[*simp*]: *set* (*preorder t*) = *set-tree t*
⟨*proof*⟩

lemma *set-postorder*[*simp*]: *set* (*postorder t*) = *set-tree t*
⟨*proof*⟩

lemma *length-preorder*[*simp*]: *length* (*preorder t*) = *size t*
⟨*proof*⟩

lemma *length-inorder*[*simp*]: *length* (*inorder t*) = *size t*
⟨*proof*⟩

lemma *length-postorder*[*simp*]: *length* (*postorder t*) = *size t*
⟨*proof*⟩

lemma *preorder-map*: *preorder* (*map-tree f t*) = *map f* (*preorder t*)
⟨*proof*⟩

lemma *inorder-map*: *inorder* (*map-tree f t*) = *map f* (*inorder t*)
⟨*proof*⟩

lemma *postorder-map*: *postorder* (*map-tree f t*) = *map f* (*postorder t*)
⟨*proof*⟩

lemma *inorder2-inorder*: *inorder2 t xs* = *inorder t* @ *xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bst-wrt-le-if-bst*: *bst t* ⟹ *bst-wrt* (≤) *t*
⟨*proof*⟩

lemma *bst-wrt-le-iff-sorted*: *bst-wrt* (≤) *t* ⟷ *sorted* (*inorder t*)

⟨*proof*⟩

lemma *bst-iff-sorted-wrt-less*: *bst t* ⟷ *sorted-wrt* (<) (*inorder t*)
⟨*proof*⟩

105.12 *heap*

105.13 *mirror*

lemma *mirror-Leaf*[*simp*]: *mirror t* = ⟨⟩ ⟷ *t* = ⟨⟩
⟨*proof*⟩

lemma *Leaf-mirror*[*simp*]: ⟨⟩ = *mirror t* ⟷ *t* = ⟨⟩
⟨*proof*⟩

lemma *size-mirror*[*simp*]: *size*(*mirror t*) = *size t*
⟨*proof*⟩

lemma *size1-mirror*[*simp*]: *size1*(*mirror t*) = *size1 t*
⟨*proof*⟩

lemma *height-mirror*[*simp*]: *height*(*mirror t*) = *height t*
⟨*proof*⟩

lemma *min-height-mirror* [*simp*]: *min-height* (*mirror t*) = *min-height t*
⟨*proof*⟩

lemma *ipl-mirror* [*simp*]: *ipl* (*mirror t*) = *ipl t*
⟨*proof*⟩

lemma *inorder-mirror*: *inorder*(*mirror t*) = *rev*(*inorder t*)
⟨*proof*⟩

lemma *map-mirror*: *map-tree f* (*mirror t*) = *mirror* (*map-tree f t*)
⟨*proof*⟩

lemma *mirror-mirror*[*simp*]: *mirror*(*mirror t*) = *t*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-mset-tree*[*simp*]: *set-mset* (*mset-tree t*) = *set-tree t*
⟨*proof*⟩

lemma *size-mset-tree*[*simp*]: *size*(*mset-tree t*) = *size t*
⟨*proof*⟩

lemma *mset-map-tree*: *mset-tree* (*map-tree f t*) = *image-mset f* (*mset-tree t*)
⟨*proof*⟩

lemma *mset-iff-set-tree*: *x* ∈# *mset-tree t* ⟷ *x* ∈ *set-tree t*
⟨*proof*⟩

lemma *mset-preorder*[*simp*]: *mset* (*preorder t*) = *mset-tree t*
⟨*proof*⟩

lemma *mset-inorder*[*simp*]: *mset* (*inorder t*) = *mset-tree t*
⟨*proof*⟩

lemma *map-mirror*: *mset-tree* (*mirror t*) = *mset-tree t*
⟨*proof*⟩

lemma *in-subtrees-mset-iff*[*simp*]: *s* ∈# *subtrees-mset t* ⟷ *s* ∈ *subtrees t*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *min-height-size1-log*: *min-height t ≤ log 2 (size1 t)*
⟨*proof*⟩

lemma *size1-log-if-complete*: *complete t ⟹ height t = log 2 (size1 t)*
⟨*proof*⟩

lemma *min-height-size1-log-if-incomplete*:
¬ complete t ⟹ min-height t < log 2 (size1 t)
⟨*proof*⟩

lemma *min-height-acomplete*: **assumes** *acomplete t*
shows *min-height t = nat(floor(log 2 (size1 t)))*
⟨*proof*⟩

lemma *height-acomplete*: **assumes** *acomplete t*
shows *height t = nat(ceiling(log 2 (size1 t)))*
⟨*proof*⟩

lemma *acomplete-Node-if-wbal1*:
assumes *acomplete l acomplete r size l = size r + 1*
shows *acomplete ⟨l, x, r⟩*
⟨*proof*⟩

lemma *acomplete-sym*: *acomplete ⟨l, x, r⟩ ⟹ acomplete ⟨r, y, l⟩*
⟨*proof*⟩

lemma *acomplete-Node-if-wbal2*:
assumes *acomplete l acomplete r abs(int(size l) − int(size r)) ≤ 1*
shows *acomplete ⟨l, x, r⟩*
⟨*proof*⟩

lemma *acomplete-if-wbalanced*: *wbalanced t ⟹ acomplete t*
⟨*proof*⟩

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*
⟨*proof*⟩

setup-lifting *type-definition-commute*

lemma *apply-commute-commute*: *apply-commute f x y = apply-commute f y x*
⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *equivp-eq-upair*: *equivp eq-upair*
⟨*proof*⟩

quotient-type '*a uprod* = '*a* × '*a* / *eq-upair* ⟨*proof*⟩

lift-definition *Upair* :: '*a* ⇒ '*a* ⇒ '*a uprod* **is** *Pair* **parametric** *Pair-transfer*[*of A A* **for** *A*] ⟨*proof*⟩

lemma *uprod-exhaust* [*case-names Upair*, *cases type*: *uprod*]:
obtains *a b* **where** *x* = *Upair a b*
⟨*proof*⟩

lemma *Upair-inject* [*simp*]: *Upair a b* = *Upair c d* ⟷ *a* = *c* ∧ *b* = *d* ∨ *a* = *d* ∧ *b* = *c*
⟨*proof*⟩

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*] ⟨*proof*⟩

lemma *case-uprod-simps* [*simp*, *code*]: *case-uprod f* (*Upair x y*) = *apply-commute f x y*
⟨*proof*⟩

lemma *uprod-split*: *P* (*case-uprod f x*) ⟷ (∀ *a b*. *x* = *Upair a b* ⟶ *P* (*apply-commute f a b*))
⟨*proof*⟩

lemma *uprod-split-asm*: *P* (*case-uprod f x*) ⟷ ¬ (∃ *a b*. *x* = *Upair a b* ∧ ¬ *P*

(*apply-commute f a b*))
⟨*proof*⟩

lift-definition *not-equal* :: ('*a*, *bool*) *commute* **is** (≠) ⟨*proof*⟩

lemma *apply-not-equal* [*simp*]: *apply-commute not-equal x y* ⟷ *x* ≠ *y*
⟨*proof*⟩

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*
⟨*proof*⟩

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*})
⟨*proof*⟩

lift-definition *set-uprod* :: '*a uprod* ⇒ '*a set* **is** λ(*a*, *b*). {*a*, *b*}
parametric *set-uprod-parametric'* ⟨*proof*⟩

lemma *set-uprod-simps* [*simp*, *code*]: *set-uprod* (*Upair x y*) = {*x*, *y*}
⟨*proof*⟩

lemma *finite-set-uprod* [*simp*]: *finite* (*set-uprod x*)
⟨*proof*⟩ **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*)
⟨*proof*⟩

lift-definition *map-uprod* :: ('*a* ⇒ '*b*) ⇒ '*a uprod* ⇒ '*b uprod* **is** λ*f*. *map-prod f f*
parametric *map-uprod-parametric'* ⟨*proof*⟩

lemma *map-uprod-simps* [*simp*, *code*]: *map-uprod f* (*Upair x y*) = *Upair* (*f x*) (*f y*)
⟨*proof*⟩ **lemma** *rel-uprod-transfer'*:
((*A* ===> *B* ===> (=)) ===> *rel-prod A A* ===> *rel-prod B B* ===> (=))
(λ*R* (*a*, *b*) (*c*, *d*). *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c*) (λ*R* (*a*, *b*) (*c*, *d*). *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c*)
⟨*proof*⟩

lift-definition *rel-uprod* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a uprod* ⇒ '*b uprod* ⇒ *bool*
is λ*R* (*a*, *b*) (*c*, *d*). *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c* **parametric** *rel-uprod-transfer'*
⟨*proof*⟩

lemma *rel-uprod-simps* [*simp*, *code*]:

rel-uprod R (*Upair a b*) (*Upair c d*) ⟷ *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c*
⟨*proof*⟩

lemma *Upair-parametric* [*transfer-rule*]: (*A* ===> *A* ===> *rel-uprod A*) *Upair Upair*
⟨*proof*⟩

lemma *case-uprod-parametric* [*transfer-rule*]:
(*rel-commute A B* ===> *rel-uprod A* ===> *B*) *case-uprod case-uprod*
⟨*proof*⟩

end

bnf *uprod*: *'a uprod*
map: *map-uprod*
sets: *set-uprod*
bd: *natLeq*
rel: *rel-uprod*
⟨*proof*⟩

lemma *pred-uprod-code* [*simp*, *code*]: *pred-uprod P* (*Upair x y*) ⟷ *P x* ∧ *P y*
⟨*proof*⟩

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*
⟨*proof*⟩

instance ⟨*proof*⟩
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*})
⟨*proof*⟩

context begin
private lift-definition *upair-inv* :: *'a uprod* ⇒ *'a*
is λ(*x*, *y*). *if x* = *y then x else undefined* ⟨*proof*⟩

lemma *finite-UNIV-prod* [*simp*]:
finite (*UNIV* :: *'a uprod set*) ⟷ *finite* (*UNIV* :: *'a set*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

lift-definition *one-word* :: ‹*'a word*›
is *1* ⟨*proof*⟩

lift-definition *plus-word* :: ‹*'a word* ⇒ *'a word* ⇒ *'a word*›
is ‹(+)›
⟨*proof*⟩

lift-definition *minus-word* :: ‹*'a word* ⇒ *'a word* ⇒ *'a word*›
is ‹(−)›
⟨*proof*⟩

lift-definition *uminus-word* :: ‹*'a word* ⇒ *'a word*›
is *uminus*
⟨*proof*⟩

lift-definition *times-word* :: ‹*'a word* ⇒ *'a word* ⇒ *'a word*›
is ‹(∗)›
⟨*proof*⟩

instance
⟨*proof*⟩

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*) (^) (^)›
⟨*proof*⟩

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) *of-bool of-bool*›
⟨*proof*⟩

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) *numeral numeral*›
⟨*proof*⟩

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) *int of-nat*›
⟨*proof*⟩

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) (λ*k*. *k*) *of-int*›
⟨*proof*⟩

lemma [*transfer-rule*]:

‹(*pcr-word* ===> (⟷)) *even* ((*dvd*) *2* :: *'a*::*len word* ⇒ *bool*)›
⟨*proof*⟩

end

lemma *exp-eq-zero-iff* [*simp*]:
‹*2* ^ *n* = (*0* :: *'a*::*len word*) ⟷ *n* ≥ *LENGTH*(*'a*)›
⟨*proof*⟩

lemma *word-exp-length-eq-0* [*simp*]:
‹(*2* :: *'a*::*len word*) ^ *LENGTH*(*'a*) = *0*›
⟨*proof*⟩

108.2.3 Basic tool setup

⟨*ML*⟩

108.2.4 Basic code generation setup

context
begin

qualified lift-definition *the-int* :: ‹*'a*::*len word* ⇒ *int*›
is ‹*take-bit LENGTH*(*'a*)› ⟨*proof*⟩

end

lemma [*code abstype*]:
‹*Word.Word* (*Word.the-int w*) = *w*›
⟨*proof*⟩

lemma *Word-eq-word-of-int* [*code-post*, *simp*]:
‹*Word.Word* = *of-int*›
⟨*proof*⟩

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 ‹λ*k l*. *take-bit LENGTH*(*'a*) *k* = *take-bit LENGTH*(*'a*) *l*›
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma [*code*]:
‹*HOL.equal v w* ⟷ *HOL.equal* (*Word.the-int v*) (*Word.the-int w*)›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int 0* = *0*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int 1* = *1*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›

⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *word-of-nat-eq-0-iff*:
‹*word-of-nat n* = (*0* :: *'a*::*len word*) ⟷ *2* ^ *LENGTH*(*'a*) *dvd n*›
⟨*proof*⟩

lemma *word-of-int-eq-0-iff*:
‹*word-of-int k* = (*0* :: *'a*::*len word*) ⟷ *2* ^ *LENGTH*(*'a*) *dvd k*›
⟨*proof*⟩

context *semiring-1*
begin

lift-definition *unsigned* :: ‹*'b*::*len word* ⇒ *'a*›
is ‹*of-nat* ∘ *nat* ∘ *take-bit LENGTH*(*'b*)›
⟨*proof*⟩

lemma *unsigned-0* [*simp*]:
‹*unsigned 0* = *0*›
⟨*proof*⟩

lemma *unsigned-1* [*simp*]:
‹*unsigned 1* = *1*›
⟨*proof*⟩

lemma *unsigned-numeral* [*simp*]:
‹*unsigned* (*numeral n* :: *'b*::*len word*) = *of-nat* (*take-bit LENGTH*(*'b*) (*numeral n*))›
⟨*proof*⟩

lemma *unsigned-neg-numeral* [*simp*]:
‹*unsigned* (− *numeral n* :: *'b*::*len word*) = *of-nat* (*nat* (*take-bit LENGTH*(*'b*) (− *numeral n*)))›
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *unsigned-of-nat*:
‹*unsigned* (*word-of-nat n* :: *'b*::*len word*) = *of-nat* (*take-bit LENGTH*(*'b*) *n*)›
⟨*proof*⟩

lemma *unsigned-of-int*:
‹*unsigned* (*word-of-int k* :: *'b*::*len word*) = *of-nat* (*nat* (*take-bit LENGTH*('*b*) *k*))›
⟨*proof*⟩

end

context *semiring-char-0*
begin

lemma *unsigned-word-eqI*:
‹*v* = *w*› **if** ‹*unsigned v* = *unsigned w*›
⟨*proof*⟩

lemma *word-eq-iff-unsigned*:
‹*v* = *w* ⟷ *unsigned v* = *unsigned w*›
⟨*proof*⟩

lemma *inj-unsigned* [*simp*]:
‹*inj unsigned*›
⟨*proof*⟩

lemma *unsigned-eq-0-iff*:
‹*unsigned w* = *0* ⟷ *w* = *0*›
⟨*proof*⟩

end

context *ring-1*
begin

lift-definition *signed* :: ‹*'b*::*len word* ⇒ *'a*›
is ‹*of-int* ∘ *signed-take-bit* (*LENGTH*('*b*) − *Suc 0*)›
⟨*proof*⟩

lemma *signed-0* [*simp*]:
‹*signed 0* = *0*›
⟨*proof*⟩

lemma *signed-1* [*simp*]:
‹*signed* (*1* :: *'b*::*len word*) = (*if LENGTH*('*b*) = *1 then* − *1 else 1*)›
⟨*proof*⟩

lemma *signed-minus-1* [*simp*]:
‹*signed* (− *1* :: *'b*::*len word*) = − *1*›
⟨*proof*⟩

lemma *signed-numeral* [*simp*]:
‹*signed* (*numeral n* :: *'b*::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('*b*) − *1*)
(*numeral n*))›

⟨*proof*⟩

lemma *signed-neg-numeral* [*simp*]:
‹*signed* (− *numeral n* :: 'b::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('b) − *1*) (− *numeral n*))›
⟨*proof*⟩

lemma *signed-of-nat*:
‹*signed* (*word-of-nat n* :: 'b::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('b) − *Suc 0*) (*int n*))›
⟨*proof*⟩

lemma *signed-of-int*:
‹*signed* (*word-of-int n* :: 'b::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('b) − *Suc 0*) *n*)›
⟨*proof*⟩

end

context *ring-char-0*
begin

lemma *signed-word-eqI*:
‹*v* = *w*› **if** ‹*signed v* = *signed w*›
⟨*proof*⟩

lemma *word-eq-iff-signed*:
‹*v* = *w* ⟷ *signed v* = *signed w*›
⟨*proof*⟩

lemma *inj-signed* [*simp*]:
‹*inj signed*›
⟨*proof*⟩

lemma *signed-eq-0-iff*:
‹*signed w* = *0* ⟷ *w* = *0*›
⟨*proof*⟩

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*)›
  ⟨*proof*⟩

lemma [*transfer-rule*]:
  ‹(*pcr-word* ===> (=)) (*take-bit LENGTH*(*'a*)) (*uint* :: *'a*::*len word* ⇒ *int*)›
  ⟨*proof*⟩

lemma [*transfer-rule*]:
  ‹(*pcr-word* ===> (=)) (*signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*)) (*sint* :: *'a*::*len word* ⇒ *int*)›
  ⟨*proof*⟩

lemma [*transfer-rule*]:
  ‹(*pcr-word* ===> *pcr-word*) (*take-bit LENGTH*(*'a*)) (*ucast* :: *'a*::*len word* ⇒ *'b*::*len word*)›
⟨*proof*⟩

lemma [*transfer-rule*]:
  ‹(*pcr-word* ===> *pcr-word*) (*signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*)) (*scast* :: *'a*::*len word* ⇒ *'b*::*len word*)›
⟨*proof*⟩

end

lemma *of-nat-unat* [*simp*]:
  ‹*of-nat* (*unat w*) = *unsigned w*›
  ⟨*proof*⟩

lemma *of-int-uint* [*simp*]:
  ‹*of-int* (*uint w*) = *unsigned w*›
  ⟨*proof*⟩

lemma *of-int-sint* [*simp*]:
  ‹*of-int* (*sint a*) = *signed a*›
  ⟨*proof*⟩

lemma *nat-uint-eq* [*simp*]:

‹*nat* (*uint w*) = *unat w*›
⟨*proof*⟩

lemma *nat-of-natural-unsigned-eq* [*simp*]:
‹*nat-of-natural* (*unsigned w*) = *unat w*›
⟨*proof*⟩

lemma *int-of-integer-unsigned-eq* [*simp*]:
‹*int-of-integer* (*unsigned w*) = *uint w*›
⟨*proof*⟩

lemma *int-of-integer-signed-eq* [*simp*]:
‹*int-of-integer* (*signed w*) = *sint w*›
⟨*proof*⟩

lemma *sgn-uint-eq* [*simp*]:
‹*sgn* (*uint w*) = *of-bool* (*w* ≠ *0*)›
⟨*proof*⟩

Aliasses only for code generation

context
begin

qualified lift-definition *of-int* :: ‹*int* ⇒ *'a*::*len word*›
is ‹*take-bit LENGTH*(*'a*)› ⟨*proof*⟩ **lift-definition** *of-nat* :: ‹*nat* ⇒ *'a*::*len word*›
is ‹*int* ∘ *take-bit LENGTH*(*'a*)› ⟨*proof*⟩ **lift-definition** *the-nat* :: ‹*'a*::*len word* ⇒ *nat*›
is ‹*nat* ∘ *take-bit LENGTH*(*'a*)› ⟨*proof*⟩ **lift-definition** *the-signed-int* :: ‹*'a*::*len word* ⇒ *int*›
is ‹*signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*)› ⟨*proof*⟩ **lift-definition** *cast* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
is ‹*take-bit LENGTH*(*'a*)› ⟨*proof*⟩ **lift-definition** *signed-cast* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
is ‹*signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*)› ⟨*proof*⟩

end

lemma [*code-abbrev*, *simp*]:
‹*Word.the-int* = *uint*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int* (*Word.of-int k* :: *'a*::*len word*) = *take-bit LENGTH*(*'a*) *k*›
⟨*proof*⟩

lemma [*code-abbrev*, *simp*]:
‹*Word.of-int* = *word-of-int*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int* (*Word.of-nat n* :: *'a*::*len word*) = *take-bit LENGTH*('*a*) (*int n*)›
⟨*proof*⟩

lemma [*code-abbrev*, *simp*]:
‹*Word.of-nat* = *word-of-nat*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-nat w* = *nat* (*Word.the-int w*)›
⟨*proof*⟩

lemma [*code-abbrev*, *simp*]:
‹*Word.the-nat* = *unat*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*code-abbrev*, *simp*]:
‹*Word.the-signed-int* = *sint*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int* (*Word.cast w* :: *'b*::*len word*) = *take-bit LENGTH*('*b*) (*Word.the-int w*)›
for *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma [*code-abbrev*, *simp*]:
‹*Word.cast* = *ucast*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*code-abbrev*, *simp*]:
‹*Word.signed-cast* = *scast*›
⟨*proof*⟩

lemma [*code*]:
‹*unsigned w* = *of-nat* (*nat* (*Word.the-int w*))›

⟨*proof*⟩

lemma [*code*]:
‹*signed w = of-int* (*Word.the-signed-int w*)›
⟨*proof*⟩

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*⟩

108.3.1 Basic ordering

instantiation *word* :: (*len*) *linorder*
begin

lift-definition *less-eq-word* :: '*a word* ⇒ '*a word* ⇒ *bool*
is λ*a b*. *take-bit LENGTH*('*a*) *a* ≤ *take-bit LENGTH*('*a*) *b*
⟨*proof*⟩

lift-definition *less-word* :: '*a word* ⇒ '*a word* ⇒ *bool*
is λ*a b*. *take-bit LENGTH*('*a*) *a* < *take-bit LENGTH*('*a*) *b*
⟨*proof*⟩

instance
⟨*proof*⟩

end

interpretation *word-order*: *ordering-top* ‹(≤)› ‹(<)› ‹− *1* :: '*a*::*len word*›
⟨*proof*⟩

interpretation *word-coorder*: *ordering-top* ‹(≥)› ‹(>)› ‹*0* :: '*a*::*len word*›
⟨*proof*⟩

lemma *word-of-nat-less-eq-iff*:
‹*word-of-nat m* ≤ (*word-of-nat n* :: '*a*::*len word*) ⟷ *take-bit LENGTH*('*a*) *m* ≤ *take-bit LENGTH*('*a*) *n*›
⟨*proof*⟩

lemma *word-of-int-less-eq-iff*:
‹*word-of-int k* ≤ (*word-of-int l* :: '*a*::*len word*) ⟷ *take-bit LENGTH*('*a*) *k* ≤ *take-bit LENGTH*('*a*) *l*›
⟨*proof*⟩

lemma *word-of-nat-less-iff*:

‹*word-of-nat m* < (*word-of-nat n* :: *'a*::*len word*) ⟷ *take-bit LENGTH('a) m* < *take-bit LENGTH('a) n*›
⟨*proof*⟩

lemma *word-of-int-less-iff*:
‹*word-of-int k* < (*word-of-int l* :: *'a*::*len word*) ⟷ *take-bit LENGTH('a) k* < *take-bit LENGTH('a) l*›
⟨*proof*⟩

lemma *word-le-def* [*code*]:
a ≤ *b* ⟷ *uint a* ≤ *uint b*
⟨*proof*⟩

lemma *word-less-def* [*code*]:
a < *b* ⟷ *uint a* < *uint b*
⟨*proof*⟩

lemma *word-greater-zero-iff*:
‹*a* > *0* ⟷ *a* ≠ *0*› **for** *a* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *of-nat-word-less-eq-iff*:
‹*of-nat m* ≤ (*of-nat n* :: *'a*::*len word*) ⟷ *take-bit LENGTH('a) m* ≤ *take-bit LENGTH('a) n*›
⟨*proof*⟩

lemma *of-nat-word-less-iff*:
‹*of-nat m* < (*of-nat n* :: *'a*::*len word*) ⟷ *take-bit LENGTH('a) m* < *take-bit LENGTH('a) n*›
⟨*proof*⟩

lemma *of-int-word-less-eq-iff*:
‹*of-int k* ≤ (*of-int l* :: *'a*::*len word*) ⟷ *take-bit LENGTH('a) k* ≤ *take-bit LENGTH('a) l*›
⟨*proof*⟩

lemma *of-int-word-less-iff*:
‹*of-int k* < (*of-int l* :: *'a*::*len word*) ⟷ *take-bit LENGTH('a) k* < *take-bit LENGTH('a) l*›
⟨*proof*⟩

instantiation *word* :: (*len*) *order-bot*
begin

lift-definition *bot-word* :: ‹*'a word*›
is *0* ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *bot-word-eq*:
‹*bot* = (*0* :: *'a*::*len word*)›
⟨*proof*⟩

instantiation *word* :: (*len*) *order-top*
begin

lift-definition *top-word* :: ‹*'a word*›
is ‹*− 1*› ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *top-word-eq*:
‹*top* = (*− 1* :: *'a*::*len word*)›
⟨*proof*⟩

108.4 Enumeration

lemma *inj-on-word-of-nat*:
‹*inj-on* (*word-of-nat* :: *nat* ⇒ *'a*::*len word*) {*0*..<*2* ^ *LENGTH*(*'a*)}›
⟨*proof*⟩

lemma *UNIV-word-eq-word-of-nat*:
‹(*UNIV* :: *'a*::*len word set*) = *word-of-nat* ' {*0*..<*2* ^ *LENGTH*(*'a*)}› (**is** ‹- = *?A*›)
⟨*proof*⟩

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
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *unat-div-distrib*:
‹*unat* (*v div w*) = *unat v div unat w*›
⟨*proof*⟩

lemma *unat-mod-distrib*:
‹*unat* (*v mod w*) = *unat v mod unat w*›
⟨*proof*⟩

instance *word* :: (*len*) *semiring-parity*
⟨*proof*⟩

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*⟩

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*⟩

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*›
⟨*proof*⟩

instantiation *word* :: (*len*) *semiring-bits*
begin

lift-definition *bit-word* :: ‹'*a word* ⇒ *nat* ⇒ *bool*›
is ‹λ*k n*. *n* < *LENGTH*('*a*) ∧ *bit k n*›
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *bit-word-eqI*:
‹*a* = *b*› **if** ‹⋀*n*. *n* < *LENGTH*('*a*) ⟹ *bit a n* ⟷ *bit b n*›
for *a b* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *bit-imp-le-length*: ‹*n* < *LENGTH*('*a*)› **if** ‹*bit w n*› **for** *w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *not-bit-length* [*simp*]:
‹¬ *bit w LENGTH*('*a*)› **for** *w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *finite-bit-word* [*simp*]:
‹*finite* {*n*. *bit w n*}›
for *w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *bit-numeral-word-iff* [*simp*]:
‹*bit* (*numeral w* :: '*a*::*len word*) *n*
⟷ *n* < *LENGTH*('*a*) ∧ *bit* (*numeral w* :: *int*) *n*›
⟨*proof*⟩

lemma *bit-neg-numeral-word-iff* [*simp*]:
‹*bit* (− *numeral w* :: '*a*::*len word*) *n*
⟷ *n* < *LENGTH*('*a*) ∧ *bit* (− *numeral w* :: *int*) *n*›
⟨*proof*⟩

instantiation *word* :: (*len*) *ring-bit-operations*
begin

lift-definition *not-word* :: ‹*'a word* ⇒ *'a word*›
  is *not*
  ⟨*proof*⟩

lift-definition *and-word* :: ‹*'a word* ⇒ *'a word* ⇒ *'a word*›
  is ‹*and*›
  ⟨*proof*⟩

lift-definition *or-word* :: ‹*'a word* ⇒ *'a word* ⇒ *'a word*›
  is *or*
  ⟨*proof*⟩

lift-definition *xor-word* :: ‹*'a word* ⇒ *'a word* ⇒ *'a word*›
  is *xor*
  ⟨*proof*⟩

lift-definition *mask-word* :: ‹*nat* ⇒ *'a word*›
  is *mask*
  ⟨*proof*⟩

lift-definition *set-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
  is *set-bit*
  ⟨*proof*⟩

lift-definition *unset-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
  is *unset-bit*
  ⟨*proof*⟩

lift-definition *flip-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
  is *flip-bit*
  ⟨*proof*⟩

lift-definition *push-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
  is *push-bit*
⟨*proof*⟩

lift-definition *drop-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
  is ‹λ*n*. *drop-bit n* ∘ *take-bit LENGTH*(*'a*)›
  ⟨*proof*⟩

lift-definition *take-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
  is ‹λ*n*. *take-bit* (*min LENGTH*(*'a*) *n*)›
  ⟨*proof*⟩

context
  includes *bit-operations-syntax*
begin

instance ⟨*proof*⟩

end

end

lemma [*code*]:
‹*push-bit n w = w * 2 ^ n*› **for** *w* :: ‹*'a::len word*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int (drop-bit n w) = drop-bit n (Word.the-int w)*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma [*code-abbrev*]:
‹*push-bit n 1 = (2 :: 'a::len word) ^ n*›
⟨*proof*⟩

context
includes *bit-operations-syntax*
begin

lemma [*code*]:
‹*NOT w = Word.of-int (NOT (Word.the-int w))*›
for *w* :: ‹*'a::len word*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int (v AND w) = Word.the-int v AND Word.the-int w*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int (v OR w) = Word.the-int v OR Word.the-int w*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int (v XOR w) = Word.the-int v XOR Word.the-int w*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int (mask n :: 'a::len word) = mask (min LENGTH('a) n)*›
⟨*proof*⟩

lemma [*code*]:

‹*set-bit n w = w OR push-bit n 1*› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma [*code*]:
‹*unset-bit n w = w AND NOT* (*push-bit n 1*)› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma [*code*]:
‹*flip-bit n w = w XOR push-bit n 1*› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

context
includes *lifting-syntax*
begin

lemma *set-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*) *set-bit set-bit*›
⟨*proof*⟩

lemma *unset-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*) *unset-bit unset-bit*›
⟨*proof*⟩

lemma *flip-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*) *flip-bit flip-bit*›
⟨*proof*⟩

lemma *signed-take-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*)
(λ*n k*. *signed-take-bit n* (*take-bit LENGTH*('*a*::*len*) *k*))
(*signed-take-bit* :: *nat* ⇒ '*a word* ⇒ '*a word*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

end

lemma *possible-bit-word*[*simp*]:
‹*possible-bit TYPE*((*'a* :: *len*) *word*) *m* ⟷ *m* < *LENGTH*(*'a*)›
⟨*proof*⟩

context *semiring-bit-operations*
begin

lemma *unsigned-minus-1-eq-mask*:
‹*unsigned* (− *1* :: *'b*::*len word*) = *mask LENGTH*(*'b*)›
⟨*proof*⟩

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*⟩

lemma *unsigned-take-bit-eq*:
‹*unsigned* (*take-bit n w*) = *take-bit n* (*unsigned w*)›
for *w* :: ‹*'b*::*len word*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›

⟨*proof*⟩

lemma *unsigned-or-eq*:
‹*unsigned* (*v OR w*) = *unsigned v OR unsigned w*›
for *v w* :: ‹'*b*::*len word*›
⟨*proof*⟩

lemma *unsigned-xor-eq*:
‹*unsigned* (*v XOR w*) = *unsigned v XOR unsigned w*›
for *v w* :: ‹'*b*::*len word*›
⟨*proof*⟩

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*›
⟨*proof*⟩

end

end

context *linordered-euclidean-semiring*
begin

lemma *unsigned-greater-eq* [*simp*]:
‹*0* ≤ *unsigned w*› **for** *w* :: ‹'*b*::*len word*›
⟨*proof*⟩

lemma *unsigned-less* [*simp*]:
‹*unsigned w* < *2* ^ *LENGTH*('*b*)› **for** *w* :: ‹'*b*::*len word*›
⟨*proof*⟩

end

context *linordered-semidom*
begin

lemma *word-less-eq-iff-unsigned*:

$a \leq b \longleftrightarrow$ *unsigned* $a \leq$ *unsigned* b
⟨*proof*⟩

lemma *word-less-iff-unsigned*:
$a < b \longleftrightarrow$ *unsigned* $a <$ *unsigned* b
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *signed-and-eq*:
‹*signed* (*v* *AND* *w*) = *signed* *v* *AND* *signed* *w*›
for *v* *w* :: ‹'*b*::*len* *word*›
⟨*proof*⟩

lemma *signed-or-eq*:
‹*signed* (*v* *OR* *w*) = *signed* *v* *OR* *signed* *w*›
for *v* *w* :: ‹'*b*::*len* *word*›

⟨*proof*⟩

lemma *signed-xor-eq*:
‹*signed* (*v XOR w*) = *signed v XOR signed w*›
for *v w* :: ‹'*b*::*len word*›
⟨*proof*⟩

end

end

108.6.3 More

lemma *sint-greater-eq*:
‹− (2 ^ (*LENGTH*('*a*) − *Suc 0*)) ≤ *sint w*› **for** *w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *sint-less*:
‹*sint w* < 2 ^ (*LENGTH*('*a*) − *Suc 0*)› **for** *w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *uint-div-distrib*:
‹*uint* (*v div w*) = *uint v div uint w*›
⟨*proof*⟩

lemma *unat-drop-bit-eq*:
‹*unat* (*drop-bit n w*) = *drop-bit n* (*unat w*)›
⟨*proof*⟩

lemma *uint-mod-distrib*:
‹*uint* (*v mod w*) = *uint v mod uint w*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *signed-scast-eq*:
‹*signed* (*scast* w :: $'c$::*len word*) = *signed-take-bit* (*LENGTH*($'c$) − *Suc 0*) (*signed* w)›
for w :: ‹$'b$::*len word*›
⟨*proof*⟩

end

lemma *uint-nonnegative*: $0 \leq$ *uint* w
⟨*proof*⟩

lemma *uint-bounded*: *uint* $w <$ 2 ^ *LENGTH*($'a$)
for w :: $'a$::*len word*
⟨*proof*⟩

lemma *uint-idem*: *uint* w *mod* 2 ^ *LENGTH*($'a$) = *uint* w
for w :: $'a$::*len word*
⟨*proof*⟩

lemma *word-uint-eqI*: *uint* a = *uint* $b \Longrightarrow a = b$
⟨*proof*⟩

lemma *word-uint-eq-iff*: $a = b \longleftrightarrow$ *uint* a = *uint* b
⟨*proof*⟩

lemma *uint-word-of-int-eq*:
‹*uint* (*word-of-int* k :: $'a$::*len word*) = *take-bit LENGTH*($'a$) k›
⟨*proof*⟩

lemma *uint-word-of-int*: *uint* (*word-of-int* k :: $'a$::*len word*) = k *mod* 2 ^ *LENGTH*($'a$)
⟨*proof*⟩

lemma *word-of-int-uint*: *word-of-int* (*uint* w) = w
⟨*proof*⟩

lemma *word-div-def* [*code*]:
a *div* b = *word-of-int* (*uint* a *div uint* b)
⟨*proof*⟩

lemma *word-mod-def* [*code*]:
a *mod* b = *word-of-int* (*uint* a *mod uint* b)
⟨*proof*⟩

lemma *split-word-all*: (⋀x::$'a$::*len word*. *PROP* P x) ≡ (⋀x. *PROP* P (*word-of-int* x))
⟨*proof*⟩

lemma *sint-uint*:
‹*sint w = signed-take-bit (LENGTH('a) − Suc 0) (uint w)*›
for *w* :: ‹*'a::len word*›
⟨*proof*⟩

lemma *unat-eq-nat-uint*:
‹*unat w = nat (uint w)*›
⟨*proof*⟩

lemma *ucast-eq*:
‹*ucast w = word-of-int (uint w)*›
⟨*proof*⟩

lemma *scast-eq*:
‹*scast w = word-of-int (sint w)*›
⟨*proof*⟩

lemma *uint-0-eq*:
‹*uint 0 = 0*›
⟨*proof*⟩

lemma *uint-1-eq*:
‹*uint 1 = 1*›
⟨*proof*⟩

lemma *word-m1-wi*: *− 1 = word-of-int (− 1)*
⟨*proof*⟩

lemma *uint-0-iff*: *uint x = 0 ⟷ x = 0*
⟨*proof*⟩

lemma *unat-0-iff*: *unat x = 0 ⟷ x = 0*
⟨*proof*⟩

lemma *unat-0*: *unat 0 = 0*
⟨*proof*⟩

lemma *unat-gt-0*: *0 < unat x ⟷ x ≠ 0*
⟨*proof*⟩

lemma *ucast-0*: *ucast 0 = 0*
⟨*proof*⟩

lemma *sint-0*: *sint 0 = 0*
⟨*proof*⟩

lemma *scast-0*: *scast 0 = 0*
⟨*proof*⟩

lemma *sint-n1*: *sint* (− *1*) = − *1*
⟨*proof*⟩

lemma *scast-n1*: *scast* (− *1*) = − *1*
⟨*proof*⟩

lemma *uint-1*: *uint* (*1*::*'a*::*len word*) = *1*
⟨*proof*⟩

lemma *unat-1*: *unat* (*1*::*'a*::*len word*) = *1*
⟨*proof*⟩

lemma *ucast-1*: *ucast* (*1*::*'a*::*len word*) = *1*
⟨*proof*⟩

instantiation *word* :: (*len*) *size*
begin

lift-definition *size-word* :: ‹*'a word* ⇒ *nat*›
is ‹λ-. *LENGTH*(*'a*)› ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *word-size* [*code*]:
‹*size w* = *LENGTH*(*'a*)› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *word-size-gt-0* [*iff*]: *0* < *size w*
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemmas *lens-gt-0* = *word-size-gt-0 len-gt-0*

lemma *lens-not-0* [*iff*]:
‹*size w* ≠ *0*› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lift-definition *source-size* :: ‹(*'a*::*len word* ⇒ *'b*) ⇒ *nat*›
is ‹λ-. *LENGTH*(*'a*)› ⟨*proof*⟩

lift-definition *target-size* :: ‹(*'a* ⇒ *'b*::*len word*) ⇒ *nat*›
is ‹λ-. *LENGTH*(*'b*)› ⟨*proof*⟩

lift-definition *is-up* :: ‹(*'a*::*len word* ⇒ *'b*::*len word*) ⇒ *bool*›
is ‹λ-. *LENGTH*(*'a*) ≤ *LENGTH*(*'b*)› ⟨*proof*⟩

lift-definition *is-down* :: ‹(*'a*::*len word* ⇒ *'b*::*len word*) ⇒ *bool*›
is ‹λ-. *LENGTH*(*'a*) ≥ *LENGTH*(*'b*)› ⟨*proof*⟩

lemma *is-up-eq*:
‹*is-up f* ⟷ *source-size f* ≤ *target-size f*›
for *f* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
⟨*proof*⟩

lemma *is-down-eq*:
‹*is-down f* ⟷ *target-size f* ≤ *source-size f*›
for *f* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
⟨*proof*⟩

lift-definition *word-int-case* :: ‹(*int* ⇒ *'b*) ⇒ *'a*::*len word* ⇒ *'b*›
is ‹λ*f*. *f* ∘ *take-bit LENGTH*(*'a*)› ⟨*proof*⟩

lemma *word-int-case-eq-uint* [*code*]:
‹*word-int-case f w* = *f* (*uint w*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *mod-word-self* [*simp*]:
‹*w mod w* = *0*› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *div-word-less*:
‹*w div v* = *0*› **if** ‹*w* < *v*› **for** *w v* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *mod-word-less*:
‹*w mod v* = *w*› **if** ‹*w* < *v*› **for** *w v* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *div-word-one* [*simp*]:
‹*1 div w* = *of-bool* (*w* = *1*)› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *mod-word-one* [*simp*]:
‹*1 mod w* = *1* − *w* ∗ *of-bool* (*w* = *1*)› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *div-word-by-minus-1-eq* [*simp*]:
‹*w div − 1 = of-bool* (*w = − 1*)› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *mod-word-by-minus-1-eq* [*simp*]:
‹*w mod − 1 = w ∗ of-bool* (*w < − 1*)› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

Legacy theorems:

lemma *word-add-def* [*code*]:
a + b = word-of-int (*uint a + uint b*)
⟨*proof*⟩

lemma *word-sub-wi* [*code*]:
a − b = word-of-int (*uint a − uint b*)
⟨*proof*⟩

lemma *word-mult-def* [*code*]:
a ∗ b = word-of-int (*uint a ∗ uint b*)
⟨*proof*⟩

lemma *word-minus-def* [*code*]:
− a = word-of-int (*− uint a*)
⟨*proof*⟩

lemma *word-0-wi*:
0 = word-of-int 0
⟨*proof*⟩

lemma *word-1-wi*:
1 = word-of-int 1
⟨*proof*⟩

lift-definition *word-succ* :: *'a*::*len word ⇒ 'a word* **is** *λx. x + 1*
⟨*proof*⟩

lift-definition *word-pred* :: *'a*::*len word ⇒ 'a word* **is** *λx. x − 1*
⟨*proof*⟩

lemma *word-succ-alt* [*code*]:
word-succ a = word-of-int (*uint a + 1*)
⟨*proof*⟩

lemma *word-pred-alt* [*code*]:
word-pred a = word-of-int (*uint a − 1*)
⟨*proof*⟩

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-of-int a + word-of-int b = word-of-int (a + b)*
and *wi-hom-sub*: *word-of-int a − word-of-int b = word-of-int (a − b)*
and *wi-hom-mult*: *word-of-int a * word-of-int b = word-of-int (a * b)*
and *wi-hom-neg*: *− word-of-int a = word-of-int (− a)*
and *wi-hom-succ*: *word-succ (word-of-int a) = word-of-int (a + 1)*
and *wi-hom-pred*: *word-pred (word-of-int a) = word-of-int (a − 1)*
⟨*proof*⟩

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 ⟷ a = 0 ∨ a = 2 ^ (LENGTH('a) − Suc 0)*›
for *a* :: ‹*'a*::*len word*›
⟨*proof*⟩

108.8 Ordering

instance *word* :: (*len*) *wellorder*
⟨*proof*⟩

lemma *word-m1-ge* [*simp*]:
word-pred 0 ≥ y
⟨*proof*⟩

lemma *word-less-alt*:
a < b ⟷ uint a < uint b
⟨*proof*⟩

lemma *word-zero-le* [*simp*]:
0 ≤ y **for** *y* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-n1-ge* [*simp*]:
y ≤ − 1 **for** *y* :: *'a*::*len word*
⟨*proof*⟩

lemmas *word-not-simps* [*simp*] =
word-zero-le [*THEN leD*] *word-m1-ge* [*THEN leD*] *word-n1-ge* [*THEN leD*]

lemma *word-gt-0*:

$0 < y \longleftrightarrow 0 \neq y$
for y :: $'a$::*len word*
⟨*proof*⟩

lemma *word-gt-0-no* [*simp*]:
‹(0 :: $'a$::*len word*) < *numeral y* ⟷ (0 :: $'a$::*len word*) ≠ *numeral y*›
⟨*proof*⟩

lemma *word-le-nat-alt*:
$a \leq b \longleftrightarrow$ *unat a* ≤ *unat b*
⟨*proof*⟩

lemma *word-less-nat-alt*:
$a < b \longleftrightarrow$ *unat a* < *unat b*
⟨*proof*⟩

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*($'a$))
⟨*proof*⟩

lemma *wi-le*:
(*word-of-int n* ≤ (*word-of-int m* :: $'a$::*len word*)) =
(*n mod 2* ^ *LENGTH*($'a$) ≤ *m mod 2* ^ *LENGTH*($'a$))
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

notation
word-sle (‹$'(\leq s')$›) **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*›

⟨*proof*⟩

lemma *word-sless-alt* [*code*]:
a <s b ⟷ sint a < sint b
⟨*proof*⟩

lemma *signed-ordering*: ‹*ordering word-sle word-sless*›
⟨*proof*⟩

lemma *signed-linorder*: ‹*class.linorder word-sle word-sless*›
⟨*proof*⟩

interpretation *signed*: *linorder word-sle word-sless*
⟨*proof*⟩

lemma *word-sless-eq*:
‹*x <s y ⟷ x <=s y ∧ x ≠ y*›
⟨*proof*⟩

lemma *minus-1-sless-0* [*simp*]:
‹*− 1 <s 0*›
⟨*proof*⟩

lemma *not-0-sless-minus-1* [*simp*]:
‹*¬ 0 <s − 1*›
⟨*proof*⟩

lemma *minus-1-sless-eq-0* [*simp*]:
‹*− 1 ≤s 0*›
⟨*proof*⟩

lemma *not-0-sless-eq-minus-1* [*simp*]:
‹*¬ 0 ≤s − 1*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *take-bit-length-eq* [*simp*]:
‹*take-bit LENGTH('a) w = w*› **for** *w* :: ‹*'a::len word*›
⟨*proof*⟩

lemma *bit-word-of-int-iff*:
‹*bit* (*word-of-int k* :: *'a*::*len word*) *n* ⟷ *n* < *LENGTH*(*'a*) ∧ *bit k n*›
⟨*proof*⟩

lemma *bit-uint-iff*:
‹*bit* (*uint w*) *n* ⟷ *n* < *LENGTH*(*'a*) ∧ *bit w n*›
for *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

lemma *bit-sint-iff*:
‹*bit* (*sint w*) *n* ⟷ *n* ≥ *LENGTH*(*'a*) ∧ *bit w* (*LENGTH*(*'a*) − *1*) ∨ *bit w n*›
for *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *bit-word-iff-drop-bit-and* [*code*]:
‹*bit a n* ⟷ *drop-bit n a AND 1* = *1*› **for** *a* :: ‹*'a*::*len word*›
⟨*proof*⟩

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*)
⟨*proof*⟩

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*›
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *signed-drop-bit-of-0* [*simp*]:
‹*signed-drop-bit n 0* = *0*›
⟨*proof*⟩

lemma *signed-drop-bit-of-minus-1* [*simp*]:
‹*signed-drop-bit n* (− *1*) = − *1*›
⟨*proof*⟩

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*⟩

lemma *signed-drop-bit-0* [*simp*]:
‹*signed-drop-bit 0 w* = *w*›

⟨*proof*⟩

lemma *sint-signed-drop-bit-eq*:
‹*sint* (*signed-drop-bit n w*) = *drop-bit n* (*sint w*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *unset-bit-eq-idem-iff*:
‹*unset-bit n w* = *w* ⟷ *bit w n* ⟶ *n* ≥ *LENGTH*('*a*)›
for *w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *flip-bit-eq-idem-iff*:
‹*flip-bit n w* = *w* ⟷ *n* ≥ *LENGTH*('*a*)›
for *w* :: ‹'*a*::*len word*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *word-rotl-eq-word-rotr* [*code*]:
‹*word-rotl n* = (*word-rotr* (*LENGTH*('*a*) − *n mod LENGTH*('*a*)) :: '*a*::*len word*
⇒ '*a word*)›
⟨*proof*⟩

lemma *word-roti-eq-word-rotr-word-rotl* [*code*]:
‹*word-roti i w* =

(*if* $i \geq 0$ *then word-rotr* (*nat* i) w *else word-rotl* (*nat* $(-i)$) w)›
⟨*proof*⟩

lemma *bit-word-rotr-iff* [*bit-simps*]:
‹*bit* (*word-rotr* m w) $n \longleftrightarrow$
$n < LENGTH('a) \wedge$ *bit* w $((n + m)$ *mod* $LENGTH('a))$›
for w :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *bit-word-rotl-iff* [*bit-simps*]:
‹*bit* (*word-rotl* m w) $n \longleftrightarrow$
$n < LENGTH('a) \wedge$ *bit* w $((n + (LENGTH('a) - m \bmod LENGTH('a))) \bmod LENGTH('a))$›
for w :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *bit-word-roti-iff* [*bit-simps*]:
‹*bit* (*word-roti* k w) $n \longleftrightarrow$
$n < LENGTH('a) \wedge$ *bit* w (*nat* $((int\ n + k)$ *mod int* $LENGTH('a)))$›
for w :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *uint-word-rotr-eq*:
‹*uint* (*word-rotr* n w) = *concat-bit* $(LENGTH('a) - n \bmod LENGTH('a))$
(*drop-bit* $(n \bmod LENGTH('a))$ (*uint* w))
(*uint* (*take-bit* $(n \bmod LENGTH('a))$ w))›
for w :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma [*code*]:
‹*Word.the-int* (*word-rotr* n w) = *concat-bit* $(LENGTH('a) - n \bmod LENGTH('a))$
(*drop-bit* $(n \bmod LENGTH('a))$ (*Word.the-int* w))
(*Word.the-int* (*take-bit* $(n \bmod LENGTH('a))$ w))›
for w :: ‹'*a*::*len word*›
⟨*proof*⟩

108.13 Split and cat operations

lift-definition *word-cat* :: ‹'*a*::*len word* ⇒ '*b*::*len word* ⇒ '*c*::*len word*›
is ‹$\lambda k\ l.$ *concat-bit* $LENGTH('b)$ l (*take-bit* $LENGTH('a)$ k)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *sint-sbintrunc'*: *sint* (*word-of-int bin* :: *'a word*) = *signed-take-bit* (*LENGTH*(*'a*::*len*) − *1*) *bin*
⟨*proof*⟩

lemma *uint-sint*: *uint w* = *take-bit LENGTH*(*'a*) (*sint w*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *bintr-uint*: *LENGTH*(*'a*) ≤ *n* ⟹ *take-bit n* (*uint w*) = *uint w*
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *wi-bintr*:
LENGTH(*'a*::*len*) ≤ *n* ⟹
word-of-int (*take-bit n w*) = (*word-of-int w* :: *'a word*)
⟨*proof*⟩

lemma *word-numeral-alt*: *numeral b* = *word-of-int* (*numeral b*)
⟨*proof*⟩

declare *word-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-neg-numeral-alt*: − *numeral b* = *word-of-int* (− *numeral b*)
⟨*proof*⟩

declare *word-neg-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *uint-bintrunc* [*simp*]:
uint (*numeral bin* :: *'a word*) = *take-bit LENGTH('a::len)* (*numeral bin*)
⟨*proof*⟩

lemma *uint-bintrunc-neg* [*simp*]:
uint (− *numeral bin* :: *'a word*) = *take-bit LENGTH('a::len)* (− *numeral bin*)
⟨*proof*⟩

lemma *sint-sbintrunc* [*simp*]:
sint (*numeral bin* :: *'a word*) = *signed-take-bit* (*LENGTH('a::len)* − *1*) (*numeral bin*)
⟨*proof*⟩

lemma *sint-sbintrunc-neg* [*simp*]:
sint (− *numeral bin* :: *'a word*) = *signed-take-bit* (*LENGTH('a::len)* − *1*) (− *numeral bin*)
⟨*proof*⟩

lemma *unat-bintrunc* [*simp*]:
unat (*numeral bin* :: *'a::len word*) = *take-bit LENGTH('a)* (*numeral bin*)
⟨*proof*⟩

lemma *unat-bintrunc-neg* [*simp*]:
unat (− *numeral bin* :: *'a::len word*) = *nat* (*take-bit LENGTH('a)* (− *numeral bin*))
⟨*proof*⟩

lemma *size-0-eq*: *size w* = *0* ⟹ *v* = *w*
for *v w* :: *'a::len word*
⟨*proof*⟩

lemma *uint-ge-0* [*iff*]: *0* ≤ *uint x*
⟨*proof*⟩

lemma *uint-lt2p* [*iff*]: *uint x* < *2* ^ *LENGTH('a)*
for *x* :: *'a::len word*
⟨*proof*⟩

lemma *sint-ge*: − (*2* ^ (*LENGTH('a)* − *1*)) ≤ *sint x*
for *x* :: *'a::len word*
⟨*proof*⟩

lemma *sint-lt*: *sint x* < *2* ^ (*LENGTH('a)* − *1*)
for *x* :: *'a::len word*
⟨*proof*⟩

lemma *uint-m2p-neg*: *uint x* − *2* ^ *LENGTH('a)* < *0*
for *x* :: *'a::len word*

⟨*proof*⟩

lemma *uint-m2p-not-non-neg*: ¬ *0* ≤ *uint x* − *2* ^ *LENGTH*('*a*)
for *x* :: '*a*::*len word*
⟨*proof*⟩

lemma *lt2p-lem*: *LENGTH*('*a*) ≤ *n* ⟹ *uint w* < *2* ^ *n*
for *w* :: '*a*::*len word*
⟨*proof*⟩

lemma *uint-le-0-iff* [*simp*]: *uint x* ≤ *0* ⟷ *uint x* = *0*
⟨*proof*⟩

lemma *uint-nat*: *uint w* = *int* (*unat w*)
⟨*proof*⟩

lemma *uint-numeral*: *uint* (*numeral b* :: '*a*::*len word*) = *numeral b mod 2* ^ *LENGTH*('*a*)
⟨*proof*⟩

lemma *uint-neg-numeral*: *uint* (− *numeral b* :: '*a*::*len word*) = − *numeral b mod 2* ^ *LENGTH*('*a*)
⟨*proof*⟩

lemma *unat-numeral*: *unat* (*numeral b* :: '*a*::*len word*) = *numeral b mod 2* ^ *LENGTH*('*a*)
⟨*proof*⟩

lemma *sint-numeral*:
sint (*numeral b* :: '*a*::*len word*) =
(*numeral b* + *2* ^ (*LENGTH*('*a*) − *1*)) *mod 2* ^ *LENGTH*('*a*) − *2* ^ (*LENGTH*('*a*) − *1*)
⟨*proof*⟩

lemma *word-of-int-0* [*simp*, *code-post*]: *word-of-int 0* = *0*
⟨*proof*⟩

lemma *word-of-int-1* [*simp*, *code-post*]: *word-of-int 1* = *1*
⟨*proof*⟩

lemma *word-of-int-neg-1* [*simp*]: *word-of-int* (− *1*) = − *1*
⟨*proof*⟩

lemma *word-of-int-numeral* [*simp*] : (*word-of-int* (*numeral bin*) :: '*a*::*len word*) = *numeral bin*
⟨*proof*⟩

lemma *word-int-case-wi*:
word-int-case f (*word-of-int i* :: '*b word*) = *f* (*i mod 2* ^ *LENGTH*('*b*::*len*))

⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *uint-range-size*: *0* ≤ *uint w* ∧ *uint w* < *2* ^ *size w*
⟨*proof*⟩

lemma *sint-range-size*: − (*2* ^ (*size w* − *Suc 0*)) ≤ *sint w* ∧ *sint w* < *2* ^ (*size w* − *Suc 0*)
⟨*proof*⟩

lemma *sint-above-size*: *2* ^ (*size w* − *1*) ≤ *x* ⟹ *sint w* < *x*
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *sint-below-size*: *x* ≤ − (*2* ^ (*size w* − *1*)) ⟹ *x* ≤ *sint w*
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-unat-eq-iff*:
‹*v* = *w* ⟷ *unat v* = *unat w*›
for *v w* :: ‹*'a*::*len word*›
⟨*proof*⟩

108.15 Testing bits

lemma *bin-nth-uint-imp*: *bit* (*uint w*) *n* ⟹ *n* < *LENGTH*(*'a*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *bin-nth-sint*:
LENGTH(*'a*) ≤ *n* ⟹
bit (*sint w*) *n* = *bit* (*sint w*) (*LENGTH*(*'a*) − *1*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *num-of-bintr'*:
take-bit (*LENGTH*(*'a*::*len*)) (*numeral a* :: *int*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: *'a word*)

⟨*proof*⟩

lemma *num-of-sbintr'*:
signed-take-bit (*LENGTH*('*a*::*len*) − *1*) (*numeral a* :: *int*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: '*a word*)
⟨*proof*⟩

lemma *num-abs-bintr*:
(*numeral x* :: '*a word*) =
word-of-int (*take-bit* (*LENGTH*('*a*::*len*)) (*numeral x*))
⟨*proof*⟩

lemma *num-abs-sbintr*:
(*numeral x* :: '*a word*) =
word-of-int (*signed-take-bit* (*LENGTH*('*a*::*len*) − *1*) (*numeral x*))
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *ucast-id* [*simp*]: *ucast w* = *w*
⟨*proof*⟩

lemma *scast-id* [*simp*]: *scast w* = *w*
⟨*proof*⟩

lemma *ucast-mask-eq*:
‹*ucast* (*mask n* :: '*b word*) = *mask* (*min LENGTH*('*b*::*len*) *n*)›
⟨*proof*⟩
lemma *ucast-bintr* [*simp*]:
ucast (*numeral w* :: '*a*::*len word*) =
word-of-int (*take-bit* (*LENGTH*('*a*)) (*numeral w*))
⟨*proof*⟩

lemma *scast-sbintr* [*simp*]:
scast (*numeral w* ::'*a*::*len word*) =
word-of-int (*signed-take-bit* (*LENGTH*('*a*) − *Suc 0*) (*numeral w*))
⟨*proof*⟩

lemma *source-size*: *source-size* (*c*::'*a*::*len word* ⇒ -) = *LENGTH*('*a*)
⟨*proof*⟩

lemma *target-size*: *target-size* (*c*::- ⇒ '*b*::*len word*) = *LENGTH*('*b*)
⟨*proof*⟩

lemma *is-down*: *is-down c* $\longleftrightarrow$ *LENGTH*('*b*) $\leq$ *LENGTH*('*a*)
for *c* :: '*a*::*len word* $\Rightarrow$ '*b*::*len word*
⟨*proof*⟩

lemma *is-up*: *is-up c* $\longleftrightarrow$ *LENGTH*('*a*) $\leq$ *LENGTH*('*b*)
for *c* :: '*a*::*len word* $\Rightarrow$ '*b*::*len word*
⟨*proof*⟩

lemma *is-up-down*:
‹*is-up c* $\longleftrightarrow$ *is-down d*›
for *c* :: ‹'*a*::*len word* $\Rightarrow$ '*b*::*len word*›
and *d* :: ‹'*b*::*len word* $\Rightarrow$ '*a*::*len word*›
⟨*proof*⟩

context
fixes *dummy-types* :: ‹'*a*::*len* $\times$ '*b*::*len*›
begin

private abbreviation (*input*) *UCAST* :: ‹'*a*::*len word* $\Rightarrow$ '*b*::*len word*›
where ‹*UCAST* == *ucast*›

private abbreviation (*input*) *SCAST* :: ‹'*a*::*len word* $\Rightarrow$ '*b*::*len word*›
where ‹*SCAST* == *scast*›

lemma *down-cast-same*:
‹*UCAST* = *scast*› **if** ‹*is-down UCAST*›
⟨*proof*⟩

lemma *sint-up-scast*:
‹*sint* (*SCAST w*) = *sint w*› **if** ‹*is-up SCAST*›
⟨*proof*⟩

lemma *uint-up-ucast*:
‹*uint* (*UCAST w*) = *uint w*› **if** ‹*is-up UCAST*›
⟨*proof*⟩

lemma *ucast-up-ucast*:
‹*ucast* (*UCAST w*) = *ucast w*› **if** ‹*is-up UCAST*›
⟨*proof*⟩

lemma *ucast-up-ucast-id*:
‹*ucast* (*UCAST w*) = *w*› **if** ‹*is-up UCAST*›
⟨*proof*⟩

lemma *scast-up-scast*:
‹*scast* (*SCAST w*) = *scast w*› **if** ‹*is-up SCAST*›
⟨*proof*⟩

lemma *scast-up-scast-id*:
‹*scast* (*SCAST w*) = *w*› **if** ‹*is-up SCAST*›
⟨*proof*⟩

lemma *isduu*:
‹*is-up UCAST*› **if** ‹*is-down d*›
for *d* :: ‹'*b word* ⇒ '*a word*›
⟨*proof*⟩

lemma *isdus*:
‹*is-up SCAST*› **if** ‹*is-down d*›
for *d* :: ‹'*b word* ⇒ '*a word*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *up-scast-surj*:
‹*surj* (*scast* :: '*b word* ⇒ '*a word*)› **if** ‹*is-up SCAST*›
⟨*proof*⟩

lemma *down-ucast-inj*:
‹*inj-on UCAST A*› **if** ‹*is-down* (*ucast* :: '*b word* ⇒ '*a word*)›
⟨*proof*⟩

lemma *down-scast-inj*:
‹*inj-on SCAST A*› **if** ‹*is-down* (*scast* :: '*b word* ⇒ '*a word*)›
⟨*proof*⟩

lemma *ucast-down-wi*:
‹*UCAST* (*word-of-int x*) = *word-of-int x*› **if** ‹*is-down UCAST*›
⟨*proof*⟩

lemma *ucast-down-no*:
‹*UCAST* (*numeral bin*) = *numeral bin*› **if** ‹*is-down UCAST*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

lemma *unat-div*:
‹*unat* (*x div y*) = *unat x div unat y*›
⟨*proof*⟩

lemma *unat-mod*:
‹*unat* (*x mod y*) = *unat x mod unat y*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *take-bit-word-beyond-length-eq*:
‹*take-bit n w* = *w*› **if** ‹*LENGTH*(*'a*) ≤ *n*› **for** *w* :: ‹*'a*::*len word*›
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mask-Suc-0*: *mask (Suc 0) = 1*
⟨*proof*⟩

lemma *bin-last-bintrunc*: *odd (take-bit l n) ⟷ l > 0 ∧ odd n*
⟨*proof*⟩

lemma *push-bit-word-beyond* [*simp*]:
‹*push-bit n w = 0*› **if** ‹*LENGTH('a) ≤ n*› **for** *w* :: ‹*'a::len word*›
⟨*proof*⟩

lemma *drop-bit-word-beyond* [*simp*]:
‹*drop-bit n w = 0*› **if** ‹*LENGTH('a) ≤ n*› **for** *w* :: ‹*'a::len word*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

lemma *of-nat-inverse*:
‹*word-of-nat r = a ⟹ r < 2 ^ LENGTH('a) ⟹ unat a = r*›
for *a* :: ‹*'a::len word*›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *uint-cong*: *x = y ⟹ uint x = uint y*
⟨*proof*⟩

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)*
⟨*proof*⟩

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*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *sint-word-diff*: *sint (a − b) = signed-take-bit (LENGTH('a) − 1) (sint a − sint b)*
⟨*proof*⟩

lemma *sint-word-mult*: *sint (a * b) = signed-take-bit (LENGTH('a) − 1) (sint a * sint b)*
⟨*proof*⟩

lemma *sint-word-minus*: *sint (− a) = signed-take-bit (LENGTH('a) − 1) (− sint a)*
⟨*proof*⟩

lemma *sint-word-succ*: *sint* (*word-succ a*) = *signed-take-bit* (*LENGTH*('*a*) − *1*) (*sint a* + *1*)
⟨*proof*⟩

lemma *sint-word-pred*: *sint* (*word-pred a*) = *signed-take-bit* (*LENGTH*('*a*) − *1*) (*sint a* − *1*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-sp-01* [*simp*]:
word-succ (− *1*) = *0* ∧ *word-succ 0* = *1* ∧ *word-pred 0* = − *1* ∧ *word-pred 1* = *0*
⟨*proof*⟩
lemma *word-of-int-Ex*: ∃ *y*. *x* = *word-of-int y*
⟨*proof*⟩

108.18 Order on fixed-length words

lift-definition *udvd* :: ‹'*a*::*len word* ⇒ '*a*::*len word* ⇒ *bool*› (**infixl** ‹*udvd*› *50*)
is ‹λ*k l*. *take-bit LENGTH*('*a*) *k dvd take-bit LENGTH*('*a*) *l*› ⟨*proof*⟩

lemma *udvd-iff-dvd*:
‹*x udvd y* ⟷ *unat x dvd unat y*›
⟨*proof*⟩

lemma *udvd-iff-dvd-int*:
‹*v udvd w* ⟷ *uint v dvd uint w*›

⟨*proof*⟩

lemma *udvdI* [*intro*]:
‹*v udvd w*› **if** ‹*unat w* = *unat v* ∗ *unat u*›
⟨*proof*⟩

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*⟩

lemma *udvd-imp-mod-eq-0*:
‹*w mod v* = *0*› **if** ‹*v udvd w*›
⟨*proof*⟩

lemma *mod-eq-0-imp-udvd* [*intro?*]:
‹*v udvd w*› **if** ‹*w mod v* = *0*›
⟨*proof*⟩

lemma *udvd-imp-dvd*:
‹*v dvd w*› **if** ‹*v udvd w*› **for** *v w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *exp-dvd-iff-exp-udvd*:
‹*2* ^ *n dvd w* ⟷ *2* ^ *n udvd w*› **for** *v w* :: ‹'*a*::*len word*›
⟨*proof*⟩

lemma *udvd-nat-alt*:
‹*a udvd b* ⟷ (∃ *n*. *unat b* = *n* ∗ *unat a*)›
⟨*proof*⟩

lemma *udvd-unfold-int*:
‹*a udvd b* ⟷ (∃ *n*≥*0*. *uint b* = *n* ∗ *uint a*)›
⟨*proof*⟩

lemma *unat-minus-one*:
‹*unat* (*w* − *1*) = *unat w* − *1*› **if** ‹*w* ≠ *0*›
⟨*proof*⟩

lemma *measure-unat*: *p* ≠ *0* ⟹ *unat* (*p* − *1*) < *unat p*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *uint-mult-lem*:
(*uint x * uint y < 2 ^ LENGTH('a)*) =
(*uint (x * y) = uint x * uint y*)
for *x y* :: *'a::len word*
⟨*proof*⟩

lemma *uint-sub-lem*: *uint x ≥ uint y ⟷ uint (x − y) = uint x − uint y*
⟨*proof*⟩

lemma *uint-add-le*: *uint (x + y) ≤ uint x + uint y*
⟨*proof*⟩

lemma *uint-sub-ge*: *uint (x − y) ≥ uint x − uint y*
⟨*proof*⟩

lemma *int-mod-ge*: ‹*a ≤ a mod n*› **if** ‹*a < n*› ‹*0 < n*›
for *a n* :: *int*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-of-int-inverse*:
word-of-int r = a ⟹ 0 ≤ r ⟹ r < 2 ^ LENGTH('a) ⟹ uint a = r
for *a :: 'a::len word*
⟨*proof*⟩

lemma *unat-split*: *P (unat x) ⟷ (∀ n. of-nat n = x ∧ n < 2^LENGTH('a) ⟶ P n)*
for *x :: 'a::len word*
⟨*proof*⟩

lemma *unat-split-asm*: *P (unat x) ⟷ (∄ n. of-nat n = x ∧ n < 2^LENGTH('a) ∧ ¬ P n)*
for *x :: 'a::len word*
⟨*proof*⟩

lemma *un-ui-le*:
‹*unat a ≤ unat b ⟷ uint a ≤ uint b*›
⟨*proof*⟩

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›*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

108.20 Some proof tool support

lemma *power-False-cong*: $False \Longrightarrow a \mathbin{\hat{}} b = c \mathbin{\hat{}} d$
⟨*proof*⟩

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*⟩

108.21 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*: $x \leq x + y \longleftrightarrow uint\ x + uint\ y < 2 \mathbin{\hat{}} size\ x$
for *x y* :: *′a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *no-olen-add′*: $x \leq y + x \longleftrightarrow uint\ y + uint\ x < 2 \mathbin{\hat{}} LENGTH('a)$
for *x y* :: *′a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-le-sub1*: $x \neq 0 \Longrightarrow 1 \leq x \longleftrightarrow 0 \leq x - 1$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *sub-wrap-lt*: $x < x - z \longleftrightarrow x < z$
for $x\ z :: 'a::len\ word$
⟨*proof*⟩

lemma *sub-wrap*: $x \leq x - z \longleftrightarrow z = 0 \vee x < z$
for $x\ z :: 'a::len\ word$
⟨*proof*⟩

lemma *plus-minus-not-NULL-ab*: $x \leq ab - c \Longrightarrow c \leq ab \Longrightarrow c \neq 0 \Longrightarrow x + c \neq 0$
for $x\ ab\ c :: 'a::len\ word$
⟨*proof*⟩

lemma *plus-minus-no-overflow-ab*: $x \leq ab - c \Longrightarrow c \leq ab \Longrightarrow x \leq x + c$
for $x\ ab\ c :: 'a::len\ word$
⟨*proof*⟩

lemma *le-minus'*: $a + c \leq b \Longrightarrow a \leq a + c \Longrightarrow c \leq b - a$
for $a\ b\ c :: 'a::len\ word$
⟨*proof*⟩

lemma *le-plus'*: $a \leq b \Longrightarrow c \leq b - a \Longrightarrow a + c \leq b$
for $a\ b\ c :: 'a::len\ word$
⟨*proof*⟩

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 $x\ y\ z :: 'a::len\ word$
⟨*proof*⟩

lemma *word-less-minus-cancel*: $y - x < z - x \Longrightarrow x \leq z \Longrightarrow y < z$
for $x\ y\ z :: 'a::len\ word$
⟨*proof*⟩

lemma *word-less-minus-mono-left*: $y < z \Longrightarrow x \leq y \Longrightarrow y - x < z - x$
for $x\ y\ z :: 'a::len\ word$
⟨*proof*⟩

lemma *word-less-minus-mono*: $a < c \Longrightarrow d < b \Longrightarrow a - b < a \Longrightarrow c - d < c \Longrightarrow a - b < c - d$
for $a\ b\ c\ d :: 'a::len\ word$

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-less-add-right*: $x < y - z \Longrightarrow z \leq y \Longrightarrow x + z < y$
for *x y z* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-less-sub-right*: $x < y + z \Longrightarrow y \leq x \Longrightarrow x - y < z$
for *x y z* :: *'a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inc-le*: $i < m \Longrightarrow i + 1 \leq m$
for *i m* :: *'a*::*len word*
⟨*proof*⟩

lemma *less-imp-less-eq-dec*:
‹$v \leq w - 1$› **if** ‹$v < w$› **for** $v\ w$ :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *inc-less-eq-triv-imp*:
‹$w = -1$› **if** ‹$w + 1 \leq w$› **for** w :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *less-eq-dec-triv-imp*:
‹$w = 0$› **if** ‹$w \leq w - 1$› **for** w :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *inc-less-eq-iff*:
‹$v + 1 \leq w \longleftrightarrow v = -1 \vee v < w$› **for** $v\ w$ :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *less-eq-dec-iff*:
‹$v \leq w - 1 \longleftrightarrow w = 0 \vee v < w$› **for** $v\ w$ :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *inc-i*: $1 \leq i \Longrightarrow i < m \Longrightarrow 1 \leq i + 1 \wedge i + 1 \leq m$
for $i\ m$ :: $'a$::*len word*
⟨*proof*⟩

lemma *dec-less-imp-less-eq*:
‹$v \leq w$› **if** ‹$v - 1 < w$› **for** $v\ w$ :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *less-inc-imp-less-eq*:
‹$v \leq w$› **if** ‹$v < w + 1$› **for** $v\ w$ :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *less-eq-dec-self-iff-eq*:
‹$w \leq w - 1 \longleftrightarrow w = 0$› **for** w :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *inc-less-eq-self-iff-eq*:
‹$w + 1 \leq w \longleftrightarrow w = -1$› **for** w :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *udvd-incr-lem*:
$⟦up < uq;\ up = ua + n * uint\ K;\ uq = ua + n' * uint\ K⟧$
$\Longrightarrow up + uint\ K \leq uq$
⟨*proof*⟩

lemma *udvd-incr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p + K \leq q$
⟨*proof*⟩

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*⟩

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'*: $xy < k \Longrightarrow z\ udvd\ xy \Longrightarrow z\ udvd\ k \Longrightarrow xy \leq k - z$
⟨*proof*⟩

lemma *udvd-incr2-K*:
$p < a + s \Longrightarrow a \leq a + s \Longrightarrow K\ udvd\ s \Longrightarrow K\ udvd\ p - a \Longrightarrow a \leq p \Longrightarrow$
$0 < K \Longrightarrow p \leq p + K \wedge p + K \leq a + s$
⟨*proof*⟩

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-of-int* $x =$ *of-nat* (*nat* x)
⟨*proof*⟩

note that *iszero-def* is only for class *comm-semiring-1-cancel*, which requires word length ≥ 1, ie *'a::len word*

lemma *iszero-word-no* [*simp*]:
iszero (*numeral bin* :: *'a::len word*) =
iszero (*take-bit LENGTH('a)* (*numeral bin* :: *int*))
⟨*proof*⟩

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* ≤ *w div 2*› **if** ‹*v* ≤ *w*› **for** *v w* :: ‹*'a::len word*›
⟨*proof*⟩

lemma *word-half-less-imp-less-eq*:
‹*v* ≤ *w*› **if** ‹*v div 2* < *w div 2*› **for** *v w* :: ‹*'a::len word*›
⟨*proof*⟩

108.23 Word and nat

lemma *word-nchotomy*: $\forall w :: 'a::len\ word.\ \exists n.\ w =$ *of-nat* $n \wedge n < 2\ \hat{}\ LENGTH('a)$
⟨*proof*⟩

lemma *of-nat-eq*: *of-nat n = w ⟷ (∃ q. n = unat w + q * 2 ^ LENGTH('a))*
for *w :: 'a::len word*
⟨*proof*⟩

lemma *of-nat-eq-size*: *of-nat n = w ⟷ (∃ q. n = unat w + q * 2 ^ size w)*
⟨*proof*⟩

lemma *of-nat-0*: *of-nat m = (0::'a::len word) ⟷ (∃ q. m = q * 2 ^ LENGTH('a))*
⟨*proof*⟩

lemma *of-nat-2p* [*simp*]: *of-nat (2 ^ LENGTH('a)) = (0::'a::len word)*
⟨*proof*⟩

lemma *of-nat-gt-0*: *of-nat k ≠ 0 ⟹ 0 < k*
⟨*proof*⟩

lemma *of-nat-neq-0*: *0 < k ⟹ k < 2 ^ LENGTH('a::len) ⟹ of-nat k ≠ (0 :: 'a word)*
⟨*proof*⟩

lemma *Abs-fnat-hom-add*: *of-nat a + of-nat b = of-nat (a + b)*
⟨*proof*⟩

lemma *Abs-fnat-hom-mult*: *of-nat a * of-nat b = (of-nat (a * b) :: 'a::len word)*
⟨*proof*⟩

lemma *Abs-fnat-hom-Suc*: *word-succ (of-nat a) = of-nat (Suc a)*
⟨*proof*⟩

lemma *Abs-fnat-hom-0*: *(0::'a::len word) = of-nat 0*
⟨*proof*⟩

lemma *Abs-fnat-hom-1*: *(1::'a::len word) = of-nat (Suc 0)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *word-arith-nat-mult*: *a * b = of-nat (unat a * unat b)*
⟨*proof*⟩

lemma *word-arith-nat-Suc*: *word-succ a = of-nat (Suc (unat a))*
⟨*proof*⟩

lemma *word-arith-nat-div*: *a div b = of-nat (unat a div unat b)*
⟨*proof*⟩

lemma *word-arith-nat-mod*: *a mod b = of-nat (unat a mod unat b)*
⟨*proof*⟩

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*⟩

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)*›
⟨*proof*⟩

lemma *unat-cong*: *x = y ⟹ unat x = unat y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *unat-mult-lem*:
*unat x * unat y < 2 ^ LENGTH('a) ⟷ unat (x * y) = unat x * unat y*
for *x y :: 'a::len word*
⟨*proof*⟩

lemma *le-no-overflow*: *x ≤ b ⟹ a ≤ a + b ⟹ x ≤ a + b*
for *a b x :: 'a::len word*
⟨*proof*⟩

lemma *uint-div*:
‹*uint (x div y) = uint x div uint y*›
⟨*proof*⟩

lemma *uint-mod*:
‹*uint* (*x mod y*) = *uint x mod uint y*›
⟨*proof*⟩

lemma *no-plus-overflow-unat-size*: $x \leq x + y \longleftrightarrow unat\ x + unat\ y < 2\ \hat{}\ size\ x$
for *x y* :: *'a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *div-lt'*: $i \leq k\ div\ x \Longrightarrow unat\ i * unat\ x < 2\ \hat{}\ LENGTH('a)$
for *i k x* :: *'a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *div-le-mult*: $[\![i \leq k\ div\ x;\ 0 < x]\!] \Longrightarrow i * x \leq k$
for *i k x* :: *'a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-div-mult-le*: *a div b* * *b* ≤ *a*
for *a b* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-mod-less-divisor*: *0* < *n* ⟹ *m mod n* < *n*
for *m n* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-of-int-power-hom*: *word-of-int a* ^ *n* = (*word-of-int* (*a* ^ *n*) :: *'a*::*len word*)
⟨*proof*⟩

lemma *word-arith-power-alt*: *a* ^ *n* = (*word-of-int* (*uint a* ^ *n*) :: *'a*::*len word*)
⟨*proof*⟩

lemma *unatSuc*: *1* + *n* ≠ *0* ⟹ *unat* (*1* + *n*) = *Suc* (*unat n*)
for *n* :: *'a*::*len word*
⟨*proof*⟩

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*)}›
⟨*proof*⟩

lemma *range-uint*: ‹*range* (*uint* :: *'a word* ⇒ *int*) = {*0*..<*2* ^ *LENGTH*(*'a*::*len*)}›
⟨*proof*⟩

lemma *UNIV-eq*: ‹(*UNIV* :: *'a word set*) = *word-of-int* ' {*0*..<*2* ^ *LENGTH*(*'a*::*len*)}›
⟨*proof*⟩

lemma *card-word*: *CARD*(*'a word*) = *2* ^ *LENGTH*(*'a*::*len*)
⟨*proof*⟩

lemma *card-word-size*: *CARD*(*'a word*) = *2* ^ *size x*
for *x* :: *'a*::*len word*
⟨*proof*⟩

end

instance *word* :: (*len*) *finite*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *word-of-int-not-numeral-eq* [*simp*]:
‹*(word-of-int (NOT (numeral bin)) :: 'a::len word) = − numeral bin − 1*›
⟨*proof*⟩

lemma *uint-and*:
‹*uint (x AND y) = uint x AND uint y*›
⟨*proof*⟩

lemma *uint-or*:
‹*uint (x OR y) = uint x OR uint y*›
⟨*proof*⟩

lemma *uint-xor*:
‹*uint (x XOR y) = uint x XOR uint y*›
⟨*proof*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-bw-same*:
x AND x = x
x OR x = x
x XOR x = 0
for *x* :: *'a*::*len word*
⟨*proof*⟩

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 $x :: 'a::len\ word$
⟨*proof*⟩

lemma *word-not-not* [*simp*]: $NOT\ (NOT\ x) = x$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *word-ao-dist*: $(x\ OR\ y)\ AND\ z = x\ AND\ z\ OR\ y\ AND\ z$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *word-oa-dist*: $x\ AND\ y\ OR\ z = (x\ OR\ z)\ AND\ (y\ OR\ z)$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *word-add-not* [*simp*]: $x + NOT\ x = -1$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *word-plus-and-or* [*simp*]: $(x\ AND\ y) + (x\ OR\ y) = x + y$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *leoa*: $w = x\ OR\ y \Longrightarrow y = w\ AND\ y$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *leao*: $w' = x'\ AND\ y' \Longrightarrow x' = x'\ OR\ w'$
for $x' :: 'a::len\ word$
⟨*proof*⟩

lemma *word-ao-equiv*: $w = w\ OR\ w' \longleftrightarrow w' = w\ AND\ w'$
for $w\ w' :: 'a::len\ word$
⟨*proof*⟩

lemma *le-word-or2*: $x \leq x\ OR\ y$
for $x\ y :: 'a::len\ word$
⟨*proof*⟩

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*
$\longleftrightarrow n < min\ LENGTH('a)\ (length\ bs) \wedge bs\ !\ n$›

⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *word-rev-rev* [*simp*] : *word-reverse* (*word-reverse w*) = *w*
⟨*proof*⟩

lemma *word-rev-gal*: *word-reverse w* = *u* ⟹ *word-reverse u* = *w*
⟨*proof*⟩

lemma *word-rev-gal'*: *u* = *word-reverse w* ⟹ *w* = *word-reverse u*
⟨*proof*⟩

lemma *word-eq-reverseI*:
‹*v* = *w*› **if** ‹*word-reverse v* = *word-reverse w*›
⟨*proof*⟩

lemma *uint-2p*: (*0*::*'a*::*len word*) < *2* ^ *n* ⟹ *uint* (*2* ^ *n*::*'a*::*len word*) = *2* ^ *n*
⟨*proof*⟩

lemma *word-of-int-2p*: (*word-of-int* (*2* ^ *n*) :: *'a*::*len word*) = *2* ^ *n*
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *False-map2-or*: ⟦*set xs* ⊆ {*False*}; *length ys* = *length xs*⟧ ⟹ *map2* (∨) *xs ys* = *ys*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *False-map2-and*: ⟦*set xs* ⊆ {*False*}; *length ys* = *length xs*⟧ ⟹ *map2* (∧) *xs ys* = *xs*
⟨*proof*⟩

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*
⟨*proof*⟩

108.25.2 Mask

lemma *minus-1-eq-mask*:
‹*− 1 = (mask LENGTH('a) :: 'a::len word)*›
⟨*proof*⟩

lemma *mask-eq-decr-exp*:
‹*mask n = 2 ^ n − (1 :: 'a::len word)*›
⟨*proof*⟩

lemma *mask-Suc-rec*:
‹*mask (Suc n) = 2 * mask n + (1 :: 'a::len word)*›
⟨*proof*⟩

context
begin

qualified lemma *bit-mask-iff* [*bit-simps*]:
‹*bit (mask m :: 'a::len word) n ⟷ n < min LENGTH('a) m*›
⟨*proof*⟩

end

lemma *mask-bin*: *mask n = word-of-int (take-bit n (− 1))*
⟨*proof*⟩

lemma *and-mask-bintr*: *w AND mask n = word-of-int (take-bit n (uint w))*
⟨*proof*⟩

lemma *and-mask-wi*: *word-of-int i AND mask n = word-of-int (take-bit n i)*
⟨*proof*⟩

lemma *and-mask-wi'*:
word-of-int i AND mask n = (word-of-int (take-bit (min LENGTH('a) n) i) :: 'a::len word)
⟨*proof*⟩

lemma *and-mask-no*: *numeral i AND mask n = word-of-int (take-bit n (numeral i))*
⟨*proof*⟩

lemma *and-mask-mod-2p*: *w AND mask n = word-of-int (uint w mod 2 ^ n)*
⟨*proof*⟩

lemma *uint-mask-eq*:
‹*uint (mask n :: 'a::len word) = mask (min LENGTH('a) n)*›
⟨*proof*⟩

lemma *and-mask-lt-2p*: *uint (w AND mask n) < 2 ^ n*
⟨*proof*⟩

lemma *mask-eq-iff*: *w AND mask n = w ⟷ uint w < 2 ^ n*
⟨*proof*⟩

lemma *and-mask-dvd*: *2 ^ n dvd uint w ⟷ w AND mask n = 0*
⟨*proof*⟩

lemma *and-mask-dvd-nat*: *2 ^ n dvd unat w ⟷ w AND mask n = 0*
⟨*proof*⟩

lemma *word-2p-lem*: *n < size w ⟹ w < 2 ^ n = (uint w < 2 ^ n)*
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *less-mask-eq*:
fixes *x* :: *'a*::*len word*
assumes *x < 2 ^ n* **shows** *x AND mask n = x*
⟨*proof*⟩

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*›
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *mask-power-eq*: *(x AND mask n) ^ k AND mask n = x ^ k AND mask n*
for *x* :: ‹*'a*::*len word*›

⟨*proof*⟩

lemma *mask-full* [*simp*]: *mask LENGTH('a) = (− 1 :: 'a::len word)*
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *slice1-0* [*simp*] : *slice1 n 0 = 0*
⟨*proof*⟩

lemma *slice-0* [*simp*] : *slice n 0 = 0*
⟨*proof*⟩

lemma *ucast-slice1*: *ucast w = slice1 (size w) w*
⟨*proof*⟩

lemma *ucast-slice*: *ucast w = slice 0 w*
⟨*proof*⟩

lemma *slice-id*: *slice 0 t = t*
⟨*proof*⟩

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*⟩

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)
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *revcast-slice1* [*OF refl*]: *rc = revcast w* ⟹ *slice1 (size rc) w = rc*
⟨*proof*⟩

lemma *revcast-rev-ucast* [*OF refl refl refl*]:
cs = [rc, uc] ⟹ *rc = revcast (word-reverse w)* ⟹ *uc = ucast w* ⟹
rc = word-reverse uc
⟨*proof*⟩

lemma *revcast-ucast*: *revcast w = word-reverse (ucast (word-reverse w))*
⟨*proof*⟩

lemma *ucast-revcast*: *ucast w = word-reverse (revcast (word-reverse w))*
⟨*proof*⟩

lemma *ucast-rev-revcast*: *ucast (word-reverse w) = word-reverse (revcast w)*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-cat-split-alt*: ⟦*size w* ≤ *size u* + *size v*; *word-split w* = (*u*,*v*)⟧ ⟹ *word-cat u v* = *w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *slice-cat1* [*OF refl*]:
⟦*wc* = *word-cat a b*; *size a* + *size b* ≤ *size wc*⟧ ⟹ *slice* (*size b*) *wc* = *a*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*⟩

108.27 Rotation

lemma *word-rotr-word-rotr-eq*: ‹*word-rotr m* (*word-rotr n w*) = *word-rotr* (*m* + *n*) *w*›
⟨*proof*⟩

lemma *word-rot-lem*: $[\![l + k = d + k \ mod \ l;\ n < l]\!] \Longrightarrow ((d + n) \ mod \ l) = n$ **for** *l*::*nat*
⟨*proof*⟩

lemma *word-rot-rl* [*simp*]: ‹*word-rotl k* (*word-rotr k v*) = *v*›
⟨*proof*⟩

lemma *word-rot-lr* [*simp*]: ‹*word-rotr k* (*word-rotl k v*) = *v*›
⟨*proof*⟩

lemma *word-rot-gal*:
‹*word-rotr n v* = *w* ⟷ *word-rotl n w* = *v*›
⟨*proof*⟩

lemma *word-rot-gal'*:
‹*w* = *word-rotr n v* ⟷ *v* = *word-rotl n w*›
⟨*proof*⟩

lemma *word-reverse-word-rotl*:
‹*word-reverse* (*word-rotl n w*) = *word-rotr n* (*word-reverse w*)› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *word-reverse-word-rotr*:
‹*word-reverse* (*word-rotr n w*) = *word-rotl n* (*word-reverse w*)›
⟨*proof*⟩

lemma *word-rotl-rev*:
‹*word-rotl n w* = *word-reverse* (*word-rotr n* (*word-reverse w*))›
⟨*proof*⟩

lemma *word-rotr-rev*:
‹*word-rotr n w* = *word-reverse* (*word-rotl n* (*word-reverse w*))›
⟨*proof*⟩

lemma *word-roti-0* [*simp*]: *word-roti 0 w* = *w*
⟨*proof*⟩

lemma *word-roti-add*: *word-roti* (*m* + *n*) *w* = *word-roti m* (*word-roti n w*)
⟨*proof*⟩

lemma *word-roti-conv-mod'*:
word-roti n w = *word-roti* (*n mod int* (*size w*)) *w*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *word-nat-cases* [*cases type*: *word*]:
fixes *x* :: *'a::len word*
obtains *n* **where** *x = of-nat n* **and** *n < 2^LENGTH('a)*
⟨*proof*⟩

lemma *max-word-max* [*intro*!]:
‹$n \leq -1$› **for** n :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *word-of-int-2p-len*: *word-of-int* $(2 \hat{} LENGTH('a)) = (0::'a::len\ word)$
⟨*proof*⟩

lemma *word-pow-0*: $(2::'a::len\ word) \hat{} LENGTH('a) = 0$
⟨*proof*⟩

lemma *max-word-wrap*:
‹$x + 1 = 0 \Longrightarrow x = -1$› **for** x :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *word-and-max*:
‹$x\ AND\ -1 = x$› **for** x :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *word-or-max*:
‹$x\ OR\ -1 = -1$› **for** x :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *word-ao-dist2*: $x\ AND\ (y\ OR\ z) = x\ AND\ y\ OR\ x\ AND\ z$
for $x\ y\ z$ :: $'a$::*len word*
⟨*proof*⟩

lemma *word-oa-dist2*: $x\ OR\ y\ AND\ z = (x\ OR\ y)\ AND\ (x\ OR\ z)$
for $x\ y\ z$ :: $'a$::*len word*
⟨*proof*⟩

lemma *word-and-not* [*simp*]: $x\ AND\ NOT\ x = 0$
for x :: $'a$::*len word*
⟨*proof*⟩

lemma *word-or-not* [*simp*]:
‹$x\ OR\ NOT\ x = -1$› **for** x :: ‹$'a$::*len word*›
⟨*proof*⟩

lemma *word-xor-and-or*: $x\ XOR\ y = x\ AND\ NOT\ y\ OR\ NOT\ x\ AND\ y$
for $x\ y$ :: $'a$::*len word*
⟨*proof*⟩

lemma *uint-lt-0* [*simp*]: $uint\ x < 0 = False$
⟨*proof*⟩

lemma *word-less-1* [*simp*]: $x < 1 \longleftrightarrow x = 0$
for x :: $'a$::*len word*
⟨*proof*⟩

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)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-neq-0-conv*: $w \neq 0 \longleftrightarrow 0 < w$
for $w ::$ *'a::len word*
⟨*proof*⟩

lemma *max-lt*: *unat* (*max a b div c*) = *unat* (*max a b*) *div unat c*
for $c ::$ *'a::len word*
⟨*proof*⟩

lemma *uint-sub-if-size*:
uint $(x - y) =$
(*if uint* $y \leq$ *uint* x
then uint $x -$ *uint* y
else uint $x -$ *uint* $y +$ *2^size* x)
⟨*proof*⟩

lemma *unat-sub*:
‹*unat* $(a - b) =$ *unat* $a -$ *unat* b›
if ‹$b \leq a$›
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *word-of-int-inj*:
‹(*word-of-int* $x ::$ *'a::len word*) = *word-of-int* $y \longleftrightarrow x = y$›
if ‹$0 \leq x \wedge x <$ *2 ^ LENGTH('a)*› ‹$0 \leq y \wedge y <$ *2 ^ LENGTH('a)*›
⟨*proof*⟩

lemma *word-le-less-eq*: $x \leq y \longleftrightarrow x = y \vee x < y$
for $x\ y ::$ *'z::len word*

⟨*proof*⟩

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*⟩

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'*
⟨*proof*⟩

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*⟩

lemma *word-induct*: *P 0* ⟹ (⋀*n*. *P n* ⟹ *P* (*1* + *n*)) ⟹ *P m*
for *P* :: *'a*::*len word* ⇒ *bool*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-rec-Pred*: *n* ≠ *0* ⟹ *word-rec z s n* = *s* (*n* − *1*) (*word-rec z s* (*n* − *1*))

⟨*proof*⟩

lemma *word-rec-in*: *f* (*word-rec z* (λ-. *f*) *n*) = *word-rec* (*f z*) (λ-. *f*) *n*
⟨*proof*⟩

lemma *word-rec-in2*: *f n* (*word-rec z f n*) = *word-rec* (*f 0 z*) (*f* ∘ (+) *1*) *n*
⟨*proof*⟩

lemma *word-rec-twice*:
m ≤ *n* ⟹ *word-rec z f n* = *word-rec* (*word-rec z f* (*n* − *m*)) (*f* ∘ (+) (*n* − *m*)) *m*
⟨*proof*⟩

lemma *word-rec-id*: *word-rec z* (λ-. *id*) *n* = *z*
⟨*proof*⟩

lemma *word-rec-id-eq*: (⋀*m*. *m* < *n* ⟹ *f m* = *id*) ⟹ *word-rec z f n* = *z*
⟨*proof*⟩

lemma *word-rec-max*:
assumes ∀ *m*≥*n*. *m* ≠ − *1* ⟶ *f m* = *id*
shows *word-rec z f* (− *1*) = *word-rec z f n*
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *of-int-signed-take-bit*:
‹*of-int* (*signed-take-bit n k*) = *signed-take-bit n* (*of-int k*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *concat-bit-of-one-2* [*simp*]:
‹*concat-bit n k 1* = *set-bit n* (*take-bit n k*)›
⟨*proof*⟩

lemma *concat-bit-numeral-of-minus-one-1* [*simp*]:
‹*concat-bit* (*numeral m*) (− *1*) *l* = *push-bit* (*numeral m*) *l OR mask* (*numeral m*)›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

end

lemma *word-cat-0-left* [*simp*]:
‹*word-cat 0 w* = *ucast w*›
⟨*proof*⟩

108.31 Executable intervals

instance *word* :: (*len*) ‹{*interval-top*, *interval-bot*}›
⟨*proof*⟩

108.32 Tool support

⟨*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*› ⟨*proof*⟩

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
⟨*proof*⟩

end

free-constructors *case-bit* **for** ‹*0*::*bit*› | ‹*1*::*bit*›
⟨*proof*⟩

lemma *bit-not-zero-iff* [*simp*]:
‹$a \neq 0 \longleftrightarrow a = 1$› **for** a :: *bit*
⟨*proof*⟩

lemma *bit-not-one-iff* [*simp*]:
‹$a \neq 1 \longleftrightarrow a = 0$› **for** a :: *bit*
⟨*proof*⟩

instantiation *bit* :: *semidom-modulo*
begin

definition *plus-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
where ‹$a + b$ = *Abs-bit* (*Rep-bit* $a \neq$ *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* $a \wedge$ *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* $a \wedge \neg$ *Rep-bit* b)›

instance
⟨*proof*⟩

end

lemma *bit-2-eq-0* [*simp*]:
‹$2 = (0::bit)$›
⟨*proof*⟩

instance *bit* :: *semiring-parity*
⟨*proof*⟩

lemma *Abs-bit-eq-of-bool* [*code-abbrev*]:
‹*Abs-bit* = *of-bool*›
⟨*proof*⟩

lemma *Rep-bit-eq-odd*:
‹*Rep-bit* = *odd*›
⟨*proof*⟩

lemma *Rep-bit-iff-odd* [*code-abbrev*]:
‹*Rep-bit* $b \longleftrightarrow$ *odd* b›

⟨*proof*⟩

lemma *Not-Rep-bit-iff-even* [*code-abbrev*]:
‹¬ *Rep-bit b* ⟷ *even b*›
⟨*proof*⟩

lemma *Not-Not-Rep-bit* [*code-unfold*]:
‹¬ ¬ *Rep-bit b* ⟷ *Rep-bit b*›
⟨*proof*⟩

code-datatype ‹*0*::*bit*› ‹*1*::*bit*›

lemma *Abs-bit-code* [*code*]:
‹*Abs-bit False* = *0*›
‹*Abs-bit True* = *1* ›
⟨*proof*⟩

lemma *Rep-bit-code* [*code*]:
‹*Rep-bit 0* ⟷ *False*›
‹*Rep-bit 1* ⟷ *True*›
⟨*proof*⟩

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*
⟨*proof*⟩

end

lemma *modulo-bit-unfold* [*simp*, *code*]:
‹*a mod b* = *of-bool* (*odd a* ∧ *even b*)› **for** *a b* :: *bit*
⟨*proof*⟩

lemma *power-bit-unfold* [*simp*]:
‹*a* ^ *n* = *of-bool* (*odd a* ∨ *n* = *0*)› **for** *a* :: *bit*
⟨*proof*⟩

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
  ⟨*proof*⟩

end

instantiation *bit* :: *semiring-bits*
begin

definition *bit-bit* :: ‹*bit* ⇒ *nat* ⇒ *bool*›
  where [*simp*]: ‹*bit-bit b n* ⟷ *odd b* ∧ *n* = *0*›

instance
  ⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *add-bit-eq-xor* [*simp*, *code*]:
‹(+) = (*Bit-Operations.xor* :: *bit ⇒* -)›
⟨*proof*⟩

lemma *mult-bit-eq-and* [*simp*, *code*]:
‹(∗) = (*Bit-Operations.and* :: *bit ⇒* -)›
⟨*proof*⟩

lemma *bit-numeral-even* [*simp*]:
‹*numeral* (*Num.Bit0 n*) = (*0* :: *bit*)›
⟨*proof*⟩

lemma *bit-numeral-odd* [*simp*]:
‹*numeral* (*Num.Bit1 n*) = (*1* :: *bit*)›
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *fst-mono*: $x \leq y \Longrightarrow fst\ x \leq fst\ y$
⟨*proof*⟩

lemma *snd-mono*: $x \leq y \Longrightarrow snd\ x \leq snd\ y$
⟨*proof*⟩

lemma *Pair-mono*: $x \leq x' \Longrightarrow y \leq y' \Longrightarrow (x,\ y) \leq (x',\ y')$
⟨*proof*⟩

lemma *Pair-le* [*simp*]: $(a,\ b) \leq (c,\ d) \longleftrightarrow a \leq c \wedge b \leq d$
⟨*proof*⟩

lemma *atLeastAtMost-prod-eq*: $\{a..b\} = \{fst\ a..fst\ b\} \times \{snd\ a..snd\ b\}$
⟨*proof*⟩

instance *prod* :: (*preorder*, *preorder*) *preorder*
⟨*proof*⟩

instance *prod* :: (*order*, *order*) *order*
⟨*proof*⟩

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*]: $inf\ (a,\ b)\ (c,\ d) = (inf\ a\ c,\ inf\ b\ d)$
⟨*proof*⟩

lemma *fst-inf* [*simp*]: $fst\ (inf\ x\ y) = inf\ (fst\ x)\ (fst\ y)$
⟨*proof*⟩

lemma *snd-inf* [*simp*]: $snd\ (inf\ x\ y) = inf\ (snd\ x)\ (snd\ y)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *prod* :: (*semilattice-inf*, *semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fst-sup* [*simp*]: *fst* (*sup x y*) = *sup* (*fst x*) (*fst y*)
⟨*proof*⟩

lemma *snd-sup* [*simp*]: *snd* (*sup x y*) = *sup* (*snd x*) (*snd y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *prod* :: (*semilattice-sup*, *semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *prod* :: (*lattice*, *lattice*) *lattice* ⟨*proof*⟩

instance *prod* :: (*distrib-lattice*, *distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

110.3 Top and bottom elements

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ⟨*proof*⟩

end

lemma *fst-top* [*simp*]: *fst top* = *top*
⟨*proof*⟩

lemma *snd-top* [*simp*]: *snd top* = *top*
⟨*proof*⟩

lemma *Pair-top-top*: (*top*, *top*) = *top*
⟨*proof*⟩

instance *prod* :: (*order-top*, *order-top*) *order-top*
⟨*proof*⟩

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ⟨*proof*⟩

end

lemma *fst-bot* [*simp*]: *fst bot* = *bot*
⟨*proof*⟩

lemma *snd-bot* [*simp*]: *snd bot* = *bot*
⟨*proof*⟩

lemma *Pair-bot-bot*: (*bot*, *bot*) = *bot*
⟨*proof*⟩

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
⟨*proof*⟩

instance *prod* :: (*bounded-lattice*, *bounded-lattice*) *bounded-lattice* ⟨*proof*⟩

instance *prod* :: (*boolean-algebra*, *boolean-algebra*) *boolean-algebra*
⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *prod* :: (*Sup*, *Sup*) *Sup*

begin

definition *Sup A* = (*SUP x∈A. fst x*, *SUP x∈A. snd x*)

instance ⟨*proof*⟩

end

instance *prod* :: (*conditionally-complete-lattice*, *conditionally-complete-lattice*) *conditionally-complete-lattice*
⟨*proof*⟩

instance *prod* :: (*complete-lattice*, *complete-lattice*) *complete-lattice*
⟨*proof*⟩

lemma *fst-Inf*: *fst* (*Inf A*) = (*INF x∈A. fst x*)
⟨*proof*⟩

lemma *fst-INF*: *fst* (*INF x∈A. f x*) = (*INF x∈A. fst* (*f x*))
⟨*proof*⟩

lemma *fst-Sup*: *fst* (*Sup A*) = (*SUP x∈A. fst x*)
⟨*proof*⟩

lemma *fst-SUP*: *fst* (*SUP x∈A. f x*) = (*SUP x∈A. fst* (*f x*))
⟨*proof*⟩

lemma *snd-Inf*: *snd* (*Inf A*) = (*INF x∈A. snd x*)
⟨*proof*⟩

lemma *snd-INF*: *snd* (*INF x∈A. f x*) = (*INF x∈A. snd* (*f x*))
⟨*proof*⟩

lemma *snd-Sup*: *snd* (*Sup A*) = (*SUP x∈A. snd x*)
⟨*proof*⟩

lemma *snd-SUP*: *snd* (*SUP x∈A. f x*) = (*SUP x∈A. snd* (*f x*))
⟨*proof*⟩

lemma *INF-Pair*: (*INF x∈A.* (*f x*, *g x*)) = (*INF x∈A. f x*, *INF x∈A. g x*)
⟨*proof*⟩

lemma *SUP-Pair*: (*SUP x∈A.* (*f x*, *g x*)) = (*SUP x∈A. f x*, *SUP x∈A. g x*)
⟨*proof*⟩

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*))

⟨*proof*⟩

lemma *SUP-prod-alt-def*:
Sup (*f* ' *A*) = (*Sup* ((*fst* ∘ *f*) ' *A*), *Sup*((*snd* ∘ *f*) ' *A*))
⟨*proof*⟩

110.5 Complete distributive lattices

instance *prod* :: (*complete-distrib-lattice*, *complete-distrib-lattice*) *complete-distrib-lattice*

⟨*proof*⟩

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* (λ*x*. *fst* (*F* (*x*, *lfp* (λ*y*. *snd* (*F* (*x*, *y*)))))),
(*lfp* (λ*y*. *snd* (*F* (*lfp* (λ*x*. *fst* (*F* (*x*, *lfp* (λ*y*. *snd* (*F* (*x*, *y*)))))), *y*)))))
(**is** *lfp F* = (*?x*, *?y*))
⟨*proof*⟩

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*⟩

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*) ≤ *x*
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *order-bot*
⟨*proof*⟩

lemma *finite-lattice-complete-top-greatest*: (*top*::'*a*::*finite-lattice-complete*) ≥ *x*
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *order-top*
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *bounded-lattice* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-empty*: *Sup* {} = (*bot* :: '*a*::*finite-lattice-complete*)
⟨*proof*⟩

lemma *finite-lattice-complete-Inf-insert*:
fixes *A* :: '*a*::*finite-lattice-complete set*
shows *Inf* (*insert x A*) = *inf x* (*Inf A*)
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-insert*:
fixes *A* :: '*a*::*finite-lattice-complete set*
shows *Sup* (*insert x A*) = *sup x* (*Sup A*)
⟨*proof*⟩

lemma *finite-lattice-complete-Inf-lower*:
(*x*::'*a*::*finite-lattice-complete*) ∈ *A* ⟹ *Inf A* ≤ *x*
⟨*proof*⟩

lemma *finite-lattice-complete-Inf-greatest*:
∀ *x*::'*a*::*finite-lattice-complete* ∈ *A*. *z* ≤ *x* ⟹ *z* ≤ *Inf A*
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-upper*:

$(x::'a::\mathit{finite\text{-}lattice\text{-}complete}) \in A \Longrightarrow \mathit{Sup}\ A \geq x$
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-least*:
$\forall\, x::'a::\mathit{finite\text{-}lattice\text{-}complete} \in A.\ z \geq x \Longrightarrow z \geq \mathit{Sup}\ A$
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *complete-lattice*
⟨*proof*⟩

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
$(\mathit{bot} :: ('a::\mathit{finite\text{-}lattice\text{-}complete} \times 'b::\mathit{finite\text{-}lattice\text{-}complete})) =$
Inf-fin UNIV
⟨*proof*⟩

lemma *finite-top-prod*:
$(\mathit{top} :: ('a::\mathit{finite\text{-}lattice\text{-}complete} \times 'b::\mathit{finite\text{-}lattice\text{-}complete})) =$
Sup-fin UNIV
⟨*proof*⟩

lemma *finite-Inf-prod*:
$\mathit{Inf}(A :: ('a::\mathit{finite\text{-}lattice\text{-}complete} \times 'b::\mathit{finite\text{-}lattice\text{-}complete})\ \mathit{set}) =$
Finite-Set.fold inf top A
⟨*proof*⟩

lemma *finite-Sup-prod*:
$\mathit{Sup}\ (A :: ('a::\mathit{finite\text{-}lattice\text{-}complete} \times 'b::\mathit{finite\text{-}lattice\text{-}complete})\ \mathit{set}) =$
Finite-Set.fold sup bot A
⟨*proof*⟩

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
⟨*proof*⟩

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

lemma *finite-bot-fun*: $(\mathit{bot} :: ('a::\mathit{finite} \Rightarrow 'b::\mathit{finite\text{-}lattice\text{-}complete})) = \mathit{Inf\text{-}fin\ UNIV}$
⟨*proof*⟩

lemma *finite-top-fun*: $(\mathit{top} :: ('a::\mathit{finite} \Rightarrow 'b::\mathit{finite\text{-}lattice\text{-}complete})) = \mathit{Sup\text{-}fin\ UNIV}$
⟨*proof*⟩

lemma *finite-Inf-fun*:
$\mathit{Inf}\ (A::('a::\mathit{finite} \Rightarrow 'b::\mathit{finite\text{-}lattice\text{-}complete})\ \mathit{set}) =$
Finite-Set.fold inf top A
⟨*proof*⟩

lemma *finite-Sup-fun*:

Sup (*A*::('*a*::*finite* ⇒ '*b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
⟨*proof*⟩

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finite-distrib-lattice-complete-inf-Sup*:
inf (*x*::'*a*::*finite-distrib-lattice-complete*) (*Sup A*) = (*SUP y*∈*A*. *inf x y*)
⟨*proof*⟩

context *finite-distrib-lattice-complete*
begin
subclass *finite-distrib-lattice*
⟨*proof*⟩
end

instance *finite-distrib-lattice-complete* ⊆ *complete-distrib-lattice* ⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *linorder-lattice-inf-le2*: *inf* (*x*::'*a*::*linorder-lattice*) *y* ≤ *y*
⟨*proof*⟩

lemma *linorder-lattice-inf-greatest*:
(*x*::'*a*::*linorder-lattice*) ≤ *y* ⟹ *x* ≤ *z* ⟹ *x* ≤ *inf y z*
⟨*proof*⟩

lemma *linorder-lattice-sup-ge1*: *sup* (*x*::'*a*::*linorder-lattice*) *y* ≥ *x*
⟨*proof*⟩

lemma *linorder-lattice-sup-ge2*: *sup* (*x*::'*a*::*linorder-lattice*) *y* ≥ *y*
⟨*proof*⟩

lemma *linorder-lattice-sup-least*:
(*x*::'*a*::*linorder-lattice*) ≥ *y* ⟹ *x* ≥ *z* ⟹ *x* ≥ *sup y z*
⟨*proof*⟩

lemma *linorder-lattice-sup-inf-distrib1*:
sup (*x*::'*a*::*linorder-lattice*) (*inf y z*) = *inf* (*sup x y*) (*sup x z*)
⟨*proof*⟩

instance *linorder-lattice* ⊆ *distrib-lattice*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

end

112 Lexicographic order on lists

theory *List-Lexorder*
imports *Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def: $xs < ys \longleftrightarrow (xs, ys) \in lexord\ \{(u, v).\ u < v\}$

definition
list-le-def: $(xs :: \text{-}\ list) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

instance ⟨*proof*⟩

end

instance *list* :: (*order*) *order*
⟨*proof*⟩

instance *list* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition $(inf :: 'a\ list \Rightarrow \text{-}) = min$

definition $(sup :: 'a\ list \Rightarrow \text{-}) = max$

instance
⟨*proof*⟩

end

lemma *not-less-Nil* [*simp*]: $\neg\ x < []$
⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: $[] < a \# x$
⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$
⟨*proof*⟩

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
⟨*proof*⟩

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
⟨*proof*⟩

end

lemma *less-list-code* [*code*]:
$xs < ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{equal, order\}) \# xs \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
⟨*proof*⟩

lemma *less-eq-list-code* [*code*]:
$x \# xs \leq ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{equal, order\}\ list) \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$
⟨*proof*⟩

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 ⟨*proof*⟩

end

instance *list* :: (*order*) *order*
⟨*proof*⟩

instance *list* :: (*linorder*) *linorder*
⟨*proof*⟩

instance *list* :: (*wellorder*) *wellorder*
⟨*proof*⟩

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition (*inf* :: *'a list* ⇒ -) = *min*

definition (*sup* :: *'a list* ⇒ -) = *max*

instance
⟨*proof*⟩

end

lemma *not-less-Nil* [*simp*]: ¬ *x* < []
⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
⟨*proof*⟩

lemma *Cons-less-Cons*: *a* # *x* < *b* # *y* ⟷ *length x* < *length y* ∨ *length x* = *length y* ∧ (*a* < *b* ∨ *a* = *b* ∧ *x* < *y*)
⟨*proof*⟩

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: [] ≤ *x*
⟨*proof*⟩

lemma *Cons-le-Cons*: *a* # *x* ≤ *b* # *y* ⟷ *length x* < *length y* ∨ *length x* = *length y* ∧ (*a* < *b* ∨ *a* = *b* ∧ *x* ≤ *y*)
⟨*proof*⟩

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *less-list-def'*: $xs < ys \longleftrightarrow$ *strict-prefix xs ys* **for** *xs ys* :: *'a list*
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *less-eq-prod-simp* [*simp*, *code*]:
$(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$
⟨*proof*⟩

lemma *less-prod-simp* [*simp*, *code*]:
$(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$
⟨*proof*⟩

A stronger version for partial orders.

lemma *less-prod-def'*:
fixes $x\ y :: 'a{::}order \times 'b{::}ord$
shows $x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x < snd\ y$
⟨*proof*⟩

instance *prod* :: (*preorder*, *preorder*) *preorder*
⟨*proof*⟩

instance *prod* :: (*order*, *order*) *order*
⟨*proof*⟩

instance *prod* :: (*linorder*, *linorder*) *linorder*
⟨*proof*⟩

instantiation *prod* :: (*linorder*, *linorder*) *distrib-lattice*
begin

definition
$(inf :: 'a \times 'b \Rightarrow - \Rightarrow -) = min$

definition
$(sup :: 'a \times 'b \Rightarrow - \Rightarrow -) = max$

instance
⟨*proof*⟩

end

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ⟨*proof*⟩

end

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
⟨*proof*⟩

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ⟨*proof*⟩

end

instance *prod* :: (*order-top*, *order-top*) *order-top*
⟨*proof*⟩

instance *prod* :: (*wellorder*, *wellorder*) *wellorder*
⟨*proof*⟩

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 subse-

quence 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 ⟨*proof*⟩

end

instance *list* :: (*type*) *order*
⟨*proof*⟩

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*›
  ⟨*proof*⟩

lemma *less-eq-list-below-empty* [*code*]:
  ‹*x* # *xs* ≤ [] ⟷ *False*›
  ⟨*proof*⟩

lemma *le-list-Cons2-iff* [*simp*, *code*]:
  ‹*x* # *xs* ≤ *y* # *ys* ⟷ (*if x* = *y then xs* ≤ *ys else x* # *xs* ≤ *ys*)›
  ⟨*proof*⟩

lemma *less-list-empty* [*simp*]:
  ‹[] < *xs* ⟷ *xs* ≠ []›
  ⟨*proof*⟩

lemma *less-list-empty-Cons* [*code*]:
  ‹[] < *x* # *xs* ⟷ *True*›
  ⟨*proof*⟩

lemma *less-list-below-empty* [*simp*, *code*]:
  ‹*xs* < [] ⟷ *False*›
  ⟨*proof*⟩

lemma *less-list-Cons2-iff* [*code*]:
  ‹*x* # *xs* < *y* # *ys* ⟷ (*if x* = *y then xs* < *ys else x* # *xs* ≤ *ys*)›

⟨*proof*⟩

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*: $xs < ys \Longrightarrow length\ xs < length\ ys$
⟨*proof*⟩

lemma *less-list-drop*: $xs < ys \Longrightarrow xs < x \# ys$
⟨*proof*⟩

lemma *less-list-take-iff*: $x \# xs < x \# ys \longleftrightarrow xs < ys$
⟨*proof*⟩

lemma *less-list-drop-many*: $xs < ys \Longrightarrow xs < zs @ ys$
⟨*proof*⟩

lemma *less-list-take-many-iff*: $zs @ xs < zs @ ys \longleftrightarrow xs < ys$
⟨*proof*⟩

lemma *less-list-rev-take*: $xs @ zs < ys @ zs \longleftrightarrow xs < ys$
⟨*proof*⟩

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 'a*(| - |*'*))))› [*900*, *0*] *900*)

end

named-theorems *datatype-record-update*

⟨*ML*⟩

end

118 Implementation of mappings with Association Lists

theory *AList-Mapping*
imports *AList Mapping*
begin

lift-definition *Mapping* :: $('a \times 'b)$ *list* $\Rightarrow$ $('a, 'b)$ *mapping* **is** *map-of* ⟨*proof*⟩

code-datatype *Mapping*

lemma *lookup-Mapping* [*simp*, *code*]: *Mapping.lookup* (*Mapping xs*) = *map-of xs*
⟨*proof*⟩

lemma *keys-Mapping* [*simp*, *code*]: *Mapping.keys* (*Mapping xs*) = *set* (*map fst xs*)
⟨*proof*⟩

lemma *empty-Mapping* [*code*]: *Mapping.empty* = *Mapping* []
⟨*proof*⟩

lemma *is-empty-Mapping* [*code*]: *Mapping.is-empty* (*Mapping xs*) ⟷ *List.null xs*
⟨*proof*⟩

lemma *update-Mapping* [*code*]: *Mapping.update k v* (*Mapping xs*) = *Mapping* (*AList.update k v xs*)
⟨*proof*⟩

lemma *delete-Mapping* [*code*]: *Mapping.delete k* (*Mapping xs*) = *Mapping* (*AList.delete k xs*)
⟨*proof*⟩

lemma *ordered-keys-Mapping* [*code*]:
Mapping.ordered-keys (*Mapping xs*) = *sort* (*remdups* (*map fst xs*))
⟨*proof*⟩

lemma *entries-Mapping* [*code*]:
Mapping.entries (*Mapping xs*) = *set* (*AList.clearjunk xs*)
⟨*proof*⟩

lemma *ordered-entries-Mapping* [*code*]:
Mapping.ordered-entries (*Mapping xs*) = *sort-key fst* (*AList.clearjunk xs*)
⟨*proof*⟩

lemma *fold-Mapping* [*code*]:
Mapping.fold f (*Mapping xs*) *a* = *List.fold* (*case-prod f*) (*sort-key fst* (*AList.clearjunk xs*)) *a*
⟨*proof*⟩

lemma *size-Mapping* [*code*]: *Mapping.size* (*Mapping xs*) = *length* (*remdups* (*map*

fst xs))
⟨*proof*⟩

lemma *tabulate-Mapping* [*code*]: *Mapping.tabulate ks f* = *Mapping* (*map* (λ*k*. (*k*, *f k*)) *ks*)
⟨*proof*⟩

lemma *bulkload-Mapping* [*code*]:
Mapping.bulkload vs = *Mapping* (*map* (λ*n*. (*n*, *vs* ! *n*)) [*0*..<*length vs*])
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *filter-Mapping* [*code*]:
Mapping.filter P (*Mapping xs*) = *Mapping* (*filter* (λ(*k*,*v*). *P k v*) (*AList.clearjunk xs*))
⟨*proof*⟩

lemma [*code nbe*]: *HOL.equal* (*x* :: (*'a*, *'b*) *mapping*) *x* ⟷ *True*

⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *char-of-integer-code* [*code*]:
‹*integer-of-char* (*char-of-integer k*) = (*if* 0 ≤ *k* ∧ *k* < 256 *then k else k mod* 256)›
⟨*proof*⟩

lemma *of-char-code* [*code*]:
‹*of-char c* = *of-nat* (*nat-of-integer* (*integer-of-char c*))›
⟨*proof*⟩

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)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *digit-0-code* [*code*]:
‹*digit0 c* ⟷ *bit* (*integer-of-char c*) *0*›
⟨*proof*⟩

lemma *digit-1-code* [*code*]:
‹*digit1 c* ⟷ *bit* (*integer-of-char c*) *1*›
⟨*proof*⟩

lemma *digit-2-code* [*code*]:
‹*digit2 c* ⟷ *bit* (*integer-of-char c*) *2*›
⟨*proof*⟩

lemma *digit-3-code* [*code*]:
‹*digit3 c* ⟷ *bit* (*integer-of-char c*) *3*›
⟨*proof*⟩

lemma *digit-4-code* [*code*]:
‹*digit4 c* ⟷ *bit* (*integer-of-char c*) *4*›
⟨*proof*⟩

lemma *digit-5-code* [*code*]:
‹*digit5 c* ⟷ *bit* (*integer-of-char c*) *5*›
⟨*proof*⟩

lemma *digit-6-code* [*code*]:
‹*digit6 c* ⟷ *bit* (*integer-of-char c*) *6*›
⟨*proof*⟩

lemma *digit-7-code* [*code*]:
‹*digit7 c* ⟷ *bit* (*integer-of-char c*) *7*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma *equal-char-code* [*code*]:
‹*HOL.equal c d* ⟷ *integer-of-char c* = *integer-of-char d*›
⟨*proof*⟩

lemma *less-eq-char-code* [*code*]:
‹*c* ≤ *d* ⟷ *integer-of-char c* ≤ *integer-of-char d*› (**is** ‹*?P* ⟷ *?Q*›)
⟨*proof*⟩

lemma *less-char-code* [*code*]:
‹*c* < *d* ⟷ *integer-of-char c* < *integer-of-char d*› (**is** ‹*?P* ⟷ *?Q*›)
⟨*proof*⟩

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*)›
⟨*proof*⟩

⟨*ML*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

The rule above is built into a preprocessor that is plugged into the code generator.

⟨*ML*⟩

119.3 Candidates which need special treatment

lemma *drop-bit-int-code* [*code*]:
‹*drop-bit n k* = *k div 2* ^ *n*› **for** *k* :: *int*
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*code*]:
(*1*::*nat*) = *Numeral1*
⟨*proof*⟩

lemma [*code-abbrev*]: *Numeral1* = (*1*::*nat*)
⟨*proof*⟩

lemma [*code*]:
Suc n = *n* + *1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩ **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))
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *equal-nat-refl* [*code nbe*]:
HOL.equal (n::nat) n ⟷ True
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

end

120.3 Conversions

lemma *of-nat-code* [*code*]:
of-nat 0 = *0*
of-nat (*nat-of-num k*) = *numeral k*
⟨*proof*⟩

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*⟩

122 Setup for Numerals

⟨*ML*⟩

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*
⟨*proof*⟩

lemma [*code*]:
Int.Pos = *int-of-integer* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code*]:
Int.Neg = *int-of-integer* ∘ *uminus* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (*numeral k*) = *Int.Pos k*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (− *numeral k*) = *Int.Neg k*
⟨*proof*⟩

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*
⟨*proof*⟩

end

lemma [*code*, *symmetric*, *code-post*]:
0 = *int-of-integer 0*
⟨*proof*⟩

lemma [*code*, *symmetric*, *code-post*]:
1 = *int-of-integer 1*
⟨*proof*⟩

lemma [*code-post*]:

int-of-integer (− *1*) = − *1*
⟨*proof*⟩

lemma [*code*]:
k + *l* = *int-of-integer* (*of-int k* + *of-int l*)
⟨*proof*⟩

lemma [*code*]:
− *k* = *int-of-integer* (− *of-int k*)
⟨*proof*⟩

lemma [*code*]:
k − *l* = *int-of-integer* (*of-int k* − *of-int l*)
⟨*proof*⟩

lemma [*code*]:
Int.dup k = *int-of-integer* (*Code-Numeral.dup* (*of-int k*))
⟨*proof*⟩

declare [[*code drop*: *Int.sub*]]

lemma [*code*]:
k ∗ *l* = *int-of-integer* (*of-int k* ∗ *of-int l*)
⟨*proof*⟩

lemma [*code*]:
k div l = *int-of-integer* (*of-int k div of-int l*)
⟨*proof*⟩

lemma [*code*]:
k mod l = *int-of-integer* (*of-int k mod of-int l*)
⟨*proof*⟩

lemma [*code*]:
divmod m n = *map-prod int-of-integer int-of-integer* (*divmod m n*)
⟨*proof*⟩

lemma [*code*]:
HOL.equal k l = *HOL.equal* (*of-int k* :: *integer*) (*of-int l*)
⟨*proof*⟩

lemma [*code*]:
k ≤ *l* ⟷ (*of-int k* :: *integer*) ≤ *of-int l*
⟨*proof*⟩

lemma [*code*]:
k < *l* ⟷ (*of-int k* :: *integer*) < *of-int l*
⟨*proof*⟩

lemma *gcd-int-of-integer* [*code*]:
gcd (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*gcd x y*)
⟨*proof*⟩

lemma *lcm-int-of-integer* [*code*]:
lcm (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*lcm x y*)
⟨*proof*⟩

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*⟩

declare *of-int-code-if* [*code*]

lemma [*code*]:
nat = *nat-of-integer* ∘ *of-int*
including *integer.lifting* ⟨*proof*⟩

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* ⟨*proof*⟩

lemma [*code*]:
int-of-char = *int-of-integer* ∘ *integer-of-char*
including *integer.lifting* ⟨*proof*⟩

context
includes *integer.lifting* **and** *bit-operations-syntax*
begin

lemma [*code*]:
‹*bit* (*int-of-integer k*) *n* ⟷ *bit k n*›
⟨*proof*⟩

lemma [*code*]:
‹*NOT* (*int-of-integer k*) = *int-of-integer* (*NOT k*)›

⟨*proof*⟩

lemma [*code*]:
‹*int-of-integer k AND int-of-integer l = int-of-integer (k AND l)*›
⟨*proof*⟩

lemma [*code*]:
‹*int-of-integer k OR int-of-integer l = int-of-integer (k OR l)*›
⟨*proof*⟩

lemma [*code*]:
‹*int-of-integer k XOR int-of-integer l = int-of-integer (k XOR l)*›
⟨*proof*⟩

lemma [*code*]:
‹*push-bit n (int-of-integer k) = int-of-integer (push-bit n k)*›
⟨*proof*⟩

lemma [*code*]:
‹*drop-bit n (int-of-integer k) = int-of-integer (drop-bit n k)*›
⟨*proof*⟩

lemma [*code*]:
‹*take-bit n (int-of-integer k) = int-of-integer (take-bit n k)*›
⟨*proof*⟩

lemma [*code*]:
‹*mask n = int-of-integer (mask n)*›
⟨*proof*⟩

lemma [*code*]:
‹*set-bit n (int-of-integer k) = int-of-integer (set-bit n k)*›
⟨*proof*⟩

lemma [*code*]:
‹*unset-bit n (int-of-integer k) = int-of-integer (unset-bit n k)*›
⟨*proof*⟩

lemma [*code*]:
‹*flip-bit n (int-of-integer k) = int-of-integer (flip-bit n k)*›
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma [*code-unfold del*]: − *numeral k* ≡ *real-of-rat* (− *numeral k*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩ **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*
⟨*proof*⟩

lemma [*code*]: ‹*inverse* r = *1* / r› **for** r :: *real*
⟨*proof*⟩

lemma [*code*]: ‹*Ratreal* r = (*let* (p, q) = *quotient-of* r *in real-of-int* p / *real-of-int* q)›
⟨*proof*⟩

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
⟨*proof*⟩
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*
⟨*proof*⟩

lemma [*code-post*]:
Nat 0 = 0
Nat 1 = 1
Nat (*numeral k*) = *numeral k*

⟨*proof*⟩

lemma [*code-abbrev*]:
integer-of-nat = *of-nat*
⟨*proof*⟩

lemma [*code-unfold*]:
Int.nat (*int-of-integer k*) = *nat-of-integer k*
⟨*proof*⟩

lemma [*code abstype*]:
Code-Target-Nat.Nat (*integer-of-nat n*) = *n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat-of-integer k*) = *max 0 k*
⟨*proof*⟩

lemma [*code-abbrev*]:
nat-of-integer (*numeral k*) = *nat-of-num k*
⟨*proof*⟩

context
begin

qualified definition *natural* :: *num* ⇒ *nat*
where [*simp*]: *natural* = *nat-of-num*

lemma [*code-computation-unfold*]:
numeral = *natural*
nat-of-num = *natural*
⟨*proof*⟩

end

lemma [*code abstract*]:
integer-of-nat (*nat-of-num n*) = *integer-of-num n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat 0 = *0*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat 1 = *1*
⟨*proof*⟩

lemma [*code*]:
Suc n = *n* + *1*

⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* + *n*) = *of-nat m* + *of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* − *n*) = *max 0* (*of-nat m* − *of-nat n*)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m div n*) = *of-nat m div of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m mod n*) = *of-nat m mod of-nat n*
⟨*proof*⟩

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))
⟨*proof*⟩

end

lemma [*code*]:
divmod m n = *map-prod nat-of-integer nat-of-integer* (*divmod m n*)
⟨*proof*⟩

lemma [*code*]:
HOL.equal m n = *HOL.equal* (*of-nat m* :: *integer*) (*of-nat n*)
⟨*proof*⟩

lemma [*code*]:
m ≤ *n* ⟷ (*of-nat m* :: *integer*) ≤ *of-nat n*
⟨*proof*⟩

lemma [*code*]:
m < *n* ⟷ (*of-nat m* :: *integer*) < *of-nat n*
⟨*proof*⟩

lemma *num-of-nat-code* [*code*]:
num-of-nat = *num-of-integer* ∘ *of-nat*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat k*) = *max 0* (*integer-of-int k*)
including *integer.lifting* ⟨*proof*⟩

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* ⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat-of-char c*) = *integer-of-char c*
⟨*proof*⟩

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*))
⟨*proof*⟩

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* ⟨*proof*⟩

context
includes *integer.lifting* **and** *bit-operations-syntax*
begin

lemma [*code*]:
‹*bit m n* ⟷ *bit* (*integer-of-nat m*) *n*›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*m AND n*) = *integer-of-nat m AND integer-of-nat n*›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*m OR n*) = *integer-of-nat m OR integer-of-nat n*›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*m XOR n*) = *integer-of-nat m XOR integer-of-nat n*›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*push-bit n m*) = *push-bit n* (*integer-of-nat m*)›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*drop-bit n m*) = *drop-bit n* (*integer-of-nat m*)›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*take-bit n m*) = *take-bit n* (*integer-of-nat m*)›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*mask n*) = *mask n*›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*set-bit n m*) = *set-bit n* (*integer-of-nat m*)›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*unset-bit n m*) = *unset-bit n* (*integer-of-nat m*)›
⟨*proof*⟩

lemma [*code*]:
‹*integer-of-nat* (*flip-bit n m*) = *flip-bit n* (*integer-of-nat m*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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
⟨*proof*⟩
end

lemma *nonnegative-complex-is-real*: ‹$(x::complex) \geq 0 \Longrightarrow x \in \mathbb{R}$›
⟨*proof*⟩

lemma *complex-is-real-iff-compare0*: ‹$(x::complex) \in \mathbb{R} \longleftrightarrow x \leq 0 \vee x \geq 0$›
⟨*proof*⟩

instance *complex* :: *ordered-comm-ring*
⟨*proof*⟩

instance *complex* :: *ordered-real-vector*
⟨*proof*⟩

instance *complex* :: *ordered-cancel-comm-semiring*
⟨*proof*⟩

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*))
⟨*proof*⟩

127.2 Type ('*key*, '*value*) *alist*

typedef ('*key*, '*value*) *alist* = {*xs* :: ('*key* × '*value*) *list*. (*distinct* ∘ *map fst*) *xs*}
morphisms *impl-of Alist*

⟨*proof*⟩

setup-lifting *type-definition-alist*

lemma *alist-ext*: *impl-of xs* = *impl-of ys* ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *alist-eq-iff*: *xs* = *ys* ⟷ *impl-of xs* = *impl-of ys*
⟨*proof*⟩

lemma *impl-of-distinct* [*simp*, *intro*]: *distinct* (*map fst* (*impl-of xs*))
⟨*proof*⟩

lemma *impl-of-Alist*:
‹*impl-of* (*Alist xs*) = *xs*› **if** ‹*distinct* (*map fst xs*)›
⟨*proof*⟩

lemma *Alist-impl-of* [*code abstype*]: *Alist* (*impl-of xs*) = *xs*
⟨*proof*⟩

127.3 Primitive operations

lift-definition *lookup* :: ('*key*, '*value*) *alist* ⇒ '*key* ⇒ '*value option* **is** *map-of*
⟨*proof*⟩

lift-definition *empty* :: ('*key*, '*value*) *alist* **is** []
⟨*proof*⟩

lift-definition *update* :: '*key* ⇒ '*value* ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *AList.update*
⟨*proof*⟩

lift-definition *delete* :: '*key* ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *AList.delete*
⟨*proof*⟩

lift-definition *map-entry* ::
'*key* ⇒ ('*value* ⇒ '*value*) ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *AList.map-entry*
⟨*proof*⟩

lift-definition *filter* :: ('*key* × '*value* ⇒ *bool*) ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *List.filter*
⟨*proof*⟩

lift-definition *map-default* ::
'*key* ⇒ '*value* ⇒ ('*value* ⇒ '*value*) ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*

is *AList.map-default*
⟨*proof*⟩

127.4 Abstract operation properties

lemma *lookup-empty* [*simp*]: *lookup empty k* = *None*
⟨*proof*⟩

lemma *lookup-update*:
lookup (*update k1 v xs*) *k2* = (*if k1* = *k2 then Some v else lookup xs k2*)
⟨*proof*⟩

lemma *lookup-update-eq* [*simp*]:
k1 = *k2* ⟹ *lookup* (*update k1 v xs*) *k2* = *Some v*
⟨*proof*⟩

lemma *lookup-update-neq* [*simp*]:
k1 ≠ *k2* ⟹ *lookup* (*update k1 v xs*) *k2* = *lookup xs k2*
⟨*proof*⟩

lemma *update-update-eq* [*simp*]:
k1 = *k2* ⟹ *update k2 v2* (*update k1 v1 xs*) = *update k2 v2 xs*
⟨*proof*⟩

lemma *lookup-delete* [*simp*]: *lookup* (*delete k al*) = (*lookup al*)(*k* := *None*)
⟨*proof*⟩

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
⟨*proof*⟩

end

127.5.2 Size

instantiation *alist* :: (*type*, *type*) *size*
begin

definition *size* (*al* :: ('*a*, '*b*) *alist*) = *length* (*impl-of al*)

instance ⟨*proof*⟩

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 ⟨*proof*⟩

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
(λa. Quickcheck-Exhaustive.exhaustive
(λk. Quickcheck-Exhaustive.exhaustive (λv. f (update k v a)) (i − 1))
(i − 1))
(i − 1))

instance ⟨*proof*⟩

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
(λa.
Quickcheck-Exhaustive.full-exhaustive
(λk. Quickcheck-Exhaustive.full-exhaustive (λv. f (valterm-update k v
a)) (i − 1))
(i − 1))
(i − 1))

instance ⟨*proof*⟩

end

128 alist is a BNF

lift-bnf (*dead 'k*, *set*: *'v*) *alist* [*wits*: [] :: (*'k* × *'v*) *list*] **for** *map*: *map rel*: *rel*
⟨*proof*⟩

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* ::
  $('key \Rightarrow 'val \times 'val \Rightarrow 'val) \Rightarrow$
    $('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
 where *join-raw f xs ys* = *foldr* $(\lambda(k, v).$ *map-default k v* $(\lambda v'.$ *f k* $(v', v)))$ *ys xs*

lemma *join-raw-Nil* [*simp*]: *join-raw f xs* [] = *xs*
 ⟨*proof*⟩

lemma *join-raw-Cons* [*simp*]:
 join-raw f xs $((k, v)$ # *ys*) = *map-default k v* $(\lambda v'.$ *f k* $(v', v))$ (*join-raw f xs ys*)
 ⟨*proof*⟩

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'*))))
 ⟨*proof*⟩

lemma *distinct-join-raw*:
 assumes *distinct* (*map fst xs*)
 shows *distinct* (*map fst* (*join-raw f xs ys*))
 ⟨*proof*⟩

definition *subtract-entries-raw xs ys* = *foldr* $(\lambda(k, v).$ *AList.map-entry k* $(\lambda v'.$ *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'*)))
⟨*proof*⟩

lemma *distinct-subtract-entries-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*subtract-entries-raw xs ys*))
⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *subtract-entries* :: ('*a*, ('*b* :: *minus*)) *alist* ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist*
is *subtract-entries-raw*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *count-of-empty*: *x* ∉ *fst* ' *set xs* ⟹ *count-of xs x* = *0*
⟨*proof*⟩

lemma *count-of-filter*: *count-of* (*List.filter* (*P* ∘ *fst*) *xs*) *x* = (*if P x then count-of xs x else 0*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Bag-eq*:
‹*Bag ms* = (∑ (*a*, *n*)←*alist.impl-of ms*. *replicate-mset n a*)›
for *ms* :: ‹('*a*, *nat*) *alist*›
⟨*proof*⟩

lemma *Mempty-Bag* [*code*]: {#} = *Bag* (*DAList.empty*)
⟨*proof*⟩

lift-definition *is-empty-Bag-impl* :: ('*a*, *nat*) *alist* ⇒ *bool* **is**
λ*xs*. *list-all* (λ*x*. *snd x* = *0*) *xs* ⟨*proof*⟩

lemma *is-empty-Bag* [*code*]: *Multiset.is-empty* (*Bag xs*) ⟷ *is-empty-Bag-impl xs*
⟨*proof*⟩

lemma *union-Bag* [*code*]: *Bag xs* + *Bag ys* = *Bag* (*join* (λ*x* (*n1*, *n2*). *n1* + *n2*)
xs ys)
⟨*proof*⟩

lemma *add-mset-Bag* [*code*]: *add-mset x* (*Bag xs*) =
Bag (*join* (λ*x* (*n1*, *n2*). *n1* + *n2*) (*DAList.update x 1 DAList.empty*) *xs*)
⟨*proof*⟩

lemma *minus-Bag* [*code*]: *Bag xs* − *Bag ys* = *Bag* (*subtract-entries xs ys*)
⟨*proof*⟩

lemma *filter-Bag* [*code*]: *filter-mset P* (*Bag xs*) = *Bag* (*DAList.filter* (*P* ∘ *fst*) *xs*)
⟨*proof*⟩

lemma *mset-eq* [*code*]: *HOL.equal* (*m1*::'*a*::*equal multiset*) *m2* ⟷ *m1* ⊆# *m2* ∧
m2 ⊆# *m1*
⟨*proof*⟩

By default the code for < is (*xs* < *ys*) = (*xs* ≤ *ys* ∧ *xs* ≠ *ys*). With equality implemented by ≤, this leads to three calls of ≤. Here is a more efficient version:

lemma *mset-less*[*code*]: *xs* ⊂# (*ys* :: *'a multiset*) ⟷ *xs* ⊆# *ys* ∧ ¬ *ys* ⊆# *xs*
⟨*proof*⟩

lemma *mset-less-eq-Bag0*:
Bag xs ⊆# *A* ⟷ (∀(*x*, *n*) ∈ *set* (*DAList.impl-of xs*). *count-of* (*DAList.impl-of xs*) *x* ≤ *count A x*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *mset-less-eq-Bag* [*code*]:
Bag xs ⊆# (*A* :: *'a multiset*) ⟷ (∀(*x*, *n*) ∈ *set* (*DAList.impl-of xs*). *n* ≤ *count A x*)
⟨*proof*⟩

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*
⟨*proof*⟩

end

context
begin

private lift-definition *single-alist-entry* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *alist* **is** λ*a b*. [(*a*, *b*)]
⟨*proof*⟩

lemma *image-mset-Bag* [*code*]:
image-mset f (*Bag ms*) =
DAList-Multiset.fold (λ*a n m*. *Bag* (*single-alist-entry* (*f a*) *n*) + *m*) {#} *ms*
⟨*proof*⟩

end

— we cannot use λ*a 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* (λ*a n*. (((+) *a*) ^^ *n*)) *0 ms*
⟨*proof*⟩
lemma *prod-mset-Bag*[*code*]: *prod-mset* (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. (((∗) *a*) ^^ *n*)) *1 ms*
⟨*proof*⟩

lemma *size-fold*: *size A* = *fold-mset* (λ-. *Suc*) *0 A* (**is** - = *fold-mset ?f* - -)
⟨*proof*⟩

lemma *size-Bag*[*code*]: *size* (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. (+) *n*) *0 ms*
⟨*proof*⟩

lemma *set-mset-fold*: *set-mset A* = *fold-mset insert* {} *A* (**is** - = *fold-mset ?f* - -)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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 ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *entry-in-tree-keys*:
  assumes (*k*, *v*) ∈ *set* (*entries t*)
  shows *k* ∈ *set* (*keys t*)
⟨*proof*⟩

lemma *keys-entries*:
  k ∈ *set* (*keys t*) ⟷ (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
  ⟨*proof*⟩

lemma *non-empty-rbt-keys*:
$t \neq rbt.Empty \Longrightarrow keys\ t \neq []$
⟨*proof*⟩

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** ‹|«› *50*)
where $t\ |\!\ll\ x \equiv rbt\text{-}less\ x\ t$

definition *rbt-greater* :: $'a \Rightarrow ('a, 'b)\ rbt \Rightarrow bool$ (**infix** ‹«|› *50*)
where
rbt-greater-prop: $rbt\text{-}greater\ k\ t = (\forall x{\in}set\ (keys\ t).\ k < x)$

lemma *rbt-less-simps* [*simp*]:
$Empty\ |\!\ll\ k = True$
$Branch\ c\ lt\ kt\ v\ rt\ |\!\ll\ k \longleftrightarrow kt < k \wedge lt\ |\!\ll\ k \wedge rt\ |\!\ll\ k$
⟨*proof*⟩

lemma *rbt-greater-simps* [*simp*]:
$k\ «\!|\ Empty = True$
$k\ «\!|\ (Branch\ c\ lt\ kt\ v\ rt) \longleftrightarrow k < kt \wedge k\ «\!|\ lt \wedge k\ «\!|\ rt$
⟨*proof*⟩

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*: $u \leq v \Longrightarrow v\ «\!|\ r \Longrightarrow u\ «\!|\ r$
and *rbt-greater-trans*: $x < y \Longrightarrow y\ «\!|\ t \Longrightarrow x\ «\!|\ t$
⟨*proof*⟩

primrec *rbt-sorted* :: $('a, 'b)\ rbt \Rightarrow bool$
where
$rbt\text{-}sorted\ Empty = True$
| $rbt\text{-}sorted\ (Branch\ c\ l\ k\ v\ r) = (l\ |\!\ll\ k \wedge k\ «\!|\ r \wedge rbt\text{-}sorted\ l \wedge rbt\text{-}sorted\ r)$

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:
rbt-sorted t $\Longrightarrow$ *List.sorted* (*map fst* (*entries t*))
⟨*proof*⟩

lemma *distinct-entries*:
rbt-sorted t $\Longrightarrow$ *distinct* (*map fst* (*entries t*))
⟨*proof*⟩

lemma *distinct-keys*:
rbt-sorted t $\Longrightarrow$ *distinct* (*keys t*)
⟨*proof*⟩

130.2.3 Tree lookup

primrec (**in** *ord*) *rbt-lookup* :: ('*a*, '*b*) *rbt* $\Rightarrow$ '*a* $\rightharpoonup$ '*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* $\Longrightarrow$ *dom* (*rbt-lookup t*) = *set* (*keys t*)
⟨*proof*⟩

lemma *dom-rbt-lookup-Branch*:
rbt-sorted (*Branch c t1 k v t2*) $\Longrightarrow$
dom (*rbt-lookup* (*Branch c t1 k v t2*))
= *Set.insert k* (*dom* (*rbt-lookup t1*) ∪ *dom* (*rbt-lookup t2*))
⟨*proof*⟩

lemma *finite-dom-rbt-lookup* [*simp*, *intro*!]: *finite* (*dom* (*rbt-lookup t*))
⟨*proof*⟩

end

context *ord* **begin**

lemma *rbt-lookup-rbt-less*[*simp*]: *t* |« *k* $\Longrightarrow$ *rbt-lookup t k* = *None*
⟨*proof*⟩

lemma *rbt-lookup-rbt-greater*[*simp*]: *k* «| *t* $\Longrightarrow$ *rbt-lookup t k* = *None*
⟨*proof*⟩

lemma *rbt-lookup-Empty*: *rbt-lookup Empty* = *Map.empty*
⟨*proof*⟩

end

context *linorder* **begin**

lemma *map-of-entries*:
rbt-sorted t $\Longrightarrow$ *map-of* (*entries t*) = *rbt-lookup t*
⟨*proof*⟩

lemma *rbt-lookup-in-tree*: *rbt-sorted t* $\Longrightarrow$ *rbt-lookup t k* = *Some v* $\longleftrightarrow$ (*k*, *v*) ∈ *set* (*entries t*)
⟨*proof*⟩

lemma *set-entries-inject*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*
shows *set* (*entries t1*) = *set* (*entries t2*) $\longleftrightarrow$ *entries t1* = *entries t2*
⟨*proof*⟩

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*⟩

lemma *entries-rbt-lookup*:
assumes *rbt-sorted t1 rbt-sorted t2*
shows *entries t1* = *entries t2* $\longleftrightarrow$ *rbt-lookup t1* = *rbt-lookup t2*
⟨*proof*⟩

lemma *rbt-lookup-from-in-tree*:
assumes *rbt-sorted t1 rbt-sorted t2*
and $\bigwedge$*v*. (*k*, *v*) ∈ *set* (*entries t1*) $\longleftrightarrow$ (*k*, *v*) ∈ *set* (*entries t2*)
shows *rbt-lookup t1 k* = *rbt-lookup t2 k*
⟨*proof*⟩

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*) $\longleftrightarrow$ *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* ⟨*proof*⟩

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* ⟨*proof*⟩

theorem *Empty-is-rbt* [*simp*]:
is-rbt Empty ⟨*proof*⟩

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*⟧ ⟹ *inv1* (*balance l k v r*)
⟨*proof*⟩

lemma *balance-bheight*: *bheight l* = *bheight r* ⟹ *bheight* (*balance l k v r*) = *Suc* (*bheight l*)

⟨*proof*⟩

lemma *balance-inv2*:
assumes *inv2 l inv2 r bheight l = bheight r*
shows *inv2 (balance l k v r)*
⟨*proof*⟩

context *ord* **begin**

lemma *balance-rbt-greater*[*simp*]: *(v «| balance a k x b) = (v «| a ∧ v «| b ∧ v < k)*
⟨*proof*⟩

lemma *balance-rbt-less*[*simp*]: *(balance a k x b |« v) = (a |« v ∧ b |« v ∧ k < v)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *entries-balance* [*simp*]:
entries (balance l k v r) = entries l @ (k, v) # entries r
⟨*proof*⟩

lemma *keys-balance* [*simp*]:
keys (balance l k v r) = keys l @ k # keys r
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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)* ⟨*proof*⟩

lemma *paint-inv1*[*simp*]: *inv1l t* $\Longrightarrow$ *inv1* (*paint B t*) ⟨*proof*⟩
lemma *paint-inv2*[*simp*]: *inv2 t* $\Longrightarrow$ *inv2* (*paint c t*) ⟨*proof*⟩
lemma *paint-color-of*[*simp*]: *color-of* (*paint B t*) = *B* ⟨*proof*⟩
lemma *paint-in-tree*[*simp*]: *entry-in-tree k x* (*paint c t*) = *entry-in-tree k x t* ⟨*proof*⟩

context *ord* **begin**

lemma *paint-rbt-sorted*[*simp*]: *rbt-sorted t* $\Longrightarrow$ *rbt-sorted* (*paint c t*) ⟨*proof*⟩
lemma *paint-rbt-lookup*[*simp*]: *rbt-lookup* (*paint c t*) = *rbt-lookup t* ⟨*proof*⟩
lemma *paint-rbt-greater*[*simp*]: (*v* «| *paint c t*) = (*v* «| *t*) ⟨*proof*⟩
lemma *paint-rbt-less*[*simp*]: (*paint c t* |« *v*) = (*t* |« *v*) ⟨*proof*⟩

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* $\Longrightarrow$ *inv1* (*rbt-ins f k x t*) *inv1l* (*rbt-ins f k x t*)
⟨*proof*⟩

end

context *linorder* **begin**

lemma *ins-rbt-greater*[*simp*]: (*v* «| *rbt-ins f* (*k* :: '*a*) *x t*) = (*v* «| *t* ∧ *k* > *v*)
⟨*proof*⟩
lemma *ins-rbt-less*[*simp*]: (*rbt-ins f k x t* |« *v*) = (*t* |« *v* ∧ *k* < *v*)
⟨*proof*⟩
lemma *ins-rbt-sorted*[*simp*]: *rbt-sorted t* $\Longrightarrow$ *rbt-sorted* (*rbt-ins f k x t*)
⟨*proof*⟩

lemma *keys-ins*: *set* (*keys* (*rbt-ins f k v t*)) = { *k* } ∪ *set* (*keys t*)
⟨*proof*⟩

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*

⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *rbt-insertwk-is-rbt*:
assumes *inv*: *is-rbt t*
shows *is-rbt* (*rbt-insert-with-key f k x t*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rbt-insertw-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert-with f k v t*)
⟨*proof*⟩
theorem *rbt-insertw-is-rbt*: *is-rbt t* ⟹ *is-rbt* (*rbt-insert-with f k v t*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbt-insert-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert k v t*)
⟨*proof*⟩

theorem *rbt-insert-is-rbt* [*simp*]: *is-rbt t* ⟹ *is-rbt* (*rbt-insert k v t*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-insert*: *is-rbt t* ⟹ *rbt-lookup* (*rbt-insert k v t*) = (*rbt-lookup t*)(*k*↦*v*)
⟨*proof*⟩

end

130.4 Deletion

lemma *bheight-paintR'*[*simp*]: *color-of t* = *B* ⟹ *bheight* (*paint R t*) = *bheight t* − *1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *balance-left-inv1*: ⟦*inv1l a*; *inv1 b*; *color-of b* = *B*⟧ ⟹ *inv1* (*balance-left a k x b*)
⟨*proof*⟩

lemma *balance-left-inv1l*: ⟦ *inv1l lt*; *inv1 rt* ⟧ ⟹ *inv1l* (*balance-left lt k x rt*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *balance-left-rbt-less*:
fixes $k :: 'a$
assumes a |« k b |« k $x < k$
shows *balance-left* a x t b |« k
⟨*proof*⟩

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 $l \vee k = a \wedge v = b \vee$ *entry-in-tree* k v r)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *balance-right-inv1*: ⟦*inv1* a; *inv1l* b; *color-of* $a = B$⟧ ⟹ *inv1* (*balance-right* a k x b)
⟨*proof*⟩

lemma *balance-right-inv1l*: ⟦ *inv1* lt; *inv1l* rt ⟧ ⟹*inv1l* (*balance-right* lt k x rt)
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *balance-right-rbt-less*:
fixes $k :: 'a$
assumes a |« k b |« k $x < k$
shows *balance-right* a x t b |« k
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

context *linorder* **begin**

lemma *combine-rbt-greater*[*simp*]:
fixes $k :: 'a$
assumes k «| l k «| r
shows k «| *combine* l r
⟨*proof*⟩

lemma *combine-rbt-less*[*simp*]:
fixes $k :: 'a$
assumes l |« k r |« k
shows *combine* l r |« k
⟨*proof*⟩

lemma *combine-rbt-sorted*:
fixes $k :: 'a$
assumes *rbt-sorted* l *rbt-sorted* r l |« k k «| r
shows *rbt-sorted* (*combine* l r)
⟨*proof*⟩

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)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma
rbt-del-from-left-rbt-less: ⟦ *lt* |« *v*; *rt* |« *v*; *k* < *v*⟧ ⟹ *rbt-del-from-left x lt k y rt* |« *v*
and *rbt-del-from-right-rbt-less*: ⟦*lt* |« *v*; *rt* |« *v*; *k* < *v*⟧ ⟹ *rbt-del-from-right x lt k y rt* |« *v*
and *rbt-del-rbt-less*: *lt* |« *v* ⟹ *rbt-del x lt* |« *v*
⟨*proof*⟩

lemma *rbt-del-from-left-rbt-greater*: ⟦*v* «| *lt*; *v* «| *rt*; *k* > *v*⟧ ⟹ *v* «| *rbt-del-from-left x lt k y rt*
and *rbt-del-from-right-rbt-greater*: ⟦*v* «| *lt*; *v* «| *rt*; *k* > *v*⟧ ⟹ *v* «| *rbt-del-from-right x lt k y rt*
and *rbt-del-rbt-greater*: *v* «| *lt* ⟹ *v* «| *rbt-del x lt*
⟨*proof*⟩

lemma ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |« *k*; *k* «| *rt*⟧ ⟹ *rbt-sorted* (*rbt-del-from-left x lt k y rt*)
and ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |« *k*; *k* «| *rt*⟧ ⟹ *rbt-sorted* (*rbt-del-from-right x lt k y rt*)
and *rbt-del-rbt-sorted*: *rbt-sorted lt* ⟹ *rbt-sorted* (*rbt-del x lt*)
⟨*proof*⟩

lemma ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |« *kt*; *kt* «| *rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* = *bheight rt*; *x* < *kt*⟧ ⟹ *entry-in-tree k v* (*rbt-del-from-left x lt kt y rt*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |« *kt*; *kt* «| *rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* = *bheight rt*; *x* > *kt*⟧ ⟹ *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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-delete*:
assumes *is-rbt*: *is-rbt t*
shows *rbt-lookup* (*rbt-delete k t*) = (*rbt-lookup t*)|‘(−{*k*})
⟨*proof*⟩

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* ⟨*proof*⟩
lemma *rbt-map-entry-inv1*: *inv1* (*rbt-map-entry k f t*) = *inv1 t* ⟨*proof*⟩
lemma *rbt-map-entry-inv2*: *inv2* (*rbt-map-entry k f t*) = *inv2 t bheight* (*rbt-map-entry k f t*) = *bheight t* ⟨*proof*⟩
lemma *rbt-map-entry-rbt-greater*: *rbt-greater a* (*rbt-map-entry k f t*) = *rbt-greater a t* ⟨*proof*⟩
lemma *rbt-map-entry-rbt-less*: *rbt-less a* (*rbt-map-entry k f t*) = *rbt-less a t* ⟨*proof*⟩
lemma *rbt-map-entry-rbt-sorted*: *rbt-sorted* (*rbt-map-entry k f t*) = *rbt-sorted t*
⟨*proof*⟩

theorem *rbt-map-entry-is-rbt* [*simp*]: *is-rbt* (*rbt-map-entry k f t*) = *is-rbt t*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩
lemma *map-keys* [*simp*]: *keys* (*map f t*) = *keys t* ⟨*proof*⟩
lemma *map-color-of*: *color-of* (*map f t*) = *color-of t* ⟨*proof*⟩
lemma *map-inv1*: *inv1* (*map f t*) = *inv1 t* ⟨*proof*⟩
lemma *map-inv2*: *inv2* (*map f t*) = *inv2 t bheight* (*map f t*) = *bheight t* ⟨*proof*⟩

context *ord* **begin**

lemma *map-rbt-greater*: *rbt-greater k* (*map f t*) = *rbt-greater k t* ⟨*proof*⟩
lemma *map-rbt-less*: *rbt-less k* (*map f t*) = *rbt-less k t* ⟨*proof*⟩
lemma *map-rbt-sorted*: *rbt-sorted* (*map f t*) = *rbt-sorted t* ⟨*proof*⟩
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map f t*) = *is-rbt t*
⟨*proof*⟩

end

theorem (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup* (*map f t*) *x* = *map-option* (*f x*) (*rbt-lookup t x*)
⟨*proof*⟩

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*
⟨*proof*⟩

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))
⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-bulkload*:
rbt-lookup (*rbt-bulkload xs*) = *map-of xs*
⟨*proof*⟩

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'))
⟨proof⟩

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'))
⟨proof⟩

lemma Suc-double-half: Suc (2 * n) div 2 = n
⟨proof⟩

lemma div2-plus-div2: n div 2 + n div 2 = (n :: nat) - n mod 2
⟨proof⟩

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
⟨proof⟩
```

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'*))
⟨*proof*⟩

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'*))
⟨*proof*⟩

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*⟩

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*⟩

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))
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *bheight-rbtreeify-f-eq-plog2I:*
⟦ rbtreeify-f n kvs = (t, kvs'); n ≤ length kvs ⟧
⟹ bheight t = plog2 n
⟨*proof*⟩

lemma *bheight-rbtreeify-g-eq-plog2I:*
⟦ rbtreeify-g n kvs = (t, kvs'); n ≤ Suc (length kvs) ⟧
⟹ bheight t = plog2 n
⟨*proof*⟩

hide-const (**open**) *plog2*

lemma *inv2-rbtreeify-f*: $n \leq$ *length kvs*
$\Longrightarrow$ *inv2* (*fst* (*rbtreeify-f n kvs*))
and *inv2-rbtreeify-g*: $n \leq$ *Suc* (*length kvs*)
$\Longrightarrow$ *inv2* (*fst* (*rbtreeify-g n kvs*))
⟨*proof*⟩

lemma $n \leq$ *length kvs* $\Longrightarrow$ *True*
and *color-of-rbtreeify-g*:
⟦ $n \leq$ *Suc* (*length kvs*); $0 < n$ ⟧
$\Longrightarrow$ *color-of* (*fst* (*rbtreeify-g n kvs*)) = *B*
⟨*proof*⟩

lemma *entries-rbtreeify-f-append*:
$n \leq$ *length kvs*
$\Longrightarrow$ *entries* (*fst* (*rbtreeify-f n kvs*)) @ *snd* (*rbtreeify-f n kvs*) = *kvs*
and *entries-rbtreeify-g-append*:
$n \leq$ *Suc* (*length kvs*)
$\Longrightarrow$ *entries* (*fst* (*rbtreeify-g n kvs*)) @ *snd* (*rbtreeify-g n kvs*) = *kvs*
⟨*proof*⟩

lemma *length-entries-rbtreeify-f*:
$n \leq$ *length kvs* $\Longrightarrow$ *length* (*entries* (*fst* (*rbtreeify-f n kvs*))) = *n*
and *length-entries-rbtreeify-g*:
$n \leq$ *Suc* (*length kvs*) $\Longrightarrow$ *length* (*entries* (*fst* (*rbtreeify-g n kvs*))) = $n - 1$
⟨*proof*⟩

lemma *rbtreeify-f-conv-drop*:
$n \leq$ *length kvs* $\Longrightarrow$ *snd* (*rbtreeify-f n kvs*) = *drop n kvs*
⟨*proof*⟩

lemma *rbtreeify-g-conv-drop*:
$n \leq$ *Suc* (*length kvs*) $\Longrightarrow$ *snd* (*rbtreeify-g n kvs*) = *drop* ($n - 1$) *kvs*
⟨*proof*⟩

lemma *entries-rbtreeify-f* [*simp*]:
$n \leq$ *length kvs* $\Longrightarrow$ *entries* (*fst* (*rbtreeify-f n kvs*)) = *take n kvs*
⟨*proof*⟩

lemma *entries-rbtreeify-g* [*simp*]:
$n \leq$ *Suc* (*length kvs*) $\Longrightarrow$
entries (*fst* (*rbtreeify-g n kvs*)) = *take* ($n - 1$) *kvs*
⟨*proof*⟩

lemma *keys-rbtreeify-f* [*simp*]: $n \leq$ *length kvs*
$\Longrightarrow$ *keys* (*fst* (*rbtreeify-f n kvs*)) = *take n* (*map fst kvs*)
⟨*proof*⟩

lemma *keys-rbtreeify-g* [*simp*]: $n \leq Suc\ (length\ kvs)$
$\Longrightarrow keys\ (fst\ (rbtreeify\text{-}g\ n\ kvs)) = take\ (n - 1)\ (map\ fst\ kvs)$
⟨*proof*⟩

lemma *rbtreeify-fD*:
$[\![\ rbtreeify\text{-}f\ n\ kvs = (t,\ kvs');\ n \leq length\ kvs\ ]\!]$
$\Longrightarrow entries\ t = take\ n\ kvs \wedge kvs' = drop\ n\ kvs$
⟨*proof*⟩

lemma *rbtreeify-gD*:
$[\![\ rbtreeify\text{-}g\ n\ kvs = (t,\ kvs');\ n \leq Suc\ (length\ kvs)\ ]\!]$
$\Longrightarrow entries\ t = take\ (n - 1)\ kvs \wedge kvs' = drop\ (n - 1)\ kvs$
⟨*proof*⟩

lemma *entries-rbtreeify* [*simp*]: $entries\ (rbtreeify\ kvs) = kvs$
⟨*proof*⟩

context *linorder* **begin**

lemma *rbt-sorted-rbtreeify-f*:
$[\![\ n \leq length\ kvs;\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!]$
$\Longrightarrow rbt\text{-}sorted\ (fst\ (rbtreeify\text{-}f\ n\ kvs))$
and *rbt-sorted-rbtreeify-g*:
$[\![\ n \leq Suc\ (length\ kvs);\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!]$
$\Longrightarrow rbt\text{-}sorted\ (fst\ (rbtreeify\text{-}g\ n\ kvs))$
⟨*proof*⟩

lemma *rbt-sorted-rbtreeify*:
$[\![\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!] \Longrightarrow rbt\text{-}sorted\ (rbtreeify\ kvs)$
⟨*proof*⟩

lemma *is-rbt-rbtreeify*:
$[\![\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!]$
$\Longrightarrow is\text{-}rbt\ (rbtreeify\ kvs)$
⟨*proof*⟩

lemma *rbt-lookup-rbtreeify*:
$[\![\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!] \Longrightarrow$
$rbt\text{-}lookup\ (rbtreeify\ kvs) = map\text{-}of\ kvs$
⟨*proof*⟩

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* :: $('a,\ 'b)\ rbt \Rightarrow ('a,\ 'b)\ rbt$
where
$skip\text{-}red\ (Branch\ color.R\ l\ k\ v\ r) = l$
| $skip\text{-}red\ t = t$

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*
⟨*proof*⟩
termination ⟨*proof*⟩

function *sinter-with* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \times 'b)$ *list* $\Rightarrow ('a \times 'b)$ *list* $\Rightarrow$ $('a \times 'b)$ *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* - [] = []
⟨*proof*⟩
termination ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sorted-sunion-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *sorted* (*map fst* (*sunion-with f xs ys*))
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *set-fst-sinter-with-subset1*:
set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst xs*)
⟨*proof*⟩

lemma *set-fst-sinter-with-subset2*:
set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst ys*)
⟨*proof*⟩

lemma *sorted-sinter-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *sorted* (*map fst* (*sinter-with f xs ys*))
⟨*proof*⟩

lemma *distinct-sinter-with* [*simp*]:
⟦ *distinct* (*map fst xs*); *distinct* (*map fst ys*) ⟧
⟹ *distinct* (*map fst* (*sinter-with f xs ys*))
⟨*proof*⟩

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*))
⟨*proof*⟩

end

lemma *distinct-map-of-rev*: *distinct* (*map fst xs*) ⟹ *map-of* (*rev xs*) = *map-of xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *set-map-filter*: *set* (*List.map-filter P xs*) = *the* ‘ (*P* ‘ *set xs* − {*None*})
⟨*proof*⟩

definition *is-rbt-empty* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where**
is-rbt-empty t ⟷ (*case t of RBT-Impl.Empty* ⇒ *True* | - ⇒ *False*)

lemma *is-rbt-empty-prop*[*simp*]: *is-rbt-empty t* ⟷ *t* = *RBT-Impl.Empty*
⟨*proof*⟩

definition *small-rbt* :: ('a, 'b) *rbt* ⇒ *bool* **where**
small-rbt t ⟷ *bheight t* < *4*

definition *flip-rbt* :: ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* ⇒ *bool* **where**
flip-rbt t1 t2 ⟷ *bheight t2* < *bheight t1*

abbreviation (*input*) *MR* **where** *MR l a b r* ≡ *Branch RBT-Impl.R l a b r*
abbreviation (*input*) *MB* **where** *MB l a b r* ≡ *Branch RBT-Impl.B l a b r*

fun *rbt-baliL* :: ('a, 'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('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* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('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* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('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* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('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* ⇒ ('a, 'b) *rbt* ⇒ ('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 ⇒ (*MR* (*MR t1 a b u2*) *a' b'* (*MR u3 a'' b'' t4*))
| *t23* ⇒ *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*
⟨*proof*⟩

lemma *size-baliL*[*simp*]: *size* (*rbt-baliL t1 a b t2*) = *Suc* (*size t1* + *size t2*)
⟨*proof*⟩

lemma *size-baliR*[*simp*]: *size* (*rbt-baliR t1 a b t2*) = *Suc* (*size t1* + *size t2*)
⟨*proof*⟩

lemma *size-baldL*[*simp*]: *size* (*rbt-baldL t1 a b t2*) = *Suc* (*size t1* + *size t2*)
⟨*proof*⟩

lemma *size-baldR*[*simp*]: *size* (*rbt-baldR t1 a b t2*) = *Suc* (*size t1* + *size t2*)
⟨*proof*⟩

lemma *size-rbt-app*[*simp*]: *size* (*rbt-app t1 t2*) = *size t1* + *size t2*
⟨*proof*⟩

lemma *size-rbt-joinL*[*simp*]: *size* (*rbt-joinL t1 a b t2*) = *Suc* (*size t1* + *size t2*)
⟨*proof*⟩

lemma *size-rbt-joinR*[*simp*]: *size* (*rbt-joinR t1 a b t2*) = *Suc* (*size t1* + *size t2*)
⟨*proof*⟩

lemma *size-rbt-join*[*simp*]: *size* (*rbt-join t1 a b t2*) = *Suc* (*size t1* + *size t2*)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *paint2*: *paint c2* (*paint c1 t*) = *paint c2 t*
⟨*proof*⟩

lemma *inv1-rbt-baliL*: *inv1l l* ⟹ *inv1 r* ⟹ *inv1* (*rbt-baliL l a b r*)
⟨*proof*⟩

lemma *inv1-rbt-baliR*: *inv1 l* ⟹ *inv1l r* ⟹ *inv1* (*rbt-baliR l a b r*)
⟨*proof*⟩

lemma *rbt-bheight-rbt-baliL*: *bheight l* = *bheight r* ⟹ *bheight* (*rbt-baliL l a b r*) = *Suc* (*bheight l*)
⟨*proof*⟩

lemma *rbt-bheight-rbt-baliR*: *bheight l* = *bheight r* ⟹ *bheight* (*rbt-baliR l a b r*) = *Suc* (*bheight l*)
⟨*proof*⟩

lemma *inv2-rbt-baliL*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* = *bheight r* ⟹ *inv2* (*rbt-baliL l a b r*)
⟨*proof*⟩

lemma *inv2-rbt-baliR*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* = *bheight r* ⟹ *inv2* (*rbt-baliR l a b r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *inv2-rbt-baldL-inv1*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* + *1* = *bheight r* ⟹ *inv1 r* ⟹
inv2 (*rbt-baldL l a b r*) ∧ *bheight* (*rbt-baldL l a b r*) = *bheight r*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inv1-rbt-baldL*: *inv1l l* $\Longrightarrow$ *inv1 r* $\Longrightarrow$ *color-of r = RBT-Impl.B* $\Longrightarrow$ *inv1* (*rbt-baldL l a b r*)
⟨*proof*⟩

lemma *inv1lI*: *inv1 t* $\Longrightarrow$ *inv1l t*
⟨*proof*⟩

lemma *neq-Black*[*simp*]: (*c* $\neq$ *RBT-Impl.B*) = (*c = RBT-Impl.R*)
⟨*proof*⟩

lemma *inv1l-rbt-baldL*: *inv1l l* $\Longrightarrow$ *inv1 r* $\Longrightarrow$ *inv1l* (*rbt-baldL l a b r*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inv1-rbt-baldR*: *inv1 l* $\Longrightarrow$ *inv1l r* $\Longrightarrow$ *color-of l = RBT-Impl.B* $\Longrightarrow$ *inv1* (*rbt-baldR l a b r*)
⟨*proof*⟩

lemma *inv1l-rbt-baldR*: *inv1 l* $\Longrightarrow$ *inv1l r* $\Longrightarrow$*inv1l* (*rbt-baldR l a b r*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*) $\wedge$ *bheight* (*rbt-baldL l a b r*) = *bheight r* $\wedge$
inv1l (*rbt-baldL l a b r*) $\wedge$ (*color-of r = RBT-Impl.B* $\longrightarrow$ *inv1* (*rbt-baldL l a b r*))
⟨*proof*⟩

lemma *inv-rbt-baldR*: *inv2 l* $\Longrightarrow$ *inv2 r* $\Longrightarrow$ *bheight l = bheight r + 1* $\Longrightarrow$ *inv1 l* $\Longrightarrow$ *inv1l r* $\Longrightarrow$
inv2 (*rbt-baldR l a b r*) $\wedge$ *bheight* (*rbt-baldR l a b r*) = *bheight l* $\wedge$
inv1l (*rbt-baldR l a b r*) $\wedge$ (*color-of l = RBT-Impl.B* $\longrightarrow$ *inv1* (*rbt-baldR l a b*

r))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *bheight-rbt-joinL*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* ≤ *bheight r* ⟹
bheight (*rbt-joinL l a b r*) = *bheight r*
⟨*proof*⟩

lemma *inv2-rbt-joinL*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* ≤ *bheight r* ⟹ *inv2* (*rbt-joinL l a b r*)
⟨*proof*⟩

lemma *bheight-rbt-joinR*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* ≥ *bheight r* ⟹
bheight (*rbt-joinR l a b r*) = *bheight l*
⟨*proof*⟩

lemma *inv2-rbt-joinR*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* ≥ *bheight r* ⟹ *inv2* (*rbt-joinR l a b r*)
⟨*proof*⟩

lemma *keys-paint*[*simp*]: *RBT-Impl.keys* (*paint c t*) = *RBT-Impl.keys t*
⟨*proof*⟩

lemma *keys-rbt-baliL*: *RBT-Impl.keys* (*rbt-baliL l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
⟨*proof*⟩

lemma *keys-rbt-baliR*: *RBT-Impl.keys* (*rbt-baliR l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
⟨*proof*⟩

lemma *keys-rbt-baldL*: *RBT-Impl.keys* (*rbt-baldL l a b r*) = *RBT-Impl.keys l* @ *a*

 # RBT-Impl.keys r
⟨*proof*⟩

lemma *keys-rbt-baldR*: *RBT-Impl.keys* (*rbt-baldR l a b r*) = *RBT-Impl.keys l* @ *a # RBT-Impl.keys r*
⟨*proof*⟩

lemma *keys-rbt-app*: *RBT-Impl.keys* (*rbt-app l r*) = *RBT-Impl.keys l* @ *RBT-Impl.keys r*
⟨*proof*⟩

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*⟩

lemma *keys-rbt-joinR*: *RBT-Impl.keys* (*rbt-joinR l a b r*) = *RBT-Impl.keys l* @ *a # RBT-Impl.keys r*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *set-keys-paint*: *set* (*RBT-Impl.keys* (*paint c t*)) = *set* (*RBT-Impl.keys t*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *inv-rbt-join*: *inv-12 l* ⟹ *inv-12 r* ⟹ *inv-12* (*rbt-join l a b r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*′)
⟨*proof*⟩

lemma *rbt-split-min-inv*: *rbt-split-min t* = (*a*,*b*,*t*′) ⟹ *inv-12 t* ⟹ *t* ≠ *RBT-Impl.Empty* ⟹ *inv-12 t*′
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *inv-rbt-join2*: *inv-12 l* ⟹ *inv-12 r* ⟹ *inv-12* (*rbt-join2 l r*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rbt-split-size*: (*l2*,*β*,*r2*) = *rbt-split t2 a* ⟹ *size l2* + *size r2* ≤ *size t2*
⟨*proof*⟩

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*)))
⟨*proof*⟩
termination
⟨*proof*⟩

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*)))
⟨*proof*⟩
termination
⟨*proof*⟩

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*⟩

lemma *rbt-fold-rbt-insert*:
assumes *inv-12 t2*
shows *inv-12* (*RBT-Impl.fold* (*rbt-insert-with-key f*) *t1 t2*)
⟨*proof*⟩

lemma *rbt-union-rec*: *inv-12 t1* ⟹ *inv-12 t2* ⟹ *inv-12* (*rbt-union-rec f t1 t2*)
⟨*proof*⟩

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'*))
⟨*proof*⟩
termination
⟨*proof*⟩

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'*))
⟨*proof*⟩
termination
⟨*proof*⟩

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*⟩

lemma *rbt-rbtreeify*[*simp*]: *inv-12* (*rbtreeify kvs*)
⟨*proof*⟩

lemma *rbt-inter-rec*: *inv-12 t1* ⟹ *inv-12 t2* ⟹ *inv-12* (*rbt-inter-rec f t1 t2*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *is-rbt-fold-rbt-insertwk*:
assumes *is-rbt t1*
shows *is-rbt* (*fold* (*rbt-insert-with-key f*) *t2 t1*)
⟨*proof*⟩

lemma *rbt-delete*: *inv-12 t* ⟹ *inv-12* (*rbt-delete x t*)
⟨*proof*⟩

lemma *rbt-sorted-delete*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-delete x t*)
⟨*proof*⟩

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*⟩

lemma *rbt-minus-rec*: *inv-12 t1* ⟹ *inv-12 t2* ⟹ *inv-12* (*rbt-minus-rec t1 t2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* $r\ k$)
⟨*proof*⟩

lemma *rbt-sorted-rbt-baliR*: *rbt-sorted* $l \Longrightarrow$ *rbt-sorted* $r \Longrightarrow l\ |\!\ll\ a \Longrightarrow a\ \ll\!|\ r \Longrightarrow$
rbt-sorted (*rbt-baliR* $l\ a\ b\ r$)
⟨*proof*⟩

lemma *rbt-lookup-rbt-baliR*: *rbt-sorted* $l \Longrightarrow$ *rbt-sorted* $r \Longrightarrow l\ |\!\ll\ a \Longrightarrow a\ \ll\!|\ r \Longrightarrow$
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* $r\ k$)
⟨*proof*⟩

lemma *rbt-sorted-rbt-joinL*: *rbt-sorted* (*RBT-Impl.Branch* $c\ l\ a\ b\ r$) $\Longrightarrow$ *bheight* l $\leq$ *bheight* $r \Longrightarrow$
rbt-sorted (*rbt-joinL* $l\ a\ b\ r$)
⟨*proof*⟩

lemma *rbt-lookup-rbt-joinL*: *rbt-sorted* $l \Longrightarrow$ *rbt-sorted* $r \Longrightarrow l\ |\!\ll\ a \Longrightarrow a\ \ll\!|\ r \Longrightarrow$
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*⟩

lemma *rbt-sorted-rbt-joinR*: *rbt-sorted* $l \Longrightarrow$ *rbt-sorted* $r \Longrightarrow l\ |\!\ll\ a \Longrightarrow a\ \ll\!|\ r \Longrightarrow$
rbt-sorted (*rbt-joinR* $l\ a\ b\ r$)
⟨*proof*⟩

lemma *rbt-lookup-rbt-joinR*: *rbt-sorted* $l \Longrightarrow$ *rbt-sorted* $r \Longrightarrow l\ |\!\ll\ a \Longrightarrow a\ \ll\!|\ r \Longrightarrow$
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*⟩

lemma *rbt-sorted-paint*: *rbt-sorted* (*paint* $c\ t$) = *rbt-sorted* t
⟨*proof*⟩

lemma *rbt-sorted-rbt-join*: *rbt-sorted* (*RBT-Impl.Branch* $c\ l\ a\ b\ r$) $\Longrightarrow$
rbt-sorted (*rbt-join* $l\ a\ b\ r$)
⟨*proof*⟩

lemma *rbt-lookup-rbt-join*: *rbt-sorted* $l \Longrightarrow$ *rbt-sorted* $r \Longrightarrow l\ |\!\ll\ a \Longrightarrow a\ \ll\!|\ r \Longrightarrow$
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* $r\ k$)
⟨*proof*⟩

lemma *rbt-split-min-rbt-sorted*: *rbt-split-min* $t = (a,b,t') \Longrightarrow$ *rbt-sorted* $t \Longrightarrow t \neq$ *RBT-Impl.Empty* $\Longrightarrow$
rbt-sorted $t' \wedge (\forall x \in$ *set* (*RBT-Impl.keys* t'). $a < x$)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lemma *rbt-sorted-fold-insertwk*: *rbt-sorted t ⟹*
rbt-sorted (RBT-Impl.fold (rbt-insert-with-key f) t′ t)
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *fold-rbt-delete*:
assumes *inv-12 t1 rbt-sorted t1 rbt-sorted t2*
shows *inv-12* (*RBT-Impl.fold* (λ*k* - *t*. *rbt-delete k t*) *t2 t1*) ∧
rbt-sorted (*RBT-Impl.fold* (λ*k* - *t*. *rbt-delete k t*) *t2 t1*) ∧
rbt-lookup (*RBT-Impl.fold* (λ*k* - *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*⟩

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*⟩

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*⟩

end

context *ord* **begin**

definition *rbt-union-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$
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* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$
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)*
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *rbt-unionw-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt (rbt-union-with f lt rt)*
⟨*proof*⟩

lemma *rbt-union-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt (rbt-union lt rt)*
⟨*proof*⟩

lemma *rbt-lookup-rbt-union*:
⟦ *rbt-sorted s*; *rbt-sorted t* ⟧ ⟹
rbt-lookup (*rbt-union s t*) = *rbt-lookup s* ++ *rbt-lookup t*
⟨*proof*⟩

lemma *rbt-interwk-is-rbt* [*simp*]:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with-key f t1 t2*)
⟨*proof*⟩

lemma *rbt-interw-is-rbt*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with f t1 t2*)
⟨*proof*⟩

lemma *rbt-inter-is-rbt*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-inter t1 t2*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbt-lookup-rbt-inter*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-inter t1 t2*) = *rbt-lookup t2* |‘ *dom* (*rbt-lookup t1*)
⟨*proof*⟩

lemma *rbt-minus-is-rbt*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-minus t1 t2*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-minus*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧
⟹ *rbt-lookup* (*rbt-minus t1 t2*) = *rbt-lookup t1* |‘ (− *dom* (*rbt-lookup t2*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *entries-code* [*code*]:
entries = *gen-entries* []
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *keys-code* [*code*]:
keys = *gen-keys* []
⟨*proof*⟩

Restore original type constraints for constants

⟨*ML*⟩

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*⟩

lemma *rbt-eq-iff*:
t1 = *t2* ⟷ *impl-of t1* = *impl-of t2*
⟨*proof*⟩

lemma *rbt-eqI*:
impl-of t1 = *impl-of t2* ⟹ *t1* = *t2*
⟨*proof*⟩

lemma *is-rbt-impl-of* [*simp*, *intro*]:
is-rbt (*impl-of t*)
⟨*proof*⟩

lemma *RBT-impl-of* [*simp*, *code abstype*]:
RBT (*impl-of t*) = *t*
⟨*proof*⟩

131.2 Primitive operations

setup-lifting *type-definition-rbt*

lift-definition *lookup* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a* ⇀ '*b* **is** *rbt-lookup* ⟨*proof*⟩

lift-definition *empty* :: ('*a*::*linorder*, '*b*) *rbt* **is** *RBT-Impl.Empty*

⟨*proof*⟩

lift-definition *insert* :: *'a*::*linorder* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **is** *rbt-insert*

⟨*proof*⟩

lift-definition *delete* :: *'a*::*linorder* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **is** *rbt-delete*
⟨*proof*⟩

lift-definition *entries* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a* × *'b*) *list* **is** *RBT-Impl.entries*
⟨*proof*⟩

lift-definition *keys* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ *'a list* **is** *RBT-Impl.keys* ⟨*proof*⟩

lift-definition *bulkload* :: (*'a*::*linorder* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* **is** *rbt-bulkload*
⟨*proof*⟩

lift-definition *map-entry* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt*
is *rbt-map-entry*
⟨*proof*⟩

lift-definition *map* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'c*) *rbt* **is**
RBT-Impl.map
⟨*proof*⟩

lift-definition *fold* :: (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ *'c* ⇒ *'c* **is**
RBT-Impl.fold ⟨*proof*⟩

lift-definition *union* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **is**
rbt-union
⟨*proof*⟩

lift-definition *foldi* :: (*'c* ⇒ *bool*) ⇒ (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*) ⇒ (*'a* :: *linorder*, *'b*)
rbt ⇒ *'c* ⇒ *'c*
is *RBT-Impl.foldi* ⟨*proof*⟩

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* ⟨*proof*⟩

lift-definition *combine* :: (*'b* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* ⇒
(*'a*, *'b*) *rbt*
is *RBT-Impl.rbt-union-with* ⟨*proof*⟩

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* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a::linorder, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$ **where**
[*code*]: $filter\ P\ t = fold\ (\lambda k\ v\ t.\ if\ P\ k\ v\ then\ insert\ k\ v\ t\ else\ t)\ t\ empty$

131.4 Abstract lookup properties

lemma *lookup-RBT*:
$is\text{-}rbt\ t \Longrightarrow lookup\ (RBT\ t) = rbt\text{-}lookup\ t$
⟨*proof*⟩

lemma *lookup-impl-of*:
$rbt\text{-}lookup\ (impl\text{-}of\ t) = lookup\ t$
⟨*proof*⟩

lemma *entries-impl-of*:
$RBT\text{-}Impl.entries\ (impl\text{-}of\ t) = entries\ t$
⟨*proof*⟩

lemma *keys-impl-of*:
$RBT\text{-}Impl.keys\ (impl\text{-}of\ t) = keys\ t$
⟨*proof*⟩

lemma *lookup-keys*:
$dom\ (lookup\ t) = set\ (keys\ t)$
⟨*proof*⟩

lemma *lookup-empty* [*simp*]:
$lookup\ empty = Map.empty$
⟨*proof*⟩

lemma *lookup-insert* [*simp*]:
$lookup\ (insert\ k\ v\ t) = (lookup\ t)(k \mapsto v)$
⟨*proof*⟩

lemma *lookup-delete* [*simp*]:
$lookup\ (delete\ k\ t) = (lookup\ t)(k := None)$
⟨*proof*⟩

lemma *map-of-entries* [*simp*]:
$map\text{-}of\ (entries\ t) = lookup\ t$
⟨*proof*⟩

lemma *entries-lookup*:
$entries\ t1 = entries\ t2 \longleftrightarrow lookup\ t1 = lookup\ t2$
⟨*proof*⟩

lemma *lookup-bulkload* [*simp*]:
$lookup\ (bulkload\ xs) = map\text{-}of\ xs$
⟨*proof*⟩

lemma *lookup-map-entry* [*simp*]:
lookup (*map-entry k f t*) = (*lookup t*)(*k* := *map-option f* (*lookup t k*))
⟨*proof*⟩

lemma *lookup-map* [*simp*]:
lookup (*map f t*) *k* = *map-option* (*f k*) (*lookup t k*)
⟨*proof*⟩

lemma *lookup-combine-with-key* [*simp*]:
lookup (*combine-with-key f t1 t2*) *k* = *combine-options* (*f k*) (*lookup t1 k*) (*lookup t2 k*)
⟨*proof*⟩

lemma *combine-altdef*: *combine f t1 t2* = *combine-with-key* (λ-. *f*) *t1 t2*
⟨*proof*⟩

lemma *lookup-combine* [*simp*]:
lookup (*combine f t1 t2*) *k* = *combine-options f* (*lookup t1 k*) (*lookup t2 k*)
⟨*proof*⟩

lemma *fold-fold*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
⟨*proof*⟩

lemma *impl-of-empty*:
impl-of empty = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]:
is-empty t ⟷ *t* = *empty*
⟨*proof*⟩

lemma *RBT-lookup-empty* [*simp*]:
rbt-lookup t = *Map.empty* ⟷ *t* = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *lookup-empty-empty* [*simp*]:
lookup t = *Map.empty* ⟷ *t* = *empty*
⟨*proof*⟩

lemma *keys-empty-eq* [*simp*]:
‹*keys empty* = []›
⟨*proof*⟩

lemma *sorted-keys* [*iff*]:
sorted (*keys t*)
⟨*proof*⟩

lemma *distinct-keys* [*iff*]:
distinct (*keys t*)
⟨*proof*⟩

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite* (*dom* (*lookup t*))
⟨*proof*⟩

lemma *lookup-union*: *lookup* (*union s t*) = *lookup s* ++ *lookup t*
⟨*proof*⟩

lemma *lookup-in-tree*: (*lookup t k* = *Some v*) = ((*k*, *v*) ∈ *set* (*entries t*))
⟨*proof*⟩

lemma *keys-entries*: (*k* ∈ *set* (*keys t*)) = (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
⟨*proof*⟩

lemma *fold-def-alt*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
⟨*proof*⟩

lemma *distinct-entries*: *distinct* (*List.map fst* (*entries t*))
⟨*proof*⟩

lemma *sorted-entries*: *sorted* (*List.map fst* (*entries t*))
⟨*proof*⟩

lemma *non-empty-keys*: *t* ≠ *empty* ⟹ *keys t* ≠ []
⟨*proof*⟩

lemma *keys-def-alt*:
keys t = *List.map fst* (*entries t*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨

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*
⟨*proof*⟩

lemma *Set-set-keys*: *Set x* = *dom* (*RBT.lookup x*)
⟨*proof*⟩

lemma *finite-Set* [*simp*, *intro*!]: *finite* (*Set x*)
⟨*proof*⟩

lemma *set-keys*: *Set t* = *set*(*RBT.keys t*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *finite-fold-fold-keys*:
assumes *comp-fun-commute f*
shows *Finite-Set.fold f A* (*Set t*) = *fold-keys f t A*
⟨*proof*⟩

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*
⟨*proof*⟩

135.3 foldi and Ball

lemma *Ball-False*: *RBT-Impl.fold* (λ*k v s*. *s* ∧ *P k*) *t False* = *False*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

135.4 foldi and Bex

lemma *Bex-True*: *RBT-Impl.fold* (λ*k v s*. *s* ∨ *P k*) *t True* = *True*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *left-le-key*: *rbt-sorted* (*Branch c lt k v rt*) ⟹ (⋀*x*. *x* ∈*set* (*RBT-Impl.keys lt*) ⟹ *x* ≤ *k*)
⟨*proof*⟩

lemma *fold-min-triv*:
fixes *k* :: - :: *linorder*
shows (∀ *x*∈*set xs*. *k* ≤ *x*) ⟹ *List.fold min xs k* = *k*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\neq$ *rbt.Empty* $\Longrightarrow$ *rbt-min-opt* (*Branch c t1 k* () *t2*) = *rbt-min-opt t1*
⟨*proof*⟩

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 $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t1* = *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t1* $\neq$ *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
shows *P t*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

maximum definition *rbt-max* :: ('*a*::*linorder*, *unit*) *RBT-Impl.rbt* $\Rightarrow$ '*a*
where *rbt-max t* = *rbt-fold1-keys max t*

lemma *fold-max-triv*:
fixes *k* :: - :: *linorder*
shows ($\forall$ *x*$\in$*set xs*. *x* $\leq$ *k*) $\Longrightarrow$ *List.fold max xs k* = *k*
⟨*proof*⟩

lemma *fold-max-rev-eq*:
fixes *xs* :: ('*a* :: *linorder*) *list*
assumes *xs* $\neq$ []
shows *List.fold max* (*tl xs*) (*hd xs*) = *List.fold max* (*tl* (*rev xs*)) (*hd* (*rev xs*))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rbt-max-eq-rbt-max-opt*:
assumes *t* ≠ *RBT-Impl.Empty*
assumes *is-rbt t*
shows *rbt-max t* = *rbt-max-opt t*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *fold1-keys-def-alt*:
fold1-keys f t = *List.fold f* (*tl* (*RBT.keys t*)) (*hd* (*RBT.keys t*))
⟨*proof*⟩

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*⟩

minimum **lift-definition** *r-min* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-min*
⟨*proof*⟩

lift-definition *r-min-opt* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-min-opt* ⟨*proof*⟩

lemma *r-min-alt-def*: *r-min t* = *fold1-keys min t*
⟨*proof*⟩

lemma *r-min-eq-r-min-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-min t* = *r-min-opt t*
⟨*proof*⟩

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*⟩

maximum **lift-definition** *r-max* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-max*
⟨*proof*⟩

lift-definition *r-max-opt* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-max-opt* ⟨*proof*⟩

lemma *r-max-alt-def*: *r-max t* = *fold1-keys max t*
⟨*proof*⟩

lemma *r-max-eq-r-max-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-max t* = *r-max-opt t*
⟨*proof*⟩

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*⟩

end

136 Code equations

code-datatype *Set Coset*

declare *list.set*[*code*]

lemma *empty-Set* [*code*]:
Set.empty = Set RBT.empty
⟨*proof*⟩

lemma *UNIV-Coset* [*code*]:
UNIV = Coset RBT.empty
⟨*proof*⟩

lemma *is-empty-Set* [*code*]:
Set.is-empty (Set t) = RBT.is-empty t
⟨*proof*⟩

lemma *compl-code* [*code*]:
– Set xs = Coset xs
– Coset xs = Set xs
⟨*proof*⟩

lemma *member-code* [*code*]:
x ∈ (Set t) = (RBT.lookup t x = Some ())
x ∈ (Coset t) = (RBT.lookup t x = None)
⟨*proof*⟩

lemma *insert-code* [*code*]:
Set.insert x (Set t) = Set (RBT.insert x () t)
Set.insert x (Coset t) = Coset (RBT.delete x t)
⟨*proof*⟩

lemma *remove-code* [*code*]:
Set.remove x (Set t) = Set (RBT.delete x t)
Set.remove x (Coset t) = Coset (RBT.insert x () t)
⟨*proof*⟩

lemma *inter-Set* [*code*]:
A ∩ Set t = rbt-filter (λk. k ∈ A) t
⟨*proof*⟩

lemma *union-Set-Set* [*code*]:
Set t1 ∪ Set t2 = Set (RBT.union t1 t2)
⟨*proof*⟩

lemma *union-Set* [*code*]:
Set t ∪ A = fold-keys Set.insert t A
⟨*proof*⟩

lemma *minus-Set* [*code*]:
A − Set t = fold-keys Set.remove t A
⟨*proof*⟩

lemma *inter-Coset-Coset* [*code*]:
Coset t1 ∩ Coset t2 = Coset (RBT.union t1 t2)
⟨*proof*⟩

lemma *inter-Coset* [*code*]:
A ∩ Coset t = fold-keys Set.remove t A
⟨*proof*⟩

lemma *union-Coset* [*code*]:
Coset t ∪ A = − rbt-filter (λk. k ∉ A) t
⟨*proof*⟩

lemma *minus-Coset* [*code*]:
A − Coset t = rbt-filter (λk. k ∈ A) t
⟨*proof*⟩

lemma *filter-Set* [*code*]:
Set.filter P (Set t) = rbt-filter P t
⟨*proof*⟩

lemma *image-Set* [*code*]:
image f (Set t) = fold-keys (λk A. Set.insert (f k) A) t {}
⟨*proof*⟩

lemma *Ball-Set* [*code*]:
Ball (Set t) P ⟷ RBT.foldi (λs. s = True) (λk v s. s ∧ P k) t True
⟨*proof*⟩

lemma *Bex-Set* [*code*]:
Bex (Set t) P ⟷ RBT.foldi (λs. s = False) (λk v s. s ∨ P k) t False
⟨*proof*⟩

lemma *subset-code* [*code*]:
Set t ≤ B ⟷ (∀ x∈Set t. x ∈ B)
A ≤ Coset t ⟷ (∀ y∈Set t. y ∉ A)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *card-Set* [*code*]:
card (*Set t*) = *fold-keys* (λ- *n*. *n* + *1*) *t 0*
⟨*proof*⟩

lemma *sum-Set* [*code*]:
sum f (*Set xs*) = *fold-keys* (*plus* ∘ *f*) *xs 0*
⟨*proof*⟩

lemma *prod-Set* [*code*]:
prod f (*Set xs*) = *fold-keys* (*times* ∘ *f*) *xs 1*
⟨*proof*⟩

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*⟩

lemma *Pow-Set* [*code*]: *Pow* (*Set t*) = *fold-keys* (λ*x A*. *A* ∪ *Set.insert x* ' *A*) *t*
{{}}
⟨*proof*⟩

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*⟩

lemma *Id-on-Set* [*code*]: *Id-on* (*Set t*) = *fold-keys* (λ*x*. *Set.insert* (*x*, *x*)) *t* {}
⟨*proof*⟩

lemma *Image-Set* [*code*]:
(*Set t*) `` *S* = *fold-keys* (λ(*x*,*y*) *A*. *if x* ∈ *S then Set.insert y A else A*) *t* {}
⟨*proof*⟩

lemma *trancl-set-ntrancl* [*code*]:
trancl (*Set t*) = *ntrancl* (*card* (*Set t*) − *1*) (*Set t*)
⟨*proof*⟩

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*⟩

lemma *wf-set*: *wf* (*Set t*) = *acyclic* (*Set t*)
⟨*proof*⟩

lemma *wf-code-set*[*code*]: *wf-code* (*Set t*) = *acyclic* (*Set t*)
⟨*proof*⟩

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*⟩

lemma *Inf-fin-set-fold* [*code*]:
Inf-fin (*Set t*) = *Min* (*Set t*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *Sup-fin-set-fold* [*code*]:
Sup-fin (*Set t*) = *Max* (*Set t*)
⟨*proof*⟩

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*⟩

lemma [*code*]:
Gcd_{fin} (*Set t*) = *fold-keys gcd t* (*0*::'*a*::{*semiring-gcd*, *linorder*})
⟨*proof*⟩

lemma [*code*]:
Gcd (*Set t*) = Gcd_{fin} (*Set t*) **for** *t* :: ‹('*a*::{*semiring-Gcd*, *linorder*}, *unit*) *RBT.rbt*›
⟨*proof*⟩

lemma [*code*]:
Gcd (*Set t*) = (Gcd_{fin} (*Set t*) :: *nat*)

⟨*proof*⟩

lemma [*code*]:
Gcd (*Set t*) = (*Gcd*$_{fin}$ (*Set t*) :: *int*)
⟨*proof*⟩

lemma [*code*]:
Lcm$_{fin}$ (*Set t*) = *fold-keys lcm t* (*1*::*'a*::{*semiring-gcd*, *linorder*})
⟨*proof*⟩

lemma [*code*]:
Lcm (*Set t*) = *Lcm*$_{fin}$ (*Set t*) **for** *t* :: ‹(*'a*::{*semiring-Gcd*, *linorder*}, *unit*) *RBT.rbt*›
⟨*proof*⟩

lemma [*code*]:
Lcm (*Set t*) = (*Lcm*$_{fin}$ (*Set t*) :: *nat*)
⟨*proof*⟩

lemma [*code*]:
Lcm (*Set t*) = (*Lcm*$_{fin}$ (*Set t*) :: *int*)
⟨*proof*⟩

lemma *Least-code* [*code*]:
‹*Lattices-Big.Least* (*Set t*) = (*if RBT.is-empty t then Lattices-Big.Least-abort* {} *else Min* (*Set t*))›
⟨*proof*⟩

lemma *Greatest-code* [*code*]:
‹*Lattices-Big.Greatest* (*Set t*) = (*if RBT.is-empty t then Lattices-Big.Greatest-abort* {} *else Max* (*Set t*))›
⟨*proof*⟩

lemma *sorted-list-set* [*code*]: *sorted-list-of-set* (*Set t*) = *RBT.keys t*
⟨*proof*⟩

lemma [*code*]:
‹*Option.these A* = *the* ‘ *Set.filter* (*Not* ∘ *Option.is-none*) *A*›
⟨*proof*⟩

lemma [*code*]:
‹*Option.image-filter f A* = *Option.these* (*image f A*)›
⟨*proof*⟩

lemma [*code*]:
‹*Set.can-select P A* = *is-singleton* (*Set.filter P A*)›
⟨*proof*⟩

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*)
⟨*proof*⟩
termination
  ⟨*proof*⟩

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*)
⟨*proof*⟩
termination
  ⟨*proof*⟩

time-function *sum-to2*
termination
  ⟨*proof*⟩

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* ⇒ *nat*) ⇒ ′*a list* ⇒ *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*
⟨*proof*⟩

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* ⇒ ′*a* ⇒ ′*b*) ⇒ ′*b* ⇒ ′*a list* ⇒ ′*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* ⇒ ′*a* ⇒ ′*b*) × (′*b* ⇒ ′*a* ⇒ *nat*) ⇒ ′*b* ⇒ ′*a list* ⇒ *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*
⟨*proof*⟩

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*⟩

lemma *T-g-ub*: *T-g xs* ≤ *2* ∗ *length xs + 1*
⟨*proof*⟩

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-f*. Or one can derive the bound for *T-g* after a specific interpretation with a specific *T-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*
⟨*proof*⟩

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

⟨*ML*⟩

theory *Suc-Notation*
imports *Main*
begin

Nested *Suc* terms of depth $2 \le n \le 9$ 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*$^-$)

⟨*ML*⟩

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*) == (∧)
⟨*proof*⟩

lemma [*code-pred-inline*]:
((*A*::*bool*) ≠ (*B*::*bool*)) = ((*A* ∧ ¬ *B*) ∨ (*B* ∧ ¬ *A*))
⟨*proof*⟩

⟨*ML*⟩

145 Pairs

⟨*ML*⟩

146 Filters

⟨*ML*⟩

147 Bounded quantifiers

declare *Ball-def*[*code-pred-inline*]
declare *Bex-def*[*code-pred-inline*]

148 Operations on Predicates

lemma *Diff*[*code-pred-inline*]:
$(A - B) = (\%x.\ A\ x \wedge \neg\ B\ x)$
⟨*proof*⟩

lemma *subset-eq*[*code-pred-inline*]:
$(P :: 'a \Rightarrow bool) < (Q :: 'a \Rightarrow bool) \equiv ((\exists x.\ Q\ x \wedge (\neg\ P\ x)) \wedge (\forall x.\ P\ x \longrightarrow Q\ x))$
⟨*proof*⟩

lemma *set-equality*[*code-pred-inline*]:
$A = B \longleftrightarrow (\forall x.\ A\ x \longrightarrow B\ x) \wedge (\forall x.\ B\ x \longrightarrow A\ x)$
⟨*proof*⟩

149 Setup for Numerals

⟨*ML*⟩

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*

⟨*ML*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* [] []
⟨*proof*⟩

lemma *list-all2-ConsI* [*code-pred-intro*]: *list-all2 P xs ys ==> P x y ==> list-all2 P* (*x*#*xs*) (*y*#*ys*)
⟨*proof*⟩

code-pred [*skip-proof*] *list-all2*
⟨*proof*⟩

151.3 Alternative rules for membership in lists

lemma *in-set-member* [*code-pred-inline*]:
x ∈ *set xs* ⟷ *List.member xs x*
⟨*proof*⟩

lemma *member-intros* [*code-pred-intro*]:
List.member (*x*#*xs*) *x*
List.member xs x ⟹ *List.member* (*y*#*xs*) *x*
⟨*proof*⟩

code-pred *List.member*
⟨*proof*⟩

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

⟨*ML*⟩

153 Simplification rules for optimisation

lemma [*code-pred-simp*]: ¬ *False* == *True*
⟨*proof*⟩

lemma [*code-pred-simp*]: ¬ *True* == *False*
⟨*proof*⟩

lemma *less-nat-k-0* [*code-pred-simp*]: *less-nat k 0 == False*
⟨*proof*⟩

end

154 A Prototype of Quickcheck based on the Predicate Compiler

theory *Predicate-Compile-Quickcheck*
imports *Predicate-Compile-Alternative-Defs*
begin

⟨*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*: ∀ *R*. *wf R* ⟶
(∀ *P*. (∀ *x*. (∀ *y*. (*y*,*x*)∈*R* ⟶ *P y*) ⟶ *P x*) ⟶ (∀ *x*. *P x*))
⟨*proof*⟩

lemma *tfl-cut-def*: *Wfrec.cut f r x* ≡ (λ*y*. *if* (*y*,*x*) ∈ *r then f y else undefined*)
⟨*proof*⟩

lemma *tfl-cut-apply*: ∀ *f R*. (*x*,*a*)∈*R* ⟶ (*Wfrec.cut f R a*)(*x*) = *f*(*x*)
⟨*proof*⟩

lemma *tfl-wfrec*:
∀ *M R f*. (*f*=*wfrec R M*) ⟶ *wf R* ⟶ (∀ *x*. *f x* = *M* (*Wfrec.cut f R x*) *x*)
⟨*proof*⟩

lemma *tfl-eq-True*: (*x* = *True*) ⟶ *x*
⟨*proof*⟩

lemma *tfl-rev-eq-mp*: (*x* = *y*) ⟶ *y* ⟶ *x*
⟨*proof*⟩

lemma *tfl-simp-thm*: $(x \longrightarrow y) \longrightarrow (x = x') \longrightarrow (x' \longrightarrow y)$
⟨*proof*⟩

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
⟨*proof*⟩

lemma *tfl-imp-trans*: $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$
⟨*proof*⟩

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
⟨*proof*⟩

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$
⟨*proof*⟩

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \longrightarrow Q \Longrightarrow Q$
⟨*proof*⟩

⟨*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*⟩

end

157 Refute

theory *Refute*
imports *Main*
keywords
refute :: *diag* **and**
refute-params :: *thy-decl*
begin

⟨*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.