

The Supplemental Isabelle/HOL Library

December 12, 2021

Contents

1	Implementation of Association Lists	21
1.1	<i>update</i> and <i>updates</i>	21
1.2	<i>delete</i>	23
1.3	<i>update-with-aux</i> and <i>delete-aux</i>	24
1.4	<i>restrict</i>	26
1.5	<i>clearjunk</i>	27
1.6	<i>map-ran</i>	28
1.7	<i>merge</i>	29
1.8	<i>compose</i>	30
1.9	<i>map-entry</i>	32
1.10	<i>map-default</i>	32
2	Adhoc overloading of constants based on their types	33
3	Pointwise instantiation of functions to algebra type classes	33
4	Algebraic operations on sets	37
5	Big O notation	43
5.1	Definitions	43
5.2	Sum	47
5.3	Misc useful stuff	48
5.4	Less than or equal to	49
6	Axiomatic Declaration of Bounded Natural Functors	50
7	Generalized Corecursor Sugar (corec and friends)	50
7.1	Coinduction	51
8	A general “while” combinator	54
8.1	Partial version	54
8.2	Total version	55

9 The Bourbaki-Witt tower construction for transfinite iteration	58
9.1 Connect with the while combinator for executability on chain-finite lattices.	61
10 Order on characters	63
11 A generic phantom type	63
12 Cardinality of types	64
12.1 Preliminary lemmas	64
12.2 Cardinalities of types	64
12.3 Classes with at least 1 and 2	65
12.4 A type class for deciding finiteness of types	66
12.5 A type class for computing the cardinality of types	66
12.6 Instantiations for <i>card-UNIV</i>	66
12.7 Code setup for sets with cardinality type information	70
12.8 Eliminating pattern matches	72
13 Lazy types in generated code	73
13.1 The type <i>lazy</i>	73
13.2 Implementation	77
13.3 YXML encoding for <i>term</i>	78
13.4 Test engine and drivers	80
14 A combinator to build partial equivalence relations from a predicate and an equivalence relation	80
15 Formalisation of chain-complete partial orders, continuity and admissibility	81
15.1 Continuity	84
15.1.1 Theorem collection <i>cont-intro</i>	84
15.2 Admissibility	91
15.3 ($=$) as order	94
15.4 ccpo for products	95
15.5 Complete lattices as ccpo	100
15.6 Parallel fixpoint induction	103
16 Confluence	106
17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums	110
17.1 The datatype universe	110
17.2 Freeness: Distinctness of Constructors	112
17.3 Set Constructions	115

18 Bijections between natural numbers and other types	119
18.1 Type $\text{nat} \times \text{nat}$	119
18.2 Type $\text{nat} + \text{nat}$	121
18.3 Type int	121
18.4 Type nat list	122
18.5 Finite sets of naturals	123
18.5.1 Preliminaries	123
18.5.2 From sets to naturals	124
18.5.3 From naturals to sets	124
18.5.4 Proof of isomorphism	124
19 Encoding (almost) everything into natural numbers	125
19.1 The class of countable types	125
19.2 Conversion functions	125
19.3 Finite types are countable	126
19.4 Automatically proving countability of old-style datatypes	126
19.5 Automatically proving countability of datatypes	127
19.6 More Countable types	127
19.7 The rationals are countably infinite	128
20 Infinite Sets and Related Concepts	128
20.1 The set of natural numbers is infinite	129
20.2 The set of integers is also infinite	129
20.3 Infinitely Many and Almost All	130
20.4 Enumeration of an Infinite Set	132
20.5 Properties of <i>wellorder-class.enumerate</i> on finite sets	134
21 Countable sets	135
21.1 Predicate for countable sets	136
21.2 Enumerate a countable set	136
21.3 Closure properties of countability	139
21.4 Misc lemmas	142
21.5 Uncountable	143
22 Countable Complete Lattices	144
22.0.1 Instances of countable complete lattices	149
23 Type of (at Most) Countable Sets	149
23.1 Cardinal stuff	149
23.2 The type of countable sets	150
23.3 Additional lemmas	157
23.3.1 <i>empty</i>	157
23.3.2 <i>cinsert</i>	157
23.3.3 <i>cimage</i>	157

23.3.4	bounded quantification	157
23.3.5	<i>cUnion</i>	158
23.4	Setup for Lifting/Transfer	158
23.4.1	Relator and predicator properties	158
23.4.2	Transfer rules for the Transfer package	158
23.5	Registration as BNF	159
24	Debugging facilities for code generated towards Isabelle/ML	161
25	Sequence of Properties on Subsequences	161
26	Common discrete functions	163
26.1	Discrete logarithm	163
26.2	Discrete square root	165
27	Pi and Function Sets	166
27.1	Basic Properties of <i>Pi</i>	167
27.2	Composition With a Restricted Domain: <i>compose</i>	168
27.3	Bounded Abstraction: <i>restrict</i>	169
27.4	Bijections Between Sets	170
27.5	Extensionality	170
27.6	Cardinality	172
27.7	Extensional Function Spaces	172
27.7.1	Injective Extensional Function Spaces	175
27.7.2	Misc properties of functions, composition and restriction from HOL Light	175
27.7.3	Cardinality	176
27.8	The pigeonhole principle	176
28	Partitions and Disjoint Sets	176
28.1	Set of Disjoint Sets	177
28.1.1	Family of Disjoint Sets	177
28.2	Construct Disjoint Sequences	180
28.3	Partitions	180
28.4	Constructions of partitions	181
28.5	Finiteness of partitions	181
28.6	Equivalence of partitions and equivalence classes	182
28.7	Refinement of partitions	182
28.8	The coarsest common refinement of a set of partitions	183
29	Type of finite sets defined as a subtype of sets	183
29.1	Definition of the type	184
29.2	Basic operations and type class instantiations	184
29.3	Other operations	187
29.4	Transferred lemmas from Set.thy	188

29.5	Additional lemmas	193
29.5.1	<i>ffUnion</i>	193
29.5.2	<i>fbind</i>	193
29.5.3	<i>fsingleton</i>	193
29.5.4	<i>fempty</i>	193
29.5.5	<i>fset</i>	193
29.5.6	<i>ffilter</i>	194
29.5.7	<i>fset-of-list</i>	194
29.5.8	<i>finsert</i>	194
29.5.9	<i>fimage</i>	194
29.5.10	bounded quantification	195
29.5.11	<i>fcard</i>	195
29.5.12	<i>sorted-list-of-fset</i>	197
29.5.13	<i>ffold</i>	197
29.5.14	Group operations	198
29.5.15	Semilattice operations	199
29.6	Choice in fsets	201
29.7	Induction and Cases rules for fsets	201
29.8	Setup for Lifting/Transfer	202
29.8.1	Relator and predicator properties	202
29.8.2	Transfer rules for the Transfer package	202
29.9	BNF setup	204
29.10	Size setup	204
29.11	Advanced relator customization	205
29.11.1	Countability	205
29.12	Quickcheck setup	205
30	Type of finite maps defined as a subtype of maps	207
30.1	Auxiliary constants and lemmas over <i>map</i>	207
30.2	Abstract characterisation	209
30.3	Operations	209
30.4	BNF setup	223
30.5	<i>size</i> setup	226
30.6	Additional operations	227
30.7	Additional properties	228
30.8	Lifting/transfer setup	228
30.9	View as datatype	229
30.10	Code setup	229
30.11	Instances	231
30.12	Tests	231
31	Disjoint FSets	232

32 Lists with elements distinct as canonical example for datatype invariants	233
32.1 The type of distinct lists	233
32.2 Executable version obeying invariant	235
32.3 Induction principle and case distinction	236
32.4 Functorial structure	236
32.5 Quickcheck generators	236
32.6 BNF instance	237
33 Type of dual ordered lattices	238
33.1 Pointwise ordering	239
33.2 Binary infimum and supremum	240
33.3 Top and bottom elements	241
33.4 Complement	242
33.5 Complete lattice operations	243
34 Equipollence and Other Relations Connected with Cardinality	244
34.1 Eqpoll	245
34.2 The strict relation	247
34.3 Mapping by an injection	247
34.4 Inserting elements into sets	248
34.5 Binary sums and unions	249
34.6 Binary Cartesian products	249
34.7 General Unions	250
34.8 General Cartesian products (Pi)	250
35 Continuity and iterations	255
35.1 Continuity for complete lattices	256
35.1.1 Least fixed points in countable complete lattices	258
36 Extended natural numbers (i.e. with infinity)	259
36.1 Type definition	259
36.2 Constructors and numbers	260
36.3 Addition	262
36.4 Multiplication	262
36.5 Numerals	263
36.6 Subtraction	264
36.7 Ordering	264
36.8 Cancellation simprocs	268
36.9 Well-ordering	268
36.10 Complete Lattice	268
36.11 Traditional theorem names	269

37 Liminf and Limsup on conditionally complete lattices	270
37.0.1 <i>Liminf</i> and <i>Limsup</i>	271
37.1 More Limits	275
38 Extended real number line	276
38.1 Definition and basic properties	278
38.1.1 Addition	280
38.1.2 Linear order on <i>ereal</i>	282
38.1.3 Multiplication	288
38.1.4 Power	294
38.1.5 Subtraction	294
38.1.6 Division	298
38.2 Complete lattice	302
38.3 Extended real intervals	304
38.4 Topological space	306
38.5 Relation to <i>enat</i>	312
38.6 Limits on <i>ereal</i>	313
38.6.1 Convergent sequences	314
38.6.2 Sums	319
38.6.3 Continuity	325
38.6.4 liminf and limsup	326
38.6.5 Tests for code generator	330
39 Indicator Function	330
39.1 The type of non-negative extended real numbers	333
39.2 Defining the extended non-negative reals	336
39.3 Cancellation simprocs	339
39.4 Order with top	339
39.5 Arithmetic	342
39.6 Coercion from <i>real</i> to <i>ennreal</i>	346
39.7 Coercion from <i>ennreal</i> to <i>real</i>	350
39.8 Coercion from <i>enat</i> to <i>ennreal</i>	351
39.9 Topology on <i>ennreal</i>	352
39.10 Approximation lemmas	359
39.11 <i>ennreal</i> theorems	360
40 Logarithm of Natural Numbers	364
40.1 Preliminaries	364
40.2 Floorlog	364
40.3 Bitlen	366
41 Various algebraic structures combined with a lattice	367
41.1 Positive Part, Negative Part, Absolute Value	368

42 Floating-Point Numbers	372
42.1 Real operations preserving the representation as floating point number	372
42.2 Arithmetic operations on floating point numbers	374
42.3 Quickcheck	376
42.4 Represent floats as unique mantissa and exponent	377
42.5 Compute arithmetic operations	379
42.6 Lemmas for types <i>real</i> , <i>nat</i> , <i>int</i>	380
42.7 Rounding Real Numbers	381
42.8 Rounding Floats	382
42.9 Truncating Real Numbers	383
42.10 Truncating Floats	385
42.11 Approximation of positive rationals	387
42.12 Division	389
42.13 Approximate Addition	390
42.14 Approximate Multiplication	392
42.15 Approximate Power	393
42.16 Lemmas needed by Approximate	396
43 Pointwise instantiation of functions to division	399
43.1 Syntactic with division	399
44 Lexicographic order on functions	400
45 The <i>going-to</i> filter	401
46 Big sum and product over function bodies	404
46.1 Abstract product	404
46.2 Concrete sum	406
46.3 Concrete product	407
47 Interval Type	407
47.1 Membership	412
47.2 Quickcheck	419
48 Approximate Operations on Intervals of Floating Point Numbers	420
48.1 Intervals with Floating Point Bounds	421
48.2 intros for <i>real-interval</i>	422
48.3 bounds for lists	423
48.4 constants for code generation	426

49 Immutable Arrays with Code Generation	426
49.1 Fundamental operations	426
49.2 Generic code equations	427
49.3 Auxiliary operations for code generation	428
49.4 Code Generation for SML	429
49.5 Code Generation for Haskell	429
50 Definition of Landau symbols	430
50.1 Definition of Landau symbols	431
50.2 Landau symbols and limits	443
50.3 Flatness of real functions	449
50.4 Asymptotic Equivalence	450
50.5 Values extended by a bottom element	456
50.6 Values extended by a top element	459
50.7 Values extended by a top and a bottom element	461
51 Infinite Streams	463
51.1 prepend list to stream	464
51.2 set of streams with elements in some fixed set	464
51.3 nth, take, drop for streams	466
51.4 unary predicates lifted to streams	468
51.5 recurring stream out of a list	468
51.6 iterated application of a function	469
51.7 stream repeating a single element	470
51.8 stream of natural numbers	470
51.9 flatten a stream of lists	471
51.10 merge a stream of streams	471
51.11 product of two streams	471
51.12 interleave two streams	472
51.13 zip	472
51.14 zip via function	473
52 List prefixes, suffixes, and homeomorphic embedding	473
52.1 Prefix order on lists	473
52.2 Basic properties of prefixes	474
52.3 Prefixes	477
52.4 Longest Common Prefix	478
52.5 Parallel lists	479
52.6 Suffix order on lists	480
52.7 Suffixes	484
52.8 Homeomorphic embedding on lists	485
52.9 Subsequences (special case of homeomorphic embedding)	487
52.10 Appending elements	488
52.11 Relation to standard list operations	489

52.12	Contiguous sublists	489
52.12.1	<i>sublist</i>	489
52.12.2	<i>sublists</i>	492
52.13	Parametricity	492
53	Linear Temporal Logic on Streams	494
54	Preliminaries	494
55	Linear temporal logic	494
56	Weak vs. strong until (contributed by Michael Foster, University of Sheffield)	504
57	Lists as vectors	505
57.1	+ and −	505
57.2	Inner product	507
58	Definitions of Least Upper Bounds and Greatest Lower Bounds	507
58.1	Rules for the Relations $* \leq$ and $\leq *$	508
58.2	Rules about the Operators <i>leastP</i> , <i>ub</i> and <i>lub</i>	508
58.3	Rules about the Operators <i>greatestP</i> , <i>isLb</i> and <i>isGlb</i>	510
59	An abstract view on maps for code generation.	512
59.1	Parametricity transfer rules	512
59.2	Type definition and primitive operations	514
59.3	Functorial structure	515
59.4	Derived operations	515
59.5	Properties	517
59.5.1	<i>entries</i> , <i>ordered-entries</i> , and <i>fold</i>	525
59.6	Code generator setup	528
60	Monad notation for arbitrary types	528
61	Less common functions on lists	529
61.1	Simproc Set-Up	536
62	(Finite) Multisets	536
62.1	The type of multisets	536
62.2	Representing multisets	537
62.3	Basic operations	538
62.3.1	Conversion to set and membership	538
62.3.2	Union	541
62.3.3	Difference	541
62.3.4	Min and Max	543

62.3.5	Equality of multisets	543
62.3.6	Pointwise ordering induced by count	545
62.3.7	Intersection and bounded union	548
62.3.8	Additional intersection facts	549
62.3.9	Additional bounded union facts	551
62.4	Replicate and repeat operations	551
62.4.1	Simprocs	553
62.4.2	Conditionally complete lattice	554
62.4.3	Filter (with comprehension syntax)	555
62.4.4	Size	557
62.5	Induction and case splits	559
62.5.1	Strong induction and subset induction for multisets	560
62.6	The fold combinator	560
62.7	Image	561
62.8	Further conversions	563
62.9	More properties of the replicate and repeat operations	568
62.10	Big operators	569
62.11	Multiset as order-ignorant lists	575
62.12	The multiset order	577
62.12.1	Well-foundedness	577
62.12.2	Closure-free presentation	578
62.13	The multiset extension is cancellative for multiset union	579
62.14	Quasi-executable version of the multiset extension	579
62.14.1	Partial-order properties	580
62.14.2	Monotonicity of multiset union	580
62.14.3	Termination proofs with multiset orders	581
62.15	Legacy theorem bindings	582
62.16	Naive implementation using lists	583
62.17	BNF setup	585
62.18	Size setup	588
62.19	Lemmas about Size	588
63	More Theorems about the Multiset Order	589
63.1	Alternative Characterizations	589
63.2	Simprocs	592
63.3	Additional facts and instantiations	592
64	Non-negative, non-positive integers and reals	593
64.1	Non-positive integers	594
64.2	Non-negative reals	595
64.3	Non-positive reals	597

65 Numeral Syntax for Types	599
65.1 Numeral Types	599
65.2 <i>num1</i>	599
65.3 Locales for modular arithmetic subtypes	600
65.4 Ring class instances	602
65.5 Order instances	604
65.6 Code setup and type classes for code generation	604
65.7 Syntax	607
65.8 Examples	607
66 ω-words	607
66.1 Type declaration and elementary operations	607
66.2 Subsequence, Prefix, and Suffix	608
66.3 Prepending	611
66.4 The limit set of an ω -word	611
66.5 Index sequences and piecewise definitions	614
67 Combinator syntax for generic, open state monads (single-threaded monads)	615
67.1 Motivation	616
67.2 State transformations and combinators	616
67.3 Monad laws	617
67.4 Do-syntax	617
68 Canonical order on option type	618
69 Futures and parallel lists for code generated towards Isabelle/ML	622
69.1 Futures	622
69.2 Parallel lists	622
70 Input syntax for pattern aliases (or “as-patterns” in Haskell)	623
70.1 Definition	624
70.2 Usage	624
71 Periodic Functions	625
72 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view	627
72.1 Preliminary: auxiliary operations for <i>almost everywhere zero</i> .	627
72.2 Type definition	631
72.3 Additive structure	632
72.4 Multiplicative structure	634
72.5 Single-point mappings	636
72.6 Integral domains	637

72.7 Mapping order	638
72.8 Fundamental mapping notions	638
72.9 Degree	640
72.10 Inductive structure	641
72.11 Quasi-functorial structure	641
72.12 Canonical dense representation of $\text{nat} \Rightarrow_0 'a$	643
72.13 Canonical sparse representation of $'a \Rightarrow_0 'b$	644
72.14 Size estimation	645
72.15 Further mapping operations and properties	646
72.16 Free Abelian Groups Over a Type	647
73 Exponentiation by Squaring	650
74 Preorders with explicit equivalence relation	651
75 Additive group operations on product types	652
75.1 Operations	652
75.2 Class instances	654
76 Roots of real quadratics	654
77 Pretty syntax for Quotient operations	656
78 Quotient infrastructure for the set type	657
78.1 Contravariant set map (vimage) and set relator, rules for the Quotient package	657
79 Quotient infrastructure for the product type	658
79.1 Rules for the Quotient package	658
80 Quotient infrastructure for the option type	660
80.1 Rules for the Quotient package	660
81 Quotient infrastructure for the list type	662
81.1 Rules for the Quotient package	662
82 Quotient infrastructure for the sum type	665
82.1 Rules for the Quotient package	665
83 Quotient types	666
83.1 Equivalence relations and quotient types	667
83.2 Equality on quotients	668
83.3 Picking representing elements	668

84 Ramsey's Theorem	669
84.1 Preliminary definitions	669
84.1.1 The n -element subsets of a set A	669
84.1.2 Partition predicates	671
84.2 Finite versions of Ramsey's theorem	671
84.2.1 Trivial cases	672
84.2.2 Ramsey's theorem with two colours and arbitrary ex- ponents (hypergraph version)	672
84.2.3 Full Ramsey's theorem with multiple colours and ar- bitrary exponents	672
84.2.4 Simple graph version	672
84.3 Preliminaries	672
84.3.1 "Axiom" of Dependent Choice	672
84.3.2 Partition functions	673
84.4 Ramsey's Theorem: Infinitary Version	673
84.5 Disjunctive Well-Foundedness	674
85 Generic reflection and reification	675
86 Assigning lengths to types by type classes	675
87 Saturated arithmetic	678
87.1 The type of saturated naturals	678
88 Set Idioms	681
88.1 Idioms for being a suitable union/intersection of something .	681
88.2 The "Relative to" operator	685
88.3 Signed division: negative results rounded towards zero rather than minus infinity.	687
89 State monad	690
90 Comparators on linear quasi-orders	694
90.1 Basic properties	694
90.2 Fundamental comparator combinators	698
90.3 Direct implementations for linear orders on selected types . .	698
91 Stably sorted lists	699
92 Alternative sorting algorithms	702
92.1 Quicksort	702
92.2 Mergesort	703

93 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming	705
94 A table-based implementation of the reflexive transitive closure	705
95 Binary Tree	707
95.1 <i>map-tree</i>	709
95.2 <i>size</i>	709
95.3 <i>set-tree</i>	709
95.4 <i>subtrees</i>	709
95.5 <i>height</i> and <i>min-height</i>	710
95.6 <i>complete</i>	711
95.7 <i>acomplete</i>	711
95.8 <i>wbalanced</i>	712
95.9 <i>ipl</i>	712
95.10 List of entries	712
95.11 Binary Search Tree	713
95.12 <i>heap</i>	713
95.13 <i>mirror</i>	713
96 Multiset of Elements of Binary Tree	714
97 Unordered pairs	716
98 A type of finite bit strings	719
98.1 Preliminaries	719
98.2 Fundamentals	719
98.2.1 Type definition	719
98.2.2 Basic arithmetic	719
98.2.3 Basic tool setup	721
98.2.4 Basic code generation setup	721
98.2.5 Basic conversions	723
98.2.6 Basic ordering	729
98.3 Enumeration	730
98.4 Bit-wise operations	731
98.5 Conversions including casts	735
98.5.1 Generic unsigned conversion	735
98.5.2 Generic signed conversion	738
98.5.3 More	739
98.6 Arithmetic operations	743
98.7 Ordering	745
98.8 Bit-wise operations	747

98.9 More shift operations	749
98.10 Rotation	749
98.11 Split and cat operations	751
98.12 More on conversions	751
98.13 Testing bits	755
98.14 Word Arithmetic	758
98.15 Transferring goals from words to ints	762
98.16 Order on fixed-length words	763
98.17 Conditions for the addition (etc) of two words to overflow	764
98.18 Some proof tool support	766
98.19 More on overflows and monotonicity	767
98.20 Arithmetic type class instantiations	770
98.21 Word and nat	770
98.22 Cardinality, finiteness of set of words	774
98.23 Bitwise Operations on Words	775
98.23.1 shift functions in terms of lists of bools	779
98.23.2 Mask	781
98.23.3 Slices	783
98.23.4 Revcast	784
98.24 Split and cat	785
98.24.1 Split and slice	785
98.25 Rotation	786
98.25.1 "Word rotation commutes with bit-wise operations	787
98.26 Maximum machine word	788
98.27 Recursion combinator for words	791
98.28 Tool support	792
99 The Field of Integers mod 2	792
100 Pointwise order on product types	796
100.1 Pointwise ordering	796
100.2 Binary infimum and supremum	797
100.3 Top and bottom elements	798
100.4 Complete lattice operations	799
100.5 Complete distributive lattices	800
100.6 Bekic's Theorem	800
100.7 Finite Distributive Lattices	803
100.8 Linear Orders	804
100.9 Finite Linear Orders	805
101 Lexicographic order on lists	805
102 Lexicographic order on lists	807

103	Prefix order on lists as order class instance	808
104	Lexicographic order on product types	809
105	Subsequence Ordering	811
105.1	Definitions and basic lemmas	811
106	Records based on BNF/datatype machinery	812
107	Implementation of mappings with Association Lists	814
108	Avoidance of pattern matching on natural numbers	816
108.1	Case analysis	816
108.2	Preprocessors	816
108.3	One candidate which needs special treatment	817
109	Implementation of natural numbers as binary numerals	817
109.1	Representation	817
109.2	Basic arithmetic	818
109.3	Conversions	820
110	Code generation of prolog programs	820
111	Setup for Numerals	820
112	Implementation of integer numbers by target-language integers	820
113	Implementation of natural numbers by target-language integers	828
113.1	Implementation for <i>nat</i>	828
114	Implementation of natural and integer numbers by target-language integers	832
115	Preprocessor setup for floats implemented by target language numerals	832
116	Abstract type of association lists with unique keys	833
116.1	Preliminaries	833
116.2	Type (<i>'key, 'value</i>) <i>alist</i>	833
116.3	Primitive operations	833
116.4	Abstract operation properties	834
116.5	Further operations	835
116.5.1	Equality	835
116.5.2	Size	835

116.6	Quickcheck generators	835
117	list is a BNF	837
118	Multisets partially implemented by association lists	837
119	Implementation of Red-Black Trees	842
119.1	Datatype of RB trees	842
119.2	Tree properties	842
119.2.1	Content of a tree	842
119.2.2	Search tree properties	843
119.2.3	Tree lookup	844
119.2.4	Red-black properties	846
119.3	Insertion	847
119.4	Deletion	850
119.5	Modifying existing entries	855
119.6	Mapping all entries	856
119.7	Folding over entries	857
119.8	Bulkloading a tree	857
119.9	Building a RBT from a sorted list	858
119.10	Union and intersection of sorted associative lists	864
119.11	Code generator setup	881
120	Abstract type of RBT trees	882
120.1	Type definition	883
120.2	Primitive operations	883
120.3	Derived operations	884
120.4	Abstract lookup properties	884
120.5	Quickcheck generators	887
120.6	Hide implementation details	887
121	Implementation of mappings with Red-Black Trees	887
121.1	Data type and invariant	888
121.2	Operations	888
121.3	Invariant preservation	889
121.4	Map Semantics	889
122	Implementation of sets using RBT trees	889
123	Definition of code datatype constructors	890
124	Deletion of already existing code equations	890

125	Lemmas	890
125.1	Auxiliary lemmas	890
125.2	fold and filter	890
125.3	foldi and Ball	891
125.4	foldi and Bex	891
125.5	folding over non empty trees and selecting the minimal and maximal element	891
125.5.1	concrete	891
125.5.2	abstract	894
126	Code equations	895
127	Common constants	900
128	Pairs	900
129	Filters	901
130	Bounded quantifiers	901
131	Operations on Predicates	901
132	Setup for Numerals	901
133	Arithmetic operations	901
133.1	Arithmetic on naturals and integers	901
133.2	Inductive definitions for ordering on naturals	902
134	Alternative list definitions	902
134.1	Alternative rules for <i>length</i>	902
134.2	Alternative rules for <i>list-all2</i>	902
134.3	Alternative rules for membership in lists	903
135	Setup for String.literal	903
136	Simplification rules for optimisation	903
137A	Prototype of Quickcheck based on the Predicate Com- piler	904
138	TFL: recursive function definitions	904
138.1	Lemmas for TFL	904
138.2	Rule setup	905
139	Program extraction from proofs involving datatypes and in- ductive predicates	905

140 **Refute**

906

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 $\mapsto$ v)
  <proof>
```

```
corollary update-conv: map-of (update k v al) k' = ((map-of al)(k $\mapsto$ v)) k'
  <proof>
```

```
lemma dom-update: fst ` set (update k v al) = {k}  $\cup$  fst ` set al
  <proof>
```

```
lemma update-keys:
  map fst (update k v al) =
    (if k  $\in$  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  $\neq$  k  $\implies$  update k v [q $\leftarrow$ ps. fst q  $\neq$  a] = [q $\leftarrow$ update k v ps. fst q  $\neq$  a]
  <proof>
```

```
lemma update-triv: map-of al k = Some v  $\implies$  update k v al = al
  <proof>
```

```
lemma update-nonempty [simp]: update k v al  $\neq$  []
  <proof>
```

lemma *update-eqD*: $\text{update } k \ v \ al = \text{update } k \ v' \ al' \implies v = v'$
 ⟨proof⟩

lemma *update-last [simp]*: $\text{update } k \ v \ (\text{update } k \ v' \ al) = \text{update } k \ v \ al$
 ⟨proof⟩

Note that the lists are not necessarily the same: $\text{update } k \ v \ (\text{update } k' \ v' \ []) = [(k', v'), (k, v)]$ and $\text{update } k' \ v' \ (\text{update } k \ v \ []) = [(k, v), (k', v')]$.

lemma *update-swap*:
 $k \neq k' \implies \text{map-of } (\text{update } k \ v \ (\text{update } k' \ v' \ al)) = \text{map-of } (\text{update } k' \ v' \ (\text{update } k \ v \ al))$
 ⟨proof⟩

lemma *update-Some-unfold*:
 $\text{map-of } (\text{update } k \ v \ al) \ x = \text{Some } y \iff$
 $x = k \wedge v = y \vee x \neq k \wedge \text{map-of } al \ x = \text{Some } y$
 ⟨proof⟩

lemma *image-update [simp]*: $x \notin A \implies \text{map-of } (\text{update } x \ y \ al) \ `A = \text{map-of } al \ `A$
 ⟨proof⟩ **definition** *updates* ::
 $'key \ list \Rightarrow 'val \ list \Rightarrow ('key \times 'val) \ list \Rightarrow ('key \times 'val) \ list$
where $\text{updates } ks \ vs = \text{fold } (\text{case-prod } \text{update}) \ (\text{zip } ks \ vs)$

lemma *updates-simps [simp]*:
 $\text{updates } [] \ vs \ ps = ps$
 $\text{updates } ks \ [] \ ps = ps$
 $\text{updates } (k \# ks) \ (v \# vs) \ ps = \text{updates } ks \ vs \ (\text{update } k \ v \ ps)$
 ⟨proof⟩

lemma *updates-key-simp [simp]*:
 $\text{updates } (k \ \# \ ks) \ vs \ ps =$
 $(\text{case } vs \ \text{of } [] \Rightarrow ps \mid v \ \# \ vs \Rightarrow \text{updates } ks \ vs \ (\text{update } k \ v \ ps))$
 ⟨proof⟩

lemma *updates-conv'*: $\text{map-of } (\text{updates } ks \ vs \ al) = (\text{map-of } al)(ks[\mapsto]vs)$
 ⟨proof⟩

lemma *updates-conv*: $\text{map-of } (\text{updates } ks \ vs \ al) \ k = ((\text{map-of } al)(ks[\mapsto]vs)) \ k$
 ⟨proof⟩

lemma *distinct-updates*:
assumes $\text{distinct } (\text{map } \text{fst } al)$
shows $\text{distinct } (\text{map } \text{fst } (\text{updates } ks \ vs \ al))$
 ⟨proof⟩

lemma *updates-append1 [simp]*: $\text{size } ks < \text{size } vs \implies$
 $\text{updates } (ks@[k]) \ vs \ al = \text{update } k \ (vs[\text{size } ks]) \ (\text{updates } ks \ vs \ al)$
 ⟨proof⟩

lemma *updates-list-update-drop* [simp]:
 $\text{size } ks \leq i \implies i < \text{size } vs \implies$
 $\text{updates } ks \text{ (vs[i:=v]) } al = \text{updates } ks \text{ vs } al$
 ⟨proof⟩

lemma *update-updates-conv-if*:
 $\text{map-of (updates } xs \text{ ys (update } x \text{ y } al)) =$
 map-of
 $(\text{if } x \in \text{set (take (length ys) xs)}$
 $\text{then updates } xs \text{ ys } al$
 $\text{else (update } x \text{ y (updates } xs \text{ ys } al)))$
 ⟨proof⟩

lemma *updates-twist* [simp]:
 $k \notin \text{set } ks \implies$
 $\text{map-of (updates } ks \text{ vs (update } k \text{ v } al)) = \text{map-of (update } k \text{ v (updates } ks \text{ vs } al))$
 ⟨proof⟩

lemma *updates-apply-notin* [simp]:
 $k \notin \text{set } ks \implies \text{map-of (updates } ks \text{ vs } al) k = \text{map-of } al \text{ } k$
 ⟨proof⟩

lemma *updates-append-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies \text{updates (xs @ zs) ys } al = \text{updates } xs \text{ ys } al$
 ⟨proof⟩

lemma *updates-append2-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies \text{updates } xs \text{ (ys @ zs) } al = \text{updates } xs \text{ ys } al$
 ⟨proof⟩

1.2 delete

qualified definition *delete* :: 'key $\Rightarrow$ ('key $\times$ 'val) list $\Rightarrow$ ('key $\times$ 'val) list
where *delete-eq*: *delete* *k* = *filter* ($\lambda(k', -). k \neq k'$)

lemma *delete-simps* [simp]:
 $\text{delete } k \text{ []} = \text{[]}$
 $\text{delete } k \text{ (p \# ps)} = (\text{if fst } p = k \text{ then delete } k \text{ ps else } p \# \text{delete } k \text{ ps})$
 ⟨proof⟩

lemma *delete-conv'*: $\text{map-of (delete } k \text{ } al) = (\text{map-of } al)(k := \text{None})$
 ⟨proof⟩

corollary *delete-conv*: $\text{map-of (delete } k \text{ } al) \text{ } k' = ((\text{map-of } al)(k := \text{None})) \text{ } k'$
 ⟨proof⟩

lemma *delete-keys*: $\text{map fst (delete } k \text{ } al) = \text{removeAll } k \text{ (map fst } al)$
 ⟨proof⟩

lemma *distinct-delete*:

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

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

lemma *delete-idem*: $\text{delete } k \text{ } (\text{delete } k \text{ } al) = \text{delete } k \text{ } al$
 ⟨*proof*⟩

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

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

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

lemma *delete-update-same*: $\text{delete } k \text{ } (\text{update } k \text{ } v \text{ } al) = \text{delete } k \text{ } al$
 ⟨*proof*⟩

lemma *delete-update*: $k \neq l \implies \text{delete } l \text{ } (\text{update } k \text{ } v \text{ } al) = \text{update } k \text{ } v \text{ } (\text{delete } l \text{ } al)$
 ⟨*proof*⟩

lemma *delete-twist*: $\text{delete } x \text{ } (\text{delete } y \text{ } al) = \text{delete } y \text{ } (\text{delete } x \text{ } al)$
 ⟨*proof*⟩

lemma *length-delete-le*: $\text{length } (\text{delete } k \text{ } al) \leq \text{length } al$
 ⟨*proof*⟩

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

qualified primrec *update-with-aux* ::

$'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

where

$\text{update-with-aux } v \text{ } k \text{ } f \text{ } [] = [(k, f v)]$

$| \text{update-with-aux } v \text{ } k \text{ } f \text{ } (p \# ps) =$
 $(\text{if } (\text{fst } p = k) \text{ then } (k, f (\text{snd } p)) \# ps \text{ else } p \# \text{update-with-aux } v \text{ } k \text{ } f \text{ } ps)$

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

qualified fun *delete-aux* :: $'key \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

where

$\text{delete-aux } k \text{ } [] = []$

$| \text{delete-aux } k \text{ } ((k', v) \# xs) = (\text{if } k = k' \text{ then } xs \text{ else } (k', v) \# \text{delete-aux } k \text{ } xs)$

lemma *map-of-update-with-aux'*:

$$\begin{aligned} & \text{map-of } (\text{update-with-aux } v \ k \ f \ ps) \ k' = \\ & ((\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \text{ of } None \Rightarrow f \ v \mid Some \ v \Rightarrow f \ v))) \ k' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *map-of-update-with-aux*:

$$\begin{aligned} & \text{map-of } (\text{update-with-aux } v \ k \ f \ ps) = \\ & (\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \text{ of } None \Rightarrow f \ v \mid Some \ v \Rightarrow f \ v)) \\ & \langle \text{proof} \rangle \end{aligned}$$

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

$\langle \text{proof} \rangle$

lemma *distinct-update-with-aux [simp]*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } (\text{update-with-aux } v \ k \ f \ ps)) = \text{distinct } (\text{map } \text{fst } ps) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *set-update-with-aux*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \\ & \text{set } (\text{update-with-aux } v \ k \ f \ xs) = \\ & (\text{set } xs - \{k\} \times UNIV \cup \{(k, f \ (\text{case map-of } xs \ k \text{ of } None \Rightarrow v \mid Some \ v \Rightarrow \\ & v))\}) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *set-delete-aux*: $\text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{set } (\text{delete-aux } k \ xs) = \text{set } xs - \{k\} \times UNIV$

$\langle \text{proof} \rangle$

lemma *dom-delete-aux*: $\text{distinct } (\text{map } \text{fst } ps) \Longrightarrow \text{fst} \text{ ' set } (\text{delete-aux } k \ ps) = \text{fst} \text{ ' set } ps - \{k\}$

$\langle \text{proof} \rangle$

lemma *distinct-delete-aux [simp]*: $\text{distinct } (\text{map } \text{fst } ps) \Longrightarrow \text{distinct } (\text{map } \text{fst } (\text{delete-aux } k \ ps))$

$\langle \text{proof} \rangle$

lemma *map-of-delete-aux'*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{map-of } (\text{delete-aux } k \ xs) = (\text{map-of } xs)(k := None) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *map-of-delete-aux*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{map-of } (\text{delete-aux } k \ xs) \ k' = ((\text{map-of } xs)(k := None)) \\ & k' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *delete-aux-eq-Nil-conv*: $\text{delete-aux } k \ ts = [] \longleftrightarrow ts = [] \vee (\exists v. ts = [(k, v)])$

$\langle \text{proof} \rangle$

1.4 restrict

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

lemma restr-simps [simp]:

$\text{restrict } A [] = []$

$\text{restrict } A (p \# ps) = (\text{if } \text{fst } p \in A \text{ then } p \# \text{restrict } A \text{ ps else } \text{restrict } A \text{ ps})$

$\langle \text{proof} \rangle$

lemma restr-conv' : $\text{map-of } (\text{restrict } A \text{ al}) = ((\text{map-of } \text{al})|' A)$

$\langle \text{proof} \rangle$

corollary restr-conv : $\text{map-of } (\text{restrict } A \text{ al}) k = ((\text{map-of } \text{al})|' A) k$

$\langle \text{proof} \rangle$

lemma distinct-restr : $\text{distinct } (\text{map } \text{fst } \text{al}) \implies \text{distinct } (\text{map } \text{fst } (\text{restrict } A \text{ al}))$

$\langle \text{proof} \rangle$

lemma restr-empty [simp]:

$\text{restrict } \{\} \text{ al} = []$

$\text{restrict } A [] = []$

$\langle \text{proof} \rangle$

lemma restr-in [simp]: $x \in A \implies \text{map-of } (\text{restrict } A \text{ al}) x = \text{map-of } \text{al } x$

$\langle \text{proof} \rangle$

lemma restr-out [simp]: $x \notin A \implies \text{map-of } (\text{restrict } A \text{ al}) x = \text{None}$

$\langle \text{proof} \rangle$

lemma dom-restr [simp]: $\text{fst } ' \text{ set } (\text{restrict } A \text{ al}) = \text{fst } ' \text{ set } \text{al} \cap A$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma restr-restr [simp]: $\text{restrict } A (\text{restrict } B \text{ al}) = \text{restrict } (A \cap B) \text{ al}$

$\langle \text{proof} \rangle$

lemma restr-update [simp]:

$\text{map-of } (\text{restrict } D (\text{update } x y \text{ al})) =$

$\text{map-of } ((\text{if } x \in D \text{ then } (\text{update } x y (\text{restrict } (D - \{x\}) \text{ al})) \text{ else } \text{restrict } D \text{ al}))$

$\langle \text{proof} \rangle$

lemma restr-delete [simp]:

$\text{delete } x (\text{restrict } D \text{ al}) = (\text{if } x \in D \text{ then } \text{restrict } (D - \{x\}) \text{ al else } \text{restrict } D \text{ al})$

$\langle \text{proof} \rangle$

lemma *update-restr*:

$\text{map-of } (\text{update } x \ y \ (\text{restrict } D \ al)) = \text{map-of } (\text{update } x \ y \ (\text{restrict } (D - \{x\}) \ al))$
 $\langle \text{proof} \rangle$

lemma *update-restr-conv* [simp]:

$x \in D \implies$
 $\text{map-of } (\text{update } x \ y \ (\text{restrict } D \ al)) = \text{map-of } (\text{update } x \ y \ (\text{restrict } (D - \{x\}) \ al))$
 $\langle \text{proof} \rangle$

lemma *restr-updates* [simp]:

$\text{length } xs = \text{length } ys \implies \text{set } xs \subseteq D \implies$
 $\text{map-of } (\text{restrict } D \ (\text{updates } xs \ ys \ al)) =$
 $\text{map-of } (\text{updates } xs \ ys \ (\text{restrict } (D - \text{set } xs) \ al))$
 $\langle \text{proof} \rangle$

lemma *restr-delete-twist*: $(\text{restrict } A \ (\text{delete } a \ ps)) = \text{delete } a \ (\text{restrict } A \ ps)$
 $\langle \text{proof} \rangle$

1.5 clearjunk

qualified function *clearjunk* :: $('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

where

$\text{clearjunk } [] = []$
 $| \text{clearjunk } (p \# ps) = p \# \text{clearjunk } (\text{delete } (\text{fst } p) \ ps)$
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

lemma *map-of-clearjunk*: $\text{map-of } (\text{clearjunk } al) = \text{map-of } al$
 $\langle \text{proof} \rangle$

lemma *clearjunk-keys-set*: $\text{set } (\text{map } \text{fst } (\text{clearjunk } al)) = \text{set } (\text{map } \text{fst } al)$
 $\langle \text{proof} \rangle$

lemma *dom-clearjunk*: $\text{fst } ' \text{set } (\text{clearjunk } al) = \text{fst } ' \text{set } al$
 $\langle \text{proof} \rangle$

lemma *distinct-clearjunk* [simp]: $\text{distinct } (\text{map } \text{fst } (\text{clearjunk } al))$
 $\langle \text{proof} \rangle$

lemma *ran-clearjunk*: $\text{ran } (\text{map-of } (\text{clearjunk } al)) = \text{ran } (\text{map-of } al)$
 $\langle \text{proof} \rangle$

lemma *ran-map-of*: $\text{ran } (\text{map-of } al) = \text{snd } ' \text{set } (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *graph-map-of*: $\text{Map.graph } (\text{map-of } al) = \text{set } (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-update*: $\text{clearjunk } (\text{update } k \ v \ al) = \text{update } k \ v \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-updates*: $\text{clearjunk } (\text{updates } ks \ vs \ al) = \text{updates } ks \ vs \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-delete*: $\text{clearjunk } (\text{delete } x \ al) = \text{delete } x \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-restrict*: $\text{clearjunk } (\text{restrict } A \ al) = \text{restrict } A \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *distinct-clearjunk-id* [simp]: $\text{distinct } (\text{map } \text{fst } al) \implies \text{clearjunk } al = al$
 $\langle \text{proof} \rangle$

lemma *clearjunk-idem*: $\text{clearjunk } (\text{clearjunk } al) = \text{clearjunk } al$
 $\langle \text{proof} \rangle$

lemma *length-clearjunk*: $\text{length } (\text{clearjunk } al) \leq \text{length } al$
 $\langle \text{proof} \rangle$

lemma *delete-map*:
assumes $\bigwedge kv. \text{fst } (f \ kv) = \text{fst } kv$
shows $\text{delete } k \ (\text{map } f \ ps) = \text{map } f \ (\text{delete } k \ ps)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-map*:
assumes $\bigwedge kv. \text{fst } (f \ kv) = \text{fst } kv$
shows $\text{clearjunk } (\text{map } f \ ps) = \text{map } f \ (\text{clearjunk } ps)$
 $\langle \text{proof} \rangle$

1.6 map-ran

definition *map-ran* :: $('key \Rightarrow 'val1 \Rightarrow 'val2) \Rightarrow ('key \times 'val1) \text{ list} \Rightarrow ('key \times 'val2) \text{ list}$
where $\text{map-ran } f = \text{map } (\lambda(k, v). (k, f \ k \ v))$

lemma *map-ran-simps* [simp]:
 $\text{map-ran } f \ [] = []$
 $\text{map-ran } f \ ((k, v) \# ps) = (k, f \ k \ v) \# \text{map-ran } f \ ps$
 $\langle \text{proof} \rangle$

lemma *map-ran-Cons-sel*: $\text{map-ran } f \ (p \# ps) = (\text{fst } p, f \ (\text{fst } p) \ (\text{snd } p)) \# \text{map-ran } f \ ps$
 $\langle \text{proof} \rangle$

lemma *length-map-ran[simp]*: $\text{length } (\text{map-ran } f \text{ al}) = \text{length } \text{al}$
 $\langle \text{proof} \rangle$

lemma *map-fst-map-ran[simp]*: $\text{map } \text{fst } (\text{map-ran } f \text{ al}) = \text{map } \text{fst } \text{al}$
 $\langle \text{proof} \rangle$

lemma *dom-map-ran*: $\text{fst } ' \text{ set } (\text{map-ran } f \text{ al}) = \text{fst } ' \text{ set } \text{al}$
 $\langle \text{proof} \rangle$

lemma *map-ran-conv*: $\text{map-of } (\text{map-ran } f \text{ al}) \text{ k} = \text{map-option } (f \text{ k}) (\text{map-of } \text{al } \text{k})$
 $\langle \text{proof} \rangle$

lemma *distinct-map-ran*: $\text{distinct } (\text{map } \text{fst } \text{al}) \implies \text{distinct } (\text{map } \text{fst } (\text{map-ran } f \text{ al}))$
 $\langle \text{proof} \rangle$

lemma *map-ran-filter*: $\text{map-ran } f \text{ } [p \leftarrow \text{ps}. \text{fst } p \neq a] = [p \leftarrow \text{map-ran } f \text{ ps}. \text{fst } p \neq a]$
 $\langle \text{proof} \rangle$

lemma *clearjunk-map-ran*: $\text{clearjunk } (\text{map-ran } f \text{ al}) = \text{map-ran } f (\text{clearjunk } \text{al})$
 $\langle \text{proof} \rangle$

1.7 merge

qualified definition *merge* :: $('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$
where $\text{merge } \text{qs } \text{ps} = \text{foldr } (\lambda(k, v). \text{update } k \text{ v}) \text{ ps } \text{qs}$

lemma *merge-simps [simp]*:
 $\text{merge } \text{qs } [] = \text{qs}$
 $\text{merge } \text{qs } (p \# \text{ps}) = \text{update } (\text{fst } p) (\text{snd } p) (\text{merge } \text{qs } \text{ps})$
 $\langle \text{proof} \rangle$

lemma *merge-updates*: $\text{merge } \text{qs } \text{ps} = \text{updates } (\text{rev } (\text{map } \text{fst } \text{ps})) (\text{rev } (\text{map } \text{snd } \text{ps})) \text{ qs}$
 $\langle \text{proof} \rangle$

lemma *dom-merge*: $\text{fst } ' \text{ set } (\text{merge } \text{xs } \text{ys}) = \text{fst } ' \text{ set } \text{xs} \cup \text{fst } ' \text{ set } \text{ys}$
 $\langle \text{proof} \rangle$

lemma *distinct-merge*: $\text{distinct } (\text{map } \text{fst } \text{xs}) \implies \text{distinct } (\text{map } \text{fst } (\text{merge } \text{xs } \text{ys}))$
 $\langle \text{proof} \rangle$

lemma *clearjunk-merge*: $\text{clearjunk } (\text{merge } \text{xs } \text{ys}) = \text{merge } (\text{clearjunk } \text{xs}) \text{ ys}$
 $\langle \text{proof} \rangle$

lemma *merge-conv'*: $\text{map-of } (\text{merge } \text{xs } \text{ys}) = \text{map-of } \text{xs} ++ \text{map-of } \text{ys}$

$\langle \text{proof} \rangle$

corollary *merge-conv*: $\text{map-of } (\text{merge } xs \ ys) \ k = (\text{map-of } xs \ ++ \ \text{map-of } ys) \ k$
 $\langle \text{proof} \rangle$

lemma *merge-empty*: $\text{map-of } (\text{merge } [] \ ys) = \text{map-of } ys$
 $\langle \text{proof} \rangle$

lemma *merge-assoc* [simp]: $\text{map-of } (\text{merge } m1 \ (\text{merge } m2 \ m3)) = \text{map-of } (\text{merge } (\text{merge } m1 \ m2) \ m3)$
 $\langle \text{proof} \rangle$

lemma *merge-Some-iff*:
 $\text{map-of } (\text{merge } m \ n) \ k = \text{Some } x \longleftrightarrow$
 $\text{map-of } n \ k = \text{Some } x \vee \text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{Some } x$
 $\langle \text{proof} \rangle$

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

lemma *merge-find-right* [simp]: $\text{map-of } n \ k = \text{Some } v \implies \text{map-of } (\text{merge } m \ n) \ k = \text{Some } v$
 $\langle \text{proof} \rangle$

lemma *merge-None* [iff]: $(\text{map-of } (\text{merge } m \ n) \ k = \text{None}) = (\text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{None})$
 $\langle \text{proof} \rangle$

lemma *merge-upd* [simp]: $\text{map-of } (\text{merge } m \ (\text{update } k \ v \ n)) = \text{map-of } (\text{update } k \ v \ (\text{merge } m \ n))$
 $\langle \text{proof} \rangle$

lemma *merge-updatess* [simp]:
 $\text{map-of } (\text{merge } m \ (\text{updates } xs \ ys \ n)) = \text{map-of } (\text{updates } xs \ ys \ (\text{merge } m \ n))$
 $\langle \text{proof} \rangle$

lemma *merge-append*: $\text{map-of } (xs \ @ \ ys) = \text{map-of } (\text{merge } ys \ xs)$
 $\langle \text{proof} \rangle$

1.8 compose

qualified function *compose* :: $('key \times 'a) \ \text{list} \Rightarrow ('a \times 'b) \ \text{list} \Rightarrow ('key \times 'b) \ \text{list}$
where

$\text{compose } [] \ ys = []$
 $| \text{compose } (x \ \# \ xs) \ ys =$
 $\quad (\text{case } \text{map-of } ys \ (\text{snd } x) \ \text{of}$
 $\quad \quad \text{None} \Rightarrow \text{compose } (\text{delete } (\text{fst } x) \ xs) \ ys$
 $\quad \quad | \text{Some } v \Rightarrow (\text{fst } x, v) \ \# \ \text{compose } xs \ ys)$
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

lemma *compose-first-None* [simp]: $\text{map-of } xs \ k = \text{None} \implies \text{map-of } (\text{compose } xs \ ys) \ k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *compose-conv*: $\text{map-of } (\text{compose } xs \ ys) \ k = (\text{map-of } ys \circ_m \text{map-of } xs) \ k$
 $\langle \text{proof} \rangle$

lemma *compose-conv'*: $\text{map-of } (\text{compose } xs \ ys) = (\text{map-of } ys \circ_m \text{map-of } xs)$
 $\langle \text{proof} \rangle$

lemma *compose-first-Some* [simp]: $\text{map-of } xs \ k = \text{Some } v \implies \text{map-of } (\text{compose } xs \ ys) \ k = \text{map-of } ys \ v$
 $\langle \text{proof} \rangle$

lemma *dom-compose*: $\text{fst } \text{'set } (\text{compose } xs \ ys) \subseteq \text{fst } \text{'set } xs$
 $\langle \text{proof} \rangle$

lemma *distinct-compose*:
assumes *distinct* (map fst xs)
shows *distinct* (map fst (compose xs ys))
 $\langle \text{proof} \rangle$

lemma *compose-delete-twist*: $\text{compose } (\text{delete } k \ xs) \ ys = \text{delete } k \ (\text{compose } xs \ ys)$
 $\langle \text{proof} \rangle$

lemma *compose-clearjunk*: $\text{compose } xs \ (\text{clearjunk } ys) = \text{compose } xs \ ys$
 $\langle \text{proof} \rangle$

lemma *clearjunk-compose*: $\text{clearjunk } (\text{compose } xs \ ys) = \text{compose } (\text{clearjunk } xs) \ ys$
 $\langle \text{proof} \rangle$

lemma *compose-empty* [simp]: $\text{compose } xs \ [] = []$
 $\langle \text{proof} \rangle$

lemma *compose-Some-iff*:
 $(\text{map-of } (\text{compose } xs \ ys) \ k = \text{Some } v) \longleftrightarrow$
 $(\exists k'. \text{map-of } xs \ k = \text{Some } k' \wedge \text{map-of } ys \ k' = \text{Some } v)$
 $\langle \text{proof} \rangle$

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

1.9 *map-entry*

qualified fun *map-entry* :: 'key $\Rightarrow$ ('val $\Rightarrow$ 'val) $\Rightarrow$ ('key $\times$ 'val) list $\Rightarrow$ ('key $\times$ 'val) list

where

map-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 $\Rightarrow$ None | Some *v'* $\Rightarrow$ 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 $\Rightarrow$ 'val $\Rightarrow$ ('val $\Rightarrow$ 'val) $\Rightarrow$ ('key $\times$ 'val) list $\Rightarrow$ ('key $\times$ '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 $\Rightarrow$ Some *v* | Some *v'* $\Rightarrow$ 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 Adhoc overloading of constants based on their types

```
theory Adhoc-Overloading
  imports Main
  keywords adhoc-overloading no-adhoc-overloading :: thy-decl
begin
```

$\langle ML \rangle$

end

3 Pointwise instantiation of functions to algebra type classes

```
theory Function-Algebras
  imports Main
begin
```

Pointwise operations

```
instantiation fun :: (type, plus) plus
begin
```

```
definition  $f + g = (\lambda x. f\ x + g\ x)$ 
instance  $\langle proof \rangle$ 
```

end

```
lemma plus-fun-apply [simp]:
   $(f + g)\ x = f\ x + g\ x$ 
   $\langle proof \rangle$ 
```

```
instantiation fun :: (type, zero) zero
begin
```

```
definition  $0 = (\lambda x. 0)$ 
instance  $\langle proof \rangle$ 
```

end

```
lemma zero-fun-apply [simp]:
   $0\ x = 0$ 
   $\langle proof \rangle$ 
```

```
instantiation fun :: (type, times) times
begin
```

```
definition  $f * g = (\lambda x. f\ x * g\ x)$ 
```

instance $\langle proof \rangle$

end

lemma *times-fun-apply* [simp]:
 $(f * g) x = f x * g x$
 $\langle proof \rangle$

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

definition *1* = ($\lambda x. 1$)
instance $\langle proof \rangle$

end

lemma *one-fun-apply* [simp]:
 $1 x = 1$
 $\langle proof \rangle$

Additive structures

instance *fun* :: (type, semigroup-add) semigroup-add
 $\langle proof \rangle$

instance *fun* :: (type, cancel-semigroup-add) cancel-semigroup-add
 $\langle proof \rangle$

instance *fun* :: (type, ab-semigroup-add) ab-semigroup-add
 $\langle proof \rangle$

instance *fun* :: (type, cancel-ab-semigroup-add) cancel-ab-semigroup-add
 $\langle proof \rangle$

instance *fun* :: (type, monoid-add) monoid-add
 $\langle proof \rangle$

instance *fun* :: (type, comm-monoid-add) comm-monoid-add
 $\langle proof \rangle$

instance *fun* :: (type, cancel-comm-monoid-add) cancel-comm-monoid-add $\langle proof \rangle$

instance *fun* :: (type, group-add) group-add
 $\langle proof \rangle$

instance *fun* :: (type, ab-group-add) ab-group-add
 $\langle proof \rangle$

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 = (λx::'a. numeral n)>
<proof>

lemma *numeral-fun-apply [simp]*:
<numeral n x = numeral n>
<proof>

lemma *of-nat-fun*: *of-nat n = (λx::'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

```

4 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. See the comments at the top of `Big0.thy`.

```

instantiation set :: (plus) plus
begin

definition plus-set :: 'a::plus set  $\Rightarrow$  'a set  $\Rightarrow$  '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  $\Rightarrow$  'a set  $\Rightarrow$  '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]: $(0::'a::\text{zero set}) = \{0\}$

instance $\langle \text{proof} \rangle$

end

instantiation *set* :: (one) one
begin

definition *set-one*[simp]: $(1::'a::\text{one set}) = \{1\}$

instance $\langle \text{proof} \rangle$

end

definition *elt-set-plus* :: $'a::\text{plus} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ (**infixl** +o 70)
where $a +o B = \{c. \exists b \in B. c = a + b\}$

definition *elt-set-times* :: $'a::\text{times} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ (**infixl** *o 80)
where $a *o B = \{c. \exists b \in B. c = a * b\}$

abbreviation (*input*) *elt-set-eq* :: $'a \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ (**infix** =o 50)
where $x =o A \equiv x \in A$

instance *set* :: (semigroup-add) semigroup-add
 $\langle \text{proof} \rangle$

instance *set* :: (ab-semigroup-add) ab-semigroup-add
 $\langle \text{proof} \rangle$

instance *set* :: (monoid-add) monoid-add
 $\langle \text{proof} \rangle$

instance *set* :: (comm-monoid-add) comm-monoid-add
 $\langle \text{proof} \rangle$

instance *set* :: (semigroup-mult) semigroup-mult
 $\langle \text{proof} \rangle$

instance *set* :: (ab-semigroup-mult) ab-semigroup-mult
 $\langle \text{proof} \rangle$

instance *set* :: (monoid-mult) monoid-mult
 $\langle \text{proof} \rangle$

instance *set* :: (comm-monoid-mult) comm-monoid-mult
 $\langle \text{proof} \rangle$

lemma *set-plus-intro* [intro]: $a \in C \Longrightarrow b \in D \Longrightarrow a + b \in C + D$

$\langle \text{proof} \rangle$

lemma *set-plus-elim*:

assumes $x \in A + B$

obtains $a\ b$ **where** $x = a + b$ **and** $a \in A$ **and** $b \in B$

$\langle \text{proof} \rangle$

lemma *set-plus-intro2* [intro]: $b \in C \implies a + b \in a + o\ C$

$\langle \text{proof} \rangle$

lemma *set-plus-rearrange*: $(a + o\ C) + (b + o\ D) = (a + b) + o\ (C + D)$

for $a\ b :: 'a::\text{comm-monoid-add}$

$\langle \text{proof} \rangle$

lemma *set-plus-rearrange2*: $a + o\ (b + o\ C) = (a + b) + o\ C$

for $a\ b :: 'a::\text{semigroup-add}$

$\langle \text{proof} \rangle$

lemma *set-plus-rearrange3*: $(a + o\ B) + C = a + o\ (B + C)$

for $a :: 'a::\text{semigroup-add}$

$\langle \text{proof} \rangle$

theorem *set-plus-rearrange4*: $C + (a + o\ D) = a + o\ (C + D)$

for $a :: 'a::\text{comm-monoid-add}$

$\langle \text{proof} \rangle$

lemmas *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2 set-plus-rearrange3 set-plus-rearrange4*

lemma *set-plus-mono* [intro!]: $C \subseteq D \implies a + o\ C \subseteq a + o\ D$

$\langle \text{proof} \rangle$

lemma *set-plus-mono2* [intro]: $C \subseteq D \implies E \subseteq F \implies C + E \subseteq D + F$

for $C\ D\ E\ F :: 'a::\text{plus set}$

$\langle \text{proof} \rangle$

lemma *set-plus-mono3* [intro]: $a \in C \implies a + o\ D \subseteq C + D$

$\langle \text{proof} \rangle$

lemma *set-plus-mono4* [intro]: $a \in C \implies a + o\ D \subseteq D + C$

for $a :: 'a::\text{comm-monoid-add}$

$\langle \text{proof} \rangle$

lemma *set-plus-mono5*: $a \in C \implies B \subseteq D \implies a + o\ B \subseteq C + D$

$\langle \text{proof} \rangle$

lemma *set-plus-mono-b*: $C \subseteq D \implies x \in a + o\ C \implies x \in a + o\ D$

$\langle \text{proof} \rangle$

lemma *set-plus-mono2-b*: $C \subseteq D \implies E \subseteq F \implies x \in C + E \implies x \in D + F$
 ⟨proof⟩

lemma *set-plus-mono3-b*: $a \in C \implies x \in a + o D \implies x \in C + D$
 ⟨proof⟩

lemma *set-plus-mono4-b*: $a \in C \implies x \in a + o D \implies x \in D + C$
for $a \ x :: 'a::comm-monoid-add$
 ⟨proof⟩

lemma *set-zero-plus [simp]*: $0 + o C = C$
for $C :: 'a::comm-monoid-add set$
 ⟨proof⟩

lemma *set-zero-plus2*: $0 \in A \implies B \subseteq A + B$
for $A \ B :: 'a::comm-monoid-add set$
 ⟨proof⟩

lemma *set-plus-imp-minus*: $a \in b + o C \implies a - b \in C$
for $a \ b :: 'a::ab-group-add$
 ⟨proof⟩

lemma *set-minus-imp-plus*: $a - b \in C \implies a \in b + o C$
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]*: $a \in C \implies b \in D \implies a * b \in C * D$
 ⟨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!]*: $b \in C \implies a * b \in a * o C$
 ⟨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 $a :: 'a::\text{semigroup-mult}$
 $\langle \text{proof} \rangle$

theorem *set-times-rearrange4*: $C * (a * o D) = a * o (C * D)$
for $a :: 'a::\text{comm-monoid-mult}$
 $\langle \text{proof} \rangle$

lemmas *set-times-rearranges* = *set-times-rearrange set-times-rearrange2 set-times-rearrange3 set-times-rearrange4*

lemma *set-times-mono* [intro]: $C \subseteq D \implies a * o C \subseteq a * o D$
 $\langle \text{proof} \rangle$

lemma *set-times-mono2* [intro]: $C \subseteq D \implies E \subseteq F \implies C * E \subseteq D * F$
for $C D E F :: 'a::\text{times set}$
 $\langle \text{proof} \rangle$

lemma *set-times-mono3* [intro]: $a \in C \implies a * o D \subseteq C * D$
 $\langle \text{proof} \rangle$

lemma *set-times-mono4* [intro]: $a \in C \implies a * o D \subseteq D * C$
for $a :: 'a::\text{comm-monoid-mult}$
 $\langle \text{proof} \rangle$

lemma *set-times-mono5*: $a \in C \implies B \subseteq D \implies a * o B \subseteq C * D$
 $\langle \text{proof} \rangle$

lemma *set-times-mono-b*: $C \subseteq D \implies x \in a * o C \implies x \in a * o D$
 $\langle \text{proof} \rangle$

lemma *set-times-mono2-b*: $C \subseteq D \implies E \subseteq F \implies x \in C * E \implies x \in D * F$
 $\langle \text{proof} \rangle$

lemma *set-times-mono3-b*: $a \in C \implies x \in a * o D \implies x \in C * D$
 $\langle \text{proof} \rangle$

lemma *set-times-mono4-b*: $a \in C \implies x \in a * o D \implies x \in D * C$
for $a x :: 'a::\text{comm-monoid-mult}$
 $\langle \text{proof} \rangle$

lemma *set-one-times* [simp]: $1 * o C = C$
for $C :: 'a::\text{comm-monoid-mult set}$
 $\langle \text{proof} \rangle$

lemma *set-times-plus-distrib*: $a * o (b + o C) = (a * b) + o (a * o C)$
for $a b :: 'a::\text{semiring}$
 $\langle \text{proof} \rangle$

lemma *set-times-plus-distrib2*: $a * o (B + C) = (a * o B) + (a * o C)$
for $a :: 'a::semiring$
 $\langle proof \rangle$

lemma *set-times-plus-distrib3*: $(a + o C) * D \subseteq a * o D + C * D$
for $a :: 'a::semiring$
 $\langle proof \rangle$

lemmas *set-times-plus-distribs* =
set-times-plus-distrib
set-times-plus-distrib2

lemma *set-neg-intro*: $a \in (-\ 1) * o C \implies -\ a \in C$
for $a :: 'a::ring-1$
 $\langle proof \rangle$

lemma *set-neg-intro2*: $a \in C \implies -\ a \in (-\ 1) * o C$
for $a :: 'a::ring-1$
 $\langle proof \rangle$

lemma *set-plus-image*: $S + T = (\lambda(x, y). x + y) ` (S \times T)$
 $\langle proof \rangle$

lemma *set-times-image*: $S * T = (\lambda(x, y). x * y) ` (S \times T)$
 $\langle proof \rangle$

lemma *finite-set-plus*: $finite\ s \implies finite\ t \implies finite\ (s + t)$
 $\langle proof \rangle$

lemma *finite-set-times*: $finite\ s \implies finite\ t \implies finite\ (s * t)$
 $\langle proof \rangle$

lemma *set-sum-alt*:
assumes $fin: finite\ I$
shows $sum\ S\ I = \{sum\ s\ I \mid s. \forall i \in I. s\ i \in S\ i\}$
 $(is\ - = ?sum\ I)$
 $\langle proof \rangle$

lemma *sum-set-cond-linear*:
fixes $f :: 'a::comm-monoid-add\ set \Rightarrow 'b::comm-monoid-add\ set$
assumes $[intro!]: \bigwedge A\ B. P\ A \implies P\ B \implies P\ (A + B)\ P\ \{0\}$
and $f: \bigwedge A\ B. P\ A \implies P\ B \implies f\ (A + B) = f\ A + f\ B\ f\ \{0\} = \{0\}$
assumes $all: \bigwedge i. i \in I \implies P\ (S\ i)$
shows $f\ (sum\ S\ I) = sum\ (f \circ S)\ I$
 $\langle proof \rangle$

lemma *sum-set-linear*:
fixes $f :: 'a::comm-monoid-add\ set \Rightarrow 'b::comm-monoid-add\ set$
assumes $\bigwedge A\ B. f(A) + f(B) = f(A + B)\ f\ \{0\} = \{0\}$

shows $f \text{ (sum } S \text{ } I) = \text{sum } (f \circ S) \text{ } I$
 $\langle \text{proof} \rangle$

lemma *set-times-Un-distrib*:

$A * (B \cup C) = A * B \cup A * C$
 $(A \cup B) * C = A * C \cup B * C$
 $\langle \text{proof} \rangle$

lemma *set-times-UNION-distrib*:

$A * \bigcup (M \text{ ' } I) = \bigcup_{i \in I}. A * M \text{ } i$
 $\bigcup (M \text{ ' } I) * A = \bigcup_{i \in I}. M \text{ } i * A$
 $\langle \text{proof} \rangle$

end

5 Big O notation

theory *BigO*

imports

Complex-Main
Function-Algebras
Set-Algebras

begin

This library is designed to support asymptotic “big O” calculations, i.e. reasoning with expressions of the form $f = O(g)$ and $f = g + O(h)$. An earlier version of this library is described in detail in [1].

The main changes in this version are as follows:

- We have eliminated the O operator on sets. (Most uses of this seem to be inessential.)
- We no longer use $+$ as output syntax for $+o$
- Lemmas involving *sumr* have been replaced by more general lemmas involving ‘*sum*’.
- The library has been expanded, with e.g. support for expressions of the form $f < g + O(h)$.

Note also since the Big O library includes rules that demonstrate set inclusion, to use the automated reasoners effectively with the library one should redeclare the theorem *subsetI* as an intro rule, rather than as an *intro!* rule, for example, using **declare** *subsetI* [*del*, *intro*].

5.1 Definitions

definition *bigo* :: $('a \Rightarrow 'b :: \text{linordered-idom}) \Rightarrow ('a \Rightarrow 'b) \text{ set } ((1O'(-')))$

where $O(f:: 'a \Rightarrow 'b) = \{h. \exists c. \forall x. |h\ x| \leq c * |f\ x|\}$

lemma *bigo-pos-const*:

$(\exists c:: 'a:: \text{linordered-idom}. \forall x. |h\ x| \leq c * |f\ x|) \longleftrightarrow$
 $(\exists c. 0 < c \wedge (\forall x. |h\ x| \leq c * |f\ x|))$
 $\langle \text{proof} \rangle$

lemma *bigo-alt-def*: $O(f) = \{h. \exists c. 0 < c \wedge (\forall x. |h\ x| \leq c * |f\ x|)\}$
 $\langle \text{proof} \rangle$

lemma *bigo-elt-subset* [intro]: $f \in O(g) \implies O(f) \subseteq O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-refl* [intro]: $f \in O(f)$
 $\langle \text{proof} \rangle$

lemma *bigo-zero*: $0 \in O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-zero2*: $O(\lambda x. 0) = \{\lambda x. 0\}$
 $\langle \text{proof} \rangle$

lemma *bigo-plus-self-subset* [intro]: $O(f) + O(f) \subseteq O(f)$
 $\langle \text{proof} \rangle$

lemma *bigo-plus-idemp* [simp]: $O(f) + O(f) = O(f)$
 $\langle \text{proof} \rangle$

lemma *bigo-plus-subset* [intro]: $O(f + g) \subseteq O(f) + O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-plus-subset2* [intro]: $A \subseteq O(f) \implies B \subseteq O(f) \implies A + B \subseteq O(f)$
 $\langle \text{proof} \rangle$

lemma *bigo-plus-eq*: $\forall x. 0 \leq f\ x \implies \forall x. 0 \leq g\ x \implies O(f + g) = O(f) + O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-bounded-alt*: $\forall x. 0 \leq f\ x \implies \forall x. f\ x \leq c * g\ x \implies f \in O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-bounded*: $\forall x. 0 \leq f\ x \implies \forall x. f\ x \leq g\ x \implies f \in O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-bounded2*: $\forall x. lb\ x \leq f\ x \implies \forall x. f\ x \leq lb\ x + g\ x \implies f \in lb + o$
 $O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-abs*: $(\lambda x. |f\ x|) =_o O(f)$
 $\langle \text{proof} \rangle$

lemma *bigo-abs2*: $f =_o O(\lambda x. |f x|)$
 $\langle proof \rangle$

lemma *bigo-abs3*: $O(f) = O(\lambda x. |f x|)$
 $\langle proof \rangle$

lemma *bigo-abs4*: $f =_o g +_o O(h) \implies (\lambda x. |f x|) =_o (\lambda x. |g x|) +_o O(h)$
 $\langle proof \rangle$

lemma *bigo-abs5*: $f =_o O(g) \implies (\lambda x. |f x|) =_o O(g)$
 $\langle proof \rangle$

lemma *bigo-elt-subset2* [intro]:
 assumes *: $f \in g +_o O(h)$
 shows $O(f) \subseteq O(g) + O(h)$
 $\langle proof \rangle$

lemma *bigo-mult* [intro]: $O(f) * O(g) \subseteq O(f * g)$
 $\langle proof \rangle$

lemma *bigo-mult2* [intro]: $f *_o O(g) \subseteq O(f * g)$
 $\langle proof \rangle$

lemma *bigo-mult3*: $f \in O(h) \implies g \in O(j) \implies f * g \in O(h * j)$
 $\langle proof \rangle$

lemma *bigo-mult4* [intro]: $f \in k +_o O(h) \implies g * f \in (g * k) +_o O(g * h)$
 $\langle proof \rangle$

lemma *bigo-mult5*:
 fixes $f :: 'a \Rightarrow 'b::linordered-field$
 assumes $\forall x. f x \neq 0$
 shows $O(f * g) \subseteq f *_o O(g)$
 $\langle proof \rangle$

lemma *bigo-mult6*: $\forall x. f x \neq 0 \implies O(f * g) = f *_o O(g)$
 for $f :: 'a \Rightarrow 'b::linordered-field$
 $\langle proof \rangle$

lemma *bigo-mult7*: $\forall x. f x \neq 0 \implies O(f * g) \subseteq O(f) * O(g)$
 for $f :: 'a \Rightarrow 'b::linordered-field$
 $\langle proof \rangle$

lemma *bigo-mult8*: $\forall x. f x \neq 0 \implies O(f * g) = O(f) * O(g)$
 for $f :: 'a \Rightarrow 'b::linordered-field$
 $\langle proof \rangle$

lemma *bigo-minus* [intro]: $f \in O(g) \implies -f \in O(g)$

$\langle proof \rangle$

lemma *bigo-minus2*: $f \in g + o \ O(h) \implies -f \in -g + o \ O(h)$
 $\langle proof \rangle$

lemma *bigo-minus3*: $O(-f) = O(f)$
 $\langle proof \rangle$

lemma *bigo-plus-absorb-lemma1*:
assumes *: $f \in O(g)$
shows $f + o \ O(g) \subseteq O(g)$
 $\langle proof \rangle$

lemma *bigo-plus-absorb-lemma2*:
assumes *: $f \in O(g)$
shows $O(g) \subseteq f + o \ O(g)$
 $\langle proof \rangle$

lemma *bigo-plus-absorb [simp]*: $f \in O(g) \implies f + o \ O(g) = O(g)$
 $\langle proof \rangle$

lemma *bigo-plus-absorb2 [intro]*: $f \in O(g) \implies A \subseteq O(g) \implies f + o \ A \subseteq O(g)$
 $\langle proof \rangle$

lemma *bigo-add-commute-imp*: $f \in g + o \ O(h) \implies g \in f + o \ O(h)$
 $\langle proof \rangle$

lemma *bigo-add-commute*: $f \in g + o \ O(h) \longleftrightarrow g \in f + o \ O(h)$
 $\langle proof \rangle$

lemma *bigo-const1*: $(\lambda x. \ c) \in O(\lambda x. \ 1)$
 $\langle proof \rangle$

lemma *bigo-const2 [intro]*: $O(\lambda x. \ c) \subseteq O(\lambda x. \ 1)$
 $\langle proof \rangle$

lemma *bigo-const3*: $c \neq 0 \implies (\lambda x. \ 1) \in O(\lambda x. \ c)$
for $c :: 'a::linordered-field$
 $\langle proof \rangle$

lemma *bigo-const4*: $c \neq 0 \implies O(\lambda x. \ 1) \subseteq O(\lambda x. \ c)$
for $c :: 'a::linordered-field$
 $\langle proof \rangle$

lemma *bigo-const [simp]*: $c \neq 0 \implies O(\lambda x. \ c) = O(\lambda x. \ 1)$
for $c :: 'a::linordered-field$
 $\langle proof \rangle$

lemma *bigo-const-mult1*: $(\lambda x. \ c * f \ x) \in O(f)$

$\langle \text{proof} \rangle$

lemma *bigo-const-mult2*: $O(\lambda x. c * f x) \subseteq O(f)$
 $\langle \text{proof} \rangle$

lemma *bigo-const-mult3*: $c \neq 0 \implies f \in O(\lambda x. c * f x)$
for $c :: 'a::\text{linordered-field}$
 $\langle \text{proof} \rangle$

lemma *bigo-const-mult4*: $c \neq 0 \implies O(f) \subseteq O(\lambda x. c * f x)$
for $c :: 'a::\text{linordered-field}$
 $\langle \text{proof} \rangle$

lemma *bigo-const-mult [simp]*: $c \neq 0 \implies O(\lambda x. c * f x) = O(f)$
for $c :: 'a::\text{linordered-field}$
 $\langle \text{proof} \rangle$

lemma *bigo-const-mult5 [simp]*: $c \neq 0 \implies (\lambda x. c) *o O(f) = O(f)$
for $c :: 'a::\text{linordered-field}$
 $\langle \text{proof} \rangle$

lemma *bigo-const-mult6 [intro]*: $(\lambda x. c) *o O(f) \subseteq O(f)$
 $\langle \text{proof} \rangle$

lemma *bigo-const-mult7 [intro]*:
assumes $*: f =o O(g)$
shows $(\lambda x. c * f x) =o O(g)$
 $\langle \text{proof} \rangle$

lemma *bigo-compose1*: $f =o O(g) \implies (\lambda x. f (k x)) =o O(\lambda x. g (k x))$
 $\langle \text{proof} \rangle$

lemma *bigo-compose2*: $f =o g +o O(h) \implies (\lambda x. f (k x)) =o (\lambda x. g (k x)) +o O(\lambda x. h(k x))$
 $\langle \text{proof} \rangle$

5.2 Sum

lemma *bigo-sum-main*: $\forall x. \forall y \in A x. 0 \leq h x y \implies$
 $\exists c. \forall x. \forall y \in A x. |f x y| \leq c * h x y \implies$
 $(\lambda x. \sum y \in A x. f x y) =o O(\lambda x. \sum y \in A x. h x y)$
 $\langle \text{proof} \rangle$

lemma *bigo-sum1*: $\forall x y. 0 \leq h x y \implies$
 $\exists c. \forall x y. |f x y| \leq c * h x y \implies$
 $(\lambda x. \sum y \in A x. f x y) =o O(\lambda x. \sum y \in A x. h x y)$
 $\langle \text{proof} \rangle$

lemma *bigo-sum2*: $\forall y. 0 \leq h y \implies$

$\exists c. \forall y. |f y| \leq c * (h y) \implies$
 $(\lambda x. \sum y \in A x. f y) =_o O(\lambda x. \sum y \in A x. h y)$
 $\langle proof \rangle$

lemma *bigo-sum3*: $f =_o O(h) \implies$
 $(\lambda x. \sum y \in A x. l x y * f (k x y)) =_o O(\lambda x. \sum y \in A x. |l x y * h (k x y)|)$
 $\langle proof \rangle$

lemma *bigo-sum4*: $f =_o g +_o O(h) \implies$
 $(\lambda x. \sum y \in A x. l x y * f (k x y)) =_o$
 $(\lambda x. \sum y \in A x. l x y * g (k x y)) +_o$
 $O(\lambda x. \sum y \in A x. |l x y * h (k x y)|)$
 $\langle proof \rangle$

lemma *bigo-sum5*: $f =_o O(h) \implies \forall x y. 0 \leq l x y \implies$
 $\forall x. 0 \leq h x \implies$
 $(\lambda x. \sum y \in A x. l x y * f (k x y)) =_o$
 $O(\lambda x. \sum y \in A x. l x y * h (k x y))$
 $\langle proof \rangle$

lemma *bigo-sum6*: $f =_o g +_o O(h) \implies \forall x y. 0 \leq l x y \implies$
 $\forall x. 0 \leq h x \implies$
 $(\lambda x. \sum y \in A x. l x y * f (k x y)) =_o$
 $(\lambda x. \sum y \in A x. l x y * g (k x y)) +_o$
 $O(\lambda x. \sum y \in A x. l x y * h (k x y))$
 $\langle proof \rangle$

5.3 Misc useful stuff

lemma *bigo-useful-intro*: $A \subseteq O(f) \implies B \subseteq O(f) \implies A + B \subseteq O(f)$
 $\langle proof \rangle$

lemma *bigo-useful-add*: $f =_o O(h) \implies g =_o O(h) \implies f + g =_o O(h)$
 $\langle proof \rangle$

lemma *bigo-useful-const-mult*: $c \neq 0 \implies (\lambda x. c) * f =_o O(h) \implies f =_o O(h)$
for $c :: 'a::linordered-field$
 $\langle proof \rangle$

lemma *bigo-fix*: $(\lambda x::nat. f (x + 1)) =_o O(\lambda x. h (x + 1)) \implies f 0 = 0 \implies f =_o O(h)$
 $\langle proof \rangle$

lemma *bigo-fix2*:
 $(\lambda x. f ((x::nat) + 1)) =_o (\lambda x. g(x + 1)) +_o O(\lambda x. h(x + 1)) \implies$
 $f 0 = g 0 \implies f =_o g +_o O(h)$
 $\langle proof \rangle$

5.4 Less than or equal to

definition *lesso* :: ('a ⇒ 'b::linordered-idom) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b (infixl <_o 70)

where $f <_o g = (\lambda x. \max (f\ x - g\ x)\ 0)$

lemma *big-lesseq1*: $f =_o O(h) \implies \forall x. |g\ x| \leq |f\ x| \implies g =_o O(h)$
 ⟨proof⟩

lemma *big-lesseq2*: $f =_o O(h) \implies \forall x. |g\ x| \leq f\ x \implies g =_o O(h)$
 ⟨proof⟩

lemma *big-lesseq3*: $f =_o O(h) \implies \forall x. 0 \leq g\ x \implies \forall x. g\ x \leq f\ x \implies g =_o O(h)$
 ⟨proof⟩

lemma *big-lesseq4*: $f =_o O(h) \implies \forall x. 0 \leq g\ x \implies \forall x. g\ x \leq |f\ x| \implies g =_o O(h)$
 ⟨proof⟩

lemma *big-lesso1*: $\forall x. f\ x \leq g\ x \implies f <_o g =_o O(h)$
 ⟨proof⟩

lemma *big-lesso2*: $f =_o g +_o O(h) \implies \forall x. 0 \leq k\ x \implies \forall x. k\ x \leq f\ x \implies k <_o g =_o O(h)$
 ⟨proof⟩

lemma *big-lesso3*: $f =_o g +_o O(h) \implies \forall x. 0 \leq k\ x \implies \forall x. g\ x \leq k\ x \implies f <_o k =_o O(h)$
 ⟨proof⟩

lemma *big-lesso4*: $f <_o g =_o O(k) \implies g =_o h +_o O(k) \implies f <_o h =_o O(k)$
for $k :: 'a \Rightarrow 'b::linordered-field$
 ⟨proof⟩

lemma *big-lesso5*: $f <_o g =_o O(h) \implies \exists C. \forall x. f\ x \leq g\ x + C * |h\ x|$
 ⟨proof⟩

lemma *lesso-add*: $f <_o g =_o O(h) \implies k <_o l =_o O(h) \implies (f + k) <_o (g + l) =_o O(h)$
 ⟨proof⟩

lemma *big-LIMSEQ1*: $f =_o O(g) \implies g \longrightarrow 0 \implies f \longrightarrow 0$
for $f\ g :: nat \Rightarrow real$
 ⟨proof⟩

lemma *big-LIMSEQ2*: $f =_o g +_o O(h) \implies h \longrightarrow 0 \implies f \longrightarrow a \implies g \longrightarrow a$
for $f\ g\ h :: nat \Rightarrow real$
 ⟨proof⟩

end

6 Axiomatic Declaration of Bounded Natural Functors

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

$\langle ML \rangle$

end

7 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:  $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$ 
   $\langle proof \rangle$ 
```

```
lemma inject-refine:  $g (f x) = x \Longrightarrow g (f y) = y \Longrightarrow f x = f y \longleftrightarrow x = y$ 
   $\langle proof \rangle$ 
```

```
lemma convol-apply:  $BNF-Def.convol f g x = (f x, g x)$ 
   $\langle proof \rangle$ 
```

```
lemma Grp-UNIV-id:  $BNF-Def.Grp UNIV id = (=)$ 
   $\langle proof \rangle$ 
```

```
lemma sum-comp-cases:
  assumes  $f \circ Inl = g \circ Inl$  and  $f \circ Inr = g \circ Inr$ 
  shows  $f = g$ 
   $\langle proof \rangle$ 
```

```
lemma case-sum-Inl-Inr-L:  $case-sum (f \circ Inl) (f \circ Inr) = f$ 
   $\langle proof \rangle$ 
```

```
lemma eq-o-InrI:  $\llbracket g \circ Inl = h; case-sum h f = g \rrbracket \Longrightarrow f = g \circ Inr$ 
```

$\langle \text{proof} \rangle$

lemma *id-bnf-o*: $\text{BNF-Composition.id-bnf} \circ f = f$
 $\langle \text{proof} \rangle$

lemma *o-id-bnf*: $f \circ \text{BNF-Composition.id-bnf} = f$
 $\langle \text{proof} \rangle$

lemma *if-True-False*:
 $(\text{if } P \text{ then True else } Q) \longleftrightarrow P \vee Q$
 $(\text{if } P \text{ then False else } Q) \longleftrightarrow \neg P \wedge Q$
 $(\text{if } P \text{ then } Q \text{ else True}) \longleftrightarrow \neg P \vee Q$
 $(\text{if } P \text{ then } Q \text{ else False}) \longleftrightarrow P \wedge Q$
 $\langle \text{proof} \rangle$

lemma *if-distrib-fun*: $(\text{if } c \text{ then } f \text{ else } g) x = (\text{if } c \text{ then } f x \text{ else } g x)$
 $\langle \text{proof} \rangle$

7.1 Coinduction

lemma *eq-comp-compI*: $a \circ b = f \circ x \implies x \circ c = \text{id} \implies f = a \circ (b \circ c)$
 $\langle \text{proof} \rangle$

lemma *self-bounded-weaken-left*: $(a :: 'a :: \text{semilattice-inf}) \leq \inf a b \implies a \leq b$
 $\langle \text{proof} \rangle$

lemma *self-bounded-weaken-right*: $(a :: 'a :: \text{semilattice-inf}) \leq \inf b a \implies a \leq b$
 $\langle \text{proof} \rangle$

lemma *symp-iff*: $\text{symp } R \longleftrightarrow R = R^{-1-1}$
 $\langle \text{proof} \rangle$

lemma *equivp-inf*: $\llbracket \text{equivp } R; \text{equivp } S \rrbracket \implies \text{equivp } (\inf R S)$
 $\langle \text{proof} \rangle$

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

lemma *predicate2I-obj*: $(\forall x y. P x y \longrightarrow Q x y) \implies P \leq Q$
 $\langle \text{proof} \rangle$

lemma *predicate2D-obj*: $P \leq Q \implies P x y \longrightarrow Q x y$
 $\langle \text{proof} \rangle$

locale *cong* =
fixes *rel* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool})$
and *eval* :: $'b \Rightarrow 'a$

and $\text{retr} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$
assumes $\text{rel-mono}: \bigwedge R S. R \leq S \Longrightarrow \text{rel } R \leq \text{rel } S$
and $\text{equivp-retr}: \bigwedge R. \text{equivp } R \Longrightarrow \text{equivp } (\text{retr } R)$
and $\text{retr-eval}: \bigwedge R x y. \llbracket (\text{rel-fun } (\text{rel } R) R) \text{ eval eval}; \text{rel } (\text{inf } R (\text{retr } R)) x y \rrbracket$
 $\Longrightarrow$
 $\text{retr } R (\text{eval } x) (\text{eval } y)$
begin

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

lemma $\text{cong-retr}: \text{cong } R \Longrightarrow \text{cong } (\text{inf } R (\text{retr } R))$
 $\langle \text{proof} \rangle$

lemma $\text{cong-equivp}: \text{cong } R \Longrightarrow \text{equivp } R$
 $\langle \text{proof} \rangle$

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

lemma $\text{gen-cong-reflp}[\text{intro}, \text{simp}]: x = y \Longrightarrow \text{gen-cong } R x y$
 $\langle \text{proof} \rangle$

lemma $\text{gen-cong-symp}[\text{intro}]: \text{gen-cong } R x y \Longrightarrow \text{gen-cong } R y x$
 $\langle \text{proof} \rangle$

lemma $\text{gen-cong-transp}[\text{intro}]: \text{gen-cong } R x y \Longrightarrow \text{gen-cong } R y z \Longrightarrow \text{gen-cong } R x z$
 $\langle \text{proof} \rangle$

lemma $\text{equivp-gen-cong}: \text{equivp } (\text{gen-cong } R)$
 $\langle \text{proof} \rangle$

lemma $\text{leq-gen-cong}: R \leq \text{gen-cong } R$
 $\langle \text{proof} \rangle$

lemmas $\text{imp-gen-cong}[\text{intro}] = \text{predicate2D}[OF \text{ leq-gen-cong}]$

lemma $\text{gen-cong-minimal}: \llbracket R \leq R'; \text{cong } R' \rrbracket \Longrightarrow \text{gen-cong } R \leq R'$
 $\langle \text{proof} \rangle$

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

lemma $\text{cong-gen-cong}: \text{cong } (\text{gen-cong } R)$
 $\langle \text{proof} \rangle$

lemma $\text{gen-cong-eval-rel-fun}: \text{rel } (\text{gen-cong } R) x y \Longrightarrow \text{rel } R x y$

$(rel\_fun\ (rel\ (gen\_cong\ R))\ (gen\_cong\ R))\ eval\ eval$
 $\langle proof \rangle$

lemma *gen-cong-eval*:

$rel\ (gen\_cong\ R)\ x\ y \implies gen\_cong\ R\ (eval\ x)\ (eval\ y)$
 $\langle proof \rangle$

lemma *gen-cong-idem*: $gen\_cong\ (gen\_cong\ R) = gen\_cong\ R$

$\langle proof \rangle$

lemma *gen-cong-rho*:

$\varrho = eval \circ f \implies rel\ (gen\_cong\ R)\ (f\ x)\ (f\ y) \implies gen\_cong\ R\ (\varrho\ x)\ (\varrho\ y)$
 $\langle proof \rangle$

lemma *coinduction*:

assumes *coind*: $\forall R. R \leq retr\ R \longrightarrow R \leq (=)$

assumes *cih*: $R \leq retr\ (gen\_cong\ R)$

shows $R \leq (=)$

$\langle proof \rangle$

end

lemma *rel-sum-case-sum*:

$rel\_fun\ (rel\_sum\ R\ S)\ T\ (case\_sum\ f1\ g1)\ (case\_sum\ f2\ g2) = (rel\_fun\ R\ T\ f1\ f2$
 $\wedge\ rel\_fun\ S\ T\ g1\ g2)$
 $\langle proof \rangle$

context

fixes *rel eval rel' eval' retr emb*

assumes *base*: $cong\ rel\ eval\ retr$

and *step*: $cong\ rel'\ eval'\ retr$

and *emb*: $eval' \circ emb = eval$

and *emb-transfer*: $rel\_fun\ (rel\ R)\ (rel'\ R)\ emb\ emb$

begin

interpretation *base*: $cong\ rel\ eval\ retr\ \langle proof \rangle$

interpretation *step*: $cong\ rel'\ eval'\ retr\ \langle proof \rangle$

lemma *gen-cong-emb*: $base.gen\_cong\ R \leq step.gen\_cong\ R$

$\langle proof \rangle$

end

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

$\langle ML \rangle$

end

8 A general “while” combinator

```
theory While-Combinator
imports Main
begin
```

8.1 Partial version

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

theorem *while-option-unfold*[code]:
while-option $b\ c\ s = (\text{if } b\ s \text{ then } \text{while-option } b\ c\ (c\ s) \text{ else } \text{Some } s)$
 <proof>

lemma *while-option-stop2*:
while-option $b\ c\ s = \text{Some } t \implies \exists k. t = (c \rightsquigarrow k)\ s \wedge \neg b\ t$
 <proof>

lemma *while-option-stop*: *while-option* $b\ c\ s = \text{Some } t \implies \neg b\ t$
 <proof>

theorem *while-option-rule*:
assumes *step*: $!!s. P\ s \implies b\ s \implies P\ (c\ s)$
and *result*: *while-option* $b\ c\ s = \text{Some } t$
and *init*: $P\ s$
shows $P\ t$
 <proof>

lemma *funpow-commute*:
 $\llbracket \forall k' < k. f\ (c\ ((c \rightsquigarrow k')\ s)) = c'\ (f\ ((c \rightsquigarrow k')\ s)) \rrbracket \implies f\ ((c \rightsquigarrow k)\ s) = (c' \rightsquigarrow k)\ (f\ s)$
 <proof>

lemma *while-option-commute-invariant*:
assumes *Invariant*: $\bigwedge s. P\ s \implies b\ s \implies P\ (c\ s)$
assumes *TestCommutate*: $\bigwedge s. P\ s \implies b\ s = b'\ (f\ s)$
assumes *BodyCommutate*: $\bigwedge s. P\ s \implies b\ s \implies f\ (c\ s) = c'\ (f\ s)$
assumes *Initial*: $P\ s$
shows *map-option* $f\ (\text{while-option } b\ c\ s) = \text{while-option } b'\ c'\ (f\ s)$
 <proof>

lemma *while-option-commute*:
assumes $\bigwedge s. b\ s = b'\ (f\ s) \wedge s. \llbracket b\ s \rrbracket \implies f\ (c\ s) = c'\ (f\ s)$
shows *map-option* $f\ (\text{while-option } b\ c\ s) = \text{while-option } b'\ c'\ (f\ s)$
 <proof>

8.2 Total version

definition $\text{while} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where $\text{while } b \ c \ s = \text{the } (\text{while-option } b \ c \ s)$

lemma while-unfold [code]:
 $\text{while } b \ c \ s = (\text{if } b \ s \text{ then } \text{while } b \ c \ (c \ s) \text{ else } s)$
 $\langle \text{proof} \rangle$

lemma def-while-unfold :
assumes $f\text{def}: f == \text{while test do}$
shows $f \ x = (\text{if test } x \text{ then } f(\text{do } x) \text{ else } x)$
 $\langle \text{proof} \rangle$

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

theorem while-rule-lemma :
assumes $\text{invariant}: !!s. P \ s ==> b \ s ==> P \ (c \ s)$
and $\text{terminate}: !!s. P \ s ==> \neg b \ s ==> Q \ s$
and $\text{wf}: \text{wf } \{(t, s). P \ s \wedge b \ s \wedge t = c \ s\}$
shows $P \ s \Longrightarrow Q \ (\text{while } b \ c \ s)$
 $\langle \text{proof} \rangle$

theorem while-rule :
 $[[P \ s;$
 $!!s. [P \ s; b \ s] ==> P \ (c \ s);$
 $!!s. [P \ s; \neg b \ s] ==> Q \ s;$
 $\text{wf } r;$
 $!!s. [P \ s; b \ s] ==> (c \ s, s) \in r] ==>$
 $Q \ (\text{while } b \ c \ s)$
 $\langle \text{proof} \rangle$

Proving termination:

theorem $\text{wf-while-option-Some}$:
assumes $\text{wf } \{(t, s). (P \ s \wedge b \ s) \wedge t = c \ s\}$
and $\bigwedge s. P \ s \Longrightarrow b \ s \Longrightarrow P(c \ s)$ **and** $P \ s$
shows $\exists t. \text{while-option } b \ c \ s = \text{Some } t$
 $\langle \text{proof} \rangle$

lemma $\text{wf-rel-while-option-Some}$:
assumes $\text{wf}: \text{wf } R$
assumes $\text{smaller}: \bigwedge s. P \ s \wedge b \ s \Longrightarrow (c \ s, s) \in R$
assumes $\text{inv}: \bigwedge s. P \ s \wedge b \ s \Longrightarrow P(c \ s)$
assumes $\text{init}: P \ s$
shows $\exists t. \text{while-option } b \ c \ s = \text{Some } t$
 $\langle \text{proof} \rangle$

theorem $\text{measure-while-option-Some}$: **fixes** $f :: 's \Rightarrow \text{nat}$
shows $(\bigwedge s. P \ s \Longrightarrow b \ s \Longrightarrow P(c \ s) \wedge f(c \ s) < f \ s)$
 $\Longrightarrow P \ s \Longrightarrow \exists t. \text{while-option } b \ c \ s = \text{Some } t$
 $\langle \text{proof} \rangle$

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

lemma *while-option-finite-subset-Some*: **fixes** $C :: 'a \text{ set}$
assumes $\text{mono } f \text{ and } !!X. X \subseteq C \implies f X \subseteq C \text{ and } \text{finite } C$
shows $\exists P. \text{while-option } (\lambda A. f A \neq A) f \{\} = \text{Some } P$
 $\langle \text{proof} \rangle$

lemma *lfp-the-while-option*:
assumes $\text{mono } f \text{ and } !!X. X \subseteq C \implies f X \subseteq C \text{ and } \text{finite } C$
shows $\text{lfp } f = \text{the}(\text{while-option } (\lambda A. f A \neq A) f \{\})$
 $\langle \text{proof} \rangle$

lemma *lfp-while*:
assumes $\text{mono } f \text{ and } !!X. X \subseteq C \implies f X \subseteq C \text{ and } \text{finite } C$
shows $\text{lfp } f = \text{while } (\lambda A. f A \neq A) f \{\}$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

<proof>

lemma *gfp-the-while-option-lattice:*

fixes $f :: 'a :: \text{complete-lattice} \Rightarrow 'a$

assumes *mono f and finite (UNIV :: 'a set)*

shows $\text{gfp } f = \text{the}(\text{while-option } (\lambda A. f A \neq A) f \text{ top})$

<proof>

lemma *gfp-while-lattice:*

fixes $f :: 'a :: \text{complete-lattice} \Rightarrow 'a$

assumes *mono f and finite (UNIV :: 'a set)*

shows $\text{gfp } f = \text{while } (\lambda A. f A \neq A) f \text{ top}$

<proof>

Computing the reflexive, transitive closure by iterating a successor function. Stops when an element is found that does 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 $p :: 'a \Rightarrow \text{bool}$

and $f :: 'a \Rightarrow 'a \text{ list}$

and $x :: 'a$

begin

qualified fun *rtrancl-while-test* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow \text{bool}$

where *rtrancl-while-test* ($ws, -$) = $(ws \neq [] \wedge p(\text{hd } ws))$

qualified fun *rtrancl-while-step* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow 'a \text{ list} \times 'a \text{ set}$

where *rtrancl-while-step* (ws, Z) =

$(\text{let } x = \text{hd } ws; \text{ new} = \text{remdups } (\text{filter } (\lambda y. y \notin Z) (f x))$

$\text{in } (\text{new} @ \text{tl } ws, \text{ set new } \cup Z))$

definition *rtrancl-while* :: $('a \text{ list} * 'a \text{ set}) \text{ option}$

where *rtrancl-while* = *while-option rtrancl-while-test rtrancl-while-step* ($[x], \{x\}$)

qualified fun *rtrancl-while-invariant* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow \text{bool}$

where *rtrancl-while-invariant* (ws, Z) =

$(x \in Z \wedge \text{set } ws \subseteq Z \wedge \text{distinct } ws \wedge \{(x, y). y \in \text{set}(f x)\} \text{ “ } (Z - \text{set } ws) \subseteq Z$

$\wedge$

$Z \subseteq \{(x, y). y \in \text{set}(f x)\}^* \text{ “ } \{x\} \wedge (\forall z \in Z - \text{set } ws. p z))$

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 = []*

$then\ Z = \{(x,y). y \in set(f\ x)\}^* \text{ `` } \{x\} \wedge (\forall z \in Z. p\ z)$
 $else\ \neg p(hd\ ws) \wedge hd\ ws \in \{(x,y). y \in set(f\ x)\}^* \text{ `` } \{x\}$
 $\langle proof \rangle$

lemma *rtrancl-while-finite-Some*:
assumes *finite* $(\{(x, y). y \in set\ (f\ x)\}^* \text{ `` } \{x\})$ (**is** *finite* ?*Cl*)
shows $\exists y. rtrancl\text{-}while = Some\ y$
 $\langle proof \rangle$

end

end

9 The Bourbaki-Witt tower construction for trans-finite iteration

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

lemma *ChainsI* [*intro?*]:
 $(\bigwedge a\ b. \llbracket a \in Y; b \in Y \rrbracket \implies (a, b) \in r \vee (b, a) \in r) \implies Y \in Chains\ r$
 $\langle proof \rangle$

lemma *in-Chains-subset*: $\llbracket M \in Chains\ r; M' \subseteq M \rrbracket \implies M' \in Chains\ r$
 $\langle proof \rangle$

lemma *in-ChainsD*: $\llbracket M \in Chains\ r; x \in M; y \in M \rrbracket \implies (x, y) \in r \vee (y, x) \in r$
 $\langle proof \rangle$

lemma *Chains-FieldD*: $\llbracket M \in Chains\ r; x \in M \rrbracket \implies x \in Field\ r$
 $\langle proof \rangle$

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

lemma *partial-order-on-trans*:
 $\llbracket partial\text{-}order\text{-}on\ A\ r; (x, y) \in r; (y, z) \in r \rrbracket \implies (x, z) \in r$
 $\langle proof \rangle$

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*: $\llbracket M \in Chains\ leq; M \neq \{\}; \bigwedge x. x \in M \implies (x, z) \in leq \rrbracket \implies (lub\ M, z) \in leq$

and *lub-upper*: $\llbracket M \in \text{Chains } \text{leq}; x \in M \rrbracket \implies (x, \text{lub } M) \in \text{leq}$
and *lub-in-Field*: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\} \rrbracket \implies \text{lub } M \in \text{Field } \text{leq}$
and *increasing*: $\bigwedge x. x \in \text{Field } \text{leq} \implies (x, f x) \in \text{leq}$
begin

lemma *leq-trans*: $\llbracket (x, y) \in \text{leq}; (y, z) \in \text{leq} \rrbracket \implies (x, z) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *leq-refl*: $x \in \text{Field } \text{leq} \implies (x, x) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *leq-antisym*: $\llbracket (x, y) \in \text{leq}; (y, x) \in \text{leq} \rrbracket \implies x = y$
 $\langle \text{proof} \rangle$

inductive-set *iterates-above* :: $'a \Rightarrow 'a \text{ set}$
for a
where
 base: $a \in \text{iterates-above } a$
 step: $x \in \text{iterates-above } a \implies f x \in \text{iterates-above } a$
 Sup: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\}; \bigwedge x. x \in M \implies x \in \text{iterates-above } a \rrbracket \implies \text{lub } M \in \text{iterates-above } a$

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

lemma *fixp-above-outside*: $a \notin \text{Field } \text{leq} \implies \text{fixp-above } a = a$
 $\langle \text{proof} \rangle$

lemma *fixp-above-inside*: $a \in \text{Field } \text{leq} \implies \text{fixp-above } a = \text{lub } (\text{iterates-above } a)$
 $\langle \text{proof} \rangle$

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*: $\llbracket x \in \text{iterates-above } a; a \in \text{Field } \text{leq} \rrbracket \implies (x, f x) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *iterates-above-Field*: $\llbracket x \in \text{iterates-above } a; a \in \text{Field } \text{leq} \rrbracket \implies x \in \text{Field } \text{leq}$
 $\langle \text{proof} \rangle$

lemma *iterates-above-ge*:
 assumes $y: y \in \text{iterates-above } a$
 and $a: a \in \text{Field } \text{leq}$

shows $(a, y) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *iterates-above-lub*:
assumes $M: M \in \text{Chains leq}$
and *nempty*: $M \neq \{\}$
and *upper*: $\bigwedge y. y \in M \implies \exists z \in M. (y, z) \in \text{leq} \wedge z \in \text{iterates-above } a$
shows $\text{lub } M \in \text{iterates-above } a$
 $\langle \text{proof} \rangle$

lemma *iterates-above-successor*:
assumes $y: y \in \text{iterates-above } a$
and $a: a \in \text{Field leq}$
shows $y = a \vee y \in \text{iterates-above } (f a)$
 $\langle \text{proof} \rangle$

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

lemma *iterates-above-triangle*:
assumes $x: x \in \text{iterates-above } a$
and $y: y \in \text{iterates-above } a$
and $a: a \in \text{Field leq}$
shows $x \in \text{iterates-above } y \vee y \in \text{iterates-above } x$
 $\langle \text{proof} \rangle$

lemma *chain-iterates-above*:
assumes $a: a \in \text{Field leq}$
shows $\text{iterates-above } a \in \text{Chains leq}$ **(is ?C ∈ -)**
 $\langle \text{proof} \rangle$

lemma *fixp-iterates-above*: $\text{fixp-above } a \in \text{iterates-above } a$
 $\langle \text{proof} \rangle$

lemma *fixp-above-Field*: $a \in \text{Field leq} \implies \text{fixp-above } a \in \text{Field leq}$
 $\langle \text{proof} \rangle$

lemma *fixp-above-unfold*:
assumes $a: a \in \text{Field leq}$
shows $\text{fixp-above } a = f (\text{fixp-above } a)$ **(is ?a = f ?a)**
 $\langle \text{proof} \rangle$

end

lemma *fixp-above-induct* [*case-names adm base step*]:
assumes *adm*: *ccpo.admissible* *lub* $(\lambda x y. (x, y) \in \text{leq})$ *P*
and *base*: $P\ a$
and *step*: $\bigwedge x. P\ x \implies P\ (f\ x)$
shows $P\ (\text{fixp-above}\ a)$
 $\langle \text{proof} \rangle$
end

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

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

lemma *in-Chains-finite*: — Translation from $\llbracket \text{Complete-Partial-Order.chain } (\leq) \text{ ?A; finite ?A; ?A} \neq \{\} \rrbracket \implies \text{Sup } ?A \in ?A$.
assumes $M \in \text{Chains leq}$
and $M \neq \{\}$
and *finite* M
shows $\text{lub } M \in M$
 $\langle \text{proof} \rangle$

lemma *fun-pow-iterates-above*: $(f \smallfrown k)\ a \in \text{iterates-above } a$
 $\langle \text{proof} \rangle$

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

lemma *Chain-finite-iterates-above-fun-pow-iff*:
assumes $\forall M \in \text{Chains leq. finite } M$
shows $x \in \text{iterates-above } a \iff (\exists j. x = (f \smallfrown j)\ a)$
 $\langle \text{proof} \rangle$

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

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

lemma *funpow-Field-leq*: $(f \smallfrown k)\ a \in \text{Field leq}$
 $\langle \text{proof} \rangle$

lemma *funpow-prefix*: $j < k \implies ((f \smallfrown j)\ a, (f \smallfrown k)\ a) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *funpow-suffix*: $(f \frown \text{Suc } k) \ a = (f \frown k) \ a \implies ((f \frown (j + k)) \ a, (f \frown k) \ a) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *funpow-stability*: $(f \frown \text{Suc } k) \ a = (f \frown k) \ a \implies ((f \frown j) \ a, (f \frown k) \ a) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *funpow-in-Chains*: $\{(f \frown k) \ a \mid k. \text{True}\} \in \text{Chains leq}$
 $\langle \text{proof} \rangle$

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

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

lemma *Chain-finite-wf*: $\text{wf } \{(f \frown k) \ a, (f \frown k) \ a \mid k. f \ ((f \frown k) \ a) \neq (f \frown k) \ a\}$
 $\langle \text{proof} \rangle$

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

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

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

end

end

end

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

end

10 Order on characters

```

theory Char-ord
  imports Main
begin

instantiation char :: linorder
begin

definition less-eq-char :: char  $\Rightarrow$  char  $\Rightarrow$  bool
  where c1  $\leq$  c2  $\iff$  of-char c1  $\leq$  (of-char c2 :: nat)

definition less-char :: char  $\Rightarrow$  char  $\Rightarrow$  bool
  where c1 < c2  $\iff$  of-char c1 < (of-char c2 :: nat)

instance
  <proof>

end

lemma less-eq-char-simp [simp]:
  Char b0 b1 b2 b3 b4 b5 b6 b7  $\leq$  Char c0 c1 c2 c3 c4 c5 c6 c7
     $\iff$  foldr ( $\lambda b k. \text{of-bool } b + k * 2$ ) [b0, b1, b2, b3, b4, b5, b6, b7] 0
       $\leq$  foldr ( $\lambda b k. \text{of-bool } b + k * 2$ ) [c0, c1, c2, c3, c4, c5, c6, c7] (0::nat)
  <proof>

lemma less-char-simp [simp]:
  Char b0 b1 b2 b3 b4 b5 b6 b7 < Char c0 c1 c2 c3 c4 c5 c6 c7
     $\iff$  foldr ( $\lambda b k. \text{of-bool } b + k * 2$ ) [b0, b1, b2, b3, b4, b5, b6, b7] 0
      < foldr ( $\lambda b k. \text{of-bool } b + k * 2$ ) [c0, c1, c2, c3, c4, c5, c6, c7] (0::nat)
  <proof>

instantiation char :: distrib-lattice
begin

definition (inf :: char  $\Rightarrow$  -) = min
definition (sup :: char  $\Rightarrow$  -) = max

instance
  <proof>

end

end

```

11 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  $\circ$  of-phantom = id
  and of-phantom-comp-phantom [simp]: of-phantom  $\circ$  phantom = id
  <proof>

syntax -Phantom :: type  $\Rightarrow$  logic ((1Phantom/(1'(-))))
translations
  Phantom('t) => CONST phantom :: -  $\Rightarrow$  ('t, -) phantom

  <ML>

lemma of-phantom-inject [simp]:
  of-phantom x = of-phantom y  $\longleftrightarrow$  x = y
  <proof>

end

```

12 Cardinality of types

```

theory Cardinality
imports Phantom-Type
begin

```

12.1 Preliminary lemmas

```

lemma (in type-definition) univ:
  UNIV = Abs ' A
  <proof>

lemma (in type-definition) card: card (UNIV :: 'b set) = card A
  <proof>

```

12.2 Cardinalities of types

```

syntax -type-card :: type  $\Rightarrow$  nat ((1CARD/(1'(-))))
translations CARD('t) => CONST card (CONST UNIV :: 't set)

  <ML>

lemma card-prod [simp]: CARD('a  $\times$  'b) = CARD('a) * CARD('b)
  <proof>

```


lemma *card-UNIV-sum*: $CARD('a + 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0$
 $then\ CARD('a) + CARD('b)\ else\ 0)$
 $\langle proof \rangle$

lemma *card-sum* [simp]: $CARD('a + 'b) = CARD('a::finite) + CARD('b::finite)$
 $\langle proof \rangle$

lemma *card-UNIV-option*: $CARD('a\ option) = (if\ CARD('a) = 0\ then\ 0\ else$
 $CARD('a) + 1)$
 $\langle proof \rangle$

lemma *card-option* [simp]: $CARD('a\ option) = Suc\ CARD('a::finite)$
 $\langle proof \rangle$

lemma *card-UNIV-set*: $CARD('a\ set) = (if\ CARD('a) = 0\ then\ 0\ else\ 2 \wedge CARD('a))$
 $\langle proof \rangle$

lemma *card-set* [simp]: $CARD('a\ set) = 2 \wedge CARD('a::finite)$
 $\langle proof \rangle$

lemma *card-nat* [simp]: $CARD(nat) = 0$
 $\langle proof \rangle$

lemma *card-fun*: $CARD('a \Rightarrow 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0 \vee$
 $CARD('b) = 1\ then\ CARD('b) \wedge CARD('a)\ else\ 0)$
 $\langle proof \rangle$

corollary *finite-UNIV-fun*:
 $finite\ (UNIV :: ('a \Rightarrow 'b)\ set) \longleftrightarrow$
 $finite\ (UNIV :: 'a\ set) \wedge finite\ (UNIV :: 'b\ set) \vee CARD('b) = 1$
 $(is\ ?lhs \longleftrightarrow ?rhs)$
 $\langle proof \rangle$

lemma *card-literal*: $CARD(String.literal) = 0$
 $\langle proof \rangle$

12.3 Classes with at least 1 and 2

Class *finite* already captures "at least 1"

lemma *zero-less-card-finite* [simp]: $0 < CARD('a::finite)$
 $\langle proof \rangle$

lemma *one-le-card-finite* [simp]: $Suc\ 0 \leq CARD('a::finite)$
 $\langle proof \rangle$

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*: $2 \leq \text{CARD}('a)$

lemma *one-less-card*: $\text{Suc } 0 < \text{CARD}('a::\text{card2})$
 ⟨*proof*⟩

lemma *one-less-int-card*: $1 < \text{int } \text{CARD}('a::\text{card2})$
 ⟨*proof*⟩

12.4 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*)
 $\longleftrightarrow$ *of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*)
 ⟨*proof*⟩

12.5 A type class for computing the cardinality of types

definition *is-list-UNIV* :: 'a *list* $\Rightarrow$ *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* $\longleftrightarrow$ *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)

12.6 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)  $\wedge$  of-phantom (finite-UNIV :: 'b
finite-UNIV))
instance  $\langle$ proof $\rangle$ 
end

instantiation prod :: (card-UNIV, card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a  $\times$  'b)
    (of-phantom (card-UNIV :: 'a card-UNIV) * of-phantom (card-UNIV :: 'b card-UNIV))
instance  $\langle$ proof $\rangle$ 
end

instantiation sum :: (finite-UNIV, finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a + 'b)
    (of-phantom (finite-UNIV :: 'a finite-UNIV)  $\wedge$  of-phantom (finite-UNIV :: 'b
finite-UNIV))
instance
     $\langle$ proof $\rangle$ 
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  $\neq$  0  $\wedge$  cb  $\neq$  0 then ca + cb else 0)
instance  $\langle$ proof $\rangle$ 
end

instantiation fun :: (finite-UNIV, card-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a  $\Rightarrow$  'b)
    (let cb = of-phantom (card-UNIV :: 'b card-UNIV)
      in cb = 1  $\vee$  of-phantom (finite-UNIV :: 'a finite-UNIV)  $\wedge$  cb  $\neq$  0)
instance
     $\langle$ proof $\rangle$ 
end

instantiation fun :: (card-UNIV, card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a  $\Rightarrow$  'b)
    (let ca = of-phantom (card-UNIV :: 'a card-UNIV);
      cb = of-phantom (card-UNIV :: 'b card-UNIV)
      in if ca  $\neq$  0  $\wedge$  cb  $\neq$  0  $\vee$  cb = 1 then cb  $\wedge$  ca else 0)
instance  $\langle$ proof $\rangle$ 
end

instantiation option :: (finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a option) (of-phantom (finite-UNIV :: 'a fi-
nite-UNIV))
instance  $\langle$ proof $\rangle$ 
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.a1]
  ⟨proof⟩

```

```

lemma UNIV-finite-2: UNIV = set [finite-2.a1, finite-2.a2]
  ⟨proof⟩

```

```

lemma UNIV-finite-3: UNIV = set [finite-3.a1, finite-3.a2, finite-3.a3]
  ⟨proof⟩

```

```

lemma UNIV-finite-4: UNIV = set [finite-4.a1, finite-4.a2, finite-4.a3, finite-4.a4]
  ⟨proof⟩

```

```

lemma UNIV-finite-5:
  UNIV = set [finite-5.a1, finite-5.a2, finite-5.a3, finite-5.a4, finite-5.a5]
  ⟨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

```

12.7 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*, $(\subseteq)$, 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 [code del]: 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 $\implies$ card (- A) = card (UNIV :: 'a set) - card (A :: 'a set)

<proof>

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

begin

qualified definition *finite'* :: 'a set $\Rightarrow$ bool

where [*simp*, *code del*, *code-abbrev*]: *finite' = finite*

lemma *finite'-code* [code]:

finite' (set xs) $\longleftrightarrow$ True

finite' (List.coset xs) $\longleftrightarrow$ of-phantom (finite-UNIV :: 'a finite-UNIV)

<proof>

end

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

begin

qualified definition *card'* :: 'a set $\Rightarrow$ nat

where [*simp*, *code del*, *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 $\Rightarrow$ 'a set $\Rightarrow$ bool

where [*simp*, *code del*, *code-abbrev*]: *subset' = ($\subseteq$)*

lemma *subset'-code* [code]:

subset' A (List.coset ys) $\longleftrightarrow$ ($\forall y \in \text{set } ys. y \notin A$)

subset' (set ys) B $\longleftrightarrow$ ($\forall y \in \text{set } ys. y \in B$)

subset' (List.coset xs) (set ys) $\longleftrightarrow$ (let n = CARD('a) in n > 0 $\wedge$ card(set (xs @ ys)) = n)

<proof> **definition** *eq-set* :: 'a set $\Rightarrow$ 'a set $\Rightarrow$ bool

where [*simp*, *code del*, *code-abbrev*]: *eq-set = (=)*

lemma *eq-set-code* [code]:

fixes *ys*

defines *rhs* $\equiv$

let n = CARD('a)

in if n = 0 then False else

```

    let xs' = remdups xs; ys' = remdups ys
    in length xs' + length ys' = n  $\wedge$  ( $\forall x \in \text{set } xs'. x \notin \text{set } ys'$ )  $\wedge$  ( $\forall y \in \text{set } ys'. y \notin \text{set } xs'$ )
    shows eq-set (List.coset xs) (set ys)  $\longleftrightarrow$  rhs
    and eq-set (set ys) (List.coset xs)  $\longleftrightarrow$  rhs
    and eq-set (set xs) (set ys)  $\longleftrightarrow$  ( $\forall x \in \text{set } xs. x \in \text{set } ys$ )  $\wedge$  ( $\forall y \in \text{set } ys. y \in \text{set } xs$ )
    and eq-set (List.coset xs) (List.coset ys)  $\longleftrightarrow$  ( $\forall x \in \text{set } xs. x \in \text{set } ys$ )  $\wedge$  ( $\forall y \in \text{set } ys. y \in \text{set } xs$ )
  <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-coset-error [code]:
  card (List.coset xs) =
    Code.abort (STR "card (List.coset -) requires type class instance card-UNIV")
      ( $\lambda$ -. card (List.coset xs))
  <proof>

lemma coset-subseteq-set-code [code]:
  List.coset xs  $\subseteq$  set ys  $\longleftrightarrow$ 
  (if xs = []  $\wedge$  ys = [] then False
   else Code.abort
     (STR "subset-eq (List.coset -) (List.set -) requires type class instance card-UNIV")
     ( $\lambda$ -. List.coset xs  $\subseteq$  set ys))
  <proof>

```

```

notepad begin — test code setup
  <proof>

```

```

end

```

```

end

```

```

theory Case-Converter
  imports Main
begin

```

12.8 Eliminating pattern matches

```

definition missing-pattern-match :: String.literal  $\Rightarrow$  (unit  $\Rightarrow$  'a)  $\Rightarrow$  'a where
  [code del]: missing-pattern-match m f = f ()

```


lemma *missing-pattern-match-cong* [cong]:
 $m = m' \implies \text{missing-pattern-match } m \ f = \text{missing-pattern-match } m' \ f$
 ⟨proof⟩

lemma *missing-pattern-match-code* [code-unfold]:
 $\text{missing-pattern-match} = \text{Code.abort}$
 ⟨proof⟩

⟨ML⟩

end

13 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 [2].

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.

13.1 The type *lazy*

typedef *'a lazy* = *UNIV* :: *'a set* ⟨proof⟩
setup-lifting *type-definition-lazy*
lift-definition *delay* :: $(\text{unit} \Rightarrow 'a) \Rightarrow 'a \text{ lazy}$ **is** $\lambda f. f \ ()$ ⟨proof⟩
lift-definition *force* :: *'a lazy* $\Rightarrow 'a$ **is** $\lambda x. x$ ⟨proof⟩

code-datatype *delay*
lemma *force-delay* [code]: $\text{force } (\text{delay } f) = f \ ()$ ⟨proof⟩
lemma *delay-force*: $\text{delay } (\lambda -. \text{force } s) = s$ ⟨proof⟩

definition *termify-lazy2* :: *'a* :: *typerep lazy* $\Rightarrow$ *term*
where *termify-lazy2* *x* =
Code-Evaluation.App (*Code-Evaluation.Const* (*STR "Code-Lazy.delay"*) (*TYPEREPR*((*unit*
 $\Rightarrow 'a) \Rightarrow 'a \text{ lazy}$)))

```
(Code-Evaluation.Const (STR "Pure.dummy-pattern") (TYPEREP((unit =>
'a))))
```

definition *termify-lazy* ::

```
(String.literal => 'typerep => 'term) =>
('term => 'term => 'term) =>
(String.literal => 'typerep => 'term => 'term) =>
'typerep => ('typerep => 'typerep => 'typerep) => ('typerep => 'typerep) =>
('a => 'term) => 'typerep => 'a :: typerep lazy => 'term => term
where termify-lazy - - - - - x = termify-lazy2 x
```

declare [[code drop: Code-Evaluation.term-of :: - lazy => -]]

lemma *term-of-lazy-code* [code]:

```
Code-Evaluation.term-of x ≡
termify-lazy
Code-Evaluation.Const Code-Evaluation.App Code-Evaluation.Abs
TYPEREP(unit) (λT U. typerep.Typerep (STR "fun") [T, U]) (λT. type-
rep.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* $\rightarrow$ (SML)

⟨signature LAZY =

sig

```
type 'a lazy;
val lazy : (unit -> 'a) -> 'a lazy;
val force : 'a lazy -> 'a;
val peek : 'a lazy -> 'a option
val termify-lazy :
(string -> 'typerep -> 'term) ->
('term -> 'term -> 'term) ->
(string -> 'typerep -> 'term -> 'term) ->
'typerep -> ('typerep -> 'typerep -> 'typerep) -> ('typerep -> 'typerep) ->
('a -> 'term) -> 'typerep -> 'a lazy -> 'term -> 'term;
end;
```

structure *Lazy* : LAZY =

struct

datatype 'a *content* =

```

    Delay of unit -> 'a
  | Value of 'a
  | Exn of exn;

datatype 'a lazy = Lazy of 'a content ref;

fun lazy f = Lazy (ref (Delay f));

fun force (Lazy x) = case !x of
  Delay f => (
    let val res = f (); val - = x := Value res; in res end
  handle exn => (x := Exn exn; raise exn))
  | Value x => x
  | Exn exn => raise exn;

fun peek (Lazy x) = case !x of
  Value x => SOME x
  | - => NONE;

fun termify-lazy const app abs unitT funT lazyT term-of T x - =
  app (const Code-Lazy.delay (funT (funT unitT T) (lazyT T)))
  (case peek x of SOME y => abs - unitT (term-of y)
  | - => const Pure.dummy-pattern (funT unitT T));

end;> 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)
<structure Termify-Lazy = struct
fun termify-lazy
  (-: string -> typ -> term) (-: term -> term -> term) (-: string -> typ ->
term -> term)
  (-: typ) (-: typ -> typ -> typ) (-: typ -> typ)
  (term-of: 'a -> term) (T: typ) (x: 'a Lazy.lazy) (-: term) =
  Const (Code-Lazy.delay, (HOLogic.unitT --> T) --> Type (Code-Lazy.lazy,
[T])) $
  (case Lazy.peek x of
    SOME (Exn.Res x) => absdummy HOLogic.unitT (term-of x)

```

```
| - => Const (Pure.dummy-pattern, Hologic.unitT --> T));
end;> 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)
⟨module Termify-Lazy : sig
  val termify-lazy :
    (string → 'typerep → 'term) →>
    ('term → 'term → 'term) →>
    (string → 'typerep → 'term → 'term) →>
    'typerep → ('typerep → 'typerep → 'typerep) →> ('typerep → 'typerep) →>
    ('a → 'term) →> 'typerep → 'a Lazy.t →> 'term →> 'term
end = struct
```

```
let termify-lazy const app abs unitT funT lazyT term-of ty x =
  app (const Code-Lazy.delay (funT (funT unitT ty) (lazyT ty)))
    (if Lazy.is-val x then abs - unitT (term-of (Lazy.force x))
     else const Pure.dummy-pattern (funT unitT ty));;
```

```
end;> for constant termify-lazy
| constant termify-lazy → (OCaml) Termify'-Lazy.termify'-lazy
```

code-reserved OCaml Lazy Termify-Lazy

code-printing

```
code-module Lazy → (Haskell) ⟨
module Lazy(Lazy, delay, force) where
```

```
newtype Lazy a = Lazy a
delay f = Lazy (f ())
force (Lazy x) = x 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)
⟨object Lazy {
  final class Lazy[A] (f: Unit => A) {
    var evaluated = false;
```

```

    lazy val x: A = f(())

    def get() : A = {
      evaluated = true;
      return x
    }
  }

  def force[A] (x: Lazy[A]) : A = {
    return x.get()
  }

  def delay[A] (f: Unit => A) : Lazy[A] = {
    return new Lazy[A] (f)
  }

  def termify-lazy[Typerep, Term, A] (
    const: String => Typerep => Term,
    app: Term => Term => Term,
    abs: String => Typerep => Term => Term,
    unitT: Typerep,
    funT: Typerep => Typerep => Typerep,
    lazyT: Typerep => Typerep,
    term-of: A => Term,
    ty: Typerep,
    x: Lazy[A],
    dummy: Term) : Term = {
    if (x.evaluated)
      app(const(Code-Lazy.delay)(funT(funT(unitT)(ty))(lazyT(ty))))(abs(-)(unitT)(term-of(x.get())))
    else
      app(const(Code-Lazy.delay)(funT(funT(unitT)(ty))(lazyT(ty))))(const(Pure.dummy-pattern)(funT(unitT)
  }
} 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 *delays*.

lemma *delay-lazy-cong*: *delay f = delay f* *⟨proof⟩*
⟨ML⟩

13.2 Implementation

⟨ML⟩

end

```

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

```

13.3 YXML encoding for *term*

```

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

```

```

lemma yot-anything:  $x = (y :: \text{yaml-of-term})$ 
  <proof>

```

```

definition yot-empty :: yaml-of-term where [code del]: yot-empty = YXML

```

```

definition yot-literal :: String.literal  $\Rightarrow$  yaml-of-term

```

```

  where [code del]: yot-literal - = YXML

```

```

definition yot-append :: yaml-of-term  $\Rightarrow$  yaml-of-term  $\Rightarrow$  yaml-of-term

```

```

  where [code del]: yot-append - - = YXML

```

```

definition yot-concat :: yaml-of-term list  $\Rightarrow$  yaml-of-term

```

```

  where [code del]: yot-concat - = YXML

```

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

```

code-printing type-constructor yaml-of-term

```

```

   $\rightarrow$  (SML) string

```

```

  and (OCaml) string

```

```

  and (Haskell) String

```

```

  and (Scala) String

```

```

| constant yot-empty

```

```

   $\rightarrow$  (SML)

```

```

  and (OCaml)

```

```

  and (Haskell)

```

```

  and (Scala)

```

```

| constant yot-literal

```

```

   $\rightarrow$  (SML) -

```

```

  and (OCaml) -

```

```

  and (Haskell) -

```

```

  and (Scala) -

```

```

| constant yot-append

```

```

   $\rightarrow$  (SML) String.concat [(-), (-)]

```

```

  and (OCaml) String.concat [(-); (-)]

```

```

  and (Haskell) infixr 5 ++

```

```

  and (Scala) infixl 5 +

```

```

| constant yot-concat

```

```

   $\rightarrow$  (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*: $x = (y :: \text{xml-tree})$
 $\langle \text{proof} \rangle$

context begin
 $\langle \text{ML} \rangle$

type-synonym *attributes* = (*String.literal* $\times$ *String.literal*) *list*
type-synonym *body* = *xml-tree list*

definition *Elem* :: *String.literal* $\Rightarrow$ *attributes* $\Rightarrow$ *xml-tree list* $\Rightarrow$ *xml-tree*
where [*code del*]: *Elem* - - - = *XML-Tree*

definition *Text* :: *String.literal* $\Rightarrow$ *xml-tree*
where [*code del*]: *Text* - = *XML-Tree*

definition *node* :: *xml-tree list* $\Rightarrow$ *xml-tree*
where *node ts* = *Elem* (*STR* "'") [] *ts*

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

definition *list where list f xs* = *map* (*node* $\circ$ *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* $\Rightarrow$ *yxml-of-term* $\Rightarrow$ *yxml-of-term*
where [*code del*]: *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 ($\lambda(a, x)$ *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* $\Rightarrow$ *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* $\Rightarrow$ *xml.body*
where [code del]: *xml-of-typ* - = [XML-Tree]

definition *xml-of-term* :: *Code-Evaluation.term* $\Rightarrow$ *xml.body*
where [code del]: *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* $\Rightarrow$ *yxml-of-term*
where *yxml-string-of-term* = *yxml-string-of-body* $\circ$ *xml-of-term*

13.4 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* :: ($'a \Rightarrow \text{bool}$) $\Rightarrow$ ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow$ $'a \Rightarrow 'a \Rightarrow \text{bool}$
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** ≈ 50)
 $\langle \text{proof} \rangle$

lemma *combine-per-top* [*simp*]: *combine-per* $\top R = R$
 $\langle \text{proof} \rangle$

lemma *combine-per-eq* [*simp*]: *combine-per* $P \text{ HOL.eq} = \text{HOL.eq} \sqcap (\lambda x y. P x)$
 $\langle \text{proof} \rangle$

lemma *symp-combine-per*: *symp* $R \Longrightarrow \text{symp} (\text{combine-per } P R)$
 $\langle \text{proof} \rangle$

lemma *transp-combine-per*: *transp* $R \Longrightarrow \text{transp} (\text{combine-per } P R)$
 $\langle \text{proof} \rangle$

lemma *combine-perI*: $P x \Longrightarrow P y \Longrightarrow x \approx y \Longrightarrow \text{combine-per } P R x y$ **for** R
(**infixl** ≈ 50)
 $\langle \text{proof} \rangle$

lemma *symp-combine-per-symp*: *symp* $R \Longrightarrow \text{symp} (\text{combine-per } P R)$
 $\langle \text{proof} \rangle$

lemma *transp-combine-per-transp*: *transp* $R \Longrightarrow \text{transp} (\text{combine-per } P R)$
 $\langle \text{proof} \rangle$

lemma *equivp-combine-per-part-equivp* [*intro?*]:
fixes R (**infixl** ≈ 50)
assumes $\exists x. P x$ **and** *equivp* R
shows *part-equivp* (*combine-per* $P R$)
 $\langle \text{proof} \rangle$

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 $((A \text{ ===> } A \text{ ===> } (=)) \text{ ===> } \text{rel-set } A \text{ ===> } (=)) \text{ Complete-Partial-Order.chain}$
Complete-Partial-Order.chain
 ⟨*proof*⟩

lemma *linorder-chain* [*simp, intro!*]:

fixes $Y :: - :: \text{linorder set}$

shows *Complete-Partial-Order.chain* $(\leq) \ Y$
 ⟨*proof*⟩

lemma *fun-lub-apply*: $\bigwedge \text{Sup. fun-lub Sup } Y \ x = \text{Sup } ((\lambda f. f \ x) \ ' \ Y)$
 ⟨*proof*⟩

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* $\{\} = (\lambda -. \text{lub } \{\})$
 ⟨*proof*⟩

lemma *chain-fun-ordD*:

assumes *Complete-Partial-Order.chain* $(\text{fun-ord } le) \ Y$

shows *Complete-Partial-Order.chain* $le \ ((\lambda f. f \ x) \ ' \ Y)$
 ⟨*proof*⟩

lemma *chain-Diff*:

Complete-Partial-Order.chain ord A

$\implies \text{Complete-Partial-Order.chain ord } (A - B)$
 ⟨*proof*⟩

lemma *chain-rel-prodD1*:

Complete-Partial-Order.chain $(\text{rel-prod } \text{orda } \text{ordb}) \ Y$

$\implies \text{Complete-Partial-Order.chain } \text{orda } (\text{fst } ' \ Y)$
 ⟨*proof*⟩

lemma *chain-rel-prodD2*:

Complete-Partial-Order.chain $(\text{rel-prod } \text{orda } \text{ordb}) \ Y$

$\implies \text{Complete-Partial-Order.chain } \text{ordb } (\text{snd } ' \ Y)$
 ⟨*proof*⟩

context *ccpo* **begin**

lemma *ccpo-fun*: *class.ccpo* $(\text{fun-lub Sup}) \ (\text{fun-ord } (\leq)) \ (\text{mk-less } (\text{fun-ord } (\leq)))$
 ⟨*proof*⟩

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain* $(\leq) \ Y \implies \text{Sup } Y \leq x$
 $\longleftrightarrow (\forall y \in Y. \ y \leq x)$
 ⟨*proof*⟩

lemma *Sup-minus-bot*:

assumes *chain*: *Complete-Partial-Order.chain* ($\leq$) *A*

shows $\sqcup (A - \{\sqcup \{\}\}) = \sqcup A$
 (**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*: $\bigwedge f. f \in Y \implies \text{monotone } le-b (\leq) f$

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*: $\bigwedge y. y \in Y \implies \text{monotone } le-b (\leq) (\lambda x. f x y)$

and *mono2*: $\bigwedge x a b. \llbracket x \in Y; a \sqsubseteq b; a \in Y; b \in Y \rrbracket \implies f x a \leq f x b$

begin

lemma *Sup-mono*:

assumes *le*: $x \sqsubseteq y$ **and** *x*: $x \in Y$ **and** *y*: $y \in Y$

shows $\sqcup (f x \text{ ‘ } Y) \leq \sqcup (f y \text{ ‘ } Y)$ (**is** $\leq$ *?rhs*)

<proof>

lemma *diag-Sup*: $\sqcup ((\lambda x. \sqcup (f x \text{ ‘ } Y)) \text{ ‘ } Y) = \sqcup ((\lambda x. f x x) \text{ ‘ } Y)$ (**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*: $\bigwedge x y. \llbracket x \sqsubseteq y; x \in Y \rrbracket \implies f x \leq f y$

shows *Sup* ($f \text{ ‘ } Y$) $\leq f (\bigvee Y)$

<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*: $\bigwedge f. f \in Z \implies \text{monotone } (\sqsubseteq) (\leq) f$

shows $\sqcup ((\lambda x. \sqcup (x \text{ ‘ } Y)) \text{ ‘ } Z) = \sqcup ((\lambda x. \sqcup ((\lambda f. f x) \text{ ‘ } Z)) \text{ ‘ } Y)$
 (**is** *?lhs* = *?rhs*)

<proof>

lemma *fixp-mono*:

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

and f : *monotone* $(\leq) (\leq) f$
and g : *monotone* $(\leq) (\leq) g$
shows $\text{ccpo-class.fixp } f \leq \text{ccpo-class.fixp } g$
 $\langle \text{proof} \rangle$

context fixes $\text{ordb} :: 'b \Rightarrow 'b \Rightarrow \text{bool}$ (**infix** $\sqsubseteq 60$) **begin**

lemma *iterates-mono*:

assumes f : $f \in \text{ccpo.iterates } (\text{fun-lub } \text{Sup}) (\text{fun-ord } (\leq)) F$
and mono : $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \implies \text{monotone } (\sqsubseteq) (\leq) (F f)$
shows $\text{monotone } (\sqsubseteq) (\leq) f$
 $\langle \text{proof} \rangle$

lemma *fixp-preserves-mono*:

assumes mono : $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. F f x)$
and mono2 : $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \implies \text{monotone } (\sqsubseteq) (\leq) (F f)$
shows $\text{monotone } (\sqsubseteq) (\leq) (\text{ccpo.fixp } (\text{fun-lub } \text{Sup}) (\text{fun-ord } (\leq)) F)$
 (**is** *monotone* - - ?*fixp*)
 $\langle \text{proof} \rangle$

end

end

lemma *monotone2monotone*:

assumes 2 : $\bigwedge x. \text{monotone } \text{ordb } \text{ordc } (\lambda y. f x y)$
and t : $\text{monotone } \text{orda } \text{ordb } (\lambda x. t x)$
and 1 : $\bigwedge y. \text{monotone } \text{orda } \text{ordc } (\lambda x. f x y)$
and trans : $\text{transp } \text{ordc}$
shows $\text{monotone } \text{orda } \text{ordc } (\lambda x. f x (t x))$
 $\langle \text{proof} \rangle$

15.1 Continuity

definition $\text{cont} :: ('a \text{ set} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \text{ set} \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$

where

$\text{cont } \text{luba } \text{orda } \text{lubb } \text{ordb } f \longleftrightarrow$
 $(\forall Y. \text{Complete-Partial-Order.chain } \text{orda } Y \longrightarrow Y \neq \{\} \longrightarrow f (\text{luba } Y) = \text{lubb } (f ` Y))$

definition $\text{mcont} :: ('a \text{ set} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \text{ set} \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$

where

$\text{mcont } \text{luba } \text{orda } \text{lubb } \text{ordb } f \longleftrightarrow$
 $\text{monotone } \text{orda } \text{ordb } f \wedge \text{cont } \text{luba } \text{orda } \text{lubb } \text{ordb } f$

15.1.1 Theorem collection *cont-intro*

named-theorems *cont-intro continuity and admissibility intro rules*

$\langle ML \rangle$

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

declare *if-mono*[*simp*]

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

lemma *monotone-applyI*:
 monotone orda ordb F $\implies$ *monotone (fun-ord orda) ordb* ($\lambda f. F (f x)$)
 $\langle proof \rangle$

lemma *monotone-if-fun* [*partial-function-mono*]:
 $\llbracket \text{monotone (fun-ord orda) (fun-ord ordb) } F; \text{monotone (fun-ord orda) (fun-ord ordb) } G \rrbracket$
 $\implies \text{monotone (fun-ord orda) (fun-ord ordb) } (\lambda f n. \text{if } c \text{ } n \text{ then } F f n \text{ else } G f n)$
 $\langle proof \rangle$

lemma *monotone-fun-apply-fun* [*partial-function-mono*]:
 monotone (fun-ord (fun-ord ord)) (fun-ord ord) ($\lambda f n. f t (g n)$)
 $\langle proof \rangle$

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

context *preorder* **begin**

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

lemma *monotone-const* [*simp*, *cont-intro*]: *monotone ord* ($\leq$) ($\lambda -. c$)
 $\langle proof \rangle$

end

lemma *transp-le* [*cont-intro*, *simp*]:
 class.preorder ord (mk-less ord) $\implies \text{transp ord}$
 $\langle proof \rangle$

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

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

lemma *transp-le* [*cont-intro*, *simp*]: *transp leq*
 $\langle proof \rangle$

lemma *preorder* [*cont-intro*, *simp*]: *class.preorder leq (mk-less leq)*
 $\langle proof \rangle$

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

end

lemma *contI* [*intro?*]:
 $(\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } \text{orda } Y; Y \neq \{\} \rrbracket \implies f (\text{luba } Y) = \text{lubb}$
 $(f \text{ ' } Y))$
 $\implies \text{cont luba } \text{orda } \text{lubb } \text{ordb } f$
 $\langle proof \rangle$

lemma *contD*:
 $\llbracket \text{cont luba } \text{orda } \text{lubb } \text{ordb } f; \text{Complete-Partial-Order.chain } \text{orda } Y; Y \neq \{\} \rrbracket$
 $\implies f (\text{luba } Y) = \text{lubb } (f \text{ ' } Y)$
 $\langle proof \rangle$

lemma *cont-id* [*simp*, *cont-intro*]: $\bigwedge \text{Sup. cont Sup ord Sup ord id}$
 $\langle proof \rangle$

lemma *cont-id'* [*simp*, *cont-intro*]: $\bigwedge \text{Sup. cont Sup ord Sup ord } (\lambda x. x)$
 $\langle proof \rangle$

lemma *cont-applyI* [*cont-intro*]:
assumes *cont*: *cont luba orda lubb ordb g*
shows *cont (fun-lub lub) (fun-ord orda) lubb ordb* $(\lambda f. g (f x))$
 $\langle proof \rangle$

lemma *call-cont*: *cont (fun-lub lub) (fun-ord ord) lub ord* $(\lambda f. f t)$
 $\langle proof \rangle$

lemma *cont-if* [*cont-intro*]:
 $\llbracket \text{cont luba } \text{orda } \text{lubb } \text{ordb } f; \text{cont luba } \text{orda } \text{lubb } \text{ordb } g \rrbracket$
 $\implies \text{cont luba } \text{orda } \text{lubb } \text{ordb } (\lambda x. \text{if } c \text{ then } f x \text{ else } g x)$
 $\langle proof \rangle$

lemma *mcontI* [*intro?*]:
 $\llbracket \text{monotone } \text{orda } \text{ordb } f; \text{cont luba } \text{orda } \text{lubb } \text{ordb } f \rrbracket \implies \text{mcont luba } \text{orda } \text{lubb}$
 $\text{ordb } f$
 $\langle proof \rangle$

lemma *mcont-mono*: *mcont luba orda lubb ordb f* $\implies \text{monotone } \text{orda } \text{ordb } f$
 $\langle proof \rangle$

lemma *mcont-cont* [*simp*]: *mcont luba orda lubb ordb f* $\implies$ *cont luba orda lubb ordb f*
 $\langle \text{proof} \rangle$

lemma *mcont-monoD*:
 $\llbracket \text{mcont luba orda lubb ordb } f; \text{ orda } x \ y \rrbracket \implies \text{ordb } (f \ x) \ (f \ y)$
 $\langle \text{proof} \rangle$

lemma *mcont-contD*:
 $\llbracket \text{mcont luba orda lubb ordb } f; \text{Complete-Partial-Order.chain orda } Y; Y \neq \{\} \rrbracket$
 $\implies f \ (\text{luba } Y) = \text{lubb } (f \ ' \ Y)$
 $\langle \text{proof} \rangle$

lemma *mcont-call* [*cont-intro*, *simp*]:
mcont (fun-lub lub) (fun-ord ord) lub ord ($\lambda f. f \ t$)
 $\langle \text{proof} \rangle$

lemma *mcont-id'* [*cont-intro*, *simp*]: *mcont lub ord lub ord* ($\lambda x. x$)
 $\langle \text{proof} \rangle$

lemma *mcont-applyI*:
mcont luba orda lubb ordb ($\lambda x. F \ x$) $\implies$ *mcont (fun-lub luba) (fun-ord orda) lubb ordb* ($\lambda f. F \ (f \ x)$)
 $\langle \text{proof} \rangle$

lemma *mcont-if* [*cont-intro*, *simp*]:
 $\llbracket \text{mcont luba orda lubb ordb } (\lambda x. f \ x); \text{mcont luba orda lubb ordb } (\lambda x. g \ x) \rrbracket$
 $\implies \text{mcont luba orda lubb ordb } (\lambda x. \text{if } c \text{ then } f \ x \text{ else } g \ x)$
 $\langle \text{proof} \rangle$

lemma *cont-fun-lub-apply*:
cont luba orda (fun-lub lubb) (fun-ord ordb) f $\longleftrightarrow$ ($\forall x. \text{cont luba orda lubb ordb } (\lambda y. f \ y \ x)$)
 $\langle \text{proof} \rangle$

lemma *mcont-fun-lub-apply*:
mcont luba orda (fun-lub lubb) (fun-ord ordb) f $\longleftrightarrow$ ($\forall x. \text{mcont luba orda lubb ordb } (\lambda y. f \ y \ x)$)
 $\langle \text{proof} \rangle$

context *ccpo* **begin**

lemma *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup* ($\leq$) ($\lambda x. c$)
 $\langle \text{proof} \rangle$

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda Sup ($\leq$) ($\lambda x. c$)
 $\langle \text{proof} \rangle$

lemma *cont-apply*:

assumes 2: $\bigwedge x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)$
and *t*: $\text{cont luba orda lubb ordb } (\lambda x. t x)$
and 1: $\bigwedge y. \text{cont luba orda Sup } (\leq) (\lambda x. f x y)$
and *mono*: $\text{monotone orda ordb } (\lambda x. t x)$
and *mono2*: $\bigwedge x. \text{monotone ordb } (\leq) (\lambda y. f x y)$
and *mono1*: $\bigwedge y. \text{monotone orda } (\leq) (\lambda x. f x y)$
shows $\text{cont luba orda Sup } (\leq) (\lambda x. f x (t x))$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont'*:

$\llbracket \bigwedge x. \text{mcont lub' ord' Sup } (\leq) (\lambda y. f x y);$
 $\bigwedge y. \text{mcont lub ord Sup } (\leq) (\lambda x. f x y);$
 $\text{mcont lub ord lub' ord' } (\lambda y. t y) \rrbracket$
 $\implies \text{mcont lub ord Sup } (\leq) (\lambda x. f x (t x))$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont*:

$\llbracket \text{mcont lub' ord' Sup } (\leq) (\lambda x. f x); \text{mcont lub ord lub' ord' } (\lambda x. t x) \rrbracket$
 $\implies \text{mcont lub ord Sup } (\leq) (\lambda x. f (t x))$
 $\langle \text{proof} \rangle$

context

fixes *ord* :: 'b $\Rightarrow$ 'b $\Rightarrow$ bool (**infix** $\sqsubseteq$ 60)
and *lub* :: 'b set $\Rightarrow$ 'b ($\bigvee$)

begin

lemma *cont-fun-lub-Sup*:

assumes *chainM*: $\text{Complete-Partial-Order.chain } (\text{fun-ord } (\leq)) M$
and *mcont* [rule-format]: $\forall f \in M. \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) f$
shows $\text{cont lub } (\sqsubseteq) \text{ Sup } (\leq) (\text{fun-lub Sup } M)$
 $\langle \text{proof} \rangle$

lemma *mcont-fun-lub-Sup*:

$\llbracket \text{Complete-Partial-Order.chain } (\text{fun-ord } (\leq)) M;$
 $\forall f \in M. \text{mcont lub ord Sup } (\leq) f \rrbracket$
 $\implies \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (\text{fun-lub Sup } M)$
 $\langle \text{proof} \rangle$

lemma *iterates-mcont*:

assumes *f*: $f \in \text{ccpo.iterates } (\text{fun-lub Sup}) (\text{fun-ord } (\leq)) F$
and *mono*: $\bigwedge f. \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) f \implies \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (F f)$
shows $\text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) f$
 $\langle \text{proof} \rangle$

lemma *fixp-preserves-mcont*:

assumes *mono*: $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. F f x)$
and *mcont*: $\bigwedge f. \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) f \implies \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (F f)$
shows $\text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (\text{ccpo.fixp } (\text{fun-lub Sup}) (\text{fun-ord } (\leq)) F)$

(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: $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. U (F (C f)) x)$
 and eq: $f \equiv C (\text{ccpo.fixp } (\text{fun-lub Sup}) (\text{fun-ord } (\leq)) (\lambda f. U (F (C f))))$
 and inverse: $\bigwedge f. U (C f) = f$
 begin

lemma fixp-preserves-mono-uc:

assumes mono2: $\bigwedge f. \text{monotone ord } (\leq) (U f) \implies \text{monotone ord } (\leq) (U (F f))$
 shows $\text{monotone ord } (\leq) (U f)$
 <proof>

lemma fixp-preserves-mcont-uc:

assumes mcont: $\bigwedge f. \text{mcont lubb ordb Sup } (\leq) (U f) \implies \text{mcont lubb ordb Sup } (\leq) (U (F f))$
 shows $\text{mcont lubb ordb Sup } (\leq) (U f)$
 <proof>

end

lemmas fixp-preserves-mono1 = fixp-preserves-mono-uc[of $\lambda x. x - \lambda x. x$, OF - - refl]

lemmas fixp-preserves-mono2 =
 fixp-preserves-mono-uc[of case-prod - curry, unfolded case-prod-curry curry-case-prod, OF - - refl]

lemmas fixp-preserves-mono3 =
 fixp-preserves-mono-uc[of $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$, unfolded case-prod-curry curry-case-prod, OF - - refl]

lemmas fixp-preserves-mono4 =
 fixp-preserves-mono-uc[of $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$, unfolded case-prod-curry curry-case-prod, OF - - refl]

lemmas fixp-preserves-mcont1 = fixp-preserves-mcont-uc[of $\lambda x. x - \lambda x. x$, OF - - refl]

lemmas fixp-preserves-mcont2 =
 fixp-preserves-mcont-uc[of case-prod - curry, unfolded case-prod-curry curry-case-prod, OF - - refl]

lemmas fixp-preserves-mcont3 =
 fixp-preserves-mcont-uc[of $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$, unfolded case-prod-curry curry-case-prod, OF - - refl]

lemmas fixp-preserves-mcont4 =
 fixp-preserves-mcont-uc[of $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$, unfolded case-prod-curry curry-case-prod, OF - - refl]

end

lemma (in *preorder*) *monotone-if-bot*:

fixes *bot*

assumes *mono*: $\bigwedge x y. \llbracket x \leq y; \neg (x \leq \text{bound}) \rrbracket \implies \text{ord } (f x) (f y)$

and *bot*: $\bigwedge x. \neg x \leq \text{bound} \implies \text{ord bot } (f x) \text{ ord bot bot}$

shows *monotone* $(\leq) \text{ ord } (\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x)$

<proof>

lemma (in *ccpo*) *mcont-if-bot*:

fixes *bot* **and** *lub* $(\bigvee)$ **and** *ord* (infix $\sqsubseteq$ 60)

assumes *ccpo*: *class.ccpo* *lub* $(\sqsubseteq)$ *lt*

and *mono*: $\bigwedge x y. \llbracket x \leq y; \neg x \leq \text{bound} \rrbracket \implies f x \sqsubseteq f y$

and *cont*: $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } (\leq) Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg x \leq \text{bound} \rrbracket \implies f (\bigsqcup Y) = \bigvee (f ` Y)$

and *bot*: $\bigwedge x. \neg x \leq \text{bound} \implies \text{bot} \sqsubseteq f x$

shows *mcont* *Sup* $(\leq) \text{ lub } (\sqsubseteq) (\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x)$ (**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*: $\bigwedge x. g\ x = (\text{if } \text{leq } x\ \text{bound then } \text{bot else } f\ x)$

and *mono*: $\bigwedge x\ y. \llbracket \text{leq } x\ y; \neg \text{leq } x\ \text{bound} \rrbracket \implies \text{ord } (f\ x)\ (f\ y)$

and *bot*: $\bigwedge x. \neg \text{leq } x\ \text{bound} \implies \text{ord } \text{bot}\ (f\ x)\ \text{ord } \text{bot}\ \text{bot}$

shows *monotone leq ord g*

<proof>

lemma *mcont-if-bot*:

fixes *bot*

assumes *ccpo*: *class.ccpo lub' ord (mk-less ord)*

and *bot*: $\bigwedge x. \neg \text{leq } x\ \text{bound} \implies \text{ord } \text{bot}\ (f\ x)$

and *g*: $\bigwedge x. g\ x = (\text{if } \text{leq } x\ \text{bound then } \text{bot else } f\ x)$

and *mono*: $\bigwedge x\ y. \llbracket \text{leq } x\ y; \neg \text{leq } x\ \text{bound} \rrbracket \implies \text{ord } (f\ x)\ (f\ y)$

and *cont*: $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain leq } Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg \text{leq } x\ \text{bound} \rrbracket \implies f\ (\text{lub } Y) = \text{lub}'\ (f\ ` Y)$

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*]:

$\llbracket \text{class.ccpo lub ord (mk-less ord)}; \text{ccpo.admissible lub ord } P; \text{ccpo.admissible lub ord } Q \rrbracket$

$\implies \text{ccpo.admissible lub ord } (\lambda x. P\ x \vee Q\ x)$

<proof>

lemma *admissible-imp'* [*cont-intro*]:

$\llbracket \text{class.ccpo lub ord (mk-less ord)}; \rrbracket$

$ccpo.admissible \text{ lub ord } (\lambda x. \neg P x);$
 $ccpo.admissible \text{ lub ord } (\lambda x. Q x) \parallel$
 $\implies cppo.admissible \text{ lub ord } (\lambda x. P x \longrightarrow Q x)$
 $\langle proof \rangle$

lemma *admissible-imp* [*cont-intro*]:
 $(Q \implies cppo.admissible \text{ lub ord } (\lambda x. P x))$
 $\implies cppo.admissible \text{ lub ord } (\lambda x. Q \longrightarrow P x)$
 $\langle proof \rangle$

lemma *admissible-not-mem'* [*THEN admissible-subst, cont-intro, simp*]:
shows *admissible-not-mem*: $ccpo.admissible \text{ Union } (\subseteq) (\lambda A. x \notin A)$
 $\langle proof \rangle$

lemma *admissible-eqI*:
assumes $f: cont \text{ luba orda lub ord } (\lambda x. f x)$
and $g: cont \text{ luba orda lub ord } (\lambda x. g x)$
shows $ccpo.admissible \text{ luba orda } (\lambda x. f x = g x)$
 $\langle proof \rangle$

corollary *admissible-eq-mcontI* [*cont-intro*]:
 $\parallel mcont \text{ luba orda lub ord } (\lambda x. f x);$
 $mcont \text{ luba orda lub ord } (\lambda x. g x) \parallel$
 $\implies cppo.admissible \text{ luba orda } (\lambda x. f x = g x)$
 $\langle proof \rangle$

lemma *admissible-iff* [*cont-intro, simp*]:
 $\parallel cppo.admissible \text{ lub ord } (\lambda x. P x \longrightarrow Q x); cppo.admissible \text{ lub ord } (\lambda x. Q x \longrightarrow$
 $P x) \parallel$
 $\implies cppo.admissible \text{ lub ord } (\lambda x. P x \longleftrightarrow Q x)$
 $\langle proof \rangle$

context *ccpo* **begin**

lemma *admissible-leI*:
assumes $f: mcont \text{ luba orda Sup } (\leq) (\lambda x. f x)$
and $g: mcont \text{ luba orda Sup } (\leq) (\lambda x. g x)$
shows $ccpo.admissible \text{ luba orda } (\lambda x. f x \leq g x)$
 $\langle proof \rangle$

end

lemma *admissible-leI*:
fixes ord (**infix** $\sqsubseteq$ 60) **and** lub ($\bigvee$)
assumes $class.ccpo \text{ lub } (\sqsubseteq) (mk-less (\sqsubseteq))$
and $mcont \text{ luba orda lub } (\sqsubseteq) (\lambda x. f x)$
and $mcont \text{ luba orda lub } (\sqsubseteq) (\lambda x. g x)$
shows $ccpo.admissible \text{ luba orda } (\lambda x. f x \sqsubseteq g x)$
 $\langle proof \rangle$

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

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible* *Sup* ($\leq$) ($\lambda x. \neg (\leq) x y$)
 $\langle \text{proof} \rangle$

end

lemma (**in** *preorder*) *preorder* [*cont-intro*, *simp*]: *class.preorder* ($\leq$) (*mk-less* ($\leq$))
 $\langle \text{proof} \rangle$

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

$\langle ML \rangle$

inductive *compact* :: (*'a set* $\Rightarrow$ *'a*) $\Rightarrow$ (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a* $\Rightarrow$ *bool*
for *lub ord x*
where *compact*:
 $\llbracket \text{ccpo.admissible } \text{lub } \text{ord } (\lambda y. \neg \text{ord } x y);$
 $\text{ccpo.admissible } \text{lub } \text{ord } (\lambda y. x \neq y) \rrbracket$
 $\implies \text{compact } \text{lub } \text{ord } x$

$\langle ML \rangle$

context *ccpo* **begin**

lemma *compactI*:
assumes *ccpo.admissible* *Sup* ($\leq$) ($\lambda y. \neg x \leq y$)
shows *ccpo.compact* *Sup* ($\leq$) *x*
 $\langle \text{proof} \rangle$

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

end

lemma *admissible-compact-neq'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-compact-neq*: *ccpo.compact lub ord k* $\implies$ *ccpo.admissible lub*

ord ($\lambda x. k \neq x$)
 $\langle \text{proof} \rangle$

lemma *admissible-neq-compact'* [*THEN* *admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-neq-compact*: *ccpo.compact lub ord k* $\implies$ *ccpo.admissible lub*
ord ($\lambda x. x \neq k$)
 $\langle \text{proof} \rangle$

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*: $\bigwedge x. \llbracket x \leq \text{ccpo-class.fixp } f; P x \rrbracket \implies P (f x)$
shows *P* (*ccpo-class.fixp f*)
 $\langle \text{proof} \rangle$

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* :: $('b \Rightarrow 'a) \Rightarrow \text{bool}$
assumes *mono*: $\bigwedge x. \text{mono-body } (\lambda f. U (F (C f)) x)$
and *eq*: $f \equiv C (\text{fixp-fun } (\lambda f. U (F (C f))))$
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*: $\bigwedge f'. \llbracket P (U f'); \text{le-fun } (U f') (U f) \rrbracket \implies P (U (F f'))$
shows *P* (*U f*)
 $\langle \text{proof} \rangle$

end

15.3 (=) as order

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

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* |]
 $\Rightarrow$ 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 lub*a; *lub-singleton lub*b |] $\Rightarrow$ *lub-singleton* (*prod-lub lub*a *lub*b)
 <proof>

lemma *prod-lub-empty* [simp]: *prod-lub lub*a *lub*b {} = (*lub*a {}, *lub*b {})
 <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*: $\bigwedge a. \text{monotone } \text{ordb } \text{ordc } (\lambda b. f(a, b))$
and *mono1*: $\bigwedge b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f(a, b))$
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* ($\lambda a. f(a, b)$)
 <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* ($\lambda b. f(a, b)$)
 <proof>

lemma *monotone-case-prodI*:

$\llbracket \bigwedge a. \text{monotone } \text{ordb } \text{ordc } (f\ a); \bigwedge b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f\ a\ b);$
class.preorder *orda* (*mk-less* *orda*); *class.preorder* *ordb* (*mk-less* *ordb*);
class.preorder *ordc* (*mk-less* *ordc*) $\rrbracket$
 $\implies \text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } (\text{case-prod } f)$
 <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* ($\lambda a. f\ a\ b$)
 <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 $\longleftrightarrow$
($\forall a.$ monotone ordb ordc ($\lambda b.$ f (a, b))) $\wedge$
($\forall b.$ monotone orda ordc ($\lambda a.$ f (a, b)))

<proof>

lemma *monotone-case-prod-iff [simp]:*

monotone (rel-prod orda ordb) ordc (case-prod f) $\longleftrightarrow$
($\forall a.$ monotone ordb ordc (f a)) $\wedge$ ($\forall b.$ monotone orda ordc ($\lambda a.$ f a b))

<proof>

end

lemma *monotone-case-prod-apply-iff:*

monotone orda ordb ($\lambda x.$ (case-prod f x) y) $\longleftrightarrow$ monotone orda ordb (case-prod
($\lambda a b.$ f a b y))

<proof>

lemma *monotone-case-prod-applyD:*

monotone orda ordb ($\lambda x.$ (case-prod f x) y)
 $\implies$ *monotone orda ordb (case-prod ($\lambda a b.$ f a b y))*

<proof>

lemma *monotone-case-prod-applyI:*

monotone orda ordb (case-prod ($\lambda a b.$ f a b y))
 $\implies$ *monotone orda ordb ($\lambda x.$ (case-prod f x) y)*

<proof>

lemma *cont-case-prod-apply-iff:*

cont luba orda lubb ordb ($\lambda x.$ (case-prod f x) y) $\longleftrightarrow$ cont luba orda lubb ordb
(case-prod ($\lambda a b.$ f a b y))

<proof>

lemma *cont-case-prod-applyI:*

cont luba orda lubb ordb (case-prod ($\lambda a b.$ f a b y))
 $\implies$ *cont luba orda lubb ordb ($\lambda x.$ (case-prod f x) y)*

<proof>

lemma *cont-case-prod-applyD:*

cont luba orda lubb ordb ($\lambda x.$ (case-prod f x) y)

$\implies \text{cont luba orda lubb ordb (case-prod } (\lambda a \ b. f \ a \ b \ y))$
 $\langle \text{proof} \rangle$

lemma *mcont-case-prod-apply-iff [simp]*:
 $\text{mcont luba orda lubb ordb } (\lambda x. (\text{case-prod } f \ x) \ y) \longleftrightarrow$
 $\text{mcont luba orda lubb ordb (case-prod } (\lambda a \ b. f \ a \ b \ y))$
 $\langle \text{proof} \rangle$

lemma *cont-prodD1*:
assumes *cont*: $\text{cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc } f$
and *class.preorder orda (mk-less orda)*
and *luba*: $\text{lub-singleton luba}$
shows $\text{cont lubb ordb lubc ordc } (\lambda y. f \ (x, y))$
 $\langle \text{proof} \rangle$

lemma *cont-prodD2*:
assumes *cont*: $\text{cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc } f$
and *class.preorder ordb (mk-less ordb)*
and *lubb*: $\text{lub-singleton lubb}$
shows $\text{cont luba orda lubc ordc } (\lambda x. f \ (x, y))$
 $\langle \text{proof} \rangle$

lemma *cont-case-prodD1*:
assumes *cont* $(\text{prod-lub luba lubb) (rel-prod orda ordb) lubc ordc (case-prod } f)$
and *class.preorder orda (mk-less orda)*
and *lub-singleton luba*
shows $\text{cont lubb ordb lubc ordc } (f \ x)$
 $\langle \text{proof} \rangle$

lemma *cont-case-prodD2*:
assumes *cont* $(\text{prod-lub luba lubb) (rel-prod orda ordb) lubc ordc (case-prod } f)$
and *class.preorder ordb (mk-less ordb)*
and *lub-singleton lubb*
shows $\text{cont luba orda lubc ordc } (\lambda x. f \ x \ y)$
 $\langle \text{proof} \rangle$

context *ccpo* **begin**

lemma *cont-prodI*:
assumes *mono*: $\text{monotone (rel-prod orda ordb) } (\leq) \ f$
and *cont1*: $\bigwedge x. \text{cont lubb ordb Sup } (\leq) \ (\lambda y. f \ (x, y))$
and *cont2*: $\bigwedge y. \text{cont luba orda Sup } (\leq) \ (\lambda x. f \ (x, y))$
and *class.preorder orda (mk-less orda)*
and *class.preorder ordb (mk-less ordb)*
shows $\text{cont (prod-lub luba lubb) (rel-prod orda ordb) Sup } (\leq) \ f$
 $\langle \text{proof} \rangle$

lemma *cont-case-prodI*:
assumes *monotone* $(\text{rel-prod orda ordb) } (\leq) \ (\text{case-prod } f)$

and $\bigwedge x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)$
and $\bigwedge y. \text{cont luba orda Sup } (\leq) (\lambda x. f x y)$
and $\text{class.preorder orda (mk-less orda)}$
and $\text{class.preorder ordb (mk-less ordb)}$
shows $\text{cont (prod-lub luba lubb) (rel-prod orda ordb) Sup } (\leq) (\text{case-prod } f)$
 $\langle \text{proof} \rangle$

lemma *cont-case-prod-iff*:

$\llbracket \text{monotone (rel-prod orda ordb) } (\leq) (\text{case-prod } f);$
 $\text{class.preorder orda (mk-less orda); lub-singleton luba};$
 $\text{class.preorder ordb (mk-less ordb); lub-singleton lubb} \rrbracket$
 $\implies \text{cont (prod-lub luba lubb) (rel-prod orda ordb) Sup } (\leq) (\text{case-prod } f) \longleftrightarrow$
 $(\forall x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)) \wedge (\forall y. \text{cont luba orda Sup } (\leq) (\lambda x. f x y))$
 $\langle \text{proof} \rangle$

end

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

lemma *mono2mono2*:

assumes $f: \text{monotone (rel-prod ordb ordc) leq } (\lambda(x, y). f x y)$
and $t: \text{monotone orda ordb } (\lambda x. t x)$
and $t': \text{monotone orda ordc } (\lambda x. t' x)$
shows $\text{monotone orda leq } (\lambda x. f (t x) (t' x))$
 $\langle \text{proof} \rangle$

lemma *cont-case-prodI* [*cont-intro*]:

$\llbracket \text{monotone (rel-prod orda ordb) leq (case-prod } f);$
 $\bigwedge x. \text{cont lubb ordb lub leq } (\lambda y. f x y);$
 $\bigwedge y. \text{cont luba orda lub leq } (\lambda x. f x y);$
 $\text{class.preorder orda (mk-less orda);}$
 $\text{class.preorder ordb (mk-less ordb)} \rrbracket$
 $\implies \text{cont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod } f)$
 $\langle \text{proof} \rangle$

lemma *cont-case-prod-iff*:

$\llbracket \text{monotone (rel-prod orda ordb) leq (case-prod } f);$
 $\text{class.preorder orda (mk-less orda); lub-singleton luba};$
 $\text{class.preorder ordb (mk-less ordb); lub-singleton lubb} \rrbracket$
 $\implies \text{cont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod } f) \longleftrightarrow$
 $(\forall x. \text{cont lubb ordb lub leq } (\lambda y. f x y)) \wedge (\forall y. \text{cont luba orda lub leq } (\lambda x. f x y))$
 $\langle \text{proof} \rangle$

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

$\llbracket \text{class.preorder orda (mk-less orda); lub-singleton luba};$
 $\text{class.preorder ordb (mk-less ordb); lub-singleton lubb} \rrbracket$
 $\implies \text{mcont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod } f) \longleftrightarrow$
 $(\forall x. \text{mcont lubb ordb lub leq } (\lambda y. f x y)) \wedge (\forall y. \text{mcont luba orda lub leq } (\lambda x. f x y))$

$y))$
 $\langle \text{proof} \rangle$

end

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

15.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: *class.ccpo* *Sup* $(\leq)$ $(<)$
 $\langle \text{proof} \rangle$

lemma *complete-lattice-ccpo'*: *class.ccpo* *Sup* $(\leq)$ $(\text{mk-less } (\leq))$
 $\langle \text{proof} \rangle$

lemma *complete-lattice-partial-function-definitions*:
partial-function-definitions $(\leq)$ *Sup*
 $\langle \text{proof} \rangle$

lemma *complete-lattice-partial-function-definitions-dual*:
partial-function-definitions $(\geq)$ *Inf*
 $\langle \text{proof} \rangle$

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 $f: \text{monotone } \text{ord } (\leq) (\lambda x. f x)$
and $g: \text{monotone } \text{ord } (\leq) (\lambda x. g x)$
shows $\text{monotone } \text{ord } (\leq) (\lambda x. f x \sqcap g x)$
 $\langle \text{proof} \rangle$

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

lemma *mono2mono-sup*:
assumes $f: \text{monotone } \text{ord } (\leq) (\lambda x. f x)$
and $g: \text{monotone } \text{ord } (\leq) (\lambda x. g x)$
shows $\text{monotone } \text{ord } (\leq) (\lambda x. f x \sqcup g x)$
 $\langle \text{proof} \rangle$

lemma *Sup-image-sup*:
assumes $Y \neq \{\}$

shows $\sqcup ((\sqcup) x \text{ ‘ } Y) = x \sqcup \sqcup Y$
 $\langle \text{proof} \rangle$

lemma *mcont-sup1*: $mcont\ Sup (\leq) Sup (\leq) (\lambda y. x \sqcup y)$
 $\langle \text{proof} \rangle$

lemma *mcont-sup2*: $mcont\ Sup (\leq) Sup (\leq) (\lambda x. x \sqcup y)$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-sup* [*cont-intro*, *simp*]:
 $\llbracket mcont\ lub\ ord\ Sup (\leq) (\lambda x. f\ x);$
 $mcont\ lub\ ord\ Sup (\leq) (\lambda x. g\ x) \rrbracket$
 $\implies mcont\ lub\ ord\ Sup (\leq) (\lambda x. f\ x \sqcup g\ x)$
 $\langle \text{proof} \rangle$

end

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

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

lemma *mcont-inf1*: $mcont\ Sup (\leq) Sup (\leq) (\lambda y. x \sqcap y)$
 $\langle \text{proof} \rangle$

lemma *mcont-inf2*: $mcont\ Sup (\leq) Sup (\leq) (\lambda x. x \sqcap y)$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-inf* [*cont-intro*, *simp*]:
 $\llbracket mcont\ lub\ ord\ Sup (\leq) (\lambda x. f\ x);$
 $mcont\ lub\ ord\ Sup (\leq) (\lambda x. g\ x) \rrbracket$
 $\implies mcont\ lub\ ord\ Sup (\leq) (\lambda x. f\ x \sqcap g\ x)$
 $\langle \text{proof} \rangle$

end

interpretation *lfp*: *partial-function-definitions* ($\leq$) :: - :: *complete-lattice* $\Rightarrow$ - *Sup*
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

interpretation *gfp*: *partial-function-definitions* ($\geq$) :: - :: *complete-lattice* $\Rightarrow$ - *Inf*
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *insert-mono* [*partial-function-mono*]:
 $monotone\ (fun\ ord\ (\subseteq))\ (\subseteq)\ A \implies monotone\ (fun\ ord\ (\subseteq))\ (\subseteq)\ (\lambda y. insert\ x$
 $(A\ y))$
 $\langle \text{proof} \rangle$

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

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

lemma *mono2mono-image* [*THEN lfp.mono2mono, cont-intro, simp*]:
shows *monotone-image*: *monotone* ($\subseteq$) ($\subseteq$) ($((\cdot) f)$)
 ⟨*proof*⟩

lemma *cont-image*: *cont* *Union* ($\subseteq$) *Union* ($\subseteq$) ($((\cdot) f)$)
 ⟨*proof*⟩

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

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro, simp*]:
monotone ord ($\subseteq$) *f* $\implies$ *monotone ord* ($\leq$) ($\lambda x. \bigsqcup f x$)
 ⟨*proof*⟩

lemma *cont-Sup*:
assumes *cont lub ord* *Union* ($\subseteq$) *f*
shows *cont lub ord* *Sup* ($\leq$) ($\lambda x. \bigsqcup f x$)
 ⟨*proof*⟩

lemma *mcont-Sup*: *mcont lub ord* *Union* ($\subseteq$) *f* $\implies$ *mcont lub ord* *Sup* ($\leq$) ($\lambda x. \bigsqcup f x$)
 ⟨*proof*⟩

lemma *monotone-SUP*:
 $\llbracket \text{monotone ord } (\subseteq) f; \bigwedge y. \text{monotone ord } (\leq) (\lambda x. g x y) \rrbracket \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup_{y \in f x} g x y)$
 ⟨*proof*⟩

lemma *monotone-SUP2*:
 $(\bigwedge y. y \in A \implies \text{monotone ord } (\leq) (\lambda x. g x y)) \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup_{y \in A} g x y)$
 ⟨*proof*⟩

lemma *cont-SUP*:
assumes *f*: *mcont lub ord* *Union* ($\subseteq$) *f*
and *g*: $\bigwedge y. \text{mcont lub ord } \text{Sup } (\leq) (\lambda x. g x y)$
shows *cont lub ord* *Sup* ($\leq$) ($\lambda x. \bigsqcup_{y \in f x} g x y$)

$\langle \text{proof} \rangle$

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

$\llbracket \text{mcont lub ord Union } (\subseteq) f; \bigwedge y. \text{mcont lub ord Sup } (\leq) (\lambda x. g \ x \ y) \rrbracket$
 $\implies \text{mcont lub ord Sup } (\leq) (\lambda x. \bigsqcup y \in f \ x. g \ x \ y)$

$\langle \text{proof} \rangle$

end

lemma *admissible-Ball* [*cont-intro*, *simp*]:

$\llbracket \bigwedge x. \text{ccpo.admissible lub ord } (\lambda A. P \ A \ x);$
 $\text{mcont lub ord Union } (\subseteq) f;$
 $\text{class.ccpo lub ord } (\text{mk-less ord}) \rrbracket$
 $\implies \text{ccpo.admissible lub ord } (\lambda A. \forall x \in f \ A. P \ A \ x)$

$\langle \text{proof} \rangle$

lemma *admissible-Bex'* [*THEN admissible-subst*, *cont-intro*, *simp*]:

shows *admissible-Bex*: $\text{ccpo.admissible Union } (\subseteq) (\lambda A. \exists x \in A. P \ x)$

$\langle \text{proof} \rangle$

15.6 Parallel fixpoint induction

context

fixes *luba* :: 'a set $\Rightarrow$ 'a
and *orda* :: 'a $\Rightarrow$ 'a $\Rightarrow$ bool
and *lubb* :: 'b set $\Rightarrow$ 'b
and *ordb* :: 'b $\Rightarrow$ 'b $\Rightarrow$ 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* $\langle \text{proof} \rangle$

interpretation *b*: *ccpo lubb ordb mk-less ordb* $\langle \text{proof} \rangle$

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')

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *monotone-map-prod* [*simp*]:

monotone (*rel-prod orda ordb*) (*rel-prod ordc ordd*) (*map-prod f g*) $\longleftrightarrow$
monotone orda ordc f $\wedge$ *monotone ordb ordd g*

$\langle \text{proof} \rangle$

lemma *parallel-fixp-induct*:

assumes *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) ($\lambda x. P$
(fst x) (snd x))
and *f*: *monotone orda orda f*
and *g*: *monotone ordb ordb g*
and *bot*: *P (luba {}) (lubb {})*
and *step*: $\bigwedge x y. P x y \implies P (f x) (g y)$
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*: $\bigwedge x. \text{monotone (fun-ord orda) orda } (\lambda f. U1 (F (C1 f)) x)$
and *G*: $\bigwedge y. \text{monotone (fun-ord ordb) ordb } (\lambda g. U2 (G (C2 g)) y)$
and *eq1*: $f \equiv C1 (\text{ccpo.fixp (fun-lub luba) (fun-ord orda) } (\lambda f. U1 (F (C1 f))))$
and *eq2*: $g \equiv C2 (\text{ccpo.fixp (fun-lub lubb) (fun-ord ordb) } (\lambda g. U2 (G (C2 g))))$
and *inverse*: $\bigwedge f. U1 (C1 f) = f$
and *inverse2*: $\bigwedge g. U2 (C2 g) = g$
and *adm*: *ccpo.admissible* (*prod-lub (fun-lub luba) (fun-lub lubb)*) (*rel-prod (fun-ord*
orda) (fun-ord ordb)) ($\lambda x. P (\text{fst } x) (\text{snd } x)$)
and *bot*: *P (λ-. luba {}) (λ-. lubb {})*
and *step*: $\bigwedge f g. P (U1 f) (U2 g) \implies P (U1 (F f)) (U2 (G g))$
shows *P (U1 f) (U2 g)*
<proof>

lemmas *parallel-fixp-induct-1-1 = parallel-fixp-induct-uc*[

of - - - λx. x - λx. x λx. x - λx. x,
OF - - - - - refl refl]

lemmas *parallel-fixp-induct-2-2 = parallel-fixp-induct-uc*[

of - - - case-prod - curry case-prod - curry,
where *P*= $\lambda f g. P (\text{curry } f) (\text{curry } g)$,
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
 $\implies \text{mcont lub ord luba orda } (\lambda x. \text{fst } (t x))$
<proof>

lemma *monotone-snd*: *monotone (rel-prod orda ordb) ordb snd*
 $\langle \text{proof} \rangle$

lemma *mcont-snd*: *mcont (prod-lub luba lubb) (rel-prod orda ordb) lubb ordb snd*
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-snd* [*cont-intro*, *simp*]:
 $\text{mcont lub ord (prod-lub luba lubb) (rel-prod orda ordb) } t$
 $\implies \text{mcont lub ord lubb ordb } (\lambda x. \text{snd } (t \ x))$
 $\langle \text{proof} \rangle$

lemma *monotone-Pair*:
 $\llbracket \text{monotone ord orda } f; \text{monotone ord ordb } g \rrbracket$
 $\implies \text{monotone ord (rel-prod orda ordb) } (\lambda x. (f \ x, g \ x))$
 $\langle \text{proof} \rangle$

lemma *cont-Pair*:
 $\llbracket \text{cont lub ord luba orda } f; \text{cont lub ord lubb ordb } g \rrbracket$
 $\implies \text{cont lub ord (prod-lub luba lubb) (rel-prod orda ordb) } (\lambda x. (f \ x, g \ x))$
 $\langle \text{proof} \rangle$

lemma *mcont-Pair*:
 $\llbracket \text{mcont lub ord luba orda } f; \text{mcont lub ord lubb ordb } g \rrbracket$
 $\implies \text{mcont lub ord (prod-lub luba lubb) (rel-prod orda ordb) } (\lambda x. (f \ x, g \ x))$
 $\langle \text{proof} \rangle$

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*]:
 $\text{mono-option } B \implies \text{mono-option } (\lambda f. \text{map-option } g \ (B \ f))$
 $\langle \text{proof} \rangle$

lemma *compact-flat-lub* [*cont-intro*]: *ccpo.compact (flat-lub x) (flat-ord x) y*
 $\langle \text{proof} \rangle$

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* $\implies$ *A* $\implies$ (=)) (=) (=)
 ⟨*proof*⟩

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

lemma *is-equality-Rel*: *is-equality* *A* $\implies$ *Transfer.Rel A t t*
 ⟨*proof*⟩

lemma *Rel-Rel-match*: *Transfer.Rel R x y* $\implies$ *Rel-match R x y*
 ⟨*proof*⟩

lemma *Rel-match-Rel*: *Rel-match R x y* $\implies$ *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* $\implies$ *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* :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ **where**
strong-confluentp $r \longleftrightarrow r^{-1-1} \text{ OO } r \leq r^{**} \text{ OO } (r^{-1-1})^{==}$

lemma *semiconfluentpI* [intro?]:

semiconfluentp r **if** $\bigwedge x y z. \llbracket r x y; r^{**} x z \rrbracket \Longrightarrow \exists u. r^{**} y u \wedge r^{**} z u$
 ⟨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** $\bigwedge x y z. \llbracket r^{**} x y; r^{**} x z \rrbracket \Longrightarrow \exists u. r^{**} y u \wedge r^{**} z u$
 ⟨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** $\bigwedge x y z. \llbracket r x y; r x z \rrbracket \Longrightarrow \exists u. r^{**} y u \wedge r^{==} z u$
 ⟨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* $r \longleftrightarrow \text{semiconfluentp } r$
 ⟨proof⟩

lemma *confluentp-conv-strong-confluentp-rtranclp*:

confluentp $r \longleftrightarrow \text{strong-confluentp } (r^{**})$
 ⟨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* $r = r^{**} \text{ OO } r^{-1-1**}$ **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 $\forall A \in \mathcal{A}. \text{finite } A$
shows $\exists \mathcal{B} \subseteq \mathcal{A}. \text{finite } \mathcal{B} \wedge (\bigcap \mathcal{B}) = (\bigcap \mathcal{A})$
 $\langle \text{proof} \rangle$

locale *wide-intersection-finite* =
fixes $E :: 'Fa \Rightarrow 'Fa \Rightarrow \text{bool}$
and $\text{mapFa} :: ('a \Rightarrow 'a) \Rightarrow 'Fa \Rightarrow 'Fa$
and $\text{setFa} :: 'Fa \Rightarrow 'a \text{ set}$
assumes *equiv*: $\text{equivp } E$
and *map-E*: $E \ x \ y \implies E \ (\text{mapFa } f \ x) \ (\text{mapFa } f \ y)$
and *map-id*: $\text{mapFa } \text{id} \ x = x$
and *map-cong*: $\forall a \in \text{setFa } x. f \ a = g \ a \implies \text{mapFa } f \ x = \text{mapFa } g \ x$
and *set-map*: $\text{setFa } (\text{mapFa } f \ x) = f \ ` \ \text{setFa } x$
and *finite*: $\text{finite } (\text{setFa } x)$
begin

lemma *binary-intersection*:
assumes $E \ y \ z$ **and** $y: \text{setFa } y \subseteq Y$ **and** $z: \text{setFa } z \subseteq Z$ **and** $a: a \in Y \ a \in Z$
shows $\exists x. E \ x \ y \wedge \text{setFa } x \subseteq Y \wedge \text{setFa } x \subseteq Z$
 $\langle \text{proof} \rangle$

lemma *finite-intersection*:
assumes $E: \forall y \in A. E \ y \ z$
and *fin*: $\text{finite } A$
and *sub*: $\forall y \in A. \text{setFa } y \subseteq Y \ y \wedge a \in Y \ y$
shows $\exists x. E \ x \ z \wedge (\forall y \in A. \text{setFa } x \subseteq Y \ y)$
 $\langle \text{proof} \rangle$

lemma *wide-intersection*:
assumes *inter-nonempty*: $\bigcap Ss \neq \{\}$
shows $(\bigcap As \in Ss. \{(x, x'). E \ x \ x'\} \text{ “ } \{x. \text{setFa } x \subseteq As\} \subseteq \{(x, x'). E \ x \ x'\} \text{ “ } \{x. \text{setFa } x \subseteq \bigcap Ss\} \text{ (is ?lhs } \subseteq \text{ ?rhs)})$
 $\langle \text{proof} \rangle$

end

Subdistributivity for quotients via confluence

lemma *rtrancp-transp-reflp*: $R^{**} = R$ **if** $\text{transp } R \ \text{reflp } R$
 $\langle \text{proof} \rangle$

lemma *rtrancp-equivp*: $R^{**} = R$ **if** $\text{equivp } R$

$\langle proof \rangle$

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 $\pi\text{-Faba} :: 'Fab \Rightarrow 'Fa$
and $\pi\text{-Fabb} :: 'Fab \Rightarrow 'Fb$
and $\pi\text{-Fbcb} :: 'Fbc \Rightarrow 'Fb$
and $\pi\text{-Fbcc} :: 'Fbc \Rightarrow 'Fc$
and $rel\text{-Fab} :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'Fa \Rightarrow 'Fb \Rightarrow bool$
and $rel\text{-Fbc} :: ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'Fb \Rightarrow 'Fc \Rightarrow bool$
and $rel\text{-Fac} :: ('a \Rightarrow 'c \Rightarrow bool) \Rightarrow 'Fa \Rightarrow 'Fc \Rightarrow bool$
and $set\text{-Fab} :: 'Fab \Rightarrow ('a \times 'b) \text{ set}$
and $set\text{-Fbc} :: 'Fbc \Rightarrow ('b \times 'c) \text{ set}$
assumes *confluent*: *confluentp* Rb
and *retract1-ab*: $\bigwedge x y. Rb (\pi\text{-Fabb } x) y \implies \exists z. Eab x z \wedge y = \pi\text{-Fabb } z \wedge$
set-Fab $z \subseteq set\text{-Fab } x$
and *retract1-bc*: $\bigwedge x y. Rb (\pi\text{-Fbcb } x) y \implies \exists z. Ebc x z \wedge y = \pi\text{-Fbcb } z \wedge$
set-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-comp*: $\bigwedge A B. rel\text{-Fac } (A \text{ OO } B) = rel\text{-Fab } A \text{ OO } rel\text{-Fbc } B$
and $\pi\text{-Faba-respect}$: *rel-fun* $Eab Ea \pi\text{-Faba } \pi\text{-Faba}$
and $\pi\text{-Fbcc-respect}$: *rel-fun* $Ebc Ec \pi\text{-Fbcc } \pi\text{-Fbcc}$
begin

lemma *retract-ab*: $Rb^{**} (\pi\text{-Fabb } x) y \implies \exists z. Eab x z \wedge y = \pi\text{-Fabb } z \wedge set\text{-Fab}$
 $z \subseteq set\text{-Fab } x$
 $\langle proof \rangle$

lemma *retract-bc*: $Rb^{**} (\pi\text{-Fbcb } x) y \implies \exists z. Ebc x z \wedge y = \pi\text{-Fbcb } z \wedge set\text{-Fbc } z$
 $\subseteq set\text{-Fbc } x$
 $\langle proof \rangle$

lemma *subdistributivity*: $rel\text{-Fab } A \text{ OO } Eb \text{ OO } rel\text{-Fbc } B \leq Ea \text{ OO } rel\text{-Fac } (A \text{ OO } B) \text{ OO } Ec$
 $\langle proof \rangle$

end

end

17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

```
theory Old-Datatype
imports Main
begin
```

17.1 The datatype universe

definition $Node = \{p. \exists f x k. p = (f :: nat \Rightarrow 'b + nat, x :: 'a + nat) \wedge f k = Inr\ 0\}$

```
typedef ('a, 'b) node = Node :: ((nat => '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 $Push :: [('b + nat), nat \Rightarrow ('b + nat)] \Rightarrow (nat \Rightarrow ('b + nat))$

where $Push == (\%b\ h. case\ nat\ b\ h)$

definition $Push\text{-}Node :: [('b + nat), ('a, 'b)\ node] \Rightarrow ('a, 'b)\ node$
 where $Push\text{-}Node == (\%n\ x. Abs\text{-}Node\ (apfst\ (Push\ n)\ (Rep\text{-}Node\ x)))$

definition $Atom :: ('a + nat) \Rightarrow ('a, 'b)\ dtree$

where $Atom == (\%x. \{Abs\text{-}Node((\%k. Inr\ 0, x))\})$

definition $Scons :: [('a, 'b)\ dtree, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$

where $Scons\ M\ N == (Push\text{-}Node\ (Inr\ 1)\ 'M)\ Un\ (Push\text{-}Node\ (Inr\ (Suc\ 1))\ 'N)$

definition $Leaf :: 'a \Rightarrow ('a, 'b)\ dtree$

where $Leaf == Atom \circ Inl$

definition $Numb :: nat \Rightarrow ('a, 'b)\ dtree$

where $Numb == Atom \circ Inr$

definition $In0 :: ('a, 'b) dtree \Rightarrow ('a, 'b) dtree$

where $In0(M) == Scons (Numb 0) M$

definition $In1 :: ('a, 'b) dtree \Rightarrow ('a, 'b) dtree$

where $In1(M) == Scons (Numb 1) M$

definition $Lim :: ('b \Rightarrow ('a, 'b) dtree) \Rightarrow ('a, 'b) dtree$

where $Lim f == \bigcup \{z. \exists x. z = Push\text{-}Node (Inl x) ' (f x)\}$

definition $ndepth :: ('a, 'b) node \Rightarrow nat$

where $ndepth(n) == (\% (f, x). LEAST k. f k = Inr 0) (Rep\text{-}Node n)$

definition $ntrunc :: [nat, ('a, 'b) dtree] \Rightarrow ('a, 'b) dtree$

where $ntrunc k N == \{n. n \in N \wedge ndepth(n) < k\}$

definition $uprod :: [(('a, 'b) dtree set, ('a, 'b) dtree set)] \Rightarrow ('a, 'b) dtree set$

where $uprod A B == UN x:A. UN y:B. \{ Scons x y \}$

definition $usum :: [(('a, 'b) dtree set, ('a, 'b) dtree set)] \Rightarrow ('a, 'b) dtree set$

where $usum A B == In0'A Un In1'B$

definition $Split :: [(('a, 'b) dtree, ('a, 'b) dtree)] \Rightarrow 'c, ('a, 'b) dtree] \Rightarrow 'c$

where $Split c M == THE u. \exists x y. M = Scons x y \wedge u = c x y$

definition $Case :: [(('a, 'b) dtree)] \Rightarrow 'c, [(('a, 'b) dtree)] \Rightarrow 'c, ('a, 'b) dtree] \Rightarrow 'c$

where $Case c d M == THE u. (\exists x. M = In0(x) \wedge u = c(x)) \vee (\exists y. M = In1(y) \wedge u = d(y))$

definition $dprod :: [(('a, 'b) dtree * ('a, 'b) dtree)set, (('a, 'b) dtree * ('a, 'b) dtree)set]$

$\Rightarrow (('a, 'b) dtree * ('a, 'b) dtree)set$

where $dprod r s == UN (x, x'):r. UN (y, y'):s. \{(Scons x y, Scons x' y')\}$

definition $dsum :: [(('a, 'b) dtree * ('a, 'b) dtree)set, (('a, 'b) dtree * ('a, 'b) dtree)set]$

$\Rightarrow (('a, 'b) dtree * ('a, 'b) dtree)set$

where $dsum r s == (UN (x, x'):r. \{(In0(x), In0(x'))\}) Un (UN (y, y'):s. \{(In1(y), In1(y'))\})$

lemma $apfst\text{-}convE$:

$[| q = apfst f p; !!x y. [| p = (x, y); q = (f(x), y) |] ==> R$

$|] ==> R$

$\langle proof \rangle$

lemma *Push-inject1*: $\text{Push } i \ f = \text{Push } j \ g \implies i=j$
 $\langle \text{proof} \rangle$

lemma *Push-inject2*: $\text{Push } i \ f = \text{Push } j \ g \implies f=g$
 $\langle \text{proof} \rangle$

lemma *Push-inject*:
 $\llbracket \text{Push } i \ f = \text{Push } j \ g; \llbracket i=j; \ f=g \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *Push-neq-K0*: $\text{Push } (\text{Inr } (\text{Suc } k)) \ f = (\%z. \text{Inr } 0) \implies P$
 $\langle \text{proof} \rangle$

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

lemma *Node-K0-I*: $(\lambda k. \text{Inr } 0, a) \in \text{Node}$
 $\langle \text{proof} \rangle$

lemma *Node-Push-I*: $p \in \text{Node} \implies \text{apfst } (\text{Push } i) \ p \in \text{Node}$
 $\langle \text{proof} \rangle$

17.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [iff]: $\text{Scons } M \ N \neq \text{Atom}(a)$
 $\langle \text{proof} \rangle$

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

lemma *inj-Atom*: $\text{inj}(\text{Atom})$
 $\langle \text{proof} \rangle$

lemmas *Atom-inject* = *inj-Atom* [THEN injD]

lemma *Atom-Atom-eq* [iff]: $(\text{Atom}(a) = \text{Atom}(b)) = (a=b)$
 $\langle \text{proof} \rangle$

lemma *inj-Leaf*: $\text{inj}(\text{Leaf})$
 $\langle \text{proof} \rangle$

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

lemma *inj-Numb*: $\text{inj}(\text{Numb})$
 $\langle \text{proof} \rangle$

lemmas *Numb-inject* $[\text{dest!}] = \text{inj-Numb} [\text{THEN injD}]$

lemma *Push-Node-inject*:

$$[[\text{Push-Node } i \ m = \text{Push-Node } j \ n; \ [[i=j; \ m=n]] ==> P]] ==> P$$
 $\langle \text{proof} \rangle$

lemma *Scons-inject-lemma1*: $\text{Scons } M \ N <= \text{Scons } M' \ N' ==> M <= M'$
 $\langle \text{proof} \rangle$

lemma *Scons-inject-lemma2*: $\text{Scons } M \ N <= \text{Scons } M' \ N' ==> N <= N'$
 $\langle \text{proof} \rangle$

lemma *Scons-inject1*: $\text{Scons } M \ N = \text{Scons } M' \ N' ==> M = M'$
 $\langle \text{proof} \rangle$

lemma *Scons-inject2*: $\text{Scons } M \ N = \text{Scons } M' \ N' ==> N = N'$
 $\langle \text{proof} \rangle$

lemma *Scons-inject*:

$$[[\text{Scons } M \ N = \text{Scons } M' \ N'; \ [[M=M'; \ N=N']] ==> P]] ==> P$$
 $\langle \text{proof} \rangle$

lemma *Scons-Scons-eq* $[\text{iff}]$: $(\text{Scons } M \ N = \text{Scons } M' \ N') = (M=M' \wedge N=N')$
 $\langle \text{proof} \rangle$

lemma *Scons-not-Leaf* $[\text{iff}]$: $\text{Scons } M \ N \neq \text{Leaf}(a)$
 $\langle \text{proof} \rangle$

lemmas *Leaf-not-Scons* $[\text{iff}] = \text{Scons-not-Leaf} [\text{THEN not-sym}]$

lemma *Scons-not-Numb* $[\text{iff}]$: $\text{Scons } M \ N \neq \text{Numb}(k)$
 $\langle \text{proof} \rangle$

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

lemma *Leaf-not-Numb* [iff]: $\text{Leaf}(a) \neq \text{Numb}(k)$
 $\langle \text{proof} \rangle$

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

lemma *ndepth-K0*: $\text{ndepth} (\text{Abs-Node}(\%k. \text{Inr } 0, x)) = 0$
 $\langle \text{proof} \rangle$

lemma *ndepth-Push-Node-aux*:
 $\text{case-nat} (\text{Inr} (\text{Suc } i)) f k = \text{Inr } 0 \longrightarrow \text{Suc}(\text{LEAST } x. f x = \text{Inr } 0) \leq k$
 $\langle \text{proof} \rangle$

lemma *ndepth-Push-Node*:
 $\text{ndepth} (\text{Push-Node} (\text{Inr} (\text{Suc } i)) n) = \text{Suc}(\text{ndepth}(n))$
 $\langle \text{proof} \rangle$

lemma *ntrunc-0* [simp]: $\text{ntrunc } 0 M = \{\}$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Atom* [simp]: $\text{ntrunc} (\text{Suc } k) (\text{Atom } a) = \text{Atom}(a)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Leaf* [simp]: $\text{ntrunc} (\text{Suc } k) (\text{Leaf } a) = \text{Leaf}(a)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Numb* [simp]: $\text{ntrunc} (\text{Suc } k) (\text{Numb } i) = \text{Numb}(i)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Scons* [simp]:
 $\text{ntrunc} (\text{Suc } k) (\text{Scons } M N) = \text{Scons} (\text{ntrunc } k M) (\text{ntrunc } k N)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-one-In0* [simp]: $\text{ntrunc} (\text{Suc } 0) (\text{In0 } M) = \{\}$
 $\langle \text{proof} \rangle$

lemma *ntrunc-In0* [simp]: $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In0 } M) = \text{In0 } (\text{ntrunc } (\text{Suc } k) M)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-one-In1* [simp]: $\text{ntrunc } (\text{Suc } 0) (\text{In1 } M) = \{\}$
 $\langle \text{proof} \rangle$

lemma *ntrunc-In1* [simp]: $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In1 } M) = \text{In1 } (\text{ntrunc } (\text{Suc } k) M)$
 $\langle \text{proof} \rangle$

17.3 Set Constructions

lemma *uprodI* [intro!]: $\llbracket M \in A; N \in B \rrbracket \implies \text{Scons } M N \in \text{uprod } A B$
 $\langle \text{proof} \rangle$

lemma *uprodE* [elim!]:
 $\llbracket c \in \text{uprod } A B;$
 $\quad \bigwedge x y. \llbracket x \in A; y \in B; c = \text{Scons } x y \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *uprodE2*: $\llbracket \text{Scons } M N \in \text{uprod } A B; \llbracket M \in A; N \in B \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *usum-In0I* [intro]: $M \in A \implies \text{In0}(M) \in \text{usum } A B$
 $\langle \text{proof} \rangle$

lemma *usum-In1I* [intro]: $N \in B \implies \text{In1}(N) \in \text{usum } A B$
 $\langle \text{proof} \rangle$

lemma *usumE* [elim!]:
 $\llbracket u \in \text{usum } A B;$
 $\quad \bigwedge x. \llbracket x \in A; u = \text{In0}(x) \rrbracket \implies P;$
 $\quad \bigwedge y. \llbracket y \in B; u = \text{In1}(y) \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *In0-not-In1* [iff]: $\text{In0}(M) \neq \text{In1}(N)$

$\langle proof \rangle$

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

lemma *In0-inject*: $In0(M) = In0(N) ==> M=N$
 $\langle proof \rangle$

lemma *In1-inject*: $In1(M) = In1(N) ==> M=N$
 $\langle proof \rangle$

lemma *In0-eq* [iff]: $(In0\ M = In0\ N) = (M=N)$
 $\langle proof \rangle$

lemma *In1-eq* [iff]: $(In1\ M = In1\ N) = (M=N)$
 $\langle proof \rangle$

lemma *inj-In0*: $inj\ In0$
 $\langle proof \rangle$

lemma *inj-In1*: $inj\ In1$
 $\langle proof \rangle$

lemma *Lim-inject*: $Lim\ f = Lim\ g ==> f = g$
 $\langle proof \rangle$

lemma *ntrunc-subsetI*: $ntrunc\ k\ M <= M$
 $\langle proof \rangle$

lemma *ntrunc-subsetD*: $(!!k. ntrunc\ k\ M <= N) ==> M <= N$
 $\langle proof \rangle$

lemma *ntrunc-equality*: $(!!k. ntrunc\ k\ M = ntrunc\ k\ N) ==> M=N$
 $\langle proof \rangle$

lemma *ntrunc-o-equality*:
 $[!k. (ntrunc(k) \circ h1) = (ntrunc(k) \circ h2)] ==> h1=h2$
 $\langle proof \rangle$

lemma *uprod-mono*: $[! A <= A'; B <= B'] ==> uprod\ A\ B <= uprod\ A'\ B'$

$\langle proof \rangle$

lemma *usum-mono*: $[| A \leq A'; B \leq B' |] ==> usum A B \leq usum A' B'$
 $\langle proof \rangle$

lemma *Scons-mono*: $[| M \leq M'; N \leq N' |] ==> Scons M N \leq Scons M' N'$
 $\langle proof \rangle$

lemma *In0-mono*: $M \leq N ==> In0(M) \leq In0(N)$
 $\langle proof \rangle$

lemma *In1-mono*: $M \leq N ==> In1(M) \leq In1(N)$
 $\langle proof \rangle$

lemma *Split [simp]*: $Split\ c\ (Scons\ M\ N) = c\ M\ N$
 $\langle proof \rangle$

lemma *Case-In0 [simp]*: $Case\ c\ d\ (In0\ M) = c(M)$
 $\langle proof \rangle$

lemma *Case-In1 [simp]*: $Case\ c\ d\ (In1\ N) = d(N)$
 $\langle proof \rangle$

lemma *ntrunc-UN1*: $ntrunc\ k\ (UN\ x.\ f(x)) = (UN\ x.\ ntrunc\ k\ (f\ x))$
 $\langle proof \rangle$

lemma *Scons-UN1-x*: $Scons\ (UN\ x.\ f\ x)\ M = (UN\ x.\ Scons\ (f\ x)\ M)$
 $\langle proof \rangle$

lemma *Scons-UN1-y*: $Scons\ M\ (UN\ x.\ f\ x) = (UN\ x.\ Scons\ M\ (f\ x))$
 $\langle proof \rangle$

lemma *In0-UN1*: $In0\ (UN\ x.\ f(x)) = (UN\ x.\ In0(f(x)))$
 $\langle proof \rangle$

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

lemma *dprodI [intro!]*:

$\llbracket (M, M') \in r; (N, N') \in s \rrbracket \implies (Scons\ M\ N, Scons\ M'\ N') \in dprod\ r\ s$
 $\langle proof \rangle$

lemma *dprodE* [*elim!*]:

$\llbracket c \in dprod\ r\ s;$
 $\bigwedge x\ y\ x'\ y'. \llbracket (x, x') \in r; (y, y') \in s;$
 $c = (Scons\ x\ y, Scons\ x'\ y') \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

lemma *dsum-In0I* [*intro*]: $(M, M') \in r \implies (In0(M), In0(M')) \in dsum\ r\ s$
 $\langle proof \rangle$

lemma *dsum-In1I* [*intro*]: $(N, N') \in s \implies (In1(N), In1(N')) \in dsum\ r\ s$
 $\langle proof \rangle$

lemma *dsumE* [*elim!*]:

$\llbracket w \in dsum\ r\ s;$
 $\bigwedge x\ x'. \llbracket (x, x') \in r; w = (In0(x), In0(x')) \rrbracket \implies P;$
 $\bigwedge y\ y'. \llbracket (y, y') \in s; w = (In1(y), In1(y')) \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

lemma *dprod-mono*: $\llbracket r \leq r'; s \leq s' \rrbracket \implies dprod\ r\ s \leq dprod\ r'\ s'$
 $\langle proof \rangle$

lemma *dsum-mono*: $\llbracket r \leq r'; s \leq s' \rrbracket \implies dsum\ r\ s \leq dsum\ r'\ s'$
 $\langle proof \rangle$

lemma *dprod-Sigma*: $(dprod\ (A \times B)\ (C \times D)) \leq (uprod\ A\ C) \times (uprod\ B\ D)$
 $\langle proof \rangle$

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

lemma *dprod-subset-Sigma2*:

$(dprod\ (Sigma\ A\ B)\ (Sigma\ C\ D)) \leq Sigma\ (uprod\ A\ C)\ (Split\ (\%x\ y. uprod\ (B\ x)\ (D\ y)))$
 $\langle proof \rangle$

lemma *dsum-Sigma*: $(dsum\ (A \times B)\ (C \times D)) \leq (usum\ A\ C) \times (usum\ B\ D)$
 $\langle proof \rangle$

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

lemma *Domain-dprod* [*simp*]: $Domain\ (dprod\ r\ s) = uprod\ (Domain\ r)\ (Domain\ s)$
 $\langle proof \rangle$

lemma *Domain-dsum* [*simp*]: $Domain\ (dsum\ r\ s) = usum\ (Domain\ r)\ (Domain\ s)$
 $\langle proof \rangle$

hides popular names

hide-type (**open**) *node item*

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

$\langle ML \rangle$

end

18 Bijections between natural numbers and other types

theory *Nat-Bijection*

imports *Main*

begin

18.1 Type $nat \times nat$

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition *triangle* :: $nat \Rightarrow nat$
where $triangle\ n = (n * Suc\ n) \div 2$

lemma *triangle-0* [*simp*]: $triangle\ 0 = 0$
 $\langle proof \rangle$

lemma *triangle-Suc* [*simp*]: $triangle\ (Suc\ n) = triangle\ n + Suc\ n$
 $\langle proof \rangle$

definition *prod-encode* :: $nat \times nat \Rightarrow nat$
where $prod-encode = (\lambda(m, n). triangle\ (m + n) + m)$

In this auxiliary function, $triangle\ k + m$ is an invariant.

```

fun prod-decode-aux :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\times$  nat
  where prod-decode-aux k m =
    (if m  $\leq$  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  $\Rightarrow$  nat  $\times$  nat
  where prod-decode = prod-decode-aux 0

lemma prod-encode-prod-decode-aux: prod-encode (prod-decode-aux k m) = triangle
  k + m
  <proof>

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: prod-encode x = prod-encode y  $\longleftrightarrow$  x = y
  <proof>

lemma prod-decode-eq: prod-decode x = prod-decode y  $\longleftrightarrow$  x = y
  <proof>

```

Ordering properties

lemma *le-prod-encode-1*: $a \leq \text{prod-encode } (a, b)$
 $\langle \text{proof} \rangle$

lemma *le-prod-encode-2*: $b \leq \text{prod-encode } (a, b)$
 $\langle \text{proof} \rangle$

18.2 Type $\text{nat} + \text{nat}$

definition *sum-encode* :: $\text{nat} + \text{nat} \Rightarrow \text{nat}$
where *sum-encode* $x = (\text{case } x \text{ of } \text{Inl } a \Rightarrow 2 * a \mid \text{Inr } b \Rightarrow \text{Suc } (2 * b))$

definition *sum-decode* :: $\text{nat} \Rightarrow \text{nat} + \text{nat}$
where *sum-decode* $n = (\text{if even } n \text{ then } \text{Inl } (n \text{ div } 2) \text{ else } \text{Inr } (n \text{ div } 2))$

lemma *sum-encode-inverse* [*simp*]: $\text{sum-decode } (\text{sum-encode } x) = x$
 $\langle \text{proof} \rangle$

lemma *sum-decode-inverse* [*simp*]: $\text{sum-encode } (\text{sum-decode } n) = n$
 $\langle \text{proof} \rangle$

lemma *inj-sum-encode*: *inj-on* *sum-encode* A
 $\langle \text{proof} \rangle$

lemma *inj-sum-decode*: *inj-on* *sum-decode* A
 $\langle \text{proof} \rangle$

lemma *surj-sum-encode*: *surj* *sum-encode*
 $\langle \text{proof} \rangle$

lemma *surj-sum-decode*: *surj* *sum-decode*
 $\langle \text{proof} \rangle$

lemma *bij-sum-encode*: *bij* *sum-encode*
 $\langle \text{proof} \rangle$

lemma *bij-sum-decode*: *bij* *sum-decode*
 $\langle \text{proof} \rangle$

lemma *sum-encode-eq*: $\text{sum-encode } x = \text{sum-encode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *sum-decode-eq*: $\text{sum-decode } x = \text{sum-decode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

18.3 Type int

definition *int-encode* :: $\text{int} \Rightarrow \text{nat}$
where *int-encode* $i = \text{sum-encode } (\text{if } 0 \leq i \text{ then } \text{Inl } (\text{nat } i) \text{ else } \text{Inr } (\text{nat } (-i - 1)))$

definition *int-decode* :: *nat* $\Rightarrow$ *int*

where *int-decode* *n* = (case *sum-decode* *n* of *Inl* *a* $\Rightarrow$ *int* *a* | *Inr* *b* $\Rightarrow$ - *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* $\longleftrightarrow$ *x* = *y*
 ⟨*proof*⟩

lemma *int-decode-eq*: *int-decode* *x* = *int-decode* *y* $\longleftrightarrow$ *x* = *y*
 ⟨*proof*⟩

18.4 Type *nat list*

fun *list-encode* :: *nat list* $\Rightarrow$ *nat*

where

list-encode [] = 0

| *list-encode* (*x* # *xs*) = *Suc* (*prod-encode* (*x*, *list-encode* *xs*))

function *list-decode* :: *nat* $\Rightarrow$ *nat list*

where

list-decode 0 = []

| *list-decode* (*Suc* *n*) = (case *prod-decode* *n* of (*x*, *y*) $\Rightarrow$ *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 $\longleftrightarrow$ x = y*
 ⟨proof⟩

lemma *list-decode-eq: list-decode x = list-decode y $\longleftrightarrow$ x = y*
 ⟨proof⟩

18.5 Finite sets of naturals

18.5.1 Preliminaries

lemma *finite-vimage-Suc-iff: finite (Suc -‘ F) $\longleftrightarrow$ 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 $\longleftrightarrow$ even y*
 shows *x = y*
 ⟨proof⟩

18.5.2 From sets to naturals

definition *set-encode* :: *nat set* $\Rightarrow$ *nat*
where *set-encode* = *sum* (($\cap$) 2)

lemma *set-encode-empty* [*simp*]: *set-encode* {} = 0
 ⟨*proof*⟩

lemma *set-encode-inf*: $\neg \text{finite } A \implies \text{set-encode } A = 0$
 ⟨*proof*⟩

lemma *set-encode-insert* [*simp*]: *finite A* $\implies n \notin A \implies \text{set-encode } (\text{insert } n \ A)$
 $= 2^{\wedge} n + \text{set-encode } A$
 ⟨*proof*⟩

lemma *even-set-encode-iff*: *finite A* $\implies \text{even } (\text{set-encode } A) \longleftrightarrow 0 \notin A$
 ⟨*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* $\Rightarrow$ *nat set*
where *set-decode* *x* = {*n*. *odd* (*x div 2* $\wedge$ *n*)}

lemma *set-decode-0* [*simp*]: $0 \in \text{set-decode } x \longleftrightarrow \text{odd } x$
 ⟨*proof*⟩

lemma *set-decode-Suc* [*simp*]: *Suc n* $\in \text{set-decode } x \longleftrightarrow n \in \text{set-decode } (x \text{ div } 2)$
 ⟨*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*:
 $n \notin \text{set-decode } z \implies \text{set-decode } (2^{\wedge} n + z) = \text{insert } n \ (\text{set-decode } z)$
 ⟨*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*

$\langle \text{proof} \rangle$

lemma *set-encode-inverse* [simp]: $\text{finite } A \implies \text{set-decode } (\text{set-encode } A) = A$
 $\langle \text{proof} \rangle$

lemma *inj-on-set-encode*: $\text{inj-on set-encode } (\text{Collect finite})$
 $\langle \text{proof} \rangle$

lemma *set-encode-eq*: $\text{finite } A \implies \text{finite } B \implies \text{set-encode } A = \text{set-encode } B \longleftrightarrow A = B$
 $\langle \text{proof} \rangle$

lemma *subset-decode-imp-le*:
 assumes $\text{set-decode } m \subseteq \text{set-decode } n$
 shows $m \leq n$
 $\langle \text{proof} \rangle$

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 $\text{ex-inj}: \exists \text{to-nat} :: 'a \Rightarrow \text{nat}. \text{inj to-nat}$

lemma *countable-classI*:
 fixes $f :: 'a \Rightarrow \text{nat}$
 assumes $\bigwedge x y. f x = f y \implies x = y$
 shows $\text{OFCLASS}('a, \text{countable-class})$
 $\langle \text{proof} \rangle$

19.2 Conversion functions

definition *to-nat* :: $'a::\text{countable} \Rightarrow \text{nat}$ **where**
 $\text{to-nat} = (\text{SOME } f. \text{inj } f)$

definition *from-nat* :: $\text{nat} \Rightarrow 'a::\text{countable}$ **where**
 $\text{from-nat} = \text{inv } (\text{to-nat} :: 'a \Rightarrow \text{nat})$

lemma *inj-to-nat* [simp]: inj to-nat
 $\langle \text{proof} \rangle$

lemma *inj-on-to-nat*[simp, intro]: $\text{inj-on to-nat } S$

$\langle \text{proof} \rangle$

lemma *surj-from-nat* [simp]: *surj from-nat*
 $\langle \text{proof} \rangle$

lemma *to-nat-split* [simp]: *to-nat* $x = \text{to-nat } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *from-nat-to-nat* [simp]:
from-nat (*to-nat* x) = x
 $\langle \text{proof} \rangle$

19.3 Finite types are countable

subclass (in *finite*) *countable*
 $\langle \text{proof} \rangle$

19.4 Automatically proving countability of old-style datatypes

context
begin

qualified inductive *finite-item* :: 'a *Old-Datatype.item* $\Rightarrow$ bool **where**
undefined: *finite-item* *undefined*
| *In0*: *finite-item* $x \Longrightarrow \text{finite-item } (\text{Old-Datatype.In0 } x)$
| *In1*: *finite-item* $x \Longrightarrow \text{finite-item } (\text{Old-Datatype.In1 } x)$
| *Leaf*: *finite-item* (*Old-Datatype.Leaf* a)
| *Scons*: $\llbracket \text{finite-item } x; \text{finite-item } y \rrbracket \Longrightarrow \text{finite-item } (\text{Old-Datatype.Scons } x \ y)$

qualified function *nth-item* :: nat $\Rightarrow$ ('a::countable) *Old-Datatype.item*
where

nth-item 0 = *undefined*
| *nth-item* (*Suc* n) =
(case *sum-decode* n of
 Inl $i \Rightarrow$
 (case *sum-decode* i of
 Inl $j \Rightarrow \text{Old-Datatype.In0 } (\text{nth-item } j)$
 | *Inr* $j \Rightarrow \text{Old-Datatype.In1 } (\text{nth-item } j)$)
 | *Inr* $i \Rightarrow$
 (case *sum-decode* i of
 Inl $j \Rightarrow \text{Old-Datatype.Leaf } (\text{from-nat } j)$
 | *Inr* $j \Rightarrow$
 (case *prod-decode* j of
 $(a, b) \Rightarrow \text{Old-Datatype.Scons } (\text{nth-item } a) (\text{nth-item } b)$)))
 $\langle \text{proof} \rangle$

lemma *le-sum-encode-Inl*: $x \leq y \Longrightarrow x \leq \text{sum-encode } (\text{Inl } y)$
 $\langle \text{proof} \rangle$

lemma *le-sum-encode-Inr*: $x \leq y \Longrightarrow x \leq \text{sum-encode } (\text{Inr } y)$

$\langle proof \rangle$ **termination**
 $\langle proof \rangle$

lemma *nth-item-covers*: $finite-item\ x \implies \exists n. nth-item\ n = x$
 $\langle proof \rangle$

theorem *countable-datatype*:
fixes *Rep* :: 'b $\Rightarrow$ ('a::countable) *Old-Datatype.item*
fixes *Abs* :: ('a::countable) *Old-Datatype.item* $\Rightarrow$ 'b
fixes *rep-set* :: ('a::countable) *Old-Datatype.item* $\Rightarrow$ bool
assumes *type*: *type-definition* *Rep Abs* (*Collect rep-set*)
assumes *finite-item*: $\bigwedge x. rep-set\ x \implies finite-item\ x$
shows *OFCLASS*('b, *countable-class*)
 $\langle proof \rangle$

$\langle ML \rangle$

end

19.5 Automatically proving countability of datatypes

$\langle ML \rangle$

19.6 More Countable types

Naturals

instance *nat* :: *countable*
 $\langle proof \rangle$

Pairs

instance *prod* :: (*countable*, *countable*) *countable*
 $\langle proof \rangle$

Sums

instance *sum* :: (*countable*, *countable*) *countable*
 $\langle proof \rangle$

Integers

instance *int* :: *countable*
 $\langle proof \rangle$

Options

instance *option* :: (*countable*) *countable*
 $\langle proof \rangle$

Lists

instance *list* :: (*countable*) *countable*
 $\langle proof \rangle$

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

context *field-char-0*
begin

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

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

end

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

theorem *rat-denum*: $\exists f :: \text{nat} \Rightarrow \text{rat}. \text{surj } f$
 ⟨*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:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n \geq m. n \in S)$ 
  for  $S :: \text{nat set}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma infinite-nat-iff-unbounded:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n > m. n \in S)$ 
  for  $S :: \text{nat set}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma finite-nat-iff-bounded:  $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{..<k\})$ 
  for  $S :: \text{nat set}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma finite-nat-iff-bounded-le:  $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{.. k\})$ 
  for  $S :: \text{nat set}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma finite-nat-bounded:  $\text{finite } S \implies \exists k. S \subseteq \{..<k\}$ 
  for  $S :: \text{nat set}$ 
   $\langle \text{proof} \rangle$ 

```

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:  $\forall m > k. \exists n > m. n \in S \implies \text{infinite } (S :: \text{nat set})$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma nat-not-finite:  $\text{finite } (\text{UNIV} :: \text{nat set}) \implies \text{False}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma range-inj-infinite:
  fixes  $f :: \text{nat} \Rightarrow 'a$ 
  assumes inj  $f$ 
  shows infinite  $(\text{range } f)$ 
   $\langle \text{proof} \rangle$ 

```

20.2 The set of integers is also infinite

```

lemma infinite-int-iff-infinite-nat-abs:  $\text{infinite } S \longleftrightarrow \text{infinite } ((\text{nat} \circ \text{abs}) ` S)$ 
  for  $S :: \text{int set}$ 
   $\langle \text{proof} \rangle$ 

```

```

proposition infinite-int-iff-unbounded-le:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| \geq m \wedge n \in S)$ 
  for  $S :: \text{int set}$ 

```

$\langle \text{proof} \rangle$

proposition *infinite-int-iff-unbounded*: $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| > m \wedge n \in S)$

for $S :: \text{int set}$

$\langle \text{proof} \rangle$

proposition *finite-int-iff-bounded*: $\text{finite } S \longleftrightarrow (\exists k. \text{abs } S \subseteq \{..<k\})$

for $S :: \text{int set}$

$\langle \text{proof} \rangle$

proposition *finite-int-iff-bounded-le*: $\text{finite } S \longleftrightarrow (\exists k. \text{abs } S \subseteq \{.. k\})$

for $S :: \text{int set}$

$\langle \text{proof} \rangle$

20.3 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM* [simp]: $\neg (\text{INFM } x. P \ x) \longleftrightarrow (\text{MOST } x. \neg P \ x)$

$\langle \text{proof} \rangle$

lemma *not-MOST* [simp]: $\neg (\text{MOST } x. P \ x) \longleftrightarrow (\text{INFM } x. \neg P \ x)$

$\langle \text{proof} \rangle$

lemma *INFM-const* [simp]: $(\text{INFM } x::'a. P) \longleftrightarrow P \wedge \text{infinite } (\text{UNIV}::'a \text{ set})$

$\langle \text{proof} \rangle$

lemma *MOST-const* [simp]: $(\text{MOST } x::'a. P) \longleftrightarrow P \vee \text{finite } (\text{UNIV}::'a \text{ set})$

$\langle \text{proof} \rangle$

lemma *INFM-imp-distrib*: $(\text{INFM } x. P \ x \longrightarrow Q \ x) \longleftrightarrow ((\text{MOST } x. P \ x) \longrightarrow (\text{INFM } x. Q \ x))$

$\langle \text{proof} \rangle$

lemma *MOST-imp-iff*: $\text{MOST } x. P \ x \Longrightarrow (\text{MOST } x. P \ x \longrightarrow Q \ x) \longleftrightarrow (\text{MOST } x. Q \ x)$

$\langle \text{proof} \rangle$

lemma *INFM-conjI*: $\text{INFM } x. P \ x \Longrightarrow \text{MOST } x. Q \ x \Longrightarrow \text{INFM } x. P \ x \wedge Q \ x$

$\langle \text{proof} \rangle$

Properties of quantifiers with injective functions.

lemma *INFM-inj*: $\text{INFM } x. P \ (f \ x) \Longrightarrow \text{inj } f \Longrightarrow \text{INFM } x. P \ x$

$\langle \text{proof} \rangle$

lemma *MOST-inj*: $\text{MOST } x. P \ x \Longrightarrow \text{inj } f \Longrightarrow \text{MOST } x. P \ (f \ x)$

$\langle \text{proof} \rangle$

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [simp]:

$\neg (\text{INFM } x. x = a)$
 $\neg (\text{INFM } x. a = x)$
 $\langle \text{proof} \rangle$

lemma *MOST-neq* [simp]:

$\text{MOST } x. x \neq a$
 $\text{MOST } x. a \neq x$
 $\langle \text{proof} \rangle$

lemma *INFM-neq* [simp]:

$(\text{INFM } x::'a. x \neq a) \longleftrightarrow \text{infinite } (\text{UNIV}::'a \text{ set})$
 $(\text{INFM } x::'a. a \neq x) \longleftrightarrow \text{infinite } (\text{UNIV}::'a \text{ set})$
 $\langle \text{proof} \rangle$

lemma *MOST-eq* [simp]:

$(\text{MOST } x::'a. x = a) \longleftrightarrow \text{finite } (\text{UNIV}::'a \text{ set})$
 $(\text{MOST } x::'a. a = x) \longleftrightarrow \text{finite } (\text{UNIV}::'a \text{ set})$
 $\langle \text{proof} \rangle$

lemma *MOST-eq-imp*:

$\text{MOST } x. x = a \longrightarrow P x$
 $\text{MOST } x. a = x \longrightarrow P x$
 $\langle \text{proof} \rangle$

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_{\infty} n. P n) \longleftrightarrow (\exists m. \forall n > m. P n)$

for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

lemma *MOST-nat-le*: $(\forall_{\infty} n. P n) \longleftrightarrow (\exists m. \forall n \geq m. P n)$

for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

lemma *INFM-nat*: $(\exists_{\infty} n. P n) \longleftrightarrow (\forall m. \exists n > m. P n)$

for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

lemma *INFM-nat-le*: $(\exists_{\infty} n. P n) \longleftrightarrow (\forall m. \exists n \geq m. P n)$

for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

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

$x::'a. P x$
 $\langle \text{proof} \rangle$

lemma *MOST-Suc-iff*: $(\text{MOST } n. P (\text{Suc } n)) \longleftrightarrow (\text{MOST } n. P n)$

$\langle \text{proof} \rangle$

lemma *MOST-SucI*: $MOST\ n.\ P\ n \implies MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \implies MOST\ n.\ P\ n$
 $\langle proof \rangle$

lemma *MOST-ge-nat*: $MOST\ n::nat.\ m \leq n$
 $\langle proof \rangle$

lemma *Inf-many-def*: $Inf\ many\ P \longleftrightarrow infinite\ \{x.\ P\ x\}$ $\langle proof \rangle$

lemma *Alm-all-def*: $Alm\ all\ P \longleftrightarrow \neg (INFM\ x.\ \neg P\ x)$ $\langle proof \rangle$

lemma *INFM-iff-infinite*: $(INFM\ x.\ P\ x) \longleftrightarrow infinite\ \{x.\ P\ x\}$ $\langle proof \rangle$

lemma *MOST-iff-cofinite*: $(MOST\ x.\ P\ x) \longleftrightarrow finite\ \{x.\ \neg P\ x\}$ $\langle proof \rangle$

lemma *INFM-EX*: $(\exists_{\infty} x.\ P\ x) \implies (\exists x.\ P\ x)$ $\langle proof \rangle$

lemma *ALL-MOST*: $\forall x.\ P\ x \implies \forall_{\infty} x.\ P\ x$ $\langle proof \rangle$

lemma *INFM-mono*: $\exists_{\infty} x.\ P\ x \implies (\bigwedge x.\ P\ x \implies Q\ x) \implies \exists_{\infty} x.\ Q\ x$ $\langle proof \rangle$

lemma *MOST-mono*: $\forall_{\infty} x.\ P\ x \implies (\bigwedge x.\ P\ x \implies Q\ x) \implies \forall_{\infty} x.\ Q\ x$ $\langle proof \rangle$

lemma *INFM-disj-distrib*: $(\exists_{\infty} x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_{\infty} x.\ P\ x) \vee (\exists_{\infty} x.\ Q\ x)$
 $\langle proof \rangle$

lemma *MOST-rev-mp*: $\forall_{\infty} x.\ P\ x \implies \forall_{\infty} x.\ P\ x \longrightarrow Q\ x \implies \forall_{\infty} x.\ Q\ x$ $\langle proof \rangle$

lemma *MOST-conj-distrib*: $(\forall_{\infty} x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_{\infty} x.\ P\ x) \wedge (\forall_{\infty} x.\ Q\ x)$
 $\langle proof \rangle$

lemma *MOST-conjI*: $MOST\ x.\ P\ x \implies MOST\ x.\ Q\ x \implies MOST\ x.\ P\ x \wedge Q\ x$
 $\langle proof \rangle$

lemma *INFM-finite-Bex-distrib*: $finite\ A \implies (INFM\ y.\ \exists x \in A.\ P\ x\ y) \longleftrightarrow (\exists x \in A.\ INFM\ y.\ P\ x\ y)$ $\langle proof \rangle$

lemma *MOST-finite-Ball-distrib*: $finite\ A \implies (MOST\ y.\ \forall x \in A.\ P\ x\ y) \longleftrightarrow (\forall x \in A.\ MOST\ y.\ P\ x\ y)$ $\langle proof \rangle$

lemma *INFM-E*: $INFM\ x.\ P\ x \implies (\bigwedge x.\ P\ x \implies thesis) \implies thesis$ $\langle proof \rangle$

lemma *MOST-I*: $(\bigwedge x.\ P\ x) \implies MOST\ x.\ P\ x$ $\langle proof \rangle$

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* :: $'a\ set \Rightarrow nat \Rightarrow 'a$

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'*: $enumerate\ S\ (Suc\ n) = enumerate\ (S - \{enumerate\ S\ 0\})\ n$
 $\langle proof \rangle$

lemma *enumerate-in-set*: $infinite\ S \implies enumerate\ S\ n \in S$
 $\langle proof \rangle$

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

lemma *enumerate-step*: $\text{infinite } S \implies \text{enumerate } S \ n < \text{enumerate } S \ (\text{Suc } n)$
 ⟨proof⟩

lemma *enumerate-mono*: $m < n \implies \text{infinite } S \implies \text{enumerate } S \ m < \text{enumerate } S \ n$
 ⟨proof⟩

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

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

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

lemma *enumerate-Suc''*:
 fixes $S :: 'a::\text{wellorder set}$
 assumes $\text{infinite } S$
 shows $\text{enumerate } S \ (\text{Suc } n) = (\text{LEAST } s. s \in S \wedge \text{enumerate } S \ n < s)$
 ⟨proof⟩

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

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

To generalise this, we’d need a condition that all initial segments were finite

lemma *bij-enumerate*:
 fixes $S :: \text{nat set}$
 assumes S : $\text{infinite } S$
 shows $\text{bij-betw } (\text{enumerate } S) \ \text{UNIV } S$
 ⟨proof⟩

lemma
fixes $S :: \text{nat set}$
assumes $S: \text{infinite } S$
shows $\text{range-enumerate: range (enumerate } S) = S$
and $\text{strict-mono-enumerate: strict-mono (enumerate } S)$
 $\langle \text{proof} \rangle$

A pair of weird and wonderful lemmas from HOL Light.

lemma *finite-transitivity-chain*:
assumes $\text{finite } A$
and $R: \bigwedge x. \neg R x x \bigwedge x y z. \llbracket R x y; R y z \rrbracket \implies R x z$
and $A: \bigwedge x. x \in A \implies \exists y. y \in A \wedge R x y$
shows $A = \{\}$
 $\langle \text{proof} \rangle$

corollary *Union-maximal-sets*:
assumes $\text{finite } \mathcal{F}$
shows $\bigcup \{T \in \mathcal{F}. \forall U \in \mathcal{F}. \neg T \subset U\} = \bigcup \mathcal{F}$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

20.5 Properties of *wellorder-class.enumerate* on finite sets

lemma *finite-enumerate-in-set*: $\llbracket \text{finite } S; n < \text{card } S \rrbracket \implies \text{enumerate } S n \in S$
 $\langle \text{proof} \rangle$

lemma *finite-enumerate-step*: $\llbracket \text{finite } S; \text{Suc } n < \text{card } S \rrbracket \implies \text{enumerate } S n < \text{enumerate } S (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *finite-enumerate-mono*: $\llbracket m < n; \text{finite } S; n < \text{card } S \rrbracket \implies \text{enumerate } S m < \text{enumerate } S n$
 $\langle \text{proof} \rangle$

lemma *finite-enumerate-mono-iff* [*simp*]:
 $\llbracket \text{finite } S; m < \text{card } S; n < \text{card } S \rrbracket \implies \text{enumerate } S m < \text{enumerate } S n \longleftrightarrow m < n$
 $\langle \text{proof} \rangle$

lemma *finite-le-enumerate*:
assumes $\text{finite } S n < \text{card } S$
shows $n \leq \text{enumerate } S n$
 $\langle \text{proof} \rangle$

lemma *finite-enumerate*:
assumes $fS: \text{finite } S$
shows $\exists r::\text{nat} \Rightarrow \text{nat}. \text{strict-mono-on } r \ \{..<\text{card } S\} \wedge (\forall n<\text{card } S. r n \in S)$
 $\langle \text{proof} \rangle$

lemma *finite-enumerate-Suc''*:

```

fixes  $S :: 'a::wellorder\ set$ 
assumes  $finite\ S\ Suc\ n < card\ S$ 
shows  $enumerate\ S\ (Suc\ n) = (LEAST\ s.\ s \in S \wedge enumerate\ S\ n < s)$ 
 $\langle proof \rangle$ 

```

```

lemma finite-enumerate-initial-segment:
  fixes  $S :: 'a::wellorder\ set$ 
  assumes  $finite\ S$  and  $n: n < card\ (S \cap \{..
  shows  $enumerate\ (S \cap \{..
   $\langle proof \rangle$$$ 
```

```

lemma finite-enumerate-Ex:
  fixes  $S :: 'a::wellorder\ set$ 
  assumes  $S: finite\ S$ 
  and  $s: s \in S$ 
  shows  $\exists n < card\ S.\ enumerate\ S\ n = s$ 
   $\langle proof \rangle$ 

```

```

lemma finite-enum-subset:
  assumes  $\bigwedge i.\ i < card\ X \implies enumerate\ X\ i = enumerate\ Y\ i$  and  $finite\ X\ finite$ 
   $Y\ card\ X \leq card\ Y$ 
  shows  $X \subseteq Y$ 
   $\langle proof \rangle$ 

```

```

lemma finite-enum-ext:
  assumes  $\bigwedge i.\ i < card\ X \implies enumerate\ X\ i = enumerate\ Y\ i$  and  $finite\ X\ finite$ 
   $Y\ card\ X = card\ Y$ 
  shows  $X = Y$ 
   $\langle proof \rangle$ 

```

```

lemma finite-bij-enumerate:
  fixes  $S :: 'a::wellorder\ set$ 
  assumes  $S: finite\ S$ 
  shows  $bij\ betw\ (enumerate\ S)\ \{..
   $\langle proof \rangle$$ 
```

```

lemma ex-bij-betw-strict-mono-card:
  fixes  $M :: 'a::wellorder\ set$ 
  assumes  $finite\ M$ 
  obtains  $h$  where  $bij\ betw\ h\ \{.. and  $strict\ mono\ on\ h\ \{..
   $\langle proof \rangle$$$ 
```

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 $S \longleftrightarrow (\exists f::'a \Rightarrow \text{nat}. \text{inj-on } f \ S)$

lemma *countableE*:

assumes $S: \text{countable } S$ **obtains** $f::'a \Rightarrow \text{nat}$ **where** *inj-on* $f \ S$
 $\langle \text{proof} \rangle$

lemma *countableI*: *inj-on* $(f::'a \Rightarrow \text{nat}) \ S \Longrightarrow \text{countable } S$
 $\langle \text{proof} \rangle$

lemma *countableI'*: *inj-on* $(f::'a \Rightarrow 'b::\text{countable}) \ S \Longrightarrow \text{countable } S$
 $\langle \text{proof} \rangle$

lemma *countableE-bij*:

assumes $S: \text{countable } S$ **obtains** $f::\text{nat} \Rightarrow 'a$ **and** $C::\text{nat set}$ **where** *bij-betw* $f \ C \ S$
 $\langle \text{proof} \rangle$

lemma *countableI-bij*: *bij-betw* $f \ (C::\text{nat set}) \ S \Longrightarrow \text{countable } S$
 $\langle \text{proof} \rangle$

lemma *countable-finite*: *finite* $S \Longrightarrow \text{countable } S$
 $\langle \text{proof} \rangle$

lemma *countableI-bij1*: *bij-betw* $f \ A \ B \Longrightarrow \text{countable } A \Longrightarrow \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countableI-bij2*: *bij-betw* $f \ B \ A \Longrightarrow \text{countable } A \Longrightarrow \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countable-iff-bij[simp]*: *bij-betw* $f \ A \ B \Longrightarrow \text{countable } A \longleftrightarrow \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countable-subset*: $A \subseteq B \Longrightarrow \text{countable } B \Longrightarrow \text{countable } A$
 $\langle \text{proof} \rangle$

lemma *countableI-type[intro, simp]*: *countable* $(A::'a::\text{countable set})$
 $\langle \text{proof} \rangle$

21.2 Enumerate a countable set

lemma *countableE-infinite*:

assumes *countable* S *infinite* S
obtains $e::'a \Rightarrow \text{nat}$ **where** *bij-betw* $e \ S \ \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *countable-infiniteE'*:

assumes *countable* A *infinite* A

obtains g **where** $\text{bij-betw } g \text{ (UNIV :: nat set) } A$
 $\langle \text{proof} \rangle$

lemma *countable-enum-cases:*

assumes *countable S*

obtains $(\text{finite}) f :: 'a \Rightarrow \text{nat}$ **where** $\text{finite } S \text{ bij-betw } f S \{..<\text{card } S\}$
 $| (\text{infinite}) f :: 'a \Rightarrow \text{nat}$ **where** $\text{infinite } S \text{ bij-betw } f S \text{ UNIV}$
 $\langle \text{proof} \rangle$

definition *to-nat-on* :: $'a \text{ set} \Rightarrow 'a \Rightarrow \text{nat}$ **where**

to-nat-on S = (SOME f. if finite S then bij-betw f S {.. $\text{card } S$ } else bij-betw f S UNIV)

definition *from-nat-into* :: $'a \text{ set} \Rightarrow \text{nat} \Rightarrow 'a$ **where**

from-nat-into S n = (if $n \in \text{to-nat-on } S$ ' S then inv-into S (to-nat-on S) n else SOME s. $s \in S$)

lemma *to-nat-on-finite:* $\text{finite } S \Longrightarrow \text{bij-betw } (\text{to-nat-on } S) S \{.. $\text{card } S$ \}$
 $\langle \text{proof} \rangle$

lemma *to-nat-on-infinite:* $\text{countable } S \Longrightarrow \text{infinite } S \Longrightarrow \text{bij-betw } (\text{to-nat-on } S) S \text{ UNIV}$
 $\langle \text{proof} \rangle$

lemma *bij-betw-from-nat-into-finite:* $\text{finite } S \Longrightarrow \text{bij-betw } (\text{from-nat-into } S) \{.. $\text{card } S$ \} S$
 $\langle \text{proof} \rangle$

lemma *bij-betw-from-nat-into:* $\text{countable } S \Longrightarrow \text{infinite } S \Longrightarrow \text{bij-betw } (\text{from-nat-into } S) \text{ UNIV } S$
 $\langle \text{proof} \rangle$

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 (\lambda i. h (\text{from-nat-into } A i)) \{.. $\text{card } A$ \} = F h A$
 $\langle \text{proof} \rangle$

end

lemma *countable-as-injective-image:*

assumes *countable A infinite A*

obtains $f :: \text{nat} \Rightarrow 'a$ **where** $A = \text{range } f \text{ inj } f$
 $\langle \text{proof} \rangle$

lemma *inj-on-to-nat-on[intro]:* $\text{countable } A \Longrightarrow \text{inj-on } (\text{to-nat-on } A) A$

$\langle \text{proof} \rangle$

lemma *to-nat-on-inj[simp]*:

$\text{countable } A \implies a \in A \implies b \in A \implies \text{to-nat-on } A \ a = \text{to-nat-on } A \ b \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *from-nat-into-to-nat-on[simp]*: $\text{countable } A \implies a \in A \implies \text{from-nat-into } A \ (\text{to-nat-on } A \ a) = a$
 $\langle \text{proof} \rangle$

lemma *subset-range-from-nat-into*: $\text{countable } A \implies A \subseteq \text{range } (\text{from-nat-into } A)$
 $\langle \text{proof} \rangle$

lemma *from-nat-into*: $A \neq \{\} \implies \text{from-nat-into } A \ n \in A$
 $\langle \text{proof} \rangle$

lemma *range-from-nat-into-subset*: $A \neq \{\} \implies \text{range } (\text{from-nat-into } A) \subseteq A$
 $\langle \text{proof} \rangle$

lemma *range-from-nat-into[simp]*: $A \neq \{\} \implies \text{countable } A \implies \text{range } (\text{from-nat-into } A) = A$
 $\langle \text{proof} \rangle$

lemma *image-to-nat-on*: $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A \ ' \ A = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *to-nat-on-surj*: $\text{countable } A \implies \text{infinite } A \implies \exists a \in A. \text{to-nat-on } A \ a = n$
 $\langle \text{proof} \rangle$

lemma *to-nat-on-from-nat-into[simp]*: $n \in \text{to-nat-on } A \ ' \ A \implies \text{to-nat-on } A \ (\text{from-nat-into } A \ n) = n$
 $\langle \text{proof} \rangle$

lemma *to-nat-on-from-nat-into-infinite[simp]*:
 $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A \ (\text{from-nat-into } A \ n) = n$
 $\langle \text{proof} \rangle$

lemma *from-nat-into-inj*:
 $\text{countable } A \implies m \in \text{to-nat-on } A \ ' \ A \implies n \in \text{to-nat-on } A \ ' \ A \implies$
 $\text{from-nat-into } A \ m = \text{from-nat-into } A \ n \longleftrightarrow m = n$
 $\langle \text{proof} \rangle$

lemma *from-nat-into-inj-infinite[simp]*:
 $\text{countable } A \implies \text{infinite } A \implies \text{from-nat-into } A \ m = \text{from-nat-into } A \ n \longleftrightarrow m = n$
 $\langle \text{proof} \rangle$

lemma *eq-from-nat-into-iff*:
 $\text{countable } A \implies x \in A \implies i \in \text{to-nat-on } A \ ' \ A \implies x = \text{from-nat-into } A \ i \longleftrightarrow$

i = to-nat-on *A* *x*
 ⟨proof⟩

lemma *from-nat-into-surj*: countable *A* $\implies a \in A \implies \exists n. \text{from-nat-into } A \ n = a$
 ⟨proof⟩

lemma *from-nat-into-inject*[*simp*]:
 $A \neq \{\}$ $\implies$ countable *A* $\implies B \neq \{\} \implies$ countable *B* $\implies$ from-nat-into *A* = from-nat-into *B* $\longleftrightarrow A = B$
 ⟨proof⟩

lemma *inj-on-from-nat-into*: inj-on from-nat-into ($\{A. A \neq \{\} \wedge \text{countable } A\}$)
 ⟨proof⟩

21.3 Closure properties of countability

lemma *countable-SIGMA*[*intro*, *simp*]:
 countable *I* $\implies (\bigwedge i. i \in I \implies \text{countable } (A \ i)) \implies \text{countable } (\text{SIGMA } i : I. A \ i)$
 ⟨proof⟩

lemma *countable-image*[*intro*, *simp*]:
 assumes countable *A*
 shows countable (*f*‘*A*)
 ⟨proof⟩

lemma *countable-image-inj-on*: countable (*f* ‘ *A*) $\implies$ inj-on *f* *A* $\implies$ countable *A*
 ⟨proof⟩

lemma *countable-image-inj-Int-vimage*:
 $\llbracket \text{inj-on } f \ S; \text{countable } A \rrbracket \implies \text{countable } (S \cap f^{-1} \text{‘ } A)$
 ⟨proof⟩

lemma *countable-image-inj-gen*:
 $\llbracket \text{inj-on } f \ S; \text{countable } A \rrbracket \implies \text{countable } \{x \in S. f \ x \in A\}$
 ⟨proof⟩

lemma *countable-image-inj-eq*:
 inj-on *f* *S* $\implies \text{countable}(f \text{ ‘ } S) \longleftrightarrow \text{countable } S$
 ⟨proof⟩

lemma *countable-image-inj*:
 $\llbracket \text{countable } A; \text{inj } f \rrbracket \implies \text{countable } \{x. f \ x \in A\}$
 ⟨proof⟩

lemma *countable-UN*[*intro*, *simp*]:
 fixes *I* :: ‘*i* set and *A* :: ‘*i* => ‘*a* set
 assumes *I*: countable *I*

assumes $A: \bigwedge i. i \in I \implies \text{countable } (A \ i)$
shows $\text{countable } (\bigcup_{i \in I}. A \ i)$
 $\langle \text{proof} \rangle$

lemma *countable-Un[intro]*: $\text{countable } A \implies \text{countable } B \implies \text{countable } (A \cup B)$
 $\langle \text{proof} \rangle$

lemma *countable-Un-iff[simp]*: $\text{countable } (A \cup B) \longleftrightarrow \text{countable } A \wedge \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countable-Plus[intro, simp]*:
 $\text{countable } A \implies \text{countable } B \implies \text{countable } (A <+> B)$
 $\langle \text{proof} \rangle$

lemma *countable-empty[intro, simp]*: $\text{countable } \{\}$
 $\langle \text{proof} \rangle$

lemma *countable-insert[intro, simp]*: $\text{countable } A \implies \text{countable } (\text{insert } a \ A)$
 $\langle \text{proof} \rangle$

lemma *countable-Int1[intro, simp]*: $\text{countable } A \implies \text{countable } (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *countable-Int2[intro, simp]*: $\text{countable } B \implies \text{countable } (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *countable-INT[intro, simp]*: $i \in I \implies \text{countable } (A \ i) \implies \text{countable } (\bigcap_{i \in I}. A \ i)$
 $\langle \text{proof} \rangle$

lemma *countable-Diff[intro, simp]*: $\text{countable } A \implies \text{countable } (A - B)$
 $\langle \text{proof} \rangle$

lemma *countable-insert-eq [simp]*: $\text{countable } (\text{insert } x \ A) = \text{countable } A$
 $\langle \text{proof} \rangle$

lemma *countable-vimage*: $B \subseteq \text{range } f \implies \text{countable } (f^{-1} B) \implies \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *surj-countable-vimage*: $\text{surj } f \implies \text{countable } (f^{-1} B) \implies \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countable-Collect[simp]*: $\text{countable } A \implies \text{countable } \{a \in A. \varphi \ a\}$
 $\langle \text{proof} \rangle$

lemma *countable-Image*:
assumes $\bigwedge y. y \in Y \implies \text{countable } (X \text{ “ } \{y\})$
assumes $\text{countable } Y$
shows $\text{countable } (X \text{ “ } Y)$

$\langle proof \rangle$

lemma *countable-relpow*:

fixes $X :: 'a\ rel$

assumes $Image\text{-}X: \bigwedge Y. countable\ Y \implies countable\ (X\ “\ Y)$

assumes $Y: countable\ Y$

shows $countable\ ((X\ \frown i)\ “\ Y)$

$\langle proof \rangle$

lemma *countable-funpow*:

fixes $f :: 'a\ set \Rightarrow 'a\ set$

assumes $\bigwedge A. countable\ A \implies countable\ (f\ A)$

and $countable\ A$

shows $countable\ ((f\ \frown n)\ A)$

$\langle proof \rangle$

lemma *countable-rtranc1*:

$(\bigwedge Y. countable\ Y \implies countable\ (X\ “\ Y)) \implies countable\ Y \implies countable\ (X^*\ “\ Y)$

$\langle proof \rangle$

lemma *countable-lists*[*intro, simp*]:

assumes $A: countable\ A$ **shows** $countable\ (lists\ A)$

$\langle proof \rangle$

lemma *Collect-finite-eq-lists*: $Collect\ finite = set\ “\ lists\ UNIV$

$\langle proof \rangle$

lemma *countable-Collect-finite*: $countable\ (Collect\ (finite::'a::countable\ set \Rightarrow bool))$

$\langle proof \rangle$

lemma *countable-int*: $countable\ \mathbb{Z}$

$\langle proof \rangle$

lemma *countable-rat*: $countable\ \mathbb{Q}$

$\langle proof \rangle$

lemma *Collect-finite-subset-eq-lists*: $\{A. finite\ A \wedge A \subseteq T\} = set\ “\ lists\ T$

$\langle proof \rangle$

lemma *countable-Collect-finite-subset*:

$countable\ T \implies countable\ \{A. finite\ A \wedge A \subseteq T\}$

$\langle proof \rangle$

lemma *countable-Fpow*: $countable\ S \implies countable\ (Fpow\ S)$

$\langle proof \rangle$

lemma *countable-set-option* [*simp*]: $countable\ (set\ option\ x)$

$\langle proof \rangle$

21.4 Misc lemmas

lemma *countable-subset-image*:

$\text{countable } B \wedge B \subseteq (f \text{ ‘ } A) \longleftrightarrow (\exists A'. \text{countable } A' \wedge A' \subseteq A \wedge (B = f \text{ ‘ } A'))$
(is ?lhs = ?rhs)

<proof>

lemma *ex-subset-image-inj*:

$(\exists T. T \subseteq f \text{ ‘ } S \wedge P \ T) \longleftrightarrow (\exists T. T \subseteq S \wedge \text{inj-on } f \ T \wedge P \ (f \text{ ‘ } T))$

<proof>

lemma *all-subset-image-inj*:

$(\forall T. T \subseteq f \text{ ‘ } S \longrightarrow P \ T) \longleftrightarrow (\forall T. T \subseteq S \wedge \text{inj-on } f \ T \longrightarrow P(f \text{ ‘ } T))$

<proof>

lemma *ex-countable-subset-image-inj*:

$(\exists T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \wedge P \ T) \longleftrightarrow$
 $(\exists T. \text{countable } T \wedge T \subseteq S \wedge \text{inj-on } f \ T \wedge P \ (f \text{ ‘ } T))$

<proof>

lemma *all-countable-subset-image-inj*:

$(\forall T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \longrightarrow P \ T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \wedge \text{inj-on } f \ T \longrightarrow P(f \text{ ‘ } T))$

<proof>

lemma *ex-countable-subset-image*:

$(\exists T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \wedge P \ T) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge P \ (f \text{ ‘ } T))$

<proof>

lemma *all-countable-subset-image*:

$(\forall T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \longrightarrow P \ T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \longrightarrow P(f \text{ ‘ } T))$

<proof>

lemma *countable-image-eq*:

$\text{countable}(f \text{ ‘ } S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ‘ } S = f \text{ ‘ } T)$

<proof>

lemma *countable-image-eq-inj*:

$\text{countable}(f \text{ ‘ } S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ‘ } S = f \text{ ‘ } T \wedge \text{inj-on } f \ T)$

<proof>

lemma *infinite-countable-subset'*:

assumes X : *infinite* X **shows** $\exists C \subseteq X. \text{countable } C \wedge \text{infinite } C$

<proof>

lemma *countable-all*:

assumes S : *countable* S

shows $(\forall s \in S. P \ s) \longleftrightarrow (\forall n :: \text{nat}. \text{from-nat-into } S \ n \in S \longrightarrow P \ (\text{from-nat-into } S \ n))$

$S\ n))$
 $\langle proof \rangle$

lemma *finite-sequence-to-countable-set*:

assumes *countable* X
obtains F **where** $\bigwedge i. F\ i \subseteq X \ \bigwedge i. F\ i \subseteq F\ (Suc\ i) \ \bigwedge i. finite\ (F\ i) \ (\bigcup i. F\ i) = X$
 $\langle proof \rangle$

lemma *transfer-countable*[*transfer-rule*]:

bi-unique $R \implies rel\text{-}fun\ (rel\text{-}set\ R) (=)\ countable\ countable$
 $\langle proof \rangle$

21.5 Uncountable

abbreviation *uncountable where*

uncountable $A \equiv \neg\ countable\ A$

lemma *uncountable-def*: *uncountable* $A \longleftrightarrow A \neq \{\}$ $\wedge \neg (\exists f::(nat \Rightarrow 'a). range\ f = A)$
 $\langle proof \rangle$

lemma *uncountable-bij-betw*: *bij-betw* $f\ A\ B \implies uncountable\ B \implies uncountable\ A$
 $\langle proof \rangle$

lemma *uncountable-infinite*: *uncountable* $A \implies infinite\ A$
 $\langle proof \rangle$

lemma *uncountable-minus-countable*:

uncountable $A \implies countable\ B \implies uncountable\ (A - B)$
 $\langle proof \rangle$

lemma *countable-Diff-eq* [*simp*]: *countable* $(A - \{x\}) = countable\ A$
 $\langle proof \rangle$

Every infinite set can be covered by a pairwise disjoint family of infinite sets. This version doesn’t achieve equality, as it only covers a countable subset

lemma *infinite-infinite-partition*:

assumes *infinite* A
obtains $C :: nat \Rightarrow 'a\ set$
where *pairwise* $(\lambda i\ j. disjnt\ (C\ i)\ (C\ j))\ UNIV\ (\bigcup i. C\ i) \subseteq A \ \bigwedge i. infinite\ (C\ i)$
 $\langle proof \rangle$

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*: $countable\ A \implies x \in A \implies Inf\ A \leq x$

assumes *ccInf-greatest*: $countable\ A \implies (\bigwedge x. x \in A \implies z \leq x) \implies z \leq Inf\ A$

assumes *ccSup-upper*: $countable\ A \implies x \in A \implies x \leq Sup\ A$

assumes *ccSup-least*: $countable\ A \implies (\bigwedge x. x \in A \implies x \leq z) \implies Sup\ A \leq z$

assumes *ccInf-empty* [*simp*]: $Inf\ \{\} = top$

assumes *ccSup-empty* [*simp*]: $Sup\ \{\} = bot$

begin

subclass *bounded-lattice*

<proof>

lemma *ccINF-lower*: $countable\ A \implies i \in A \implies (INF\ i \in A. f\ i) \leq f\ i$

<proof>

lemma *ccINF-greatest*: $countable\ A \implies (\bigwedge i. i \in A \implies u \leq f\ i) \implies u \leq (INF\ i \in A. f\ i)$

<proof>

lemma *ccSUP-upper*: $countable\ A \implies i \in A \implies f\ i \leq (SUP\ i \in A. f\ i)$

<proof>

lemma *ccSUP-least*: $countable\ A \implies (\bigwedge i. i \in A \implies f\ i \leq u) \implies (SUP\ i \in A. f\ i) \leq u$

<proof>

lemma *ccInf-lower2*: $countable\ A \implies u \in A \implies u \leq v \implies Inf\ A \leq v$

<proof>

lemma *ccINF-lower2*: $countable\ A \implies i \in A \implies f\ i \leq u \implies (INF\ i \in A. f\ i) \leq u$

<proof>

lemma *ccSup-upper2*: $countable\ A \implies u \in A \implies v \leq u \implies v \leq Sup\ A$

<proof>

lemma *ccSUP-upper2*: $countable\ A \implies i \in A \implies u \leq f\ i \implies u \leq (SUP\ i \in A. f\ i)$

<proof>

lemma *le-ccInf-iff*: $\text{countable } A \implies b \leq \text{Inf } A \longleftrightarrow (\forall a \in A. b \leq a)$
 ⟨proof⟩

lemma *le-ccINF-iff*: $\text{countable } A \implies u \leq (\text{INF } i \in A. f i) \longleftrightarrow (\forall i \in A. u \leq f i)$
 ⟨proof⟩

lemma *ccSup-le-iff*: $\text{countable } A \implies \text{Sup } A \leq b \longleftrightarrow (\forall a \in A. a \leq b)$
 ⟨proof⟩

lemma *ccSUP-le-iff*: $\text{countable } A \implies (\text{SUP } i \in A. f i) \leq u \longleftrightarrow (\forall i \in A. f i \leq u)$
 ⟨proof⟩

lemma *ccInf-insert [simp]*: $\text{countable } A \implies \text{Inf } (\text{insert } a A) = \text{inf } a (\text{Inf } A)$
 ⟨proof⟩

lemma *ccINF-insert [simp]*: $\text{countable } A \implies (\text{INF } x \in \text{insert } a A. f x) = \text{inf } (f a) (\text{Inf } (f ` A))$
 ⟨proof⟩

lemma *ccSup-insert [simp]*: $\text{countable } A \implies \text{Sup } (\text{insert } a A) = \text{sup } a (\text{Sup } A)$
 ⟨proof⟩

lemma *ccSUP-insert [simp]*: $\text{countable } A \implies (\text{SUP } x \in \text{insert } a A. f x) = \text{sup } (f a) (\text{Sup } (f ` A))$
 ⟨proof⟩

lemma *ccINF-empty [simp]*: $(\text{INF } x \in \{\}. f x) = \text{top}$
 ⟨proof⟩

lemma *ccSUP-empty [simp]*: $(\text{SUP } x \in \{\}. f x) = \text{bot}$
 ⟨proof⟩

lemma *ccInf-superset-mono*: $\text{countable } A \implies B \subseteq A \implies \text{Inf } A \leq \text{Inf } B$
 ⟨proof⟩

lemma *ccSup-subset-mono*: $\text{countable } B \implies A \subseteq B \implies \text{Sup } A \leq \text{Sup } B$
 ⟨proof⟩

lemma *ccInf-mono*:
 assumes [intro]: $\text{countable } B \text{ countable } A$
 assumes $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$
 shows $\text{Inf } A \leq \text{Inf } B$
 ⟨proof⟩

lemma *ccINF-mono*:
 $\text{countable } A \implies \text{countable } B \implies (\bigwedge m. m \in B \implies \exists n \in A. f n \leq g m) \implies (\text{INF } n \in A. f n) \leq (\text{INF } n \in B. g n)$
 ⟨proof⟩

lemma *ccSup-mono*:

assumes [intro]: *countable B countable A*

assumes $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$

shows $\text{Sup } A \leq \text{Sup } B$

<proof>

lemma *ccSUP-mono*:

countable A $\implies$ countable B $\implies (\bigwedge n. n \in A \implies \exists m \in B. f n \leq g m) \implies (\text{SUP } n \in A. f n) \leq (\text{SUP } n \in B. g n)$

<proof>

lemma *ccINF-superset-mono*:

countable A $\implies B \subseteq A \implies (\bigwedge x. x \in B \implies f x \leq g x) \implies (\text{INF } x \in A. f x) \leq (\text{INF } x \in B. g x)$

<proof>

lemma *ccSUP-subset-mono*:

countable B $\implies A \subseteq B \implies (\bigwedge x. x \in A \implies f x \leq g x) \implies (\text{SUP } x \in A. f x) \leq (\text{SUP } x \in B. g x)$

<proof>

lemma *less-eq-ccInf-inter*: *countable A $\implies$ countable B $\implies \text{sup } (\text{Inf } A) (\text{Inf } B) \leq \text{Inf } (A \cap B)$*

<proof>

lemma *ccSup-inter-less-eq*: *countable A $\implies$ countable B $\implies \text{Sup } (A \cap B) \leq \text{inf } (\text{Sup } A) (\text{Sup } B)$*

<proof>

lemma *ccInf-union-distrib*: *countable A $\implies$ countable B $\implies \text{Inf } (A \cup B) = \text{inf } (\text{Inf } A) (\text{Inf } B)$*

<proof>

lemma *ccINF-union*:

countable A $\implies$ countable B $\implies (\text{INF } i \in A \cup B. M i) = \text{inf } (\text{INF } i \in A. M i) (\text{INF } i \in B. M i)$

<proof>

lemma *ccSup-union-distrib*: *countable A $\implies$ countable B $\implies \text{Sup } (A \cup B) = \text{sup } (\text{Sup } A) (\text{Sup } B)$*

<proof>

lemma *ccSUP-union*:

countable A $\implies$ countable B $\implies (\text{SUP } i \in A \cup B. M i) = \text{sup } (\text{SUP } i \in A. M i) (\text{SUP } i \in B. M i)$

<proof>

lemma *ccINF-inf-distrib*: *countable A $\implies \text{inf } (\text{INF } a \in A. f a) (\text{INF } a \in A. g a) =$*

$(\text{INF } a \in A. \text{ inf } (f a) (g a))$
 $\langle \text{proof} \rangle$

lemma *ccSUP-sup-distrib*: $\text{countable } A \implies \text{sup } (\text{SUP } a \in A. f a) (\text{SUP } a \in A. g a)$
 $= (\text{SUP } a \in A. \text{sup } (f a) (g a))$
 $\langle \text{proof} \rangle$

lemma *ccINF-const [simp]*: $A \neq \{\}$ $\implies (\text{INF } i \in A. f) = f$
 $\langle \text{proof} \rangle$

lemma *ccSUP-const [simp]*: $A \neq \{\}$ $\implies (\text{SUP } i \in A. f) = f$
 $\langle \text{proof} \rangle$

lemma *ccINF-top [simp]*: $(\text{INF } x \in A. \text{top}) = \text{top}$
 $\langle \text{proof} \rangle$

lemma *ccSUP-bot [simp]*: $(\text{SUP } x \in A. \text{bot}) = \text{bot}$
 $\langle \text{proof} \rangle$

lemma *ccINF-commute*: $\text{countable } A \implies \text{countable } B \implies (\text{INF } i \in A. \text{INF } j \in B. f i j) = (\text{INF } j \in B. \text{INF } i \in A. f i j)$
 $\langle \text{proof} \rangle$

lemma *ccSUP-commute*: $\text{countable } A \implies \text{countable } B \implies (\text{SUP } i \in A. \text{SUP } j \in B. f i j) = (\text{SUP } j \in B. \text{SUP } i \in A. f i j)$
 $\langle \text{proof} \rangle$

end

context

fixes $a :: 'a :: \{\text{countable-complete-lattice}, \text{linorder}\}$

begin

lemma *less-ccSup-iff*: $\text{countable } S \implies a < \text{Sup } S \longleftrightarrow (\exists x \in S. a < x)$
 $\langle \text{proof} \rangle$

lemma *less-ccSUP-iff*: $\text{countable } A \implies a < (\text{SUP } i \in A. f i) \longleftrightarrow (\exists x \in A. a < f x)$
 $\langle \text{proof} \rangle$

lemma *ccInf-less-iff*: $\text{countable } S \implies \text{Inf } S < a \longleftrightarrow (\exists x \in S. x < a)$
 $\langle \text{proof} \rangle$

lemma *ccINF-less-iff*: $\text{countable } A \implies (\text{INF } i \in A. f i) < a \longleftrightarrow (\exists x \in A. f x < a)$
 $\langle \text{proof} \rangle$

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
assumes *sup-ccInf*: $\text{countable } B \implies \text{sup } a (\text{Inf } B) = (\text{INF } b \in B. \text{sup } a b)$

assumes *inf-ccSup*: $\text{countable } B \implies \inf a (\text{Sup } B) = (\text{SUP } b \in B. \inf a b)$
begin

lemma *sup-ccINF*:
 $\text{countable } B \implies \sup a (\text{INF } b \in B. f b) = (\text{INF } b \in B. \sup a (f b))$
 $\langle \text{proof} \rangle$

lemma *inf-ccSUP*:
 $\text{countable } B \implies \inf a (\text{SUP } b \in B. f b) = (\text{SUP } b \in B. \inf a (f b))$
 $\langle \text{proof} \rangle$

subclass *distrib-lattice*
 $\langle \text{proof} \rangle$

lemma *ccInf-sup*:
 $\text{countable } B \implies \sup (\text{Inf } B) a = (\text{INF } b \in B. \sup b a)$
 $\langle \text{proof} \rangle$

lemma *ccSup-inf*:
 $\text{countable } B \implies \inf (\text{Sup } B) a = (\text{SUP } b \in B. \inf b a)$
 $\langle \text{proof} \rangle$

lemma *ccINF-sup*:
 $\text{countable } B \implies \sup (\text{INF } b \in B. f b) a = (\text{INF } b \in B. \sup (f b) a)$
 $\langle \text{proof} \rangle$

lemma *ccSUP-inf*:
 $\text{countable } B \implies \inf (\text{SUP } b \in B. f b) a = (\text{SUP } b \in B. \inf (f b) a)$
 $\langle \text{proof} \rangle$

lemma *ccINF-sup-distrib2*:
 $\text{countable } A \implies \text{countable } B \implies \sup (\text{INF } a \in A. f a) (\text{INF } b \in B. g b) = (\text{INF } a \in A. \text{INF } b \in B. \sup (f a) (g b))$
 $\langle \text{proof} \rangle$

lemma *ccSUP-inf-distrib2*:
 $\text{countable } A \implies \text{countable } B \implies \inf (\text{SUP } a \in A. f a) (\text{SUP } b \in B. g b) = (\text{SUP } a \in A. \text{SUP } b \in B. \inf (f a) (g b))$
 $\langle \text{proof} \rangle$

context
fixes $f :: 'a \Rightarrow 'b :: \text{countable-complete-lattice}$
assumes *mono f*
begin

lemma *mono-ccInf*:
 $\text{countable } A \implies f (\text{Inf } A) \leq (\text{INF } x \in A. f x)$
 $\langle \text{proof} \rangle$

lemma *mono-ccSup*:

countable A $\implies$ (SUP $x \in A$. $f\ x$) $\leq$ f (Sup A)

<proof>

lemma *mono-ccINF*:

countable I $\implies$ f (INF $i \in I$. $A\ i$) $\leq$ (INF $x \in I$. f (A x))

<proof>

lemma *mono-ccSUP*:

countable I $\implies$ (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 $(\text{finite } A \wedge |A| < o \mid \text{UNIV}::\text{nat set}|) \vee$
 $(\text{infinite } A \wedge |A| = o \mid \text{UNIV}::\text{nat set}|)$
 $\langle \text{proof} \rangle$

lemma *countable-cases-card-of*[*elim*]:
assumes *countable A*
obtains $(\text{Fin}) \text{ finite } A \mid |A| < o \mid \text{UNIV}::\text{nat set}|$
 $\mid (\text{Inf}) \text{ infinite } A \mid |A| = o \mid \text{UNIV}::\text{nat set}|$
 $\langle \text{proof} \rangle$

lemma *countable-or*:
 $\text{countable } A \implies (\exists f::'a \Rightarrow \text{nat}. \text{finite } A \wedge \text{inj-on } f \ A) \vee (\exists f::'a \Rightarrow \text{nat}. \text{infinite } A$
 $\wedge \text{bij-betw } f \ A \ \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *countable-cases*[*elim*]:
assumes *countable A*
obtains $(\text{Fin}) f :: 'a \Rightarrow \text{nat} \text{ where } \text{finite } A \text{ inj-on } f \ A$
 $\mid (\text{Inf}) f :: 'a \Rightarrow \text{nat} \text{ where } \text{infinite } A \text{ bij-betw } f \ A \ \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *countable-ordLeq*:
assumes $|A| \leq o \mid B|$ **and** *countable B*
shows *countable A*
 $\langle \text{proof} \rangle$

lemma *countable-ordLess*:
assumes *AB*: $|A| < o \mid B|$ **and** *B*: *countable B*
shows *countable A*
 $\langle \text{proof} \rangle$

end

23.2 The type of countable sets

typedef *'a cset* = $\{A :: 'a \text{ set}. \text{countable } A\}$ **morphisms** *rcset acset*
 $\langle \text{proof} \rangle$

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 $\Rightarrow$ 'a cset $\Rightarrow$ bool
is *subset-eq* **parametric** *subset-transfer* ⟨proof⟩

definition *less-cset* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ bool
where $xs < ys \equiv xs \leq ys \wedge xs \neq (ys::'a \text{ cset})$

lemma *less-cset-transfer*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique A*
shows $((\text{pcr-cset } A) ==> (\text{pcr-cset } A) ==> (=)) (\subset) (<)$
⟨proof⟩

lift-definition *sup-cset* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ 'a cset
is *union* **parametric** *union-transfer* ⟨proof⟩

lift-definition *inf-cset* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ 'a cset
is *inter* **parametric** *inter-transfer* ⟨proof⟩

lift-definition *minus-cset* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ 'a cset
is *minus* **parametric** *Diff-transfer* ⟨proof⟩

instance ⟨proof⟩

end

abbreviation *empty* :: 'a cset **where** *empty* $\equiv$ *bot*
abbreviation *csubset-eq* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ bool **where** *csubset-eq* $xs \ ys \equiv xs \leq ys$
abbreviation *csubset* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ bool **where** *csubset* $xs \ ys \equiv xs < ys$
abbreviation *cUn* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ 'a cset **where** *cUn* $xs \ ys \equiv \text{sup } xs \ ys$
abbreviation *cInt* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ 'a cset **where** *cInt* $xs \ ys \equiv \text{inf } xs \ ys$
abbreviation *cDiff* :: 'a cset $\Rightarrow$ 'a cset $\Rightarrow$ 'a cset **where** *cDiff* $xs \ ys \equiv \text{minus } xs \ ys$

lift-definition *cin* :: 'a $\Rightarrow$ 'a cset $\Rightarrow$ bool is ($\in$) **parametric** *member-transfer*
⟨proof⟩

lift-definition *cinsert* :: 'a $\Rightarrow$ 'a cset $\Rightarrow$ 'a cset is *insert* **parametric** *Lifting-Set.insert-transfer*
⟨proof⟩

abbreviation *csingle* :: 'a $\Rightarrow$ 'a cset **where** *csingle* $x \equiv \text{cinsert } x \text{ empty}$

lift-definition *cimage* :: ('a $\Rightarrow$ 'b) $\Rightarrow$ 'a cset $\Rightarrow$ 'b cset is (') **parametric** *image-transfer*
⟨proof⟩

lift-definition *cBall* :: 'a cset $\Rightarrow$ ('a $\Rightarrow$ bool) $\Rightarrow$ bool is *Ball* **parametric** *Ball-transfer*
⟨proof⟩

lift-definition *cBex* :: 'a cset $\Rightarrow$ ('a $\Rightarrow$ bool) $\Rightarrow$ bool is *Bex* **parametric** *Bex-transfer*
⟨proof⟩

lift-definition *cUnion* :: 'a cset cset $\Rightarrow$ 'a cset is *Union* **parametric** *Union-transfer*

<proof>

abbreviation (*input*) $c\text{UNION} :: 'a \text{ cset} \Rightarrow ('a \Rightarrow 'b \text{ cset}) \Rightarrow 'b \text{ cset}$
where $c\text{UNION } A \ f \equiv c\text{Union } (c\text{image } f \ A)$

lemma *Union-conv-UNION*: $\bigcup A = \bigcup (\text{id} \cdot A)$

<proof>

lemmas $c\text{set-eqI} = \text{set-eqI}[\text{Transfer.transferred}]$
lemmas $c\text{set-eq-iff}[no-atp] = \text{set-eq-iff}[\text{Transfer.transferred}]$
lemmas $c\text{Ball}[\text{intro!}] = \text{ball}[\text{Transfer.transferred}]$
lemmas $c\text{bspec}[\text{dest?}] = \text{bspec}[\text{Transfer.transferred}]$
lemmas $c\text{BallE}[\text{elim}] = \text{ballE}[\text{Transfer.transferred}]$
lemmas $c\text{BexI}[\text{intro}] = \text{bexI}[\text{Transfer.transferred}]$
lemmas $\text{rev-cBexI}[\text{intro?}] = \text{rev-bexI}[\text{Transfer.transferred}]$
lemmas $c\text{BexCI} = \text{bexCI}[\text{Transfer.transferred}]$
lemmas $c\text{BexE}[\text{elim!}] = \text{bexE}[\text{Transfer.transferred}]$
lemmas $c\text{Ball-triv}[\text{simp}] = \text{ball-triv}[\text{Transfer.transferred}]$
lemmas $c\text{Bex-triv}[\text{simp}] = \text{bex-triv}[\text{Transfer.transferred}]$
lemmas $c\text{Bex-triv-one-point1}[\text{simp}] = \text{bex-triv-one-point1}[\text{Transfer.transferred}]$
lemmas $c\text{Bex-triv-one-point2}[\text{simp}] = \text{bex-triv-one-point2}[\text{Transfer.transferred}]$
lemmas $c\text{Bex-one-point1}[\text{simp}] = \text{bex-one-point1}[\text{Transfer.transferred}]$
lemmas $c\text{Bex-one-point2}[\text{simp}] = \text{bex-one-point2}[\text{Transfer.transferred}]$
lemmas $c\text{Ball-one-point1}[\text{simp}] = \text{ball-one-point1}[\text{Transfer.transferred}]$
lemmas $c\text{Ball-one-point2}[\text{simp}] = \text{ball-one-point2}[\text{Transfer.transferred}]$
lemmas $c\text{Ball-conj-distrib} = \text{ball-conj-distrib}[\text{Transfer.transferred}]$
lemmas $c\text{Bex-disj-distrib} = \text{bex-disj-distrib}[\text{Transfer.transferred}]$
lemmas $c\text{Ball-cong} = \text{ball-cong}[\text{Transfer.transferred}]$
lemmas $c\text{Bex-cong} = \text{bex-cong}[\text{Transfer.transferred}]$
lemmas $c\text{subsetI}[\text{intro!}] = \text{subsetI}[\text{Transfer.transferred}]$
lemmas $c\text{subsetD}[\text{elim}, \text{intro?}] = \text{subsetD}[\text{Transfer.transferred}]$
lemmas $\text{rev-csubsetD}[no-atp, \text{intro?}] = \text{rev-subsetD}[\text{Transfer.transferred}]$
lemmas $c\text{subsetCE}[no-atp, \text{elim}] = \text{subsetCE}[\text{Transfer.transferred}]$
lemmas $c\text{subset-eq}[no-atp] = \text{subset-eq}[\text{Transfer.transferred}]$
lemmas $\text{contra-csubsetD}[no-atp] = \text{contra-subsetD}[\text{Transfer.transferred}]$
lemmas $c\text{subset-refl} = \text{subset-refl}[\text{Transfer.transferred}]$
lemmas $c\text{subset-trans} = \text{subset-trans}[\text{Transfer.transferred}]$
lemmas $c\text{set-rev-mp} = \text{rev-subsetD}[\text{Transfer.transferred}]$
lemmas $c\text{set-mp} = \text{subsetD}[\text{Transfer.transferred}]$
lemmas $c\text{subset-not-fsubset-eq}[\text{code}] = \text{subset-not-subset-eq}[\text{Transfer.transferred}]$
lemmas $\text{eq-cmem-trans} = \text{eq-mem-trans}[\text{Transfer.transferred}]$
lemmas $c\text{subset-antisym}[\text{intro!}] = \text{subset-antisym}[\text{Transfer.transferred}]$
lemmas $\text{cequalityD1} = \text{equalityD1}[\text{Transfer.transferred}]$
lemmas $\text{cequalityD2} = \text{equalityD2}[\text{Transfer.transferred}]$
lemmas $\text{cequalityE} = \text{equalityE}[\text{Transfer.transferred}]$
lemmas $\text{cequalityCE}[\text{elim}] = \text{equalityCE}[\text{Transfer.transferred}]$
lemmas $\text{eqcset-imp-iff} = \text{eqset-imp-iff}[\text{Transfer.transferred}]$
lemmas $\text{eqelem-imp-iff} = \text{eqelem-imp-iff}[\text{Transfer.transferred}]$
lemmas $c\text{empty-iff}[\text{simp}] = \text{empty-iff}[\text{Transfer.transferred}]$
lemmas $c\text{empty-fsubsetI}[\text{iff}] = \text{empty-subsetI}[\text{Transfer.transferred}]$


```

lemmas equals-emptyI = equalsOI[Transfer.transferred]
lemmas equals-emptyD = equalsOD[Transfer.transferred]
lemmas cBall-empty[simp] = ball-empty[Transfer.transferred]
lemmas cBex-empty[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-finset-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-empty-left = Int-empty-left[Transfer.transferred]
lemmas cInt-empty-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-empty-left = Un-empty-left[Transfer.transferred]
lemmas cUn-empty-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-empty-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-psubset-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 *empty*

lemma *emptyE* [*elim!*]: *cin a empty* $\implies$ *P* $\langle$ *proof* $\rangle$

23.3.2 *cinsert*

lemma *countable-insert-iff*: *countable* (*insert x A*) $\longleftrightarrow$ *countable A* $\langle$ *proof* $\rangle$

lemma *set-cinsert*:

assumes *cin x A*

obtains *B* **where** *A = cinsert x B* **and** $\neg \text{cin } x \text{ } B$

$\langle$ *proof* $\rangle$

lemma *mk-disjoint-cinsert*: *cin a A* $\implies \exists B. A = \text{cinsert } a \text{ } B \wedge \neg \text{cin } a \text{ } B$ $\langle$ *proof* $\rangle$

23.3.3 *cimage*

lemma *subset-cimage-iff*: *csubset-eq B* (*cimage f A*) $\longleftrightarrow (\exists AA. \text{csubset-eq } AA \text{ } A \wedge B = \text{cimage } f \text{ } AA)$ $\langle$ *proof* $\rangle$

23.3.4 bounded quantification

lemma *cBex-simps* [*simp*, *no-atp*]:

$\bigwedge A \text{ } P \text{ } Q. \text{cBex } A (\lambda x. P \text{ } x \wedge Q) = (\text{cBex } A \text{ } P \wedge Q)$

$\bigwedge A \text{ } P \text{ } Q. \text{cBex } A (\lambda x. P \wedge Q \text{ } x) = (P \wedge \text{cBex } A \text{ } Q)$

$\bigwedge P. \text{cBex } \text{empty } P = \text{False}$

$\bigwedge a \text{ } B \text{ } P. \text{cBex } (\text{cinsert } a \text{ } B) \text{ } P = (P \text{ } a \vee \text{cBex } B \text{ } P)$

$\bigwedge A \text{ } P \text{ } f. \text{cBex } (\text{cimage } f \text{ } A) \text{ } P = \text{cBex } A (\lambda x. P \text{ } (f \text{ } x))$

$\bigwedge A \text{ } P. (\neg \text{cBex } A \text{ } P) = \text{cBall } A (\lambda x. \neg P \text{ } x)$

$\langle$ *proof* $\rangle$

lemma *cBall-simps* [*simp*, *no-atp*]:

$\bigwedge A \text{ } P \text{ } Q. \text{cBall } A (\lambda x. P \text{ } x \vee Q) = (\text{cBall } A \text{ } P \vee Q)$

$\bigwedge A \text{ } P \text{ } Q. \text{cBall } A (\lambda x. P \vee Q \text{ } x) = (P \vee \text{cBall } A \text{ } Q)$

$\bigwedge A \text{ } P \text{ } Q. \text{cBall } A (\lambda x. P \longrightarrow Q \text{ } x) = (P \longrightarrow \text{cBall } A \text{ } Q)$

$\bigwedge A \text{ } P \text{ } Q. \text{cBall } A (\lambda x. P \text{ } x \longrightarrow Q) = (\text{cBex } A \text{ } P \longrightarrow Q)$

$\bigwedge P. \text{cBall } \text{empty } P = \text{True}$

$\bigwedge a \text{ } B \text{ } P. \text{cBall } (\text{cinsert } a \text{ } B) \text{ } P = (P \text{ } a \wedge \text{cBall } B \text{ } P)$

$\bigwedge A \text{ } P \text{ } f. \text{cBall } (\text{cimage } f \text{ } A) \text{ } P = \text{cBall } A (\lambda x. P \text{ } (f \text{ } x))$

$\bigwedge A \text{ } P. (\neg \text{cBall } A \text{ } P) = \text{cBex } A (\lambda x. \neg P \text{ } x)$

$\langle$ *proof* $\rangle$

lemma *atomize-cBall*:

$(\bigwedge x. \text{cin } x \ A ==> P \ x) == \text{Trueprop } (cBall \ A \ (\lambda x. P \ x))$
 $\langle \text{proof} \rangle$

23.3.5 $cUnion$

lemma $cUNION\text{-}cimage$: $cUNION \ (cimage \ f \ A) \ g = cUNION \ A \ (g \circ f)$
 $\langle \text{proof} \rangle$

23.4 Setup for Lifting/Transfer

23.4.1 Relator and predicate properties

lift-definition $rel\text{-}cset$:: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \ cset \Rightarrow 'b \ cset \Rightarrow bool$
is $rel\text{-}set$ **parametric** $rel\text{-}set\text{-}transfer$ $\langle \text{proof} \rangle$

lemma $rel\text{-}cset\text{-}alt\text{-}def$:

$rel\text{-}cset \ R \ a \ b \longleftrightarrow$
 $(\forall t \in rcset \ a. \exists u \in rcset \ b. R \ t \ u) \wedge$
 $(\forall t \in rcset \ b. \exists u \in rcset \ a. R \ u \ t)$
 $\langle \text{proof} \rangle$

lemma $rel\text{-}cset\text{-}iff$:

$rel\text{-}cset \ R \ a \ b \longleftrightarrow$
 $(\forall t. \text{cin } t \ a \longrightarrow (\exists u. \text{cin } u \ b \wedge R \ t \ u)) \wedge$
 $(\forall t. \text{cin } t \ b \longrightarrow (\exists u. \text{cin } u \ a \wedge R \ u \ t))$
 $\langle \text{proof} \rangle$

lemma $rel\text{-}cset\text{-}cUNION$:

$\llbracket rel\text{-}cset \ Q \ A \ B; rel\text{-}fun \ Q \ (rel\text{-}cset \ R) \ f \ g \rrbracket$
 $\implies rel\text{-}cset \ R \ (cUnion \ (cimage \ f \ A)) \ (cUnion \ (cimage \ g \ B))$
 $\langle \text{proof} \rangle$

lemma $rel\text{-}cset\text{-}csingle\text{-}iff$ [$simp$]: $rel\text{-}cset \ R \ (csingle \ x) \ (csingle \ y) \longleftrightarrow R \ x \ y$
 $\langle \text{proof} \rangle$

23.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes $lifting\text{-}syntax$
begin

lemmas $cempty\text{-}parametric$ [$transfer\text{-}rule$] = $empty\text{-}transfer$ [$Transfer.transferred$]

lemma $cinsert\text{-}parametric$ [$transfer\text{-}rule$]:

$(A ==> rel\text{-}cset \ A ==> rel\text{-}cset \ A) \ cinsert \ cinsert$
 $\langle \text{proof} \rangle$

lemma $cUn\text{-}parametric$ [$transfer\text{-}rule$]:

$(rel\text{-}cset \ A ==> rel\text{-}cset \ A ==> rel\text{-}cset \ A) \ cUn \ cUn$
 $\langle \text{proof} \rangle$

lemma *cUnion-parametric* [transfer-rule]:

$(\text{rel-cset } (\text{rel-cset } A) \implies \text{rel-cset } A) \text{ cUnion cUnion}$
 $\langle \text{proof} \rangle$

lemma *cimage-parametric* [transfer-rule]:

$((A \implies B) \implies \text{rel-cset } A \implies \text{rel-cset } B) \text{ cimage cimage}$
 $\langle \text{proof} \rangle$

lemma *cBall-parametric* [transfer-rule]:

$(\text{rel-cset } A \implies (A \implies (=)) \implies (=)) \text{ cBall cBall}$
 $\langle \text{proof} \rangle$

lemma *cBex-parametric* [transfer-rule]:

$(\text{rel-cset } A \implies (A \implies (=)) \implies (=)) \text{ cBex cBex}$
 $\langle \text{proof} \rangle$

lemma *rel-cset-parametric* [transfer-rule]:

$((A \implies B \implies (=)) \implies \text{rel-cset } A \implies \text{rel-cset } B \implies (=))$
 rel-cset rel-cset
 $\langle \text{proof} \rangle$

Rules requiring bi-unique, bi-total or right-total relations

lemma *cin-parametric* [transfer-rule]:

$\text{bi-unique } A \implies (A \implies \text{rel-cset } A \implies (=)) \text{ cin cin}$
 $\langle \text{proof} \rangle$

lemma *cInt-parametric* [transfer-rule]:

$\text{bi-unique } A \implies (\text{rel-cset } A \implies \text{rel-cset } A \implies \text{rel-cset } A) \text{ cInt cInt}$
 $\langle \text{proof} \rangle$

lemma *cDiff-parametric* [transfer-rule]:

$\text{bi-unique } A \implies (\text{rel-cset } A \implies \text{rel-cset } A \implies \text{rel-cset } A) \text{ cDiff cDiff}$
 $\langle \text{proof} \rangle$

lemma *csubset-parametric* [transfer-rule]:

$\text{bi-unique } A \implies (\text{rel-cset } A \implies \text{rel-cset } A \implies (=)) \text{ csubset-eq csubset-eq}$
 $\langle \text{proof} \rangle$

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 $A :: 'a \text{ set}$

shows $|\{X. X \subseteq A \wedge \text{countable } X \wedge X \neq \{\}\}| \leq_o |\{f::\text{nat} \Rightarrow 'a. \text{range } f \subseteq A\}|$
 $\langle \text{proof} \rangle$

lemma *card-of-countable-sets-Func:*

$|\{X. X \subseteq A \wedge \text{countable } X \wedge X \neq \{\}\}| \leq_o |A| \hat{\sim}_c \text{natLeq}$
 $\langle \text{proof} \rangle$

lemma *ordLeq-countable-subsets:*

$|A| \leq_o |\{X. X \subseteq A \wedge \text{countable } X\}|$
 $\langle \text{proof} \rangle$

end

lemma *finite-countable-subset:*

finite $\{X. X \subseteq A \wedge \text{countable } X\} \longleftrightarrow \text{finite } A$
 $\langle \text{proof} \rangle$

lemma *rcset-to-rcset:* $\text{countable } A \implies \text{rcset } (\text{the-inv rcset } A) = A$

including *cset.lifting*

$\langle \text{proof} \rangle$

lemma *Collect-Int-Times:* $\{(x, y). R \ x \ y\} \cap A \times B = \{(x, y). R \ x \ y \wedge x \in A \wedge y \in B\}$

$\langle \text{proof} \rangle$

lemma *rel-cset-aux:*

$(\forall t \in \text{rcset } a. \exists u \in \text{rcset } b. R \ t \ u) \wedge (\forall t \in \text{rcset } b. \exists u \in \text{rcset } a. R \ u \ t) \longleftrightarrow$
 $((\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} \text{ (cimage fst)})^{-1-1} \text{ OO}$

$\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} \text{ (cimage snd)}) \ a \ b \text{ (is ?L = ?R)}$
 $\langle \text{proof} \rangle$ **including** *cset.lifting*

$\langle \text{proof} \rangle$

context

includes *cardinal-syntax*

begin

bnf $'a \text{ cset}$

map: *cimage*

sets: *rcset*

bd: *natLeq*

wits: *cempty*

rel: *rel-cset*

$\langle \text{proof} \rangle$ **including** *cset.lifting* $\langle \text{proof} \rangle$ **including** *cset.lifting* $\langle \text{proof} \rangle$

including *cset.lifting* $\langle \text{proof} \rangle$

end

end

24 Debugging facilities for code generated towards Isabelle/ML

```
theory Debug
imports Main
begin
```

```
context
begin
```

```
qualified definition trace :: String.literal  $\Rightarrow$  unit where
  [simp]: trace s = ()
```

```
qualified definition tracing :: String.literal  $\Rightarrow$  'a  $\Rightarrow$  'a where
  [simp]: tracing s = id
```

```
lemma [code]:
  tracing s = (let u = trace s in id)
  <proof> definition flush :: 'a  $\Rightarrow$  unit where
  [simp]: flush x = ()
```

```
qualified definition flushing :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b where
  [simp]: flushing x = id
```

```
lemma [code, code-unfold]:
  flushing x = (let u = flush x in id)
  <proof> definition timing :: String.literal  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b where
  [simp]: timing s f x = f x
```

end

code-printing

```
constant Debug.trace  $\rightarrow$  (Eval) Output.tracing
| constant Debug.flush  $\rightarrow$  (Eval) Output.tracing/ (@{make'-string} -) — note
indirection via antiquotation
| constant Debug.timing  $\rightarrow$  (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 $P :: \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{bool}$

assumes *ex-subseq*: $\bigwedge n \ s. \text{strict-mono } (s :: \text{nat} \Rightarrow \text{nat}) \implies \exists r'. \text{strict-mono } r' \wedge P \ n \ (s \circ r')$

begin

definition *reduce* **where** $\text{reduce } s \ n = (\text{SOME } r' :: \text{nat} \Rightarrow \text{nat}. \text{strict-mono } r' \wedge P \ n \ (s \circ r'))$

lemma *subseq-reduce*[*intro, simp*]:

$\text{strict-mono } s \implies \text{strict-mono } (\text{reduce } s \ n)$

$\langle \text{proof} \rangle$

lemma *reduce-holds*:

$\text{strict-mono } s \implies P \ n \ (s \circ \text{reduce } s \ n)$

$\langle \text{proof} \rangle$

primrec *seqseq* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**

$\text{seqseq } 0 = \text{id}$

| $\text{seqseq } (\text{Suc } n) = \text{seqseq } n \circ \text{reduce } (\text{seqseq } n) \ n$

lemma *subseq-seqseq*[*intro, simp*]: $\text{strict-mono } (\text{seqseq } n)$

$\langle \text{proof} \rangle$

lemma *seqseq-holds*:

$P \ n \ (\text{seqseq } (\text{Suc } n))$

$\langle \text{proof} \rangle$

definition *diagseq* :: $\text{nat} \Rightarrow \text{nat}$ **where** $\text{diagseq } i = \text{seqseq } i \ i$

lemma *diagseq-mono*: $\text{diagseq } n < \text{diagseq } (\text{Suc } n)$

$\langle \text{proof} \rangle$

lemma *subseq-diagseq*: $\text{strict-mono } \text{diagseq}$

$\langle \text{proof} \rangle$

primrec *fold-reduce* **where**

$\text{fold-reduce } n \ 0 = \text{id}$

| $\text{fold-reduce } n \ (\text{Suc } k) = \text{fold-reduce } n \ k \circ \text{reduce } (\text{seqseq } (n + k)) \ (n + k)$

lemma *subseq-fold-reduce*[*intro, simp*]: $\text{strict-mono } (\text{fold-reduce } n \ k)$

$\langle \text{proof} \rangle$

lemma *ex-subseq-reduce-index*: $\text{seqseq } (n + k) = \text{seqseq } n \circ \text{fold-reduce } n \ k$

$\langle \text{proof} \rangle$

lemma *seqseq-fold-reduce*: $\text{seqseq } n = \text{fold-reduce } 0 \ n$

$\langle \text{proof} \rangle$

lemma *diagseq-fold-reduce*: $\text{diagseq } n = \text{fold-reduce } 0 \ n \ n$
 $\langle \text{proof} \rangle$

lemma *fold-reduce-add*: $\text{fold-reduce } 0 \ (m + n) = \text{fold-reduce } 0 \ m \circ \text{fold-reduce } m \ n$
 $\langle \text{proof} \rangle$

lemma *diagseq-add*: $\text{diagseq } (k + n) = (\text{seqseq } k \circ (\text{fold-reduce } k \ n)) \ (k + n)$
 $\langle \text{proof} \rangle$

lemma *diagseq-sub*:
assumes $m \leq n$ **shows** $\text{diagseq } n = (\text{seqseq } m \circ (\text{fold-reduce } m \ (n - m))) \ n$
 $\langle \text{proof} \rangle$

lemma *subseq-diagonal-rest*: $\text{strict-mono } (\lambda x. \text{fold-reduce } k \ x \ (k + x))$
 $\langle \text{proof} \rangle$

lemma *diagseq-seqseq*: $\text{diagseq} \circ ((+) \ k) = (\text{seqseq } k \circ (\lambda x. \text{fold-reduce } k \ x \ (k + x)))$
 $\langle \text{proof} \rangle$

lemma *diagseq-holds*:
assumes *subseq-stable*: $\bigwedge r \ s \ n. \text{strict-mono } r \implies P \ n \ s \implies P \ n \ (s \circ r)$
shows $P \ k \ (\text{diagseq} \circ ((+) \ (\text{Suc } k)))$
 $\langle \text{proof} \rangle$

end

end

26 Common discrete functions

theory *Discrete*
imports *Complex-Main*
begin

26.1 Discrete logarithm

context
begin

qualified fun *log* :: $\text{nat} \Rightarrow \text{nat}$
where $[\text{simp del}]$: $\text{log } n = (\text{if } n < 2 \text{ then } 0 \text{ else } \text{Suc } (\text{log } (n \text{ div } 2)))$

lemma *log-induct* [*consumes 1, case-names one double*]:
fixes $n :: \text{nat}$
assumes $n > 0$

assumes *one*: $P\ 1$
assumes *double*: $\bigwedge n. n \geq 2 \implies P\ (n\ \text{div}\ 2) \implies P\ n$
shows $P\ n$
 $\langle \text{proof} \rangle$

lemma *log-zero* [*simp*]: $\log\ 0 = 0$
 $\langle \text{proof} \rangle$

lemma *log-one* [*simp*]: $\log\ 1 = 0$
 $\langle \text{proof} \rangle$

lemma *log-Suc-zero* [*simp*]: $\log\ (\text{Suc}\ 0) = 0$
 $\langle \text{proof} \rangle$

lemma *log-rec*: $n \geq 2 \implies \log\ n = \text{Suc}\ (\log\ (n\ \text{div}\ 2))$
 $\langle \text{proof} \rangle$

lemma *log-twice* [*simp*]: $n \neq 0 \implies \log\ (2 * n) = \text{Suc}\ (\log\ n)$
 $\langle \text{proof} \rangle$

lemma *log-half* [*simp*]: $\log\ (n\ \text{div}\ 2) = \log\ n - 1$
 $\langle \text{proof} \rangle$

lemma *log-exp* [*simp*]: $\log\ (2^\wedge n) = n$
 $\langle \text{proof} \rangle$

lemma *log-mono*: *mono* *log*
 $\langle \text{proof} \rangle$

lemma *log-exp2-le*:
assumes $n > 0$
shows $2^\wedge \log\ n \leq n$
 $\langle \text{proof} \rangle$

lemma *log-exp2-gt*: $2 * 2^\wedge \log\ n > n$
 $\langle \text{proof} \rangle$

lemma *log-exp2-ge*: $2 * 2^\wedge \log\ n \geq n$
 $\langle \text{proof} \rangle$

lemma *log-le-iff*: $m \leq n \implies \log\ m \leq \log\ n$
 $\langle \text{proof} \rangle$

lemma *log-eqI*:
assumes $n > 0\ 2^\wedge k \leq n\ n < 2 * 2^\wedge k$
shows $\log\ n = k$
 $\langle \text{proof} \rangle$

lemma *log-altdef*: $\log\ n = (\text{if}\ n = 0\ \text{then}\ 0\ \text{else}\ \text{nat}\ \lfloor \text{Transcendental.log}\ 2\ (\text{real-of-nat}\ n) \rfloor)$

$n))]$
 $\langle \text{proof} \rangle$

26.2 Discrete square root

qualified definition $\text{sqrt} :: \text{nat} \Rightarrow \text{nat}$
where $\text{sqrt } n = \text{Max } \{m. m^2 \leq n\}$

lemma sqrt-aux :
fixes $n :: \text{nat}$
shows $\text{finite } \{m. m^2 \leq n\}$ **and** $\{m. m^2 \leq n\} \neq \{\}$
 $\langle \text{proof} \rangle$

lemma sqrt-unique :
assumes $m^2 \leq n < (\text{Suc } m)^2$
shows $\text{Discrete.sqrt } n = m$
 $\langle \text{proof} \rangle$

lemma $\text{sqrt-code}[\text{code}]$: $\text{sqrt } n = \text{Max } (\text{Set.filter } (\lambda m. m^2 \leq n) \{0..n\})$
 $\langle \text{proof} \rangle$

lemma $\text{sqrt-inverse-power2}$ $[\text{simp}]$: $\text{sqrt } (n^2) = n$
 $\langle \text{proof} \rangle$

lemma sqrt-zero $[\text{simp}]$: $\text{sqrt } 0 = 0$
 $\langle \text{proof} \rangle$

lemma sqrt-one $[\text{simp}]$: $\text{sqrt } 1 = 1$
 $\langle \text{proof} \rangle$

lemma mono-sqrt : mono sqrt
 $\langle \text{proof} \rangle$

lemma mono-sqrt' : $m \leq n \implies \text{Discrete.sqrt } m \leq \text{Discrete.sqrt } n$
 $\langle \text{proof} \rangle$

lemma $\text{sqrt-greater-zero-iff}$ $[\text{simp}]$: $\text{sqrt } n > 0 \longleftrightarrow n > 0$
 $\langle \text{proof} \rangle$

lemma sqrt-power2-le $[\text{simp}]$: $(\text{sqrt } n)^2 \leq n$
 $\langle \text{proof} \rangle$

lemma sqrt-le : $\text{sqrt } n \leq n$
 $\langle \text{proof} \rangle$

Additional facts about the discrete square root, thanks to Julian Bien-darra, Manuel Eberl

lemma $\text{Suc-sqrt-power2-gt}$: $n < (\text{Suc } (\text{Discrete.sqrt } n))^2$

$\langle proof \rangle$

lemma *le-sqrt-iff*: $x \leq Discrete.sqrt\ y \longleftrightarrow x^2 \leq y$
 $\langle proof \rangle$

lemma *le-sqrtI*: $x^2 \leq y \implies x \leq Discrete.sqrt\ y$
 $\langle proof \rangle$

lemma *sqrt-le-iff*: $Discrete.sqrt\ y \leq x \longleftrightarrow (\forall z. z^2 \leq y \longrightarrow z \leq x)$
 $\langle proof \rangle$

lemma *sqrt-leI*:
 $(\bigwedge z. z^2 \leq y \implies z \leq x) \implies Discrete.sqrt\ y \leq x$
 $\langle proof \rangle$

lemma *sqrt-Suc*:
 $Discrete.sqrt\ (Suc\ n) = (if\ \exists m. Suc\ n = m^2\ then\ Suc\ (Discrete.sqrt\ n)\ else\ Discrete.sqrt\ n)$
 $\langle proof \rangle$

end

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* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ (**infixr** $\rightarrow 60$)
where $A \rightarrow B \equiv Pi\ A\ (\lambda x. B)$

syntax
 $-Pi :: pptrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set\ ((\exists \Pi\ -\in-./\ -)\ 10)$
 $-lam :: pptrn \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)\ ((\exists \lambda\ -\in-./\ -)\ [0,0,3]\ 3)$
translations
 $\Pi\ x \in A. B \equiv CONST\ Pi\ A\ (\lambda x. B)$

$$\lambda x \in A. f \Rightarrow \text{CONST restrict } (\lambda x. f) A$$

definition *compose* :: 'a set $\Rightarrow$ ('b $\Rightarrow$ 'c) $\Rightarrow$ ('a $\Rightarrow$ 'b) $\Rightarrow$ ('a $\Rightarrow$ 'c)
where *compose* A g f = ($\lambda x \in A. g (f x)$)

27.1 Basic Properties of *Pi*

lemma *Pi-I[intro!]*: ($\bigwedge x. x \in A \Rightarrow f x \in B x$) $\Rightarrow f \in \text{Pi } A B$
 $\langle \text{proof} \rangle$

lemma *Pi-I'[simp]*: ($\bigwedge x. x \in A \longrightarrow f x \in B x$) $\Rightarrow f \in \text{Pi } A B$
 $\langle \text{proof} \rangle$

lemma *funcsetI*: ($\bigwedge x. x \in A \Rightarrow f x \in B$) $\Rightarrow f \in A \rightarrow B$
 $\langle \text{proof} \rangle$

lemma *Pi-mem*: $f \in \text{Pi } A B \Rightarrow x \in A \Rightarrow f x \in B x$
 $\langle \text{proof} \rangle$

lemma *Pi-iff*: $f \in \text{Pi } I X \longleftrightarrow (\forall i \in I. f i \in X i)$
 $\langle \text{proof} \rangle$

lemma *PiE [elim]*: $f \in \text{Pi } A B \Rightarrow (f x \in B x \Rightarrow Q) \Rightarrow (x \notin A \Rightarrow Q) \Rightarrow Q$
 $\langle \text{proof} \rangle$

lemma *Pi-cong*: ($\bigwedge w. w \in A \Rightarrow f w = g w$) $\Rightarrow f \in \text{Pi } A B \longleftrightarrow g \in \text{Pi } A B$
 $\langle \text{proof} \rangle$

lemma *funcset-id [simp]*: $(\lambda x. x) \in A \rightarrow A$
 $\langle \text{proof} \rangle$

lemma *funcset-mem*: $f \in A \rightarrow B \Rightarrow x \in A \Rightarrow f x \in B$
 $\langle \text{proof} \rangle$

lemma *funcset-image*: $f \in A \rightarrow B \Rightarrow f ' A \subseteq B$
 $\langle \text{proof} \rangle$

lemma *image-subset-iff-funcset*: $F ' A \subseteq B \longleftrightarrow F \in A \rightarrow B$
 $\langle \text{proof} \rangle$

lemma *funcset-to-empty-iff*: $A \rightarrow \{\} = (\text{if } A = \{\} \text{ then UNIV else } \{\})$
 $\langle \text{proof} \rangle$

lemma *Pi-eq-empty[simp]*: $(\Pi x \in A. B x) = \{\} \longleftrightarrow (\exists x \in A. B x = \{\})$
 $\langle \text{proof} \rangle$

lemma *Pi-empty [simp]*: $\text{Pi } \{\} B = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *Pi-Int*: $Pi\ I\ E \cap Pi\ I\ F = (\Pi\ i \in I. E\ i \cap F\ i)$
 $\langle proof \rangle$

lemma *Pi-UN*:
fixes $A :: nat \Rightarrow 'i \Rightarrow 'a\ set$
assumes *finite I*
and *mono*: $\bigwedge i\ n\ m. i \in I \implies n \leq m \implies A\ n\ i \subseteq A\ m\ i$
shows $(\bigcup n. Pi\ I\ (A\ n)) = (\Pi\ i \in I. \bigcup n. A\ n\ i)$
 $\langle proof \rangle$

lemma *Pi-UNIV [simp]*: $A \rightarrow UNIV = UNIV$
 $\langle proof \rangle$

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x. x \in A \implies B\ x \subseteq C\ x) \implies Pi\ A\ B \subseteq Pi\ A\ C$
 $\langle proof \rangle$

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \implies Pi\ A\ B \subseteq Pi\ A'\ B$
 $\langle proof \rangle$

lemma *prod-final*:
assumes *1*: $fst \circ f \in Pi\ A\ B$
and *2*: $snd \circ f \in Pi\ A\ C$
shows $f \in (\Pi\ z \in A. B\ z \times C\ z)$
 $\langle proof \rangle$

lemma *Pi-split-domain[simp]*: $x \in Pi\ (I \cup J)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x \in Pi\ J\ X$
 $\langle proof \rangle$

lemma *Pi-split-insert-domain[simp]*: $x \in Pi\ (insert\ i\ I)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x\ i \in X\ i$
 $\langle proof \rangle$

lemma *Pi-cancel-fupd-range[simp]*: $i \notin I \implies x \in Pi\ I\ (B(i := b)) \longleftrightarrow x \in Pi\ I\ B$
 $\langle proof \rangle$

lemma *Pi-cancel-fupd[simp]*: $i \notin I \implies x(i := a) \in Pi\ I\ B \longleftrightarrow x \in Pi\ I\ B$
 $\langle proof \rangle$

lemma *Pi-fupd-iff*: $i \in I \implies f \in Pi\ I\ (B(i := A)) \longleftrightarrow f \in Pi\ (I - \{i\})\ B \wedge f\ i \in A$
 $\langle proof \rangle$

27.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \rightarrow B \implies g \in B \rightarrow C \implies compose\ A\ g\ f \in A \rightarrow C$
 $\langle proof \rangle$

lemma *compose-assoc*:

assumes $f \in A \rightarrow B$

shows $\text{compose } A \ h \ (\text{compose } A \ g \ f) = \text{compose } A \ (\text{compose } B \ h \ g) \ f$

$\langle \text{proof} \rangle$

lemma *compose-eq*: $x \in A \implies \text{compose } A \ g \ f \ x = g \ (f \ x)$

$\langle \text{proof} \rangle$

lemma *surj-compose*: $f \text{ ‘ } A = B \implies g \text{ ‘ } B = C \implies \text{compose } A \ g \ f \text{ ‘ } A = C$

$\langle \text{proof} \rangle$

27.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \implies (\bigwedge i. i \in J \implies f \ i = g \ i) \implies \text{restrict } f \ I$

$= \text{restrict } g \ J$

$\langle \text{proof} \rangle$

lemma *restrictI[intro!]*: $(\bigwedge x. x \in A \implies f \ x \in B \ x) \implies (\lambda x \in A. f \ x) \in \text{Pi } A \ B$

$\langle \text{proof} \rangle$

lemma *restrict-apply[simp]*: $(\lambda y \in A. f \ y) \ x = (\text{if } x \in A \text{ then } f \ x \text{ else undefined})$

$\langle \text{proof} \rangle$

lemma *restrict-apply'*: $x \in A \implies (\lambda y \in A. f \ y) \ x = f \ x$

$\langle \text{proof} \rangle$

lemma *restrict-ext*: $(\bigwedge x. x \in A \implies f \ x = g \ x) \implies (\lambda x \in A. f \ x) = (\lambda x \in A. g \ x)$

$\langle \text{proof} \rangle$

lemma *restrict-UNIV*: $\text{restrict } f \ \text{UNIV} = f$

$\langle \text{proof} \rangle$

lemma *inj-on-restrict-eq [simp]*: $\text{inj-on } (\text{restrict } f \ A) \ A = \text{inj-on } f \ A$

$\langle \text{proof} \rangle$

lemma *Id-compose*: $f \in A \rightarrow B \implies f \in \text{extensional } A \implies \text{compose } A \ (\lambda y \in B. y)$

$f = f$

$\langle \text{proof} \rangle$

lemma *compose-Id*: $g \in A \rightarrow B \implies g \in \text{extensional } A \implies \text{compose } A \ g \ (\lambda x \in A.$

$x) = g$

$\langle \text{proof} \rangle$

lemma *image-restrict-eq [simp]*: $(\text{restrict } f \ A) \text{ ‘ } A = f \text{ ‘ } A$

$\langle \text{proof} \rangle$

lemma *restrict-restrict[simp]*: $\text{restrict } (\text{restrict } f \ A) \ B = \text{restrict } f \ (A \cap B)$

$\langle \text{proof} \rangle$

lemma *restrict-fupd[simp]*: $i \notin I \implies \text{restrict } (f \ (i := x)) \ I = \text{restrict } f \ I$
 ⟨proof⟩

lemma *restrict-upd[simp]*: $i \notin I \implies (\text{restrict } f \ I)(i := y) = \text{restrict } (f(i := y))$
 (*insert i I*)
 ⟨proof⟩

lemma *restrict-Pi-cancel*: $\text{restrict } x \ I \in \text{Pi } I \ A \longleftrightarrow x \in \text{Pi } I \ A$
 ⟨proof⟩

lemma *sum-restrict' [simp]*: $\text{sum}' (\lambda i \in I. g \ i) \ I = \text{sum}' (\lambda i. g \ i) \ I$
 ⟨proof⟩

lemma *prod-restrict' [simp]*: $\text{prod}' (\lambda i \in I. g \ i) \ I = \text{prod}' (\lambda i. g \ i) \ I$
 ⟨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 $f \in A \rightarrow B$
 and $g \in B \rightarrow A$
 and $g \circ f: \bigwedge x. x \in A \implies g \ (f \ x) = x$
 and $f \circ g: \bigwedge y. y \in B \implies f \ (g \ y) = y$
 shows *bij-betw* $f \ A \ B$
 ⟨proof⟩

lemma *bij-betw-imp-funcset*: $\text{bij-betw } f \ A \ B \implies f \in A \rightarrow B$
 ⟨proof⟩

lemma *inj-on-compose*: $\text{bij-betw } f \ A \ B \implies \text{inj-on } g \ B \implies \text{inj-on } (\text{compose } A \ g \ f)$
 A
 ⟨proof⟩

lemma *bij-betw-compose*: $\text{bij-betw } f \ A \ B \implies \text{bij-betw } g \ B \ C \implies \text{bij-betw } (\text{compose } A \ g \ f) \ A \ C$
 ⟨proof⟩

lemma *bij-betw-restrict-eq [simp]*: $\text{bij-betw } (\text{restrict } f \ A) \ A \ B = \text{bij-betw } f \ A \ B$
 ⟨proof⟩

27.5 Extensionality

lemma *extensional-empty[simp]*: $\text{extensional } \{\} = \{\lambda x. \text{undefined}\}$
 ⟨proof⟩

lemma *extensional-arb*: $f \in \text{extensional } A \implies x \notin A \implies f \ x = \text{undefined}$

$\langle \text{proof} \rangle$

lemma *restrict-extensional* [simp]: *restrict f A* $\in$ *extensional A*
 $\langle \text{proof} \rangle$

lemma *compose-extensional* [simp]: *compose A f g* $\in$ *extensional A*
 $\langle \text{proof} \rangle$

lemma *extensionalityI*:
 assumes *f* $\in$ *extensional A*
 and *g* $\in$ *extensional A*
 and $\bigwedge x. x \in A \implies f\ x = g\ x$
 shows *f* = *g*
 $\langle \text{proof} \rangle$

lemma *extensional-restrict*: *f* $\in$ *extensional A* $\implies$ *restrict f A* = *f*
 $\langle \text{proof} \rangle$

lemma *extensional-subset*: *f* $\in$ *extensional A* $\implies A \subseteq B \implies f \in$ *extensional B*
 $\langle \text{proof} \rangle$

lemma *inv-into-funcset*: *f* ‘ *A* = *B* $\implies (\lambda x \in B. \text{inv-into } A\ f\ x) \in B \rightarrow A$
 $\langle \text{proof} \rangle$

lemma *compose-inv-into-id*: *bij-betw f A B* $\implies$ *compose A* $(\lambda y \in B. \text{inv-into } A\ f\ y)$
f = $(\lambda x \in A. x)$
 $\langle \text{proof} \rangle$

lemma *compose-id-inv-into*: *f* ‘ *A* = *B* $\implies$ *compose B f* $(\lambda y \in B. \text{inv-into } A\ f\ y)$
 = $(\lambda x \in B. x)$
 $\langle \text{proof} \rangle$

lemma *extensional-insert*[intro, simp]:
 assumes *a* $\in$ *extensional (insert i I)*
 shows *a*(*i* := *b*) $\in$ *extensional (insert i I)*
 $\langle \text{proof} \rangle$

lemma *extensional-Int*[simp]: *extensional I* $\cap$ *extensional I'* = *extensional (I* $\cap$ *I')*
 $\langle \text{proof} \rangle$

lemma *extensional-UNIV*[simp]: *extensional UNIV* = *UNIV*
 $\langle \text{proof} \rangle$

lemma *restrict-extensional-sub*[intro]: *A* $\subseteq$ *B* $\implies$ *restrict f A* $\in$ *extensional B*
 $\langle \text{proof} \rangle$

lemma *extensional-insert-undefined*[intro, simp]:
a $\in$ *extensional (insert i I)* $\implies a(i := \text{undefined}) \in$ *extensional I*

$\langle \text{proof} \rangle$

lemma *extensional-insert-cancel*[intro, simp]:
 $a \in \text{extensional } I \implies a \in \text{extensional } (\text{insert } i \ I)$
 $\langle \text{proof} \rangle$

27.6 Cardinality

lemma *card-inj*: $f \in A \rightarrow B \implies \text{inj-on } f \ A \implies \text{finite } B \implies \text{card } A \leq \text{card } B$
 $\langle \text{proof} \rangle$

lemma *card-bij*:
 assumes $f \in A \rightarrow B$ *inj-on* $f \ A$
 and $g \in B \rightarrow A$ *inj-on* $g \ B$
 and *finite* A *finite* B
 shows $\text{card } A = \text{card } B$
 $\langle \text{proof} \rangle$

27.7 Extensional Function Spaces

definition *PiE* :: $'a \ \text{set} \Rightarrow ('a \Rightarrow 'b \ \text{set}) \Rightarrow ('a \Rightarrow 'b) \ \text{set}$
 where $\text{PiE } S \ T = \text{Pi } S \ T \cap \text{extensional } S$

abbreviation $\text{Pi}_E \ A \ B \equiv \text{PiE } A \ B$

syntax

$-\text{PiE} :: \text{pttrn} \Rightarrow 'a \ \text{set} \Rightarrow 'b \ \text{set} \Rightarrow ('a \Rightarrow 'b) \ \text{set} \quad ((\exists \text{Pi}_E \ - \in \cdot / \ -) \ 10)$

translations

$\text{Pi}_E \ x \in A. \ B \equiv \text{CONST } \text{Pi}_E \ A \ (\lambda x. \ B)$

abbreviation *extensional-funcset* :: $'a \ \text{set} \Rightarrow 'b \ \text{set} \Rightarrow ('a \Rightarrow 'b) \ \text{set}$ (**infixr** $\rightarrow_E$ 60)

where $A \rightarrow_E B \equiv (\text{Pi}_E \ i \in A. \ B)$

lemma *extensional-funcset-def*: $\text{extensional-funcset } S \ T = (S \rightarrow T) \cap \text{extensional } S$
 $\langle \text{proof} \rangle$

lemma *PiE-empty-domain*[simp]: $\text{Pi}_E \ \{\} \ T = \{\lambda x. \ \text{undefined}\}$
 $\langle \text{proof} \rangle$

lemma *PiE-UNIV-domain*: $\text{Pi}_E \ \text{UNIV} \ T = \text{Pi } \text{UNIV} \ T$
 $\langle \text{proof} \rangle$

lemma *PiE-empty-range*[simp]: $i \in I \implies F \ i = \{\} \implies (\text{Pi}_E \ i \in I. \ F \ i) = \{\}$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-empty-iff*: $\text{Pi}_E \ I \ F = \{\} \longleftrightarrow (\exists i \in I. \ F \ i = \{\})$
 $\langle \text{proof} \rangle$

lemma *PiE-arb*: $f \in Pi_E S T \implies x \notin S \implies f x = \text{undefined}$
 $\langle \text{proof} \rangle$

lemma *PiE-mem*: $f \in Pi_E S T \implies x \in S \implies f x \in T$
 $\langle \text{proof} \rangle$

lemma *PiE-fun-upd*: $y \in T x \implies f \in Pi_E S T \implies f(x := y) \in Pi_E (\text{insert } x S)$
 T
 $\langle \text{proof} \rangle$

lemma *fun-upd-in-PiE*: $x \notin S \implies f \in Pi_E (\text{insert } x S) T \implies f(x := \text{undefined}) \in Pi_E S T$
 $\langle \text{proof} \rangle$

lemma *PiE-insert-eq*: $Pi_E (\text{insert } x S) T = (\lambda(y, g). g(x := y)) \text{ ‘ } (T x \times Pi_E S T)$
 $\langle \text{proof} \rangle$

lemma *PiE-Int*: $Pi_E I A \cap Pi_E I B = Pi_E I (\lambda x. A x \cap B x)$
 $\langle \text{proof} \rangle$

lemma *PiE-cong*: $(\bigwedge i. i \in I \implies A i = B i) \implies Pi_E I A = Pi_E I B$
 $\langle \text{proof} \rangle$

lemma *PiE-E [elim]*:
assumes $f \in Pi_E A B$
obtains $x \in A$ **and** $f x \in B$
 $| x \notin A$ **and** $f x = \text{undefined}$
 $\langle \text{proof} \rangle$

lemma *PiE-I[intro!]*:
 $(\bigwedge x. x \in A \implies f x \in B) \implies (\bigwedge x. x \notin A \implies f x = \text{undefined}) \implies f \in Pi_E A B$
 $\langle \text{proof} \rangle$

lemma *PiE-mono*: $(\bigwedge x. x \in A \implies B x \subseteq C x) \implies Pi_E A B \subseteq Pi_E A C$
 $\langle \text{proof} \rangle$

lemma *PiE-iff*: $f \in Pi_E I X \longleftrightarrow (\forall i \in I. f i \in X i) \wedge f \in \text{extensional } I$
 $\langle \text{proof} \rangle$

lemma *restrict-PiE-iff*: $\text{restrict } f I \in Pi_E I X \longleftrightarrow (\forall i \in I. f i \in X i)$
 $\langle \text{proof} \rangle$

lemma *ext-funcset-to-sing-iff [simp]*: $A \rightarrow_E \{a\} = \{\lambda x \in A. a\}$
 $\langle \text{proof} \rangle$

lemma *PiE-restrict[simp]*: $f \in Pi_E A B \implies \text{restrict } f A = f$
 $\langle \text{proof} \rangle$

lemma *restrict-PiE[simp]*: $\text{restrict } f \ I \in \text{PiE } I \ S \longleftrightarrow f \in \text{Pi } I \ S$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-subset*:
assumes *ne*: $\bigwedge i. i \in I \implies F \ i \neq \{\}$ $\bigwedge i. i \in I \implies F' \ i \neq \{\}$
and *eq*: $\text{PiE } I \ F = \text{PiE } I \ F'$
and $i \in I$
shows $F \ i \subseteq F' \ i$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-iff-not-empty*:
assumes *ne*: $\bigwedge i. i \in I \implies F \ i \neq \{\}$ $\bigwedge i. i \in I \implies F' \ i \neq \{\}$
shows $\text{PiE } I \ F = \text{PiE } I \ F' \longleftrightarrow (\forall i \in I. F \ i = F' \ i)$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-iff*:
 $\text{PiE } I \ F = \text{PiE } I \ F' \longleftrightarrow (\forall i \in I. F \ i = F' \ i) \vee ((\exists i \in I. F \ i = \{\}) \wedge (\exists i \in I. F' \ i = \{\}))$
 $\langle \text{proof} \rangle$

lemma *extensional-funcset-fun-upd-restricts-rangeI*:
 $\forall y \in S. f \ x \neq f \ y \implies f \in (\text{insert } x \ S) \rightarrow_E T \implies f(x := \text{undefined}) \in S \rightarrow_E (T - \{f \ x\})$
 $\langle \text{proof} \rangle$

lemma *extensional-funcset-fun-upd-extends-rangeI*:
assumes $a \in T \ f \in S \rightarrow_E (T - \{a\})$
shows $f(x := a) \in \text{insert } x \ S \rightarrow_E T$
 $\langle \text{proof} \rangle$

lemma *subset-PiE*:
 $\text{PiE } I \ S \subseteq \text{PiE } I \ T \longleftrightarrow \text{PiE } I \ S = \{\} \vee (\forall i \in I. S \ i \subseteq T \ i)$ (**is** ?lhs $\longleftrightarrow$ - $\vee$?rhs)
 $\langle \text{proof} \rangle$

lemma *PiE-eq*:
 $\text{PiE } I \ S = \text{PiE } I \ T \longleftrightarrow \text{PiE } I \ S = \{\} \wedge \text{PiE } I \ T = \{\} \vee (\forall i \in I. S \ i = T \ i)$
 $\langle \text{proof} \rangle$

lemma *PiE-UNIV [simp]*: $\text{PiE } \text{UNIV } (\lambda i. \text{UNIV}) = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *image-projection-PiE*:
 $(\lambda f. f \ i) \cdot (\text{PiE } I \ S) = (\text{if } \text{PiE } I \ S = \{\} \text{ then } \{\} \text{ else if } i \in I \text{ then } S \ i \text{ else } \{\text{undefined}\})$
 $\langle \text{proof} \rangle$

lemma *PiE-singleton*:

assumes $f \in \text{extensional } A$
shows $\text{PiE } A (\lambda x. \{f x\}) = \{f\}$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-singleton*: $(\Pi_E i \in I. S i) = \{\lambda i \in I. f i\} \longleftrightarrow (\forall i \in I. S i = \{f i\})$
 $\langle \text{proof} \rangle$

lemma *PiE-over-singleton-iff*: $(\Pi_E x \in \{a\}. B x) = (\bigcup b \in B a. \{\lambda x \in \{a\}. b\})$
 $\langle \text{proof} \rangle$

lemma *all-PiE-elements*:
 $(\forall z \in \text{PiE } I S. \forall i \in I. P i (z i)) \longleftrightarrow \text{PiE } I S = \{\} \vee (\forall i \in I. \forall x \in S i. P i x)$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *PiE-ext*: $\llbracket x \in \text{PiE } k s; y \in \text{PiE } k s; \bigwedge i. i \in k \implies x i = y i \rrbracket \implies x = y$
 $\langle \text{proof} \rangle$

27.7.1 Injective Extensional Function Spaces

lemma *extensional-funcset-fun-upd-inj-onI*:

assumes $f \in S \rightarrow_E (T - \{a\})$
and *inj-on* $f S$
shows *inj-on* $(f(x := a)) S$
 $\langle \text{proof} \rangle$

lemma *extensional-funcset-extend-domain-inj-on-eq*:

assumes $x \notin S$
shows $\{f. f \in (\text{insert } x S) \rightarrow_E T \wedge \text{inj-on } f (\text{insert } x S)\} =$
 $\{\lambda(y, g). g(x := y)\} \cap \{(y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge \text{inj-on } g S\}$
 $\langle \text{proof} \rangle$

lemma *extensional-funcset-extend-domain-inj-onI*:

assumes $x \notin S$
shows *inj-on* $(\lambda(y, g). g(x := y)) \{ (y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge \text{inj-on } g S \}$
 $\langle \text{proof} \rangle$

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)
 $\langle \text{proof} \rangle$

lemma *function-factors-left*:

$(\forall x y. (g x = g y) \longrightarrow (f x = f y)) \longleftrightarrow (\exists h. f = h \circ g)$
 $\langle \text{proof} \rangle$

lemma *function-factors-right-gen:*

$(\forall x. P\ x \longrightarrow (\exists y. g\ y = f\ x)) \longleftrightarrow (\exists h. \forall x. P\ x \longrightarrow f\ x = g(h\ x))$
 $\langle proof \rangle$

lemma *function-factors-right:*

$(\forall x. \exists y. g\ y = f\ x) \longleftrightarrow (\exists h. f = g \circ h)$
 $\langle proof \rangle$

lemma *restrict-compose-right:*

$restrict\ (g \circ restrict\ f\ S)\ S = restrict\ (g \circ f)\ S$
 $\langle proof \rangle$

lemma *restrict-compose-left:*

$f\ ' S \subseteq T \implies restrict\ (restrict\ g\ T \circ f)\ S = restrict\ (g \circ f)\ S$
 $\langle proof \rangle$

27.7.3 Cardinality

lemma *finite-PiE:* $finite\ S \implies (\bigwedge i. i \in S \implies finite\ (T\ i)) \implies finite\ (\Pi_E\ i \in S. T\ i)$
 $\langle proof \rangle$

lemma *inj-combinator:* $x \notin S \implies inj-on\ (\lambda(y, g). g(x := y))\ (T\ x \times Pi_E\ S\ T)$
 $\langle proof \rangle$

lemma *card-PiE:* $finite\ S \implies card\ (\Pi_E\ i \in S. T\ i) = (\prod_{i \in S. card\ (T\ i)})$
 $\langle proof \rangle$

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* $B \neq \{\}$
shows $\exists y \in B. card\ (f\ -'\ \{y\} \cap A) * card\ B \geq card\ A$
 $\langle proof \rangle$

end

28 Partitions and Disjoint Sets

theory *Disjoint-Sets*

imports *FuncSet*

begin

lemma *mono-imp-UN-eq-last:* $mono\ A \implies (\bigcup_{i \leq n. A\ i} = A\ n$

$\langle \text{proof} \rangle$

28.1 Set of Disjoint Sets

abbreviation $\text{disjoint} :: 'a \text{ set set} \Rightarrow \text{bool}$ **where** $\text{disjoint} \equiv \text{pairwise disjoint}$

lemma disjoint-def : $\text{disjoint } A \longleftrightarrow (\forall a \in A. \forall b \in A. a \neq b \longrightarrow a \cap b = \{\})$
 $\langle \text{proof} \rangle$

lemma disjointI :
 $(\bigwedge a \ b. a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}) \Longrightarrow \text{disjoint } A$
 $\langle \text{proof} \rangle$

lemma disjointD :
 $\text{disjoint } A \Longrightarrow a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}$
 $\langle \text{proof} \rangle$

lemma disjoint-image : $\text{inj-on } f \ (\bigcup A) \Longrightarrow \text{disjoint } A \Longrightarrow \text{disjoint } ((\cdot) f \cdot A)$
 $\langle \text{proof} \rangle$

lemma **assumes** $\text{disjoint } (A \cup B)$
shows disjoint-unionD1 : $\text{disjoint } A$ **and** disjoint-unionD2 : $\text{disjoint } B$
 $\langle \text{proof} \rangle$

lemma disjoint-INT :
assumes $*$: $\bigwedge i. i \in I \Longrightarrow \text{disjoint } (F i)$
shows $\text{disjoint } \{\bigcap i \in I. X i \mid X. \forall i \in I. X i \in F i\}$
 $\langle \text{proof} \rangle$

lemma $\text{diff-Union-pairwise-disjoint}$:
assumes $\text{pairwise disjoint } \mathcal{A} \ \mathcal{B} \subseteq \mathcal{A}$
shows $\bigcup \mathcal{A} - \bigcup \mathcal{B} = \bigcup (\mathcal{A} - \mathcal{B})$
 $\langle \text{proof} \rangle$

lemma $\text{Int-Union-pairwise-disjoint}$:
assumes $\text{pairwise disjoint } (\mathcal{A} \cup \mathcal{B})$
shows $\bigcup \mathcal{A} \cap \bigcup \mathcal{B} = \bigcup (\mathcal{A} \cap \mathcal{B})$
 $\langle \text{proof} \rangle$

lemma $\text{psubset-Union-pairwise-disjoint}$:
assumes $\mathcal{B}$: $\text{pairwise disjoint } \mathcal{B}$ **and** $\mathcal{A} \subset \mathcal{B} - \{\{\}\}$
shows $\bigcup \mathcal{A} \subset \bigcup \mathcal{B}$
 $\langle \text{proof} \rangle$

28.1.1 Family of Disjoint Sets

definition $\text{disjoint-family-on} :: ('i \Rightarrow 'a \text{ set}) \Rightarrow 'i \text{ set} \Rightarrow \text{bool}$ **where**
 $\text{disjoint-family-on } A \ S \longleftrightarrow (\forall m \in S. \forall n \in S. m \neq n \longrightarrow A m \cap A n = \{\})$

abbreviation $\text{disjoint-family } A \equiv \text{disjoint-family-on } A \ \text{UNIV}$

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*:
 $\text{disjoint-family-on } A \ I \implies i \in I \implies j \in I \implies i \neq j \implies A \ i \cap A \ j = \{\}$
<proof>

lemma *disjoint-family-subset*: $\text{disjoint-family } A \implies (\bigwedge x. B \ x \subseteq A \ x) \implies \text{disjoint-family } B$
<proof>

lemma *disjoint-family-on-insert*:
 $i \notin I \implies \text{disjoint-family-on } A \ (\text{insert } i \ I) \longleftrightarrow A \ i \cap (\bigcup_{i \in I}. A \ i) = \{\} \wedge \text{disjoint-family-on } A \ I$
<proof>

lemma *disjoint-family-on-bisimulation*:
assumes *disjoint-family-on f S*
and $\bigwedge n \ m. n \in S \implies m \in S \implies n \neq m \implies f \ n \cap f \ m = \{\} \implies g \ n \cap g \ m = \{\}$
shows *disjoint-family-on g S*
<proof>

lemma *disjoint-family-on-mono*:
 $A \subseteq B \implies \text{disjoint-family-on } f \ B \implies \text{disjoint-family-on } f \ A$
<proof>

lemma *disjoint-family-Suc*:
 $(\bigwedge n. A \ n \subseteq A \ (\text{Suc } n)) \implies \text{disjoint-family } (\lambda i. A \ (\text{Suc } i) - A \ i)$
<proof>

lemma *disjoint-family-on-disjoint-image*:
 $\text{disjoint-family-on } A \ I \implies \text{disjoint } (A \text{ ‘ } I)$
<proof>

lemma *disjoint-family-on-vimageI*: $\text{disjoint-family-on } F \ I \implies \text{disjoint-family-on } (\lambda i. f \text{ – ‘ } F \ i) \ I$
<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 $\bigwedge i. i \in I \implies A\ i \neq \{\}$
shows $\text{disjoint-family-on } A\ I \iff \text{disjoint } (A\ \text{' } I) \wedge \text{inj-on } A\ I$
 $\langle \text{proof} \rangle$

lemma *card-UN-disjoint'*:
assumes $\text{disjoint-family-on } A\ I \wedge i. i \in I \implies \text{finite } (A\ i) \text{ finite } I$
shows $\text{card } (\bigcup_{i \in I}. A\ i) = (\sum_{i \in I}. \text{card } (A\ i))$
 $\langle \text{proof} \rangle$

lemma *disjoint-UN*:
assumes $F: \bigwedge i. i \in I \implies \text{disjoint } (F\ i)$ **and** $*$: $\text{disjoint-family-on } (\lambda i. \bigcup (F\ i))\ I$
shows $\text{disjoint } (\bigcup_{i \in I}. F\ i)$
 $\langle \text{proof} \rangle$

lemma *distinct-list-bind*:
assumes $\text{distinct } xs \wedge x. x \in \text{set } xs \implies \text{distinct } (f\ x)$
 $\text{disjoint-family-on } (\text{set } \circ f) (\text{set } xs)$
shows $\text{distinct } (\text{List.bind } xs\ f)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-UNION-disjoint*:
assumes $\text{disj: disjoint-family-on } A'\ I$
assumes $\text{bij: } \bigwedge i. i \in I \implies \text{bij-betw } f\ (A\ i)\ (A'\ i)$
shows $\text{bij-betw } f\ (\bigcup_{i \in I}. A\ i)\ (\bigcup_{i \in I}. A'\ i)$
 $\langle \text{proof} \rangle$

lemma *disjoint-union*: $\text{disjoint } C \implies \text{disjoint } B \implies \bigcup C \cap \bigcup B = \{\} \implies \text{disjoint } (C \cup B)$
 $\langle \text{proof} \rangle$

Sum/product of the union of a finite disjoint family

context *comm-monoid-set*
begin

lemma *UNION-disjoint-family*:
assumes $\text{finite } I$ **and** $\forall i \in I. \text{finite } (A\ i)$
and $\text{disjoint-family-on } A\ I$
shows $F\ g\ (\bigcup (A\ \text{' } I)) = F\ (\lambda x. F\ g\ (A\ x))\ I$
 $\langle \text{proof} \rangle$

lemma *Union-disjoint-sets*:
assumes $\forall A \in C. \text{finite } A$ **and** $\text{disjoint } C$
shows $F\ g\ (\bigcup C) = (F \circ F)\ g\ C$
 $\langle \text{proof} \rangle$

end

The union of an infinite disjoint family of non-empty sets is infinite.

lemma *infinite-disjoint-family-imp-infinite-UNION*:

assumes $\neg \text{finite } A \wedge x. x \in A \implies f x \neq \{\}$ *disjoint-family-on* $f A$
shows $\neg \text{finite } (\bigcup (f ` A))$
 $\langle \text{proof} \rangle$

28.2 Construct Disjoint Sequences

definition *disjointed* :: $(\text{nat} \Rightarrow 'a \text{ set}) \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$ **where**
disjointed $A n = A n - (\bigcup_{i \in \{0..<n\}} A i)$

lemma *finite-UN-disjointed-eq*: $(\bigcup_{i \in \{0..<n\}} \text{disjointed } A i) = (\bigcup_{i \in \{0..<n\}} A i)$
 $\langle \text{proof} \rangle$

lemma *UN-disjointed-eq*: $(\bigcup i. \text{disjointed } A i) = (\bigcup i. A i)$
 $\langle \text{proof} \rangle$

lemma *less-disjoint-disjointed*: $m < n \implies \text{disjointed } A m \cap \text{disjointed } A n = \{\}$
 $\langle \text{proof} \rangle$

lemma *disjoint-family-disjointed*: *disjoint-family* (*disjointed* A)
 $\langle \text{proof} \rangle$

lemma *disjointed-subset*: $\text{disjointed } A n \subseteq A n$
 $\langle \text{proof} \rangle$

lemma *disjointed-0[simp]*: $\text{disjointed } A 0 = A 0$
 $\langle \text{proof} \rangle$

lemma *disjointed-mono*: $\text{mono } A \implies \text{disjointed } A (\text{Suc } n) = A (\text{Suc } n) - A n$
 $\langle \text{proof} \rangle$

28.3 Partitions

Partitions P of a set A . We explicitly disallow empty sets.

definition *partition-on* :: $'a \text{ set} \Rightarrow 'a \text{ set set} \Rightarrow \text{bool}$
where
partition-on $A P \longleftrightarrow \bigcup P = A \wedge \text{disjoint } P \wedge \{\} \notin P$

lemma *partition-onI*:
 $\bigcup P = A \implies (\bigwedge p q. p \in P \implies q \in P \implies p \neq q \implies \text{disjnt } p q) \implies \{\} \notin P$
 $\implies \text{partition-on } A P$
 $\langle \text{proof} \rangle$

lemma *partition-onD1*: $\text{partition-on } A P \implies A = \bigcup P$
 $\langle \text{proof} \rangle$

lemma *partition-onD2*: $\text{partition-on } A P \implies \text{disjoint } P$
 $\langle \text{proof} \rangle$

lemma *partition-onD3*: $\text{partition-on } A \ P \implies \{\} \notin P$
 $\langle \text{proof} \rangle$

28.4 Constructions of partitions

lemma *partition-on-empty*: $\text{partition-on } \{\} \ P \longleftrightarrow P = \{\}$
 $\langle \text{proof} \rangle$

lemma *partition-on-space*: $A \neq \{\} \implies \text{partition-on } A \ \{A\}$
 $\langle \text{proof} \rangle$

lemma *partition-on-singletons*: $\text{partition-on } A \ ((\lambda x. \{x\}) \text{ ` } A)$
 $\langle \text{proof} \rangle$

lemma *partition-on-transform*:
assumes P : $\text{partition-on } A \ P$
assumes $F\text{-UN}$: $\bigcup (F \text{ ` } P) = F (\bigcup P)$ **and** $F\text{-disjnt}$: $\bigwedge p \ q. p \in P \implies q \in P$
 $\implies \text{disjnt } p \ q \implies \text{disjnt } (F \ p) \ (F \ q)$
shows $\text{partition-on } (F \ A) \ (F \text{ ` } P - \{\{\}\})$
 $\langle \text{proof} \rangle$

lemma *partition-on-restrict*: $\text{partition-on } A \ P \implies \text{partition-on } (B \cap A) \ ((\cap) \ B \text{ ` } P - \{\{\}\})$
 $\langle \text{proof} \rangle$

lemma *partition-on-vimage*: $\text{partition-on } A \ P \implies \text{partition-on } (f \text{ ` } A) \ ((\text{ ` }) \ f \text{ ` } P - \{\{\}\})$
 $\langle \text{proof} \rangle$

lemma *partition-on-inj-image*:
assumes P : $\text{partition-on } A \ P$ **and** f : $\text{inj-on } f \ A$
shows $\text{partition-on } (f \text{ ` } A) \ ((\text{ ` }) \ f \text{ ` } P - \{\{\}\})$
 $\langle \text{proof} \rangle$

lemma *partition-on-insert*:
assumes $\text{disjnt } p \ (\bigcup P)$
shows $\text{partition-on } A \ (\text{insert } p \ P) \longleftrightarrow \text{partition-on } (A - p) \ P \wedge p \subseteq A \wedge p \neq \{\}$
 $\langle \text{proof} \rangle$

28.5 Finiteness of partitions

lemma *finitely-many-partition-on*:
assumes $\text{finite } A$
shows $\text{finite } \{P. \text{partition-on } A \ P\}$
 $\langle \text{proof} \rangle$

lemma *finite-elements*: $\text{finite } A \implies \text{partition-on } A \ P \implies \text{finite } P$
 $\langle \text{proof} \rangle$

lemma *product-partition*:

assumes *partition-on* A P **and** $\bigwedge p. p \in P \implies \text{finite } p$
shows $\text{card } A = (\sum_{p \in P. \text{card } p})$
 $\langle \text{proof} \rangle$

28.6 Equivalence of partitions and equivalence classes

lemma *partition-on-quotient*:
assumes r : *equiv* A r
shows *partition-on* A $(A // r)$
 $\langle \text{proof} \rangle$

lemma *equiv-partition-on*:
assumes P : *partition-on* A P
shows *equiv* A $\{(x, y). \exists p \in P. x \in p \wedge y \in p\}$
 $\langle \text{proof} \rangle$

lemma *partition-on-eq-quotient*:
assumes P : *partition-on* A P
shows $A // \{(x, y). \exists p \in P. x \in p \wedge y \in p\} = P$
 $\langle \text{proof} \rangle$

lemma *partition-on-alt*: *partition-on* A $P \longleftrightarrow (\exists r. \text{equiv } A \text{ } r \wedge P = A // r)$
 $\langle \text{proof} \rangle$

28.7 Refinement of partitions

definition *refines* :: 'a set $\Rightarrow$ 'a set set $\Rightarrow$ 'a set set $\Rightarrow$ bool
where *refines* A P $Q \equiv$
 $\text{partition-on } A \text{ } P \wedge \text{partition-on } A \text{ } Q \wedge (\forall X \in P. \exists Y \in Q. X \subseteq Y)$

lemma *refines-refl*: *partition-on* A $P \implies \text{refines } A \text{ } P \text{ } P$
 $\langle \text{proof} \rangle$

lemma *refines-asym1*:
assumes *refines* A P Q *refines* A Q P
shows $P \subseteq Q$
 $\langle \text{proof} \rangle$

lemma *refines-asym*: $\llbracket \text{refines } A \text{ } P \text{ } Q; \text{refines } A \text{ } Q \text{ } P \rrbracket \implies P=Q$
 $\langle \text{proof} \rangle$

lemma *refines-trans*: $\llbracket \text{refines } A \text{ } P \text{ } Q; \text{refines } A \text{ } Q \text{ } R \rrbracket \implies \text{refines } A \text{ } P \text{ } R$
 $\langle \text{proof} \rangle$

lemma *refines-obtains-subset*:
assumes *refines* A P Q $q \in Q$
shows *partition-on* q $\{p \in P. p \subseteq q\}$
 $\langle \text{proof} \rangle$

28.8 The coarsest common refinement of a set of partitions

definition *common-refinement* :: 'a set set set $\Rightarrow$ 'a set set

where *common-refinement* $\mathcal{P} \equiv (\bigcup f \in (\Pi_E P \in \mathcal{P}. P). \{\bigcap (f \text{ ' } \mathcal{P})\}) - \{\{\}\}$

With non-extensional function space

lemma *common-refinement*: *common-refinement* $\mathcal{P} = (\bigcup f \in (\Pi P \in \mathcal{P}. P). \{\bigcap (f \text{ ' } \mathcal{P})\}) - \{\{\}\}$

(**is** ?lhs = ?rhs)

<proof>

lemma *common-refinement-exists*: $\llbracket X \in \text{common-refinement } \mathcal{P}; P \in \mathcal{P} \rrbracket \Longrightarrow \exists R \in P. X \subseteq R$

<proof>

lemma *Union-common-refinement*: $\bigcup (\text{common-refinement } \mathcal{P}) = (\bigcap P \in \mathcal{P}. \bigcup P)$

<proof>

lemma *partition-on-common-refinement*:

assumes $A: \bigwedge P. P \in \mathcal{P} \Longrightarrow \text{partition-on } A P$ **and** $\mathcal{P} \neq \{\}$

shows *partition-on* $A (\text{common-refinement } \mathcal{P})$

<proof>

lemma *refines-common-refinement*:

assumes $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{partition-on } A P$ $P \in \mathcal{P}$

shows *refines* $A (\text{common-refinement } \mathcal{P}) P$

<proof>

The common refinement is itself refined by any other

lemma *common-refinement-coarsest*:

assumes $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{partition-on } A P$ $R \text{ partition-on } A R$ $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{refines } A R P$ $\mathcal{P} \neq \{\}$

shows *refines* $A R (\text{common-refinement } \mathcal{P})$

<proof>

lemma *finite-common-refinement*:

assumes *finite* $\mathcal{P}$ $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{finite } P$

shows *finite* $(\text{common-refinement } \mathcal{P})$

<proof>

lemma *card-common-refinement*:

assumes *finite* $\mathcal{P}$ $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{finite } P$

shows *card* $(\text{common-refinement } \mathcal{P}) \leq (\prod P \in \mathcal{P}. \text{card } P)$

<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 ≡ xs ≤ ys ∧ xs ≠
(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 ($\{\mid\}$) **where** $\{\mid\} \equiv \text{bot}$

abbreviation *fsubset-eq* :: 'a fset $\Rightarrow$ 'a fset $\Rightarrow$ bool (**infix** $|\subseteq|$ 50) **where** $xs |\subseteq| ys \equiv xs \leq ys$

abbreviation *fsubset* :: 'a fset $\Rightarrow$ 'a fset $\Rightarrow$ bool (**infix** $|\subset|$ 50) **where** $xs |\subset| ys \equiv xs < ys$

abbreviation *funion* :: 'a fset $\Rightarrow$ 'a fset $\Rightarrow$ 'a fset (**infixl** $|\cup|$ 65) **where** $xs |\cup| ys \equiv \text{sup } xs \text{ } ys$

abbreviation *finter* :: 'a fset $\Rightarrow$ 'a fset $\Rightarrow$ 'a fset (**infixl** $|\cap|$ 65) **where** $xs |\cap| ys \equiv \text{inf } xs \text{ } ys$

abbreviation *fminus* :: 'a fset $\Rightarrow$ 'a fset $\Rightarrow$ 'a fset (**infixl** $|-|$ 65) **where** $xs |-| ys \equiv \text{minus } xs \text{ } ys$

instantiation *fset* :: (equal) equal

begin

definition *HOL.equal* $A \ B \longleftrightarrow A |\subseteq| B \wedge B |\subseteq| A$

instance $\langle \text{proof} \rangle$

end

instantiation *fset* :: (type) conditionally-complete-lattice

begin

context includes *lifting-syntax*

begin

lemma *right-total-Inf-fset-transfer*:

assumes $[\text{transfer-rule}]$: bi-unique A **and** $[\text{transfer-rule}]$: right-total A

shows $(\text{rel-set } (\text{rel-set } A) ==> \text{rel-set } A)$

$(\lambda S. \text{ if finite } (\bigcap S \cap \text{Collect } (\text{Domainp } A)) \text{ then } \bigcap S \cap \text{Collect } (\text{Domainp } A) \text{ else } \{\})$

$(\lambda S. \text{ if finite } (\text{Inf } S) \text{ then } \text{Inf } S \text{ else } \{\})$

$\langle \text{proof} \rangle$

lemma *Inf-fset-transfer*:

assumes $[\text{transfer-rule}]$: bi-unique A **and** $[\text{transfer-rule}]$: bi-total A

shows $(\text{rel-set } (\text{rel-set } A) ==> \text{rel-set } A) (\lambda A. \text{ if finite } (\text{Inf } A) \text{ then } \text{Inf } A \text{ else } \{\})$

$(\lambda A. \text{ if finite } (\text{Inf } A) \text{ then } \text{Inf } A \text{ else } \{\})$

$\langle \text{proof} \rangle$

lift-definition *Inf-fset* :: 'a fset set $\Rightarrow$ 'a fset **is** $\lambda A. \text{ if finite } (\text{Inf } A) \text{ then } \text{Inf } A \text{ else } \{\}$

parametric *right-total-Inf-fset-transfer* *Inf-fset-transfer* $\langle \text{proof} \rangle$

lemma *Sup-fset-transfer*:

assumes $[\text{transfer-rule}]$: bi-unique A

shows $(\text{rel-set } (\text{rel-set } A) ==> \text{rel-set } A) (\lambda A. \text{ if finite } (\text{Sup } A) \text{ then } \text{Sup } A \text{ else } \{\})$

$(\lambda A. \text{ if finite } (Sup\ A) \text{ then } Sup\ A \text{ else } \{\}) \langle proof \rangle$

lift-definition $Sup\text{-}fset :: 'a\ fset\ set \Rightarrow 'a\ fset$ **is** $\lambda A. \text{ if finite } (Sup\ A) \text{ then } Sup\ A$
 $\text{else } \{\}$

parametric $Sup\text{-}fset\text{-}transfer \langle proof \rangle$

lemma $finite\text{-}Sup: \exists z. \text{ finite } z \wedge (\forall a. a \in X \longrightarrow a \leq z) \Longrightarrow \text{ finite } (Sup\ X)$
 $\langle proof \rangle$

lemma $transfer\text{-}bdd\text{-}below[transfer\text{-}rule]: (rel\text{-}set\ (pcr\text{-}fset\ (=)) \Longrightarrow (=))\ bdd\text{-}below$
 $bdd\text{-}below$
 $\langle proof \rangle$

end

instance

$\langle proof \rangle$

end

instantiation $fset :: (finite)\ complete\text{-}lattice$

begin

lift-definition $top\text{-}fset :: 'a\ fset$ **is** $UNIV$ **parametric** $right\text{-}total\text{-}UNIV\text{-}transfer$
 $UNIV\text{-}transfer$
 $\langle proof \rangle$

instance

$\langle proof \rangle$

end

instantiation $fset :: (finite)\ complete\text{-}boolean\text{-}algebra$

begin

lift-definition $uminus\text{-}fset :: 'a\ fset \Rightarrow 'a\ fset$ **is** $uminus$
parametric $right\text{-}total\text{-}Compl\text{-}transfer\ Compl\text{-}transfer \langle proof \rangle$

instance

$\langle proof \rangle$

end

abbreviation $fUNIV :: 'a::finite\ fset$ **where** $fUNIV \equiv top$

abbreviation $fuminus :: 'a::finite\ fset \Rightarrow 'a\ fset$ $(|-| - [81]\ 80)$ **where** $|-| x \equiv$
 $uminus\ x$

declare $top\text{-}fset.rep\text{-}eq[simp]$

29.3 Other operations

lift-definition $finsert :: 'a \Rightarrow 'a \text{ fset} \Rightarrow 'a \text{ fset}$ **is** *insert* **parametric** *Lifting-Set.insert-transfer*
<proof>

syntax

-insert-fset $:: \text{args} \Rightarrow 'a \text{ fset} \ (\{|(-)|\})$

translations

$\{|x, xs|\} == \text{CONST } finsert \ x \ \{|xs|\}$
 $\{|x|\} == \text{CONST } finsert \ x \ \{|\}$

lift-definition $fmember :: 'a \Rightarrow 'a \text{ fset} \Rightarrow \text{bool}$ (**infix** $|\in| \ 50$) **is** *Set.member*
parametric *member-transfer* *<proof>*

abbreviation $notin\text{-fset} :: 'a \Rightarrow 'a \text{ fset} \Rightarrow \text{bool}$ (**infix** $|\notin| \ 50$) **where** $x |\notin| S \equiv \neg (x |\in| S)$

context includes *lifting-syntax*
begin

lift-definition $ffilter :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ fset} \Rightarrow 'a \text{ fset}$ **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* *<proof>*

lift-definition $fPow :: 'a \text{ fset} \Rightarrow 'a \text{ fset fset}$ **is** *Pow* **parametric** *Pow-transfer*
<proof>

lift-definition $fcard :: 'a \text{ fset} \Rightarrow \text{nat}$ **is** *card* **parametric** *card-transfer* *<proof>*

lift-definition $fimage :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ fset} \Rightarrow 'b \text{ fset}$ (**infixr** $|\uparrow| \ 90$) **is** *image*
parametric *image-transfer* *<proof>*

lift-definition $fthe\text{-elem} :: 'a \text{ fset} \Rightarrow 'a$ **is** *the-elem* *<proof>*

lift-definition $fbind :: 'a \text{ fset} \Rightarrow ('a \Rightarrow 'b \text{ fset}) \Rightarrow 'b \text{ fset}$ **is** *Set.bind* **parametric**
bind-transfer
<proof>

lift-definition $ffUnion :: 'a \text{ fset fset} \Rightarrow 'a \text{ fset}$ **is** *Union* **parametric** *Union-transfer*
<proof>

lift-definition $fBall :: 'a \text{ fset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **is** *Ball* **parametric** *Ball-transfer*
<proof>

lift-definition $fBex :: 'a \text{ fset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **is** *Bex* **parametric** *Bex-transfer*
<proof>

lift-definition $ffold :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ fset} \Rightarrow 'b$ **is** *Finite-Set.fold* *<proof>*

lift-definition $fset\text{-of-list} :: 'a \text{ list} \Rightarrow 'a \text{ fset}$ **is** *set* *<proof>*

lift-definition *sorted-list-of-fset* :: 'a::linorder fset $\Rightarrow$ 'a list **is** sorted-list-of-set
 <proof>

29.4 Transferred lemmas from Set.thy

```

lemmas fset-eqI = set-eqI[Transfer.transferred]
lemmas fset-eq-iff[no-atp] = set-eq-iff[Transfer.transferred]
lemmas fBallI[intro!] = ballI[Transfer.transferred]
lemmas fbspec[dest?] = bspec[Transfer.transferred]
lemmas fBallE[elim] = ballE[Transfer.transferred]
lemmas fBexI[intro] = bexI[Transfer.transferred]
lemmas rev-fBexI[intro?] = rev-bexI[Transfer.transferred]
lemmas fBexCI = bexCI[Transfer.transferred]
lemmas fBexE[elim!] = bexE[Transfer.transferred]
lemmas fBall-triv[simp] = ball-triv[Transfer.transferred]
lemmas fBex-triv[simp] = bex-triv[Transfer.transferred]
lemmas fBex-triv-one-point1[simp] = bex-triv-one-point1[Transfer.transferred]
lemmas fBex-triv-one-point2[simp] = bex-triv-one-point2[Transfer.transferred]
lemmas fBex-one-point1[simp] = bex-one-point1[Transfer.transferred]
lemmas fBex-one-point2[simp] = bex-one-point2[Transfer.transferred]
lemmas fBall-one-point1[simp] = ball-one-point1[Transfer.transferred]
lemmas fBall-one-point2[simp] = ball-one-point2[Transfer.transferred]
lemmas fBall-conj-distrib = ball-conj-distrib[Transfer.transferred]
lemmas fBex-disj-distrib = bex-disj-distrib[Transfer.transferred]
lemmas fBall-cong[fundef-cong] = ball-cong[Transfer.transferred]
lemmas fBex-cong[fundef-cong] = bex-cong[Transfer.transferred]
lemmas fsubsetI[intro!] = subsetI[Transfer.transferred]
lemmas fsubsetD[elim, intro?] = subsetD[Transfer.transferred]
lemmas rev-fsubsetD[no-atp, intro?] = rev-subsetD[Transfer.transferred]
lemmas fsubsetCE[no-atp, elim] = subsetCE[Transfer.transferred]
lemmas fsubset-eq[no-atp] = subset-eq[Transfer.transferred]
lemmas contra-fsubsetD[no-atp] = contra-subsetD[Transfer.transferred]
lemmas fsubset-refl = subset-refl[Transfer.transferred]
lemmas fsubset-trans = subset-trans[Transfer.transferred]
lemmas fset-rev-mp = rev-subsetD[Transfer.transferred]
lemmas fset-mp = subsetD[Transfer.transferred]
lemmas fsubset-not-fsubset-eq[code] = subset-not-subset-eq[Transfer.transferred]
lemmas eq-fmem-trans = eq-mem-trans[Transfer.transferred]
lemmas fsubset-antisym[intro!] = subset-antisym[Transfer.transferred]
lemmas fequalityD1 = equalityD1[Transfer.transferred]
lemmas fequalityD2 = equalityD2[Transfer.transferred]
lemmas fequalityE = equalityE[Transfer.transferred]
lemmas fequalityCE[elim] = equalityCE[Transfer.transferred]
lemmas eqfset-imp-iff = eqset-imp-iff[Transfer.transferred]
lemmas eqfelem-imp-iff = eqelem-imp-iff[Transfer.transferred]
lemmas fempty-iff[simp] = empty-iff[Transfer.transferred]
lemmas fempty-fsubsetI[iff] = empty-subsetI[Transfer.transferred]
lemmas equalsffemptyI = equals0I[Transfer.transferred]
lemmas equalsffemptyD = equals0D[Transfer.transferred]

```

```

lemmas fBall-empty[simp] = ball-empty[Transfer.transferred]
lemmas fBex-empty[simp] = bex-empty[Transfer.transferred]
lemmas fPow-iff[iff] = Pow-iff[Transfer.transferred]
lemmas fPowI = PowI[Transfer.transferred]
lemmas fPowD = PowD[Transfer.transferred]
lemmas fPow-bottom = Pow-bottom[Transfer.transferred]
lemmas fPow-top = Pow-top[Transfer.transferred]
lemmas fPow-not-empty = Pow-not-empty[Transfer.transferred]
lemmas finter-iff[simp] = Int-iff[Transfer.transferred]
lemmas finterI[intro!] = IntI[Transfer.transferred]
lemmas finterD1 = IntD1[Transfer.transferred]
lemmas finterD2 = IntD2[Transfer.transferred]
lemmas finterE[elim!] = IntE[Transfer.transferred]
lemmas funion-iff[simp] = Un-iff[Transfer.transferred]
lemmas funionI1[elim?] = UnI1[Transfer.transferred]
lemmas funionI2[elim?] = UnI2[Transfer.transferred]
lemmas funionCI[intro!] = UnCI[Transfer.transferred]
lemmas funionE[elim!] = UnE[Transfer.transferred]
lemmas fminus-iff[simp] = Diff-iff[Transfer.transferred]
lemmas fminusI[intro!] = DiffI[Transfer.transferred]
lemmas fminusD1 = DiffD1[Transfer.transferred]
lemmas fminusD2 = DiffD2[Transfer.transferred]
lemmas fminusE[elim!] = DiffE[Transfer.transferred]
lemmas finsert-iff[simp] = insert-iff[Transfer.transferred]
lemmas finsertI1 = insertI1[Transfer.transferred]
lemmas finsertI2 = insertI2[Transfer.transferred]
lemmas finsertE[elim!] = insertE[Transfer.transferred]
lemmas finsertCI[intro!] = insertCI[Transfer.transferred]
lemmas fsubset-finsert-iff = subset-insert-iff[Transfer.transferred]
lemmas finsert-ident = insert-ident[Transfer.transferred]
lemmas fsingletonI[intro!,no-atp] = singletonI[Transfer.transferred]
lemmas fsingletonD[dest!,no-atp] = singletonD[Transfer.transferred]
lemmas fsingleton-iff = singleton-iff[Transfer.transferred]
lemmas fsingleton-inject[dest!] = singleton-inject[Transfer.transferred]
lemmas fsingleton-finsert-inj-eq[iff,no-atp] = singleton-insert-inj-eq[Transfer.transferred]
lemmas fsingleton-finsert-inj-eq'[iff,no-atp] = singleton-insert-inj-eq'[Transfer.transferred]
lemmas fsubset-fsingletonD = subset-singletonD[Transfer.transferred]
lemmas fminus-single-finsert = Diff-single-insert[Transfer.transferred]
lemmas fdoubleton-eq-iff = doubleton-eq-iff[Transfer.transferred]
lemmas funion-fsingleton-iff = Un-singleton-iff[Transfer.transferred]
lemmas fsingleton-funion-iff = singleton-Un-iff[Transfer.transferred]
lemmas fimage-eqI[simp, intro] = image-eqI[Transfer.transferred]
lemmas fimageI = imageI[Transfer.transferred]
lemmas rev-fimage-eqI = rev-image-eqI[Transfer.transferred]
lemmas fimageE[elim!] = imageE[Transfer.transferred]
lemmas Compr-fimage-eq = Compr-image-eq[Transfer.transferred]
lemmas fimage-funion = image-Un[Transfer.transferred]
lemmas fimage-iff = image-iff[Transfer.transferred]
lemmas fimage-fsubset-iff[no-atp] = image-subset-iff[Transfer.transferred]

```

```

lemmas fimage-fsubsetI = image-subsetI[Transfer.transferred]
lemmas fimage-ident[simp] = image-ident[Transfer.transferred]
lemmas if-split-fmem1 = if-split-mem1[Transfer.transferred]
lemmas if-split-fmem2 = if-split-mem2[Transfer.transferred]
lemmas pfssubsetI[intro!,no-atp] = psubsetI[Transfer.transferred]
lemmas pfssubsetE[elim!,no-atp] = psubsetE[Transfer.transferred]
lemmas pfssubset-finset-iff = psubset-insert-iff[Transfer.transferred]
lemmas pfssubset-eq = psubset-eq[Transfer.transferred]
lemmas pfssubset-imp-fsubset = psubset-imp-subset[Transfer.transferred]
lemmas pfssubset-trans = psubset-trans[Transfer.transferred]
lemmas pfssubsetD = psubsetD[Transfer.transferred]
lemmas pfssubset-fsubset-trans = psubset-subset-trans[Transfer.transferred]
lemmas fsubset-pfssubset-trans = subset-psubset-trans[Transfer.transferred]
lemmas pfssubset-imp-ex-fmem = psubset-imp-ex-mem[Transfer.transferred]
lemmas fimage-fPow-mono = image-Pow-mono[Transfer.transferred]
lemmas fimage-fPow-surj = image-Pow-surj[Transfer.transferred]
lemmas fsubset-finsetI = subset-insertI[Transfer.transferred]
lemmas fsubset-finsetI2 = subset-insertI2[Transfer.transferred]
lemmas fsubset-finset = subset-insert[Transfer.transferred]
lemmas funion-upper1 = Un-upper1[Transfer.transferred]
lemmas funion-upper2 = Un-upper2[Transfer.transferred]
lemmas funion-least = Un-least[Transfer.transferred]
lemmas finter-lower1 = Int-lower1[Transfer.transferred]
lemmas finter-lower2 = Int-lower2[Transfer.transferred]
lemmas finter-greatest = Int-greatest[Transfer.transferred]
lemmas fminus-fsubset = Diff-subset[Transfer.transferred]
lemmas fminus-fsubset-conv = Diff-subset-conv[Transfer.transferred]
lemmas fsubset-fempty[simp] = subset-empty[Transfer.transferred]
lemmas not-pfssubset-fempty[iff] = not-psubset-empty[Transfer.transferred]
lemmas finset-is-funion = insert-is-Un[Transfer.transferred]
lemmas finset-not-fempty[simp] = insert-not-empty[Transfer.transferred]
lemmas fempty-not-finset = empty-not-insert[Transfer.transferred]
lemmas finset-absorb = insert-absorb[Transfer.transferred]
lemmas finset-absorb2[simp] = insert-absorb2[Transfer.transferred]
lemmas finset-commute = insert-commute[Transfer.transferred]
lemmas finset-fsubset[simp] = insert-subset[Transfer.transferred]
lemmas finset-inter-finset[simp] = insert-inter-insert[Transfer.transferred]
lemmas finset-disjoint[simp,no-atp] = insert-disjoint[Transfer.transferred]
lemmas disjoint-finset[simp,no-atp] = disjoint-insert[Transfer.transferred]
lemmas fimage-fempty[simp] = image-empty[Transfer.transferred]
lemmas fimage-finset[simp] = image-insert[Transfer.transferred]
lemmas fimage-constant = image-constant[Transfer.transferred]
lemmas fimage-constant-conv = image-constant-conv[Transfer.transferred]
lemmas fimage-fimage = image-image[Transfer.transferred]
lemmas finset-fimage[simp] = insert-image[Transfer.transferred]
lemmas fimage-is-fempty[iff] = image-is-empty[Transfer.transferred]
lemmas fempty-is-fimage[iff] = empty-is-image[Transfer.transferred]
lemmas fimage-cong = image-cong[Transfer.transferred]
lemmas fimage-finter-fsubset = image-Int-subset[Transfer.transferred]

```

```

lemmas fimage-fminus-fsubset = image-diff-subset[Transfer.transferred]
lemmas finter-absorb = Int-absorb[Transfer.transferred]
lemmas finter-left-absorb = Int-left-absorb[Transfer.transferred]
lemmas finter-commute = Int-commute[Transfer.transferred]
lemmas finter-left-commute = Int-left-commute[Transfer.transferred]
lemmas finter-assoc = Int-assoc[Transfer.transferred]
lemmas finter-ac = Int-ac[Transfer.transferred]
lemmas finter-absorb1 = Int-absorb1[Transfer.transferred]
lemmas finter-absorb2 = Int-absorb2[Transfer.transferred]
lemmas finter-fempty-left = Int-empty-left[Transfer.transferred]
lemmas finter-fempty-right = Int-empty-right[Transfer.transferred]
lemmas disjoint-iff-fnot-equal = disjoint-iff-not-equal[Transfer.transferred]
lemmas finter-funion-distrib = Int-Un-distrib[Transfer.transferred]
lemmas finter-funion-distrib2 = Int-Un-distrib2[Transfer.transferred]
lemmas finter-fsubset-iff[no-atp, simp] = Int-subset-iff[Transfer.transferred]
lemmas funion-absorb = Un-absorb[Transfer.transferred]
lemmas funion-left-absorb = Un-left-absorb[Transfer.transferred]
lemmas funion-commute = Un-commute[Transfer.transferred]
lemmas funion-left-commute = Un-left-commute[Transfer.transferred]
lemmas funion-assoc = Un-assoc[Transfer.transferred]
lemmas funion-ac = Un-ac[Transfer.transferred]
lemmas funion-absorb1 = Un-absorb1[Transfer.transferred]
lemmas funion-absorb2 = Un-absorb2[Transfer.transferred]
lemmas funion-fempty-left = Un-empty-left[Transfer.transferred]
lemmas funion-fempty-right = Un-empty-right[Transfer.transferred]
lemmas funion-finsert-left[simp] = Un-insert-left[Transfer.transferred]
lemmas funion-finsert-right[simp] = Un-insert-right[Transfer.transferred]
lemmas finter-finsert-left = Int-insert-left[Transfer.transferred]
lemmas finter-finsert-left-iffempty[simp] = Int-insert-left-if0[Transfer.transferred]
lemmas finter-finsert-left-if1[simp] = Int-insert-left-if1[Transfer.transferred]
lemmas finter-finsert-right = Int-insert-right[Transfer.transferred]
lemmas finter-finsert-right-iffempty[simp] = Int-insert-right-if0[Transfer.transferred]
lemmas finter-finsert-right-if1[simp] = Int-insert-right-if1[Transfer.transferred]
lemmas funion-finter-distrib = Un-Int-distrib[Transfer.transferred]
lemmas funion-finter-distrib2 = Un-Int-distrib2[Transfer.transferred]
lemmas funion-finter-crazy = Un-Int-crazy[Transfer.transferred]
lemmas fsubset-funion-eq = subset-Un-eq[Transfer.transferred]
lemmas funion-fempty[iff] = Un-empty[Transfer.transferred]
lemmas funion-fsubset-iff[no-atp, simp] = Un-subset-iff[Transfer.transferred]
lemmas funion-fminus-finter = Un-Diff-Int[Transfer.transferred]
lemmas ffunion-empty[simp] = Union-empty[Transfer.transferred]
lemmas ffunion-mono = Union-mono[Transfer.transferred]
lemmas ffunion-insert[simp] = Union-insert[Transfer.transferred]
lemmas fminus-finter2 = Diff-Int2[Transfer.transferred]
lemmas funion-finter-assoc-eq = Un-Int-assoc-eq[Transfer.transferred]
lemmas fBall-funion = ball-Un[Transfer.transferred]
lemmas fBex-funion = bex-Un[Transfer.transferred]
lemmas fminus-eq-fempty-iff[simp, no-atp] = Diff-eq-empty-iff[Transfer.transferred]
lemmas fminus-cancel[simp] = Diff-cancel[Transfer.transferred]

```

```

lemmas fminus-idemp[simp] = Diff-idemp[Transfer.transferred]
lemmas fminus-triv = Diff-triv[Transfer.transferred]
lemmas fempty-fminus[simp] = empty-Diff[Transfer.transferred]
lemmas fminus-fempty[simp] = Diff-empty[Transfer.transferred]
lemmas fminus-finsertffempty[simp,no-atp] = Diff-insert0[Transfer.transferred]
lemmas fminus-finsert = Diff-insert[Transfer.transferred]
lemmas fminus-finsert2 = Diff-insert2[Transfer.transferred]
lemmas finsert-fminus-if = insert-Diff-if[Transfer.transferred]
lemmas finsert-fminus1[simp] = insert-Diff1[Transfer.transferred]
lemmas finsert-fminus-single[simp] = insert-Diff-single[Transfer.transferred]
lemmas finsert-fminus = insert-Diff[Transfer.transferred]
lemmas fminus-finsert-absorb = Diff-insert-absorb[Transfer.transferred]
lemmas fminus-disjoint[simp] = Diff-disjoint[Transfer.transferred]
lemmas fminus-partition = Diff-partition[Transfer.transferred]
lemmas double-fminus = double-diff[Transfer.transferred]
lemmas funion-fminus-cancel[simp] = Un-Diff-cancel[Transfer.transferred]
lemmas funion-fminus-cancel2[simp] = Un-Diff-cancel2[Transfer.transferred]
lemmas fminus-funion = Diff-Un[Transfer.transferred]
lemmas fminus-finter = Diff-Int[Transfer.transferred]
lemmas funion-fminus = Un-Diff[Transfer.transferred]
lemmas finter-fminus = Int-Diff[Transfer.transferred]
lemmas fminus-finter-distrib = Diff-Int-distrib[Transfer.transferred]
lemmas fminus-finter-distrib2 = Diff-Int-distrib2[Transfer.transferred]
lemmas fUNIV-bool[no-atp] = UNIV-bool[Transfer.transferred]
lemmas fPow-fempty[simp] = Pow-empty[Transfer.transferred]
lemmas fPow-finsert = Pow-insert[Transfer.transferred]
lemmas funion-fPow-fsubset = Un-Pow-subset[Transfer.transferred]
lemmas fPow-finter-eq[simp] = Pow-Int-eq[Transfer.transferred]
lemmas fset-eq-fsubset = set-eq-subset[Transfer.transferred]
lemmas fsubset-iff[no-atp] = subset-iff[Transfer.transferred]
lemmas fsubset-iff-pfsubset-eq = subset-iff-psubset-eq[Transfer.transferred]
lemmas all-not-fin-conv[simp] = all-not-in-conv[Transfer.transferred]
lemmas ex-fin-conv = ex-in-conv[Transfer.transferred]
lemmas fimage-mono = image-mono[Transfer.transferred]
lemmas fPow-mono = Pow-mono[Transfer.transferred]
lemmas finsert-mono = insert-mono[Transfer.transferred]
lemmas funion-mono = Un-mono[Transfer.transferred]
lemmas finter-mono = Int-mono[Transfer.transferred]
lemmas fminus-mono = Diff-mono[Transfer.transferred]
lemmas fin-mono = in-mono[Transfer.transferred]
lemmas fthe-felem-eq[simp] = the-elem-eq[Transfer.transferred]
lemmas fLeast-mono = Least-mono[Transfer.transferred]
lemmas fbind-fbind = bind-bind[Transfer.transferred]
lemmas fempty-fbind[simp] = empty-bind[Transfer.transferred]
lemmas nonfempty-fbind-const = nonempty-bind-const[Transfer.transferred]
lemmas fbind-const = bind-const[Transfer.transferred]
lemmas fmember-filter[simp] = member-filter[Transfer.transferred]
lemmas fequalityI = equalityI[Transfer.transferred]
lemmas fset-of-list-simps[simp] = set-simps[Transfer.transferred]

```


lemmas *fset-of-list-append*[simp] = *set-append*[*Transfer.transferred*]

lemmas *fset-of-list-rev*[simp] = *set-rev*[*Transfer.transferred*]

lemmas *fset-of-list-map*[simp] = *set-map*[*Transfer.transferred*]

29.5 Additional lemmas

29.5.1 *ffUnion*

lemmas *ffUnion-union-distrib*[simp] = *Union-Un-distrib*[*Transfer.transferred*]

29.5.2 *fbind*

lemma *fbind-cong*[*fundef-cong*]: $A = B \implies (\bigwedge x. x \in B \implies f x = g x) \implies fbind A f = fbind B g$

<proof>

29.5.3 *fsingleton*

lemmas *fsingletonE* = *fsingletonD* [*elim-format*]

29.5.4 *fempty*

lemma *fempty-ffilter*[simp]: *ffilter* ($\lambda \cdot. False$) $A = \{\}$

<proof>

lemma *femptyE* [*elim!*]: $a \in \{\} \implies P$

<proof>

29.5.5 *fset*

lemmas *fset-simps*[simp] = *bot-fset.rep-eq* *finset.rep-eq*

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 *notin-fset*: $x \notin S \longleftrightarrow x \notin fset\ S$ *<proof>*

lemmas *inter-fset*[simp] = *inf-fset.rep-eq*

lemmas *union-fset*[simp] = *sup-fset.rep-eq*

lemmas *minus-fset*[simp] = *minus-fset.rep-eq*

29.5.6 *ffilter***lemma** *subset-ffilter*:

$$\text{ffilter } P \ A \ |\subseteq| \text{ffilter } Q \ A = (\forall x. x \ |\in| \ A \longrightarrow P \ x \longrightarrow Q \ x)$$

<proof>

lemma *eq-ffilter*:

$$(\text{ffilter } P \ A = \text{ffilter } Q \ A) = (\forall x. x \ |\in| \ A \longrightarrow P \ x = Q \ x)$$

<proof>

lemma *pfssubset-ffilter*:

$$(\bigwedge x. x \ |\in| \ A \Longrightarrow P \ x \Longrightarrow Q \ x) \Longrightarrow (x \ |\in| \ A \wedge \neg P \ x \wedge Q \ x) \Longrightarrow$$

$$\text{ffilter } P \ A \ |\subset| \text{ffilter } Q \ A$$

<proof>

29.5.7 *fset-of-list***lemma** *fset-of-list-filter[simp]*:

$$\text{fset-of-list } (\text{filter } P \ xs) = \text{ffilter } P \ (\text{fset-of-list } xs)$$

<proof>

lemma *fset-of-list-subset[intro]*:

$$\text{set } xs \subseteq \text{set } ys \Longrightarrow \text{fset-of-list } xs \ |\subseteq| \text{fset-of-list } ys$$

<proof>

lemma *fset-of-list-elem*: $(x \ |\in| \ \text{fset-of-list } xs) \longleftrightarrow (x \in \text{set } xs)$ *<proof>***29.5.8** *fininsert***lemma** *set-fininsert*:**assumes** $x \ |\in| \ A$ **obtains** B **where** $A = \text{fininsert } x \ B$ **and** $x \ |\notin| \ B$ *<proof>***lemma** *mk-disjoint-fininsert*: $a \ |\in| \ A \Longrightarrow \exists B. A = \text{fininsert } a \ B \wedge a \ |\notin| \ B$ *<proof>***lemma** *fininsert-eq-iff*:**assumes** $a \ |\notin| \ A$ **and** $b \ |\notin| \ B$ **shows** $(\text{fininsert } a \ A = \text{fininsert } b \ B) =$ $(\text{if } a = b \text{ then } A = B \text{ else } \exists C. A = \text{fininsert } b \ C \wedge b \ |\notin| \ C \wedge B = \text{fininsert } a \ C \wedge$ $a \ |\notin| \ C)$ *<proof>***29.5.9** *fimage***lemma** *subset-fimage-iff*: $(B \ |\subseteq| \ f \ `A) = (\exists AA. AA \ |\subseteq| \ A \wedge B = f \ `AA)$ *<proof>*

29.5.10 bounded quantification**lemma** *bex-simps* [*simp*, *no-atp*]:
$$\begin{aligned}
& \bigwedge A P Q. fBex A (\lambda x. P x \wedge Q) = (fBex A P \wedge Q) \\
& \bigwedge A P Q. fBex A (\lambda x. P \wedge Q x) = (P \wedge fBex A Q) \\
& \bigwedge P. fBex \{\|\} P = False \\
& \bigwedge a B P. fBex (finsert a B) P = (P a \vee fBex B P) \\
& \bigwedge A P f. fBex (f \mid \cdot \mid A) P = fBex A (\lambda x. P (f x)) \\
& \bigwedge A P. (\neg fBex A P) = fBall A (\lambda x. \neg P x)
\end{aligned}$$

<proof>

lemma *ball-simps* [*simp*, *no-atp*]:
$$\begin{aligned}
& \bigwedge A P Q. fBall A (\lambda x. P x \vee Q) = (fBall A P \vee Q) \\
& \bigwedge A P Q. fBall A (\lambda x. P \vee Q x) = (P \vee fBall A Q) \\
& \bigwedge A P Q. fBall A (\lambda x. P \longrightarrow Q x) = (P \longrightarrow fBall A Q) \\
& \bigwedge A P Q. fBall A (\lambda x. P x \longrightarrow Q) = (fBex A P \longrightarrow Q) \\
& \bigwedge P. fBall \{\|\} P = True \\
& \bigwedge a B P. fBall (finsert a B) P = (P a \wedge fBall B P) \\
& \bigwedge A P f. fBall (f \mid \cdot \mid A) P = fBall A (\lambda x. P (f x)) \\
& \bigwedge A P. (\neg fBall A P) = fBex A (\lambda x. \neg P x)
\end{aligned}$$

<proof>

lemma *atomize-fBall*:
$$(\bigwedge x. x \in A \implies P x) == Trueprop (fBall A (\lambda x. P x))$$

<proof>

lemma *fBall-mono*[*mono*]: $P \leq Q \implies fBall S P \leq fBall S Q$ *<proof>***lemma** *fBex-mono*[*mono*]: $P \leq Q \implies fBex S P \leq fBex S Q$ *<proof>***end****29.5.11 fcard****lemma** *fcard-fempty*:
$$fcard \{\|\} = 0$$

<proof>

lemma *fcard-finsert-disjoint*:
$$x \notin A \implies fcard (finsert x A) = Suc (fcard A)$$

<proof>

lemma *fcard-finsert-if*:
$$fcard (finsert x A) = (if x \in A then fcard A else Suc (fcard A))$$

<proof>

lemma *fcard-0-eq* [*simp*, *no-atp*]:
$$fcard A = 0 \iff A = \{\|\}$$

$\langle \text{proof} \rangle$

lemma *fcard-Suc-fminus1*:

$$x \in A \implies \text{Suc} (\text{fcard} (A \mid - \mid \{|x|\})) = \text{fcard} A$$

$\langle \text{proof} \rangle$

lemma *fcard-fminus-fsingleton*:

$$x \in A \implies \text{fcard} (A \mid - \mid \{|x|\}) = \text{fcard} A - 1$$

$\langle \text{proof} \rangle$

lemma *fcard-fminus-fsingleton-if*:

$$\text{fcard} (A \mid - \mid \{|x|\}) = (\text{if } x \in A \text{ then } \text{fcard} A - 1 \text{ else } \text{fcard} A)$$

$\langle \text{proof} \rangle$

lemma *fcard-fminus-finsert[simp]*:

assumes $a \in A$ **and** $a \notin B$

shows $\text{fcard} (A \mid - \mid \text{finsert } a \ B) = \text{fcard} (A \mid - \mid B) - 1$

$\langle \text{proof} \rangle$

lemma *fcard-finsert*: $\text{fcard} (\text{finsert } x \ A) = \text{Suc} (\text{fcard} (A \mid - \mid \{|x|\}))$

$\langle \text{proof} \rangle$

lemma *fcard-finsert-le*: $\text{fcard} A \leq \text{fcard} (\text{finsert } x \ A)$

$\langle \text{proof} \rangle$

lemma *fcard-mono*:

$$A \subseteq B \implies \text{fcard} A \leq \text{fcard} B$$

$\langle \text{proof} \rangle$

lemma *fcard-seteq*: $A \subseteq B \implies \text{fcard} B \leq \text{fcard} A \implies A = B$

$\langle \text{proof} \rangle$

lemma *pfssubset-fcard-mono*: $A \subset B \implies \text{fcard} A < \text{fcard} B$

$\langle \text{proof} \rangle$

lemma *fcard-funion-finter*:

$$\text{fcard} A + \text{fcard} B = \text{fcard} (A \mid \cup \mid B) + \text{fcard} (A \mid \cap \mid B)$$

$\langle \text{proof} \rangle$

lemma *fcard-funion-disjoint*:

$$A \mid \cap \mid B = \{\mid\} \implies \text{fcard} (A \mid \cup \mid B) = \text{fcard} A + \text{fcard} B$$

$\langle \text{proof} \rangle$

lemma *fcard-funion-fsubset*:

$$B \subseteq A \implies \text{fcard} (A \mid - \mid B) = \text{fcard} A - \text{fcard} B$$

$\langle \text{proof} \rangle$

lemma *diff-fcard-le-fcard-fminus*:

$$\text{fcard} A - \text{fcard} B \leq \text{fcard} (A \mid - \mid B)$$

$\langle proof \rangle$

lemma *fcard-fminus1-less*: $x \in A \implies \text{fcard } (A \mid - \mid \{|x|\}) < \text{fcard } A$
 $\langle proof \rangle$

lemma *fcard-fminus2-less*:
 $x \in A \implies y \in A \implies \text{fcard } (A \mid - \mid \{|x|\} \mid - \mid \{|y|\}) < \text{fcard } A$
 $\langle proof \rangle$

lemma *fcard-fminus1-le*: $\text{fcard } (A \mid - \mid \{|x|\}) \leq \text{fcard } A$
 $\langle proof \rangle$

lemma *fcard-pfssubset*: $A \mid \subseteq \mid B \implies \text{fcard } A < \text{fcard } B \implies A < B$
 $\langle proof \rangle$

29.5.12 sorted-list-of-fset

lemma *sorted-list-of-fset-simps*[*simp*]:
 $\text{set } (\text{sorted-list-of-fset } S) = \text{fset } S$
 $\text{fset-of-list } (\text{sorted-list-of-fset } S) = S$
 $\langle proof \rangle$

29.5.13 ffold

context *comp-fun-commute*

begin

lemmas *ffold-empty*[*simp*] = *fold-empty*[*Transfer.transferred*]

lemma *ffold-finsert* [*simp*]:
assumes $x \notin A$
shows $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ A)$
 $\langle proof \rangle$

lemma *ffold-fun-left-comm*:
 $f \ x \ (\text{ffold } f \ z \ A) = \text{ffold } f \ (f \ x \ z) \ A$
 $\langle proof \rangle$

lemma *ffold-finsert2*:
 $x \notin A \implies \text{ffold } f \ z \ (\text{finsert } x \ A) = \text{ffold } f \ (f \ x \ z) \ A$
 $\langle proof \rangle$

lemma *ffold-rec*:
assumes $x \in A$
shows $\text{ffold } f \ z \ A = f \ x \ (\text{ffold } f \ z \ (A \mid - \mid \{|x|\}))$
 $\langle proof \rangle$

lemma *ffold-finsert-fremove*:
 $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ (A \mid - \mid \{|x|\}))$
 $\langle proof \rangle$

end

lemma *ffold-finimage*:

assumes *inj-on g (fset A)*

shows $\text{ffold } f \ z \ (g \mid^! A) = \text{ffold } (f \circ g) \ z \ A$

<proof>

lemma *ffold-cong*:

assumes *comp-fun-commute f comp-fun-commute g*

$\bigwedge x. x \mid \in \mid A \implies f \ x = g \ x$

and $s = t$ **and** $A = B$

shows $\text{ffold } f \ s \ A = \text{ffold } g \ t \ B$

<proof>

context *comp-fun-idem*

begin

lemma *ffold-fininsert-idem*:

$\text{ffold } f \ z \ (\text{fininsert } x \ A) = f \ x \ (\text{ffold } f \ z \ A)$

<proof>

declare *ffold-fininsert [simp del] ffold-fininsert-idem [simp]*

lemma *ffold-fininsert-idem2*:

$\text{ffold } f \ z \ (\text{fininsert } x \ A) = \text{ffold } f \ (f \ x \ z) \ A$

<proof>

end

29.5.14 Group operations

locale *comm-monoid-fset = comm-monoid*

begin

sublocale *set: comm-monoid-set* *<proof>*

lift-definition $F :: ('b \Rightarrow 'a) \Rightarrow 'b \text{ fset} \Rightarrow 'a \text{ is set}.F$ *<proof>*

lemmas *cong[fundef-cong] = set.cong[Transfer.transferred]*

lemma *cong-simp[cong]*:

$\llbracket A = B; \bigwedge x. x \mid \in \mid B =_{\text{simp}} \Rightarrow g \ x = h \ x \rrbracket \implies F \ g \ A = F \ h \ B$

<proof>

end

context *comm-monoid-add* **begin**

sublocale *fsum: comm-monoid-fset plus 0*

rewrites *comm-monoid-set.F plus 0 = sum*

defines $fsum = fsum.F$
 $\langle proof \rangle$

end

29.5.15 Semilattice operations

locale $semilattice-fset = semilattice$
begin

sublocale $set: semilattice-set \langle proof \rangle$

lift-definition $F :: 'a fset \Rightarrow 'a \text{ is } set.F \langle proof \rangle$

lemma $eq-fold: F (finsert\ x\ A) = ffold\ f\ x\ A$
 $\langle proof \rangle$

lemma $singleton [simp]: F \{|x|\} = x$
 $\langle proof \rangle$

lemma $insert-not-elem: x \notin A \Longrightarrow A \neq \{|\}\Longrightarrow F (finsert\ x\ A) = x * F\ A$
 $\langle proof \rangle$

lemma $in-idem: x \in A \Longrightarrow x * F\ A = F\ A$
 $\langle proof \rangle$

lemma $insert [simp]: A \neq \{|\}\Longrightarrow F (finsert\ x\ A) = x * F\ A$
 $\langle proof \rangle$

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 $fMin = fMin.F$
 $\langle proof \rangle$

sublocale $fMax: semilattice-order-fset\ max\ greater-eq\ greater$
rewrites $semilattice-set.F\ max = Max$
defines $fMax = fMax.F$
 $\langle proof \rangle$

end

lemma *mono-fMax-commute*: $\text{mono } f \implies A \neq \{\mid\} \implies f (fMax A) = fMax (f \mid A)$
 ⟨proof⟩

lemma *mono-fMin-commute*: $\text{mono } f \implies A \neq \{\mid\} \implies f (fMin A) = fMin (f \mid A)$
 ⟨proof⟩

lemma *fMax-in[simp]*: $A \neq \{\mid\} \implies fMax A \mid A$
 ⟨proof⟩

lemma *fMin-in[simp]*: $A \neq \{\mid\} \implies fMin A \mid A$
 ⟨proof⟩

lemma *fMax-ge[simp]*: $x \mid A \implies x \leq fMax A$
 ⟨proof⟩

lemma *fMin-le[simp]*: $x \mid A \implies fMin A \leq x$
 ⟨proof⟩

lemma *fMax-eqI*: $(\bigwedge y. y \mid A \implies y \leq x) \implies x \mid A \implies fMax A = x$
 ⟨proof⟩

lemma *fMin-eqI*: $(\bigwedge y. y \mid A \implies x \leq y) \implies x \mid A \implies fMin A = x$
 ⟨proof⟩

lemma *fMax-finsert[simp]*: $fMax (finsert x A) = (\text{if } A = \{\mid\} \text{ then } x \text{ else } max x (fMax A))$
 ⟨proof⟩

lemma *fMin-finsert[simp]*: $fMin (finsert x A) = (\text{if } A = \{\mid\} \text{ then } x \text{ else } min x (fMin A))$
 ⟨proof⟩

context *linorder* **begin**

lemma *fset-linorder-max-induct[case-names fempty finsert]*:
 assumes $P \{\mid\}$
 and $\bigwedge x S. [\forall y. y \mid S \longrightarrow y < x; P S] \implies P (finsert x S)$
 shows $P S$
 ⟨proof⟩

lemma *fset-linorder-min-induct[case-names fempty finsert]*:
 assumes $P \{\mid\}$
 and $\bigwedge x S. [\forall y. y \mid S \longrightarrow y > x; P S] \implies P (finsert x S)$
 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*: $\bigwedge x\ S'. S = \text{finsert}\ x\ S' \Longrightarrow P$

shows P

<proof>

lemma *fset-induct* [*case-names empty insert*]:

assumes *fempty-case*: $P\ \{\|\}$

and *finsert-case*: $\bigwedge x\ S. P\ S \Longrightarrow P\ (\text{finsert}\ x\ S)$

shows $P\ S$

<proof>

lemma *fset-induct-stronger* [*case-names empty insert, induct type: fset*]:

assumes *empty-fset-case*: $P\ \{\|\}$

and *insert-fset-case*: $\bigwedge x\ S. \llbracket x \notin S; P\ S \rrbracket \Longrightarrow P\ (\text{finsert}\ x\ S)$

shows $P\ S$

<proof>

lemma *fset-card-induct*:

assumes *empty-fset-case*: $P\ \{\|\}$

and *card-fset-Suc-case*: $\bigwedge S\ T. \text{Suc}\ (\text{fcard}\ S) = (\text{fcard}\ T) \Longrightarrow P\ S \Longrightarrow P\ T$

shows $P\ S$

<proof>

lemma *fset-strong-cases*:

obtains $xs = \{\|\}$

$\mid ys\ x$ **where** $x \notin ys$ **and** $xs = \text{finsert}\ x\ ys$

<proof>

lemma *fset-induct2*:

$P\ \{\|\}\ \{\|\} \Longrightarrow$

$(\bigwedge x\ xs. x \notin xs \Longrightarrow P\ (\text{finsert}\ x\ xs)\ \{\|\}) \Longrightarrow$

$(\bigwedge y\ ys. y \notin ys \Longrightarrow P\ \{\|\}\ (\text{finsert}\ y\ ys)) \Longrightarrow$

$(\bigwedge x\ xs\ y\ ys. \llbracket P\ xs\ ys; x \notin xs; y \notin ys \rrbracket \Longrightarrow P\ (\text{finsert}\ x\ xs)\ (\text{finsert}\ y\ ys)) \Longrightarrow$

$P\ xs\ a\ ys\ a$

<proof>

29.8 Setup for Lifting/Transfer

29.8.1 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-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 $\exists Y. \text{finite } Y \wedge \text{rel-set } R X Y \wedge \text{rel-set } S Y Z$

<proof>

29.8.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*

begin

lemmas *fempty-transfer* [*transfer-rule*] = *empty-transfer* [*Transfer.transferred*]

lemma *finsert-transfer* [*transfer-rule*]:

(*A* $====>$ *rel-fset* *A* $====>$ *rel-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*]:
 $(\text{rel-fset } A \implies \text{rel-fset } (\text{rel-fset } A)) \text{ fPow fPow}$
 $\langle \text{proof} \rangle$

lemma *rel-fset-transfer* [*transfer-rule*]:
 $((A \implies B \implies (=)) \implies \text{rel-fset } A \implies \text{rel-fset } B \implies (=))$
 rel-fset rel-fset
 $\langle \text{proof} \rangle$

lemma *bind-transfer* [*transfer-rule*]:
 $(\text{rel-fset } A \implies (A \implies \text{rel-fset } B) \implies \text{rel-fset } B) \text{ fbind fbind}$
 $\langle \text{proof} \rangle$

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(A \implies \text{rel-fset } A \implies (=)) (|\in|) (|\in|)$
 $\langle \text{proof} \rangle$

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(\text{rel-fset } A \implies \text{rel-fset } A \implies \text{rel-fset } A) \text{ finter finter}$
 $\langle \text{proof} \rangle$

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(\text{rel-fset } A \implies \text{rel-fset } A \implies \text{rel-fset } A) (|-|) (|-|)$
 $\langle \text{proof} \rangle$

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(\text{rel-fset } A \implies \text{rel-fset } A \implies (=)) (|\subseteq|) (|\subseteq|)$
 $\langle \text{proof} \rangle$

lemma *fSup-transfer* [*transfer-rule*]:
 $\text{bi-unique } A \implies (\text{rel-set } (\text{rel-fset } A) \implies \text{rel-fset } A) \text{ Sup Sup}$
 $\langle \text{proof} \rangle$

lemma *fInf-transfer* [*transfer-rule*]:
assumes *bi-unique* *A* **and** *bi-total* *A*
shows $(\text{rel-set } (\text{rel-fset } A) \implies \text{rel-fset } A) \text{ Inf Inf}$
 $\langle \text{proof} \rangle$

lemma *ffilter-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $((A \implies (=)) \implies \text{rel-fset } A \implies \text{rel-fset } A) \text{ ffilter ffilter}$
 $\langle \text{proof} \rangle$

lemma *card-transfer* [*transfer-rule*]:
 $bi\text{-}unique\ A \implies (rel\text{-}fset\ A \implies (=))\ fcard\ fcard$
 ⟨*proof*⟩

end

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

29.9 BNF setup

context
includes *fset.lifting*
begin

lemma *rel-fset-alt*:
 $rel\text{-}fset\ R\ a\ b \longleftrightarrow (\forall t \in fset\ a.\ \exists u \in fset\ b.\ R\ t\ u) \wedge (\forall t \in fset\ b.\ \exists u \in fset\ a.\ R\ u\ t)$
 ⟨*proof*⟩

lemma *fset-to-fset*: $finite\ A \implies fset\ (the\text{-}inv\ fset\ A) = A$
 ⟨*proof*⟩

lemma *rel-fset-aux*:
 $(\forall t \in fset\ a.\ \exists u \in fset\ b.\ R\ t\ u) \wedge (\forall u \in fset\ b.\ \exists t \in fset\ a.\ R\ t\ u) \longleftrightarrow$
 $((BNF\text{-}Def.Grp\ \{a.\ fset\ a \subseteq \{(a, b).\ R\ a\ b\}\}\ (fimage\ fst))^{-1-1}\ OO$
 $BNF\text{-}Def.Grp\ \{a.\ fset\ a \subseteq \{(a, b).\ R\ a\ b\}\}\ (fimage\ snd))\ a\ b\ (\text{is}\ ?L = ?R)$
 ⟨*proof*⟩

bnf *'a fset*
map: *fimage*
sets: *fset*
bd: *natLeq*
wits: $\{\|\}$
rel: *rel-fset*
 ⟨*proof*⟩

lemma *rel-fset-fset*: $rel\text{-}set\ \chi\ (fset\ A1)\ (fset\ A2) = rel\text{-}fset\ \chi\ A1\ A2$
 ⟨*proof*⟩

end

lemmas [*simp*] = *fset.map-comp fset.map-id fset.set-map*

29.10 Size setup

context includes *fset.lifting* **begin**
lift-definition *size-fset* :: $('a \Rightarrow nat) \Rightarrow 'a\ fset \Rightarrow nat$ **is** $\lambda f.\ sum\ (Suc \circ f)$ ⟨*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

lemmas size-fset-simps[simp] =
  size-fset-def[THEN meta-eq-to-obj-eq, THEN fun-cong, THEN fun-cong,
  unfolded map-fun-def comp-def id-apply]

lemmas size-fset-overloaded-simps[simp] =
  size-fset-simps[of λ-. 0, unfolded add-0-left add-0-right,
  folded size-fset-overloaded-def]

lemma fset-size-o-map: inj f  $\implies$  size-fset g  $\circ$  fimage f = size-fset (g  $\circ$  f)
  ⟨proof⟩
including fset.lifting ⟨proof⟩

⟨ML⟩

lifting-update fset.lifting
lifting-forget fset.lifting

```

29.11 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⟩

```

29.11.1 Countability

```

lemma exists-fset-of-list:  $\exists xs. \text{fset-of-list } xs = S$ 
including fset.lifting
  ⟨proof⟩

```

```

lemma fset-of-list-surj[simp, intro]: surj fset-of-list
  ⟨proof⟩

```

```

instance fset :: (countable) countable
  ⟨proof⟩

```

29.12 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 insert* $\{\cdot\}$ (*x* :: (*'a* :: *typerep* *
-)) $\{\cdot\}$ *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* ($\lambda A. f$
A orelse Quickcheck-Exhaustive.exhaustive ($\lambda x. \text{if } x \in A \text{ then } None \text{ else } f (\text{insert}$
x A)) (*i* - 1)) (*i* - 1)))

instance $\langle \text{proof} \rangle$

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 ($\lambda A. f A \text{ orelse Quickcheck-Exhaustive.full-exhaustive}$ ($\lambda x. \text{if}$
fst *x* $\in$ *fst A* *then* *None* *else* *f* (*valtermify-finsert* *x A*)) (*i* - 1)) (*i* - 1)))

instance $\langle \text{proof} \rangle$

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* $\Rightarrow$ *natural* $\Rightarrow$ *natural* $\times$ *natural* $\Rightarrow$ (*'a* *fset* $\times$ (*unit*
 $\Rightarrow$ *term*)) $\times$ *natural* $\times$ *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  $\circ \rightarrow$  ( $\lambda x.$  random-aux-fset i j  $\circ \rightarrow$  ( $\lambda s.$  Pair
(valtermify-finset x s))))))

```

lemma [code]:

```

random-aux-fset i j =
  Quickcheck-Random.collapse (Random.select-weight [(1, Pair valterm-femptyset),
    (i, Quickcheck-Random.random j  $\circ \rightarrow$  ( $\lambda x.$  random-aux-fset (i - 1) j  $\circ \rightarrow$  ( $\lambda s.$ 
Pair (valtermify-finset x s))))))
<proof>

```

definition random-fset i = random-aux-fset i i

instance <proof>

end

end

end

30 Type of finite maps defined as a subtype of maps

theory Finite-Map

imports FSet AList Conditional-Parametricity

abbrevs (= = $\subseteq_f$

begin

30.1 Auxiliary constants and lemmas over map

parametric-constant map-add-transfer[transfer-rule]: map-add-def

parametric-constant map-of-transfer[transfer-rule]: map-of-def

context includes lifting-syntax **begin**

abbreviation rel-map :: ('b $\Rightarrow$ 'c $\Rightarrow$ bool) $\Rightarrow$ ('a $\rightarrow$ 'b) $\Rightarrow$ ('a $\rightarrow$ 'c) $\Rightarrow$ bool **where**
rel-map f $\equiv$ (=) $\implies$ rel-option f

lemma ran-transfer[transfer-rule]: (rel-map A $\implies$ rel-set A) ran ran
<proof>

lemma ran-alt-def: ran m = (the $\circ$ m) ‘ dom m

<proof>

parametric-constant *dom-transfer*[*transfer-rule*]: *dom-def*

definition *map-upd* :: $'a \Rightarrow 'b \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **where**
map-upd *k v m* = *m*(*k* $\mapsto$ *v*)

parametric-constant *map-upd-transfer*[*transfer-rule*]: *map-upd-def*

definition *map-filter* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **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* :: $'a \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **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* :: $'a \text{ set} \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **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* :: $'a \text{ set} \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **where**
map-restrict-set A = *map-filter* ($\lambda a. a \in A$)

parametric-constant *map-restrict-set-transfer*[*transfer-rule*]: *map-restrict-set-def*

definition *map-pred* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \rightarrow 'b) \Rightarrow \text{bool}$ **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* :: $'a \text{ set} \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'c)$
 $\Rightarrow \text{bool}$ **where**
rel-map-on-set S P = *eq-onp* ($\lambda x. x \in S$) $==>$ *rel-option P*

definition *set-of-map* :: $('a \rightarrow 'b) \Rightarrow ('a \times 'b) \text{ set}$ **where**
set-of-map m = {(*k*, *v*) | *k v. m k* = *Some v*}

lemma *set-of-map-alt-def*: *set-of-map* $m = (\lambda k. (k, \text{the } (m\ k))) \text{ ‘ dom } m$
 $\langle \text{proof} \rangle$

lemma *set-of-map-finite*: *finite* (*dom* m) $\implies$ *finite* (*set-of-map* m)
 $\langle \text{proof} \rangle$

lemma *set-of-map-inj*: *inj* *set-of-map*
 $\langle \text{proof} \rangle$

lemma *dom-comp*: *dom* ($m \circ_m n$) $\subseteq$ *dom* n
 $\langle \text{proof} \rangle$

lemma *dom-comp-finite*: *finite* (*dom* n) $\implies$ *finite* (*dom* (*map-comp* $m\ n$))
 $\langle \text{proof} \rangle$

parametric-constant *map-comp-transfer*[*transfer-rule*]: *map-comp-def*

end

30.2 Abstract characterisation

typedef ($'a, 'b$) *fmap* = { $m. \text{finite } (\text{dom } m)$ } :: ($'a \rightarrow 'b$) *set*
morphisms *fmlookup* *Abs-fmap*
 $\langle \text{proof} \rangle$

setup-lifting *type-definition-fmap*

lemma *dom-fmlookup-finite*[*intro, simp*]: *finite* (*dom* (*fmlookup* m))
 $\langle \text{proof} \rangle$

lemma *fmap-ext*:
assumes $\bigwedge x. \text{fmlookup } m\ x = \text{fmlookup } n\ x$
shows $m = n$
 $\langle \text{proof} \rangle$

30.3 Operations

context
includes *fset.lifting*
begin

lift-definition *fmran* :: ($'a, 'b$) *fmap* $\Rightarrow$ $'b$ *fset*
is *ran*
parametric *ran-transfer*
 $\langle \text{proof} \rangle$

lemma *fmlookup-ran-iff*: $y \in | \text{fmran } m \iff (\exists x. \text{fmlookup } m\ x = \text{Some } y)$
 $\langle \text{proof} \rangle$

lemma *fmranI*: $\text{fmlookup } m \ x = \text{Some } y \implies y \in \text{fmran } m \ \langle \text{proof} \rangle$

lemma *fmranE*[*elim*]:
assumes $y \in \text{fmran } m$
obtains x **where** $\text{fmlookup } m \ x = \text{Some } y$
 $\langle \text{proof} \rangle$

lift-definition *fmdom* :: $('a, 'b) \text{ fmap} \Rightarrow 'a \text{ fset}$
is *dom*
parametric *dom-transfer*
 $\langle \text{proof} \rangle$

lemma *fmlookup-dom-iff*: $x \in \text{fmdom } m \longleftrightarrow (\exists a. \text{fmlookup } m \ x = \text{Some } a)$
 $\langle \text{proof} \rangle$

lemma *fmdom-notI*: $\text{fmlookup } m \ x = \text{None} \implies x \notin \text{fmdom } m \ \langle \text{proof} \rangle$
lemma *fmdomI*: $\text{fmlookup } m \ x = \text{Some } y \implies x \in \text{fmdom } m \ \langle \text{proof} \rangle$
lemma *fmdom-notD*[*dest*]: $x \notin \text{fmdom } m \implies \text{fmlookup } m \ x = \text{None} \ \langle \text{proof} \rangle$

lemma *fmdomE*[*elim*]:
assumes $x \in \text{fmdom } m$
obtains y **where** $\text{fmlookup } m \ x = \text{Some } y$
 $\langle \text{proof} \rangle$

lift-definition *fmdom'* :: $('a, 'b) \text{ fmap} \Rightarrow 'a \text{ set}$
is *dom*
parametric *dom-transfer*
 $\langle \text{proof} \rangle$

lemma *fmlookup-dom'-iff*: $x \in \text{fmdom}' \ m \longleftrightarrow (\exists a. \text{fmlookup } m \ x = \text{Some } a)$
 $\langle \text{proof} \rangle$

lemma *fmdom'-notI*: $\text{fmlookup } m \ x = \text{None} \implies x \notin \text{fmdom}' \ m \ \langle \text{proof} \rangle$
lemma *fmdom'I*: $\text{fmlookup } m \ x = \text{Some } y \implies x \in \text{fmdom}' \ m \ \langle \text{proof} \rangle$
lemma *fmdom'-notD*[*dest*]: $x \notin \text{fmdom}' \ m \implies \text{fmlookup } m \ x = \text{None} \ \langle \text{proof} \rangle$

lemma *fmdom'E*[*elim*]:
assumes $x \in \text{fmdom}' \ m$
obtains $x \ y$ **where** $\text{fmlookup } m \ x = \text{Some } y$
 $\langle \text{proof} \rangle$

lemma *fmdom'-alt-def*: $\text{fmdom}' \ m = \text{fset } (\text{fmdom } m)$
 $\langle \text{proof} \rangle$

lemma *finite-fmdom'*[*simp*]: $\text{finite } (\text{fmdom}' \ m)$
 $\langle \text{proof} \rangle$

lemma *dom-fmlookup*[*simp*]: $\text{dom } (\text{fmlookup } m) = \text{fmdom}' \ m$
 $\langle \text{proof} \rangle$

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 *fmrان-empty[simp]*: fmrان fmempty = fempty <proof>

lift-definition *fmupd* :: 'a ⇒ 'b ⇒ ('a, 'b) fmap ⇒ ('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 $\bigwedge x y. \text{fmlookup } m \ x = \text{Some } y \implies P \ x$
 shows $\text{fmfilter } P \ m = m$
 $\langle \text{proof} \rangle$

lemma *fmfilter-false[simp]*:
 assumes $\bigwedge x y. \text{fmlookup } m \ x = \text{Some } y \implies \neg P \ x$
 shows $\text{fmfilter } P \ m = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma *fmfilter-comp[simp]*: $\text{fmfilter } P \ (\text{fmfilter } Q \ m) = \text{fmfilter } (\lambda x. P \ x \wedge Q \ x) \ m$
 $\langle \text{proof} \rangle$

lemma *fmfilter-comm*: $\text{fmfilter } P \ (\text{fmfilter } Q \ m) = \text{fmfilter } Q \ (\text{fmfilter } P \ m)$
 $\langle \text{proof} \rangle$

lemma *fmfilter-cong[cong]*:
 assumes $\bigwedge x y. \text{fmlookup } m \ x = \text{Some } y \implies P \ x = Q \ x$
 shows $\text{fmfilter } P \ m = \text{fmfilter } Q \ m$
 $\langle \text{proof} \rangle$

lemma *fmfilter-cong'[fundef-cong]*:
 assumes $m = n \wedge \bigwedge x. x \in \text{fmdom}' \ m \implies P \ x = Q \ x$
 shows $\text{fmfilter } P \ m = \text{fmfilter } Q \ n$
 $\langle \text{proof} \rangle$

lemma *fmfilter-upd[simp]*:
 $\text{fmfilter } P \ (\text{fmupd } x \ y \ m) = (\text{if } P \ x \text{ then } \text{fmupd } x \ y \ (\text{fmfilter } P \ m) \text{ else } \text{fmfilter } P \ m)$
 $\langle \text{proof} \rangle$

lift-definition *fmdrop* :: $'a \Rightarrow ('a, 'b) \text{fmap} \Rightarrow ('a, 'b) \text{fmap}$
 is *map-drop*
 parametric *map-drop-transfer*
 $\langle \text{proof} \rangle$

lemma *fmdrop-lookup[simp]*: $\text{fmlookup } (\text{fmdrop } a \ m) \ a = \text{None}$
 $\langle \text{proof} \rangle$

lift-definition *fmdrop-set* :: $'a \text{ set} \Rightarrow ('a, 'b) \text{fmap} \Rightarrow ('a, 'b) \text{fmap}$
 is *map-drop-set*
 parametric *map-drop-set-transfer*
 $\langle \text{proof} \rangle$

lift-definition *fmdrop-fset* :: $'a \text{ fset} \Rightarrow ('a, 'b) \text{fmap} \Rightarrow ('a, 'b) \text{fmap}$
 is *map-drop-set*
 parametric *map-drop-set-transfer*

$\langle \text{proof} \rangle$

lift-definition $\text{fmrestrict-set} :: 'a \text{ set} \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow ('a, 'b) \text{ fmap}$
is map-restrict-set
parametric $\text{map-restrict-set-transfer}$
 $\langle \text{proof} \rangle$

lift-definition $\text{fmrestrict-fset} :: 'a \text{ fset} \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow ('a, 'b) \text{ fmap}$
is map-restrict-set
parametric $\text{map-restrict-set-transfer}$
 $\langle \text{proof} \rangle$

lemma fmfilter-alt-defs :
 $\text{fmdrop } a = \text{fmfilter } (\lambda a'. a' \neq a)$
 $\text{fmdrop-set } A = \text{fmfilter } (\lambda a. a \notin A)$
 $\text{fmdrop-fset } B = \text{fmfilter } (\lambda a. a \notin B)$
 $\text{fmrestrict-set } A = \text{fmfilter } (\lambda a. a \in A)$
 $\text{fmrestrict-fset } B = \text{fmfilter } (\lambda a. a \in B)$
 $\langle \text{proof} \rangle$

lemma fmdom-drop[simp] : $\text{fmdom } (\text{fmdrop } a \ m) = \text{fmdom } m - \{a\}$ $\langle \text{proof} \rangle$
lemma fmdom'-drop[simp] : $\text{fmdom}' (\text{fmdrop } a \ m) = \text{fmdom}' m - \{a\}$ $\langle \text{proof} \rangle$
lemma $\text{fmdom'-drop-set[simp]}$: $\text{fmdom}' (\text{fmdrop-set } A \ m) = \text{fmdom}' m - A$ $\langle \text{proof} \rangle$
lemma $\text{fmdom-drop-fset[simp]}$: $\text{fmdom } (\text{fmdrop-fset } A \ m) = \text{fmdom } m - A$ $\langle \text{proof} \rangle$
lemma $\text{fmdom'-restrict-set}$: $\text{fmdom}' (\text{fmrestrict-set } A \ m) \subseteq A$ $\langle \text{proof} \rangle$
lemma $\text{fmdom-restrict-fset}$: $\text{fmdom } (\text{fmrestrict-fset } A \ m) \subseteq A$ $\langle \text{proof} \rangle$

lemma fmdrop-fmupd : $\text{fmdrop } x \ (\text{fmupd } y \ z \ m) = (\text{if } x = y \text{ then } \text{fmdrop } x \ m \text{ else } \text{fmupd } y \ z \ (\text{fmdrop } x \ m))$
 $\langle \text{proof} \rangle$

lemma fmdrop-idle : $x \notin \text{fmdom } B \implies \text{fmdrop } x \ B = B$
 $\langle \text{proof} \rangle$

lemma fmdrop-idle' : $x \notin \text{fmdom}' B \implies \text{fmdrop } x \ B = B$
 $\langle \text{proof} \rangle$

lemma fmdrop-fmupd-same : $\text{fmdrop } x \ (\text{fmupd } x \ y \ m) = \text{fmdrop } x \ m$
 $\langle \text{proof} \rangle$

lemma $\text{fmdom'-restrict-set-precise}$: $\text{fmdom}' (\text{fmrestrict-set } A \ m) = \text{fmdom}' m \cap A$
 $\langle \text{proof} \rangle$

lemma $\text{fmdom'-restrict-fset-precise}$: $\text{fmdom } (\text{fmrestrict-fset } A \ m) = \text{fmdom } m \cap A$
 $\langle \text{proof} \rangle$

lemma $\text{fmdom'-drop-fset[simp]}$: $\text{fmdom}' (\text{fmdrop-fset } A \ m) = \text{fmdom}' m - \text{fset } A$

$\langle \text{proof} \rangle$

lemma *fmdom'-restrict-fset*: $\text{fmdom}' (\text{fmrestrict-fset } A \ m) \subseteq \text{fset } A$
 $\langle \text{proof} \rangle$

lemma *fmlookup-drop[simp]*:
 $\text{fmlookup } (\text{fmdrop } a \ m) \ x = (\text{if } x \neq a \text{ then } \text{fmlookup } m \ x \text{ else } \text{None})$
 $\langle \text{proof} \rangle$

lemma *fmlookup-drop-set[simp]*:
 $\text{fmlookup } (\text{fmdrop-set } A \ m) \ x = (\text{if } x \notin A \text{ then } \text{fmlookup } m \ x \text{ else } \text{None})$
 $\langle \text{proof} \rangle$

lemma *fmlookup-drop-fset[simp]*:
 $\text{fmlookup } (\text{fmdrop-fset } A \ m) \ x = (\text{if } x \notin A \text{ then } \text{fmlookup } m \ x \text{ else } \text{None})$
 $\langle \text{proof} \rangle$

lemma *fmlookup-restrict-set[simp]*:
 $\text{fmlookup } (\text{fmrestrict-set } A \ m) \ x = (\text{if } x \in A \text{ then } \text{fmlookup } m \ x \text{ else } \text{None})$
 $\langle \text{proof} \rangle$

lemma *fmlookup-restrict-fset[simp]*:
 $\text{fmlookup } (\text{fmrestrict-fset } A \ m) \ x = (\text{if } x \in A \text{ then } \text{fmlookup } m \ x \text{ else } \text{None})$
 $\langle \text{proof} \rangle$

lemma *fmrestrict-set-dom[simp]*: $\text{fmrestrict-set } (\text{fmdom}' \ m) \ m = m$
 $\langle \text{proof} \rangle$

lemma *fmrestrict-fset-dom[simp]*: $\text{fmrestrict-fset } (\text{fmdom } m) \ m = m$
 $\langle \text{proof} \rangle$

lemma *fmdrop-empty[simp]*: $\text{fmdrop } a \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma *fmdrop-set-empty[simp]*: $\text{fmdrop-set } A \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma *fmdrop-fset-empty[simp]*: $\text{fmdrop-fset } A \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma *fmdrop-fset-fmdom[simp]*: $\text{fmdrop-fset } (\text{fmdom } A) \ A = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma *fmdrop-set-fmdom[simp]*: $\text{fmdrop-set } (\text{fmdom}' \ A) \ A = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma *fmrestrict-set-empty[simp]*: $\text{fmrestrict-set } A \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

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*

$(S - \{b\}) \ m$
 $\langle proof \rangle$

lemma *fmdrop-fmrestrict-fset[simp]*: $fmdrop \ b \ (fmrestrict-fset \ S \ m) = fmrestrict-fset$
 $(S - \{ \mid b \mid \}) \ m$
 $\langle proof \rangle$

lemma *fmdrop-idem[simp]*: $fmdrop \ a \ (fmdrop \ a \ m) = fmdrop \ a \ m$
 $\langle proof \rangle$

lemma *fmdrop-set-twice[simp]*: $fmdrop-set \ S \ (fmdrop-set \ T \ m) = fmdrop-set \ (S \cup$
 $T) \ m$
 $\langle proof \rangle$

lemma *fmdrop-fset-twice[simp]*: $fmdrop-fset \ S \ (fmdrop-fset \ T \ m) = fmdrop-fset \ (S$
 $\mid \cup \mid T) \ m$
 $\langle proof \rangle$

lemma *fmdrop-set-fmdrop[simp]*: $fmdrop-set \ S \ (fmdrop \ b \ m) = fmdrop-set \ (insert$
 $b \ S) \ m$
 $\langle proof \rangle$

lemma *fmdrop-fset-fmdrop[simp]*: $fmdrop-fset \ S \ (fmdrop \ b \ m) = fmdrop-fset \ (insert$
 $b \ S) \ m$
 $\langle proof \rangle$

lift-definition *fmadd* :: $('a, 'b) \ fmap \Rightarrow ('a, 'b) \ fmap \Rightarrow ('a, 'b) \ fmap$ (**infixl** $++_f$
 100)
is *map-add*
parametric *map-add-transfer*
 $\langle proof \rangle$

lemma *fmlookup-add[simp]*:
 $fmlookup \ (m ++_f \ n) \ x = (if \ x \ \in \ fmdom \ n \ then \ fmlookup \ n \ x \ else \ fmlookup \ m$
 $x)$
 $\langle proof \rangle$

lemma *fmdom-add[simp]*: $fmdom \ (m ++_f \ n) = fmdom \ m \ \mid \cup \mid fmdom \ n \ \langle proof \rangle$

lemma *fmdom'-add[simp]*: $fmdom' \ (m ++_f \ n) = fmdom' \ m \ \cup \ fmdom' \ n \ \langle proof \rangle$

lemma *fmadd-drop-left-dom*: $fmdrop-fset \ (fmdom \ n) \ m ++_f \ n = m ++_f \ n$
 $\langle proof \rangle$

lemma *fmadd-restrict-right-dom*: $fmrestrict-fset \ (fmdom \ n) \ (m ++_f \ n) = n$
 $\langle proof \rangle$

lemma *fmfilter-add-distrib[simp]*: $fmfilter \ P \ (m ++_f \ n) = fmfilter \ P \ m ++_f \ fm-$
 $filter \ P \ n$
 $\langle proof \rangle$

lemma *fmdrop-add-distrib[simp]*: $fmdrop\ a\ (m\ ++_f\ n) = fmdrop\ a\ m\ ++_f\ fmdrop\ a\ n$
 $\langle proof \rangle$

lemma *fmdrop-set-add-distrib[simp]*: $fmdrop\text{-}set\ A\ (m\ ++_f\ n) = fmdrop\text{-}set\ A\ m\ ++_f\ fmdrop\text{-}set\ A\ n$
 $\langle proof \rangle$

lemma *fmdrop-fset-add-distrib[simp]*: $fmdrop\text{-}fset\ A\ (m\ ++_f\ n) = fmdrop\text{-}fset\ A\ m\ ++_f\ fmdrop\text{-}fset\ A\ n$
 $\langle proof \rangle$

lemma *fmrestrict-set-add-distrib[simp]*:
 $fmrestrict\text{-}set\ A\ (m\ ++_f\ n) = fmrestrict\text{-}set\ A\ m\ ++_f\ fmrestrict\text{-}set\ A\ n$
 $\langle proof \rangle$

lemma *fmrestrict-fset-add-distrib[simp]*:
 $fmrestrict\text{-}fset\ A\ (m\ ++_f\ n) = fmrestrict\text{-}fset\ A\ m\ ++_f\ fmrestrict\text{-}fset\ A\ n$
 $\langle proof \rangle$

lemma *fmadd-empty[simp]*: $fmempty\ ++_f\ m = m\ m\ ++_f\ fmempty = m$
 $\langle proof \rangle$

lemma *fmadd-idempotent[simp]*: $m\ ++_f\ m = m$
 $\langle proof \rangle$

lemma *fmadd-assoc[simp]*: $m\ ++_f\ (n\ ++_f\ p) = m\ ++_f\ n\ ++_f\ p$
 $\langle proof \rangle$

lemma *fmadd-fmupd[simp]*: $m\ ++_f\ fmupd\ a\ b\ n = fmupd\ a\ b\ (m\ ++_f\ n)$
 $\langle proof \rangle$

lift-definition *fmpr* $pred :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a, 'b) \text{fmap} \Rightarrow bool$
is *map-pred*
parametric *map-pred-transfer*
 $\langle proof \rangle$

lemma *fmpr* $predI[intro]$:
assumes $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \Longrightarrow P\ x\ y$
shows *fmpr* $pred\ P\ m$
 $\langle proof \rangle$

lemma *fmpr* $predD[dest]$: *fmpr* $pred\ P\ m \Longrightarrow fmlookup\ m\ x = Some\ y \Longrightarrow P\ x\ y$
 $\langle proof \rangle$

lemma *fmpr* $pred\text{-}iff$: *fmpr* $pred\ P\ m \longleftrightarrow (\forall x\ y. fmlookup\ m\ x = Some\ y \longrightarrow P\ x\ y)$
 $\langle proof \rangle$

lemma *fmpred-alt-def*: $fmpred\ P\ m \longleftrightarrow fBall\ (fmdom\ m)\ (\lambda x. P\ x\ (the\ (fmlookup\ m\ x)))$
 $\langle proof \rangle$

lemma *fmpred-mono-strong*:
assumes $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \implies P\ x\ y \implies Q\ x\ y$
shows $fmpred\ P\ m \implies fmpred\ Q\ m$
 $\langle proof \rangle$

lemma *fmpred-mono[mono]*: $P \leq Q \implies fmpred\ P \leq fmpred\ Q$
 $\langle proof \rangle$

lemma *fmpred-empty[intro!, simp]*: $fmpred\ P\ fmempty$
 $\langle proof \rangle$

lemma *fmpred-upd[intro]*: $fmpred\ P\ m \implies P\ x\ y \implies fmpred\ P\ (fmupd\ x\ y\ m)$
 $\langle proof \rangle$

lemma *fmpred-updD[dest]*: $fmpred\ P\ (fmupd\ x\ y\ m) \implies P\ x\ y$
 $\langle proof \rangle$

lemma *fmpred-add[intro]*: $fmpred\ P\ m \implies fmpred\ P\ n \implies fmpred\ P\ (m\ ++_f\ n)$
 $\langle proof \rangle$

lemma *fmpred-filter[intro]*: $fmpred\ P\ m \implies fmpred\ P\ (fmfilter\ Q\ m)$
 $\langle proof \rangle$

lemma *fmpred-drop[intro]*: $fmpred\ P\ m \implies fmpred\ P\ (fmdrop\ a\ m)$
 $\langle proof \rangle$

lemma *fmpred-drop-set[intro]*: $fmpred\ P\ m \implies fmpred\ P\ (fmdrop-set\ A\ m)$
 $\langle proof \rangle$

lemma *fmpred-drop-fset[intro]*: $fmpred\ P\ m \implies fmpred\ P\ (fmdrop-fset\ A\ m)$
 $\langle proof \rangle$

lemma *fmpred-restrict-set[intro]*: $fmpred\ P\ m \implies fmpred\ P\ (fmrestrict-set\ A\ m)$
 $\langle proof \rangle$

lemma *fmpred-restrict-fset[intro]*: $fmpred\ P\ m \implies fmpred\ P\ (fmrestrict-fset\ A\ m)$
 $\langle proof \rangle$

lemma *fmpred-cases[consumes 1]*:
assumes $fmpred\ P\ m$
obtains $(none)\ fmlookup\ m\ x = None \mid (some)\ y\ \mathbf{where}\ fmlookup\ m\ x = Some\ y\ P\ x\ y$
 $\langle proof \rangle$

lift-definition *fmsubset* :: $('a, 'b)\ fmap \Rightarrow ('a, 'b)\ fmap \Rightarrow bool$ (**infix** $\subseteq_f$ 50)

is map-le
 $\langle \text{proof} \rangle$

lemma *fmsubset-alt-def*: $m \subseteq_f n \longleftrightarrow \text{fmpred } (\lambda k \ v. \text{fmlookup } n \ k = \text{Some } v) \ m$
 $\langle \text{proof} \rangle$

lemma *fmsubset-pred*: $\text{fmpred } P \ m \implies n \subseteq_f m \implies \text{fmpred } P \ n$
 $\langle \text{proof} \rangle$

lemma *fmsubset-filter-mono*: $m \subseteq_f n \implies \text{fmfilter } P \ m \subseteq_f \text{fmfilter } P \ n$
 $\langle \text{proof} \rangle$

lemma *fmsubset-drop-mono*: $m \subseteq_f n \implies \text{fmdrop } a \ m \subseteq_f \text{fmdrop } a \ n$
 $\langle \text{proof} \rangle$

lemma *fmsubset-drop-set-mono*: $m \subseteq_f n \implies \text{fmdrop-set } A \ m \subseteq_f \text{fmdrop-set } A \ n$
 $\langle \text{proof} \rangle$

lemma *fmsubset-drop-fset-mono*: $m \subseteq_f n \implies \text{fmdrop-fset } A \ m \subseteq_f \text{fmdrop-fset } A \ n$
 $\langle \text{proof} \rangle$

lemma *fmsubset-restrict-set-mono*: $m \subseteq_f n \implies \text{fmrestrict-set } A \ m \subseteq_f \text{fmrestrict-set } A \ n$
 $\langle \text{proof} \rangle$

lemma *fmsubset-restrict-fset-mono*: $m \subseteq_f n \implies \text{fmrestrict-fset } A \ m \subseteq_f \text{fmrestrict-fset } A \ n$
 $\langle \text{proof} \rangle$

lemma *fmfilter-subset[simp]*: $\text{fmfilter } P \ m \subseteq_f m$
 $\langle \text{proof} \rangle$

lemma *fmsubset-drop[simp]*: $\text{fmdrop } a \ m \subseteq_f m$
 $\langle \text{proof} \rangle$

lemma *fmsubset-drop-set[simp]*: $\text{fmdrop-set } S \ m \subseteq_f m$
 $\langle \text{proof} \rangle$

lemma *fmsubset-drop-fset[simp]*: $\text{fmdrop-fset } S \ m \subseteq_f m$
 $\langle \text{proof} \rangle$

lemma *fmsubset-restrict-set[simp]*: $\text{fmrestrict-set } S \ m \subseteq_f m$
 $\langle \text{proof} \rangle$

lemma *fmsubset-restrict-fset[simp]*: $\text{fmrestrict-fset } S \ m \subseteq_f m$
 $\langle \text{proof} \rangle$

lift-definition *fset-of-fmap* :: $('a, 'b) \text{fmap} \Rightarrow ('a \times 'b) \text{fset}$ **is** *set-of-map*

$\langle proof \rangle$

lemma *fset-of-fmap-inj*[*intro*, *simp*]: *inj fset-of-fmap*
 $\langle proof \rangle$

lemma *fset-of-fmap-iff*[*simp*]: $(a, b) \in |fset-of-fmap\ m| \longleftrightarrow fmlookup\ m\ a = Some\ b$
 $\langle proof \rangle$

lemma *fset-of-fmap-iff'*[*simp*]: $(a, b) \in fset\ (fset-of-fmap\ m) \longleftrightarrow fmlookup\ m\ a = Some\ b$
 $\langle proof \rangle$

lift-definition *fmap-of-list* :: $('a \times 'b)\ list \Rightarrow ('a, 'b)\ fmap$
is *map-of*
parametric *map-of-transfer*
 $\langle proof \rangle$

lemma *fmap-of-list-simps*[*simp*]:
 $fmap-of-list\ [] = fmempty$
 $fmap-of-list\ ((k, v) \# kvs) = fmupd\ k\ v\ (fmap-of-list\ kvs)$
 $\langle proof \rangle$

lemma *fmap-of-list-app*[*simp*]: $fmap-of-list\ (xs\ @\ ys) = fmap-of-list\ ys\ ++_f\ fmap-of-list\ xs$
 $\langle proof \rangle$

lemma *fmupd-alt-def*: $fmupd\ k\ v\ m = m\ ++_f\ fmap-of-list\ [(k, v)]$
 $\langle proof \rangle$

lemma *fmprpred-of-list*[*intro*]:
assumes $\bigwedge k\ v. (k, v) \in set\ xs \implies P\ k\ v$
shows $fmprpred\ P\ (fmap-of-list\ xs)$
 $\langle proof \rangle$

lemma *fmap-of-list-SomeD*: $fmlookup\ (fmap-of-list\ xs)\ k = Some\ v \implies (k, v) \in set\ xs$
 $\langle proof \rangle$

lemma *fmdom-fmap-of-list*[*simp*]: $fmdom\ (fmap-of-list\ xs) = fset-of-list\ (map\ fst\ xs)$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *fmrel-on-fset-alt-def*: $fmrel-on-fset\ S\ P\ m\ n \longleftrightarrow fBall\ S\ (\lambda x. rel-option\ P)$

$(fmlookup\ m\ x)\ (fmlookup\ n\ x))$
 $\langle proof \rangle$

lemma *fmrel-on-fsetI[intro]*:
 assumes $\bigwedge x. x \in S \implies rel-option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$
 shows *fmrel-on-fset* $S\ P\ m\ n$
 $\langle proof \rangle$

lemma *fmrel-on-fset-mono[mono]*: $R \leq Q \implies fmrel-on-fset\ S\ R \leq fmrel-on-fset\ S\ Q$
 $\langle proof \rangle$

lemma *fmrel-on-fsetD*: $x \in S \implies fmrel-on-fset\ S\ P\ m\ n \implies rel-option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$
 $\langle proof \rangle$

lemma *fmrel-on-fsubset*: *fmrel-on-fset* $S\ R\ m\ n \implies T \subseteq S \implies fmrel-on-fset\ T\ R\ m\ n$
 $\langle proof \rangle$

lemma *fmrel-on-fset-unionI*:
fmrel-on-fset $A\ R\ m\ n \implies fmrel-on-fset\ B\ R\ m\ n \implies fmrel-on-fset\ (A \cup B)\ R\ m\ n$
 $\langle proof \rangle$

lemma *fmrel-on-fset-updateI*:
 assumes *fmrel-on-fset* $S\ P\ m\ n\ P\ v_1\ v_2$
 shows *fmrel-on-fset* $(finsert\ k\ S)\ P\ (fmupd\ k\ v_1\ m)\ (fmupd\ k\ v_2\ n)$
 $\langle proof \rangle$

lift-definition *fmimage* :: $('a, 'b)\ fmap \Rightarrow 'a\ fset \Rightarrow 'b\ fset$ **is** $\lambda m\ S. \{b \mid a\ b. m\ a = Some\ b \wedge a \in S\}$
 $\langle proof \rangle$

lemma *fmimage-alt-def*: *fmimage* $m\ S = fmran\ (fmrestrict-fset\ S\ m)$
 $\langle proof \rangle$

lemma *fmimage-empty[simp]*: *fmimage* $m\ fempty = fempty$
 $\langle proof \rangle$

lemma *fmimage-subset-ran[simp]*: *fmimage* $m\ S \subseteq fmran\ m$
 $\langle proof \rangle$

lemma *fmimage-dom[simp]*: *fmimage* $m\ (fdom\ m) = fmran\ m$
 $\langle proof \rangle$

lemma *fmimage-inter*: *fmimage* $m\ (A \cap B) \subseteq fmimage\ m\ A \cap fmimage\ m\ B$
 $\langle proof \rangle$

lemma *fmimage-inter-dom[simp]*:

$$fmimage\ m\ (fmdom\ m\ |\cap|\ A) = fmimage\ m\ A$$

$$fmimage\ m\ (A\ |\cap|\ fmdom\ m) = fmimage\ m\ A$$

<proof>

lemma *fmimage-union[simp]*: $fmimage\ m\ (A\ |\cup|\ B) = fmimage\ m\ A\ |\cup|\ fmimage\ m\ B$

<proof>

lemma *fmimage-Union[simp]*: $fmimage\ m\ (ffUnion\ A) = ffUnion\ (fmimage\ m\ |\cap|\ A)$

<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]*: $fmimage\ (fmrestrict-fset\ B\ m)\ A = fmimage\ m\ (A\ |\cap|\ B)$

<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*: $fmran\ (fmrestrict-fset\ A\ m) = fmimage\ m\ (fmdom\ m\ |\cap|\ A)$

<proof>

lemma *fmlookup-image-iff*: $y\ |\in|\ fmimage\ m\ A \longleftrightarrow (\exists x. fmlookup\ m\ x = Some\ y \wedge x\ |\in|\ A)$

<proof>

lemma *fmimageI*: $fmlookup\ m\ x = Some\ y \implies x\ |\in|\ A \implies y\ |\in|\ fmimage\ m\ A$

<proof>

lemma *fmimageE*[*elim*]:
 assumes $y \in | \text{fmimage } m \ A$
 obtains x **where** $\text{fmlookup } m \ x = \text{Some } y \ x \in | \ A$
 $\langle \text{proof} \rangle$

lift-definition *fmcomp* :: $('b, 'c) \text{ fmap} \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow ('a, 'c) \text{ fmap}$ (**infixl** $\circ_f$ 55)
 is *map-comp*
 parametric *map-comp-transfer*
 $\langle \text{proof} \rangle$

lemma *fmlookup-comp*[*simp*]: $\text{fmlookup } (m \circ_f n) \ x = \text{Option.bind } (\text{fmlookup } n \ x)$
 $(\text{fmlookup } m)$
 $\langle \text{proof} \rangle$

end

30.4 BNF setup

lift-bnf $('a, \text{fmran}': 'b) \text{ fmap}$ [*wits*: *Map.empty*]
 for *map*: *fmmap*
 rel: *fmrel*
 $\langle \text{proof} \rangle$

declare *fmap.pred-mono*[*mono*]

lemma *fmran'-alt-def*: $\text{fmran}' \ m = \text{fset } (\text{fmran } m)$
including *fset.lifting*
 $\langle \text{proof} \rangle$

lemma *fmlookup-ran'-iff*: $y \in \text{fmran}' \ m \longleftrightarrow (\exists x. \text{fmlookup } m \ x = \text{Some } y)$
 $\langle \text{proof} \rangle$

lemma *fmran'I*: $\text{fmlookup } m \ x = \text{Some } y \implies y \in \text{fmran}' \ m$ $\langle \text{proof} \rangle$

lemma *fmran'E*[*elim*]:
 assumes $y \in \text{fmran}' \ m$
 obtains x **where** $\text{fmlookup } m \ x = \text{Some } y$
 $\langle \text{proof} \rangle$

lemma *fmrel-iff*: $\text{fmrel } R \ m \ n \longleftrightarrow (\forall x. \text{rel-option } R \ (\text{fmlookup } m \ x) \ (\text{fmlookup } n \ x))$
 $\langle \text{proof} \rangle$

lemma *fmrelI*[*intro*]:
 assumes $\bigwedge x. \text{rel-option } R \ (\text{fmlookup } m \ x) \ (\text{fmlookup } n \ x)$
 shows $\text{fmrel } R \ m \ n$
 $\langle \text{proof} \rangle$

lemma *fmrel-upd[intro]*: $fmrel\ P\ m\ n \implies P\ x\ y \implies fmrel\ P\ (fmupd\ k\ x\ m)\ (fmupd\ k\ y\ n)$
 $\langle proof \rangle$

lemma *fmrelD[dest]*: $fmrel\ P\ m\ n \implies rel-option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$
 $\langle proof \rangle$

lemma *fmrel-addI[intro]*:
assumes $fmrel\ P\ m\ n\ fmrel\ P\ a\ b$
shows $fmrel\ P\ (m\ ++_f\ a)\ (n\ ++_f\ b)$
 $\langle proof \rangle$

lemma *fmrel-cases[consumes 1]*:
assumes $fmrel\ P\ m\ n$
obtains $(none)\ fmlookup\ m\ x = None\ fmlookup\ n\ x = None$
 $\quad \mid (some)\ a\ b\ \mathbf{where}\ fmlookup\ m\ x = Some\ a\ fmlookup\ n\ x = Some\ b\ P\ a\ b$
 $\langle proof \rangle$

lemma *fmrel-filter[intro]*: $fmrel\ P\ m\ n \implies fmrel\ P\ (fmfilter\ Q\ m)\ (fmfilter\ Q\ n)$
 $\langle proof \rangle$

lemma *fmrel-drop[intro]*: $fmrel\ P\ m\ n \implies fmrel\ P\ (fmdrop\ a\ m)\ (fmdrop\ a\ n)$
 $\langle proof \rangle$

lemma *fmrel-drop-set[intro]*: $fmrel\ P\ m\ n \implies fmrel\ P\ (fmdrop-set\ A\ m)\ (fmdrop-set\ A\ n)$
 $\langle proof \rangle$

lemma *fmrel-drop-fset[intro]*: $fmrel\ P\ m\ n \implies fmrel\ P\ (fmdrop-fset\ A\ m)\ (fmdrop-fset\ A\ n)$
 $\langle proof \rangle$

lemma *fmrel-restrict-set[intro]*: $fmrel\ P\ m\ n \implies fmrel\ P\ (fmrestrict-set\ A\ m)\ (fmrestrict-set\ A\ n)$
 $\langle proof \rangle$

lemma *fmrel-restrict-fset[intro]*: $fmrel\ P\ m\ n \implies fmrel\ P\ (fmrestrict-fset\ A\ m)\ (fmrestrict-fset\ A\ n)$
 $\langle proof \rangle$

lemma *fmrel-on-fset-fmrel-restrict*:
 $fmrel-on-fset\ S\ P\ m\ n \longleftrightarrow fmrel\ P\ (fmrestrict-fset\ S\ m)\ (fmrestrict-fset\ S\ n)$
 $\langle proof \rangle$

lemma *fmrel-on-fset-refl-strong*:
assumes $\bigwedge x\ y.\ x \in S \implies fmlookup\ m\ x = Some\ y \implies P\ y\ y$
shows $fmrel-on-fset\ S\ P\ m\ m$
 $\langle proof \rangle$

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* $x = \text{fmdom}\ y$

<proof>

lemma *fmrel-fmdom'-eq*: *fmrel* $P\ x\ y \implies \text{fmdom}'\ x = \text{fmdom}'\ y$

<proof>

lemma *fmrel-rel-fmran*:

assumes *fmrel* $P\ x\ y$

shows *rel-fset* $P\ (\text{fmran}\ x)\ (\text{fmran}\ y)$

<proof>

lemma *fmrel-rel-fmran'*: *fmrel* $P\ x\ y \implies \text{rel-set}\ P\ (\text{fmran}'\ x)\ (\text{fmran}'\ y)$

<proof>

lemma *pred-fmap-fmpred[simp]*: *pred-fmap* $P = \text{fmpred}\ (\lambda\cdot. P)$

<proof>

including *fset.lifting*

<proof>

lemma *pred-fmap-id[simp]*: *pred-fmap* $\text{id}\ (\text{fmmap}\ f\ m) \longleftrightarrow \text{pred-fmap}\ f\ m$

<proof>

lemma *pred-fmapD*: *pred-fmap* $P\ m \implies x \in \text{fmran}\ m \implies P\ x$

<proof>

lemma *fmlookup-map[simp]*: *fmlookup* $(\text{fmmap}\ f\ m)\ x = \text{map-option}\ f\ (\text{fmlookup}\ m\ x)$

<proof>

lemma *fmpred-map[simp]*: *fmpred* $P\ (\text{fmmap}\ f\ m) \longleftrightarrow \text{fmpred}\ (\lambda k\ v. P\ k\ (f\ v))\ m$

<proof>

lemma *fmpred-id[simp]*: *fmpred* $(\lambda\cdot. \text{id})\ (\text{fmmap}\ f\ m) \longleftrightarrow \text{fmpred}\ (\lambda\cdot. f)\ m$

<proof>

lemma *fmmap-add[simp]*: *fmmap* $f\ (m\ ++_f\ n) = \text{fmmap}\ f\ m\ ++_f\ \text{fmmap}\ f\ n$

<proof>

lemma *fmmap-empty[simp]*: *fmmap* $f\ \text{fmempty} = \text{fmempty}$

<proof>

lemma *fmdom-map[simp]*: $fmdom (fmap f m) = fmdom m$
including *fset.lifting*
 $\langle proof \rangle$

lemma *fmdom'-map[simp]*: $fmdom' (fmap f m) = fmdom' m$
 $\langle proof \rangle$

lemma *fmran-fmap[simp]*: $fmran (fmap f m) = f \mid^{\dagger} fmran m$
including *fset.lifting*
 $\langle proof \rangle$

lemma *fmran'-fmap[simp]*: $fmran' (fmap f m) = f \mid^{\dagger} fmran' m$
 $\langle proof \rangle$

lemma *fmfilter-fmap[simp]*: $fmfilter P (fmap f m) = fmap f (fmfilter P m)$
 $\langle proof \rangle$

lemma *fmdrop-fmap[simp]*: $fmdrop a (fmap f m) = fmap f (fmdrop a m)$
 $\langle proof \rangle$

lemma *fmdrop-set-fmap[simp]*: $fmdrop-set A (fmap f m) = fmap f (fmdrop-set A m)$ $\langle proof \rangle$

lemma *fmdrop-fset-fmap[simp]*: $fmdrop-fset A (fmap f m) = fmap f (fmdrop-fset A m)$ $\langle proof \rangle$

lemma *fmrestrict-set-fmap[simp]*: $fmrestrict-set A (fmap f m) = fmap f (fmrestrict-set A m)$ $\langle proof \rangle$

lemma *fmrestrict-fset-fmap[simp]*: $fmrestrict-fset A (fmap f m) = fmap f (fmrestrict-fset A m)$ $\langle proof \rangle$

lemma *fmap-subset[intro]*: $m \subseteq_f n \implies fmap f m \subseteq_f fmap f n$
 $\langle proof \rangle$

lemma *fmap-fset-of-fmap*: $fset-of-fmap (fmap f m) = (\lambda(k, v). (k, f v)) \mid^{\dagger} fset-of-fmap m$
including *fset.lifting*
 $\langle proof \rangle$

lemma *fmap-fmupd*: $fmap f (fmupd x y m) = fmupd x (f y) (fmap f m)$
 $\langle proof \rangle$

30.5 size setup

definition *size-fmap* :: $('a \Rightarrow nat) \Rightarrow ('b \Rightarrow nat) \Rightarrow ('a, 'b) fmap \Rightarrow nat$ **where**
 $[simp]$: $size-fmap f g m = size-fset (\lambda(a, b). f a + g b) (fset-of-fmap m)$

instantiation *fmap* :: $(type, type) size$ **begin**

definition *size-fmap where*

size-fmap-overloaded-def: $size-fmap = Finite-Map.size-fmap (\lambda-. 0) (\lambda-. 0)$

instance $\langle proof \rangle$

end

lemma *size-fmap-overloaded-simps*[simp]: $size\ x = size\ (fset-of-fmap\ x)$
 $\langle proof \rangle$

lemma *fmap-size-o-map*: $inj\ h \implies size-fmap\ f\ g \circ fmap\ h = size-fmap\ f\ (g \circ h)$
 $\langle proof \rangle$

$\langle ML \rangle$

30.6 Additional operations

lift-definition *fmap-keys* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b)\ fmap \Rightarrow ('a, 'c)\ fmap$ **is**
 $\lambda f\ m\ a. map-option\ (f\ a)\ (m\ a)$
 $\langle proof \rangle$

lemma *fmpred-fmap-keys*[simp]: $fmpred\ P\ (fmap-keys\ f\ m) = fmpred\ (\lambda a\ b. P\ a\ (f\ a\ b))\ m$
 $\langle proof \rangle$

lemma *fmdom-fmap-keys*[simp]: $fmdom\ (fmap-keys\ f\ m) = fmdom\ m$
including *fset.lifting*
 $\langle proof \rangle$

lemma *fmlookup-fmap-keys*[simp]: $fmlookup\ (fmap-keys\ f\ m)\ x = map-option\ (f\ x)\ (fmlookup\ m\ x)$
 $\langle proof \rangle$

lemma *fmfilter-fmap-keys*[simp]: $fmfilter\ P\ (fmap-keys\ f\ m) = fmap-keys\ f\ (fmfilter\ P\ m)$
 $\langle proof \rangle$

lemma *fmdrop-fmap-keys*[simp]: $fmdrop\ a\ (fmap-keys\ f\ m) = fmap-keys\ f\ (fmdrop\ a\ m)$
 $\langle proof \rangle$

lemma *fmdrop-set-fmap-keys*[simp]: $fmdrop-set\ A\ (fmap-keys\ f\ m) = fmap-keys\ f\ (fmdrop-set\ A\ m)$
 $\langle proof \rangle$

lemma *fmdrop-fset-fmap-keys*[simp]: $fmdrop-fset\ A\ (fmap-keys\ f\ m) = fmap-keys\ f\ (fmdrop-fset\ A\ m)$
 $\langle proof \rangle$

lemma *fmrestrict-set-fmap-keys*[simp]: $fmrestrict-set\ A\ (fmap-keys\ f\ m) = fmap-keys\ f\ (fmrestrict-set\ A\ m)$
 $\langle proof \rangle$

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]: $m \subseteq_f n \implies \text{fmmap-keys } f \ m \subseteq_f \text{fmmap-keys } f \ n$
 ⟨proof⟩

definition *sorted-list-of-fmap* :: (*'a::linorder*, *'b*) *fmap* $\Rightarrow$ (*'a* $\times$ *'b*) *list* **where**
sorted-list-of-fmap *m* = *map* ($\lambda k. (k, \text{the } (\text{fmlookup } m \ k))$) (*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⟩

including *fset.lifting*
 ⟨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* $\forall x \in S. \exists y. Q \ x \ y$
shows $\exists m. \text{fmdom}' \ m = S \wedge \text{fmpred } Q \ m$
 ⟨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* $\implies$ *fmrel* *P* $\implies$ *fmrel* *P*) *fmadd* *fmadd*
 ⟨proof⟩

lemma *fmupd-transfer*[transfer-rule]:
 ($(=) \implies P \implies \text{fmrel } P \implies \text{fmrel } P$) *fmupd* *fmupd*
 ⟨proof⟩

end

lemma *Quotient-fmap-bnf*[quot-map]:
assumes *Quotient* *R* *Abs* *Rep* *T*

shows *Quotient* (*fmrel* *R*) (*fmmmap* *Abs*) (*fmmmap* *Rep*) (*fmrel* *T*)
 ⟨*proof*⟩

30.9 View as datatype

lemma *fmap-distinct*[*simp*]:
 fmempty $\neq$ *fmupd* *k* *v* *m*
 fmupd *k* *v* *m* $\neq$ *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* $\notin$ *fmdom* *m'*
 ⟨*proof*⟩ **including** *fmap.lifting* *fset.lifting*
 ⟨*proof*⟩

lemma *fmap-induct*[*case-names fmempty fmupd, induct type: fmap*]:
 assumes *P fmempty*
 assumes ($\bigwedge x\ y\ m. P\ m \implies \text{fmlookup}\ m\ x = \text{None} \implies P\ (\text{fmupd}\ x\ y\ m)$)
 shows *P m*
 ⟨*proof*⟩

30.10 Code setup

instantiation *fmap* :: (*type*, *equal*) *equal* **begin**

definition *equal-fmap* $\equiv$ *fmrel* *HOL.equal*

instance ⟨*proof*⟩

end

lemma *fBall-alt-def*: *fBall* *S* *P* $\longleftrightarrow (\forall x. x \in S \longrightarrow P\ x)$
 ⟨*proof*⟩

lemma *fmrel-code*:
 fmrel *R* *m* *n* $\longleftrightarrow$
 fBall (*fmdom* *m*) ($\lambda x. \text{rel-option}\ R\ (\text{fmlookup}\ m\ x)\ (\text{fmlookup}\ n\ x)$) $\wedge$
 fBall (*fmdom* *n*) ($\lambda x. \text{rel-option}\ R\ (\text{fmlookup}\ m\ x)\ (\text{fmlookup}\ n\ x)$)
 ⟨*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 ($\lambda(k, -). P\ k$) 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 ($\lambda(a, b). (a, f\ a\ b)$) m)
<proof>

lemma *fmimage-of-list[code]:*
fmimage (fmap-of-list m) A = fset-of-list (map snd (filter ($\lambda(k, -). k \in A$) (AList.clearjunk m)))
<proof>

lemma *fmcomp-list[code]:*
fmap-of-list m $\circ_f$ 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** $\exists xs. \text{map-of } xs = m$
 $\langle \text{proof} \rangle$

lemma *exists-fmap-of-list*: $\exists xs. \text{fmap-of-list } xs = m$
 $\langle \text{proof} \rangle$

lemma *fmap-of-list-surj*[*simp*, *intro*]: *surj* *fmap-of-list*
 $\langle \text{proof} \rangle$

instance *fmap* :: (*countable*, *countable*) *countable*
 $\langle \text{proof} \rangle$

instance *fmap* :: (*finite*, *finite*) *finite*
 $\langle \text{proof} \rangle$

lifting-update *fmap.lifting*
lifting-forget *fmap.lifting*

30.12 Tests

export-code

fBall fmrel fmran fmran' fmdom fmdom' fmpred pred-fmap fmsubset fmupd fm-
rel-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*
 $\langle \text{proof} \rangle$

lift-definition *test2* :: *'a* $\Rightarrow$ *'b* $\Rightarrow$ (*'a*, *'b fset*) *fmap* **is** $\lambda a \ b. \text{fmupd } a \ \{b\} \ \text{fmempty}$
 $\langle \text{proof} \rangle$

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 $\Rightarrow$ 'a fset $\Rightarrow$ bool **is** *disjnt* $\langle$ proof $\rangle$

lemma *fdisjnt-alt-def*: *fdisjnt* *M N* $\longleftrightarrow$ (*M* $\mid\cap\mid$ *N* = $\{\mid\}$)
 $\langle$ proof $\rangle$

lemma *fdisjnt-insert*: *x* $\notin$ *N* $\Longrightarrow$ *fdisjnt* *M N* $\Longrightarrow$ *fdisjnt* (*finsert* *x M*) *N*
 $\langle$ proof $\rangle$

lemma *fdisjnt-subset-right*: *N'* $\mid\subseteq\mid$ *N* $\Longrightarrow$ *fdisjnt* *M N* $\Longrightarrow$ *fdisjnt* *M N'*
 $\langle$ proof $\rangle$

lemma *fdisjnt-subset-left*: *N'* $\mid\subseteq\mid$ *N* $\Longrightarrow$ *fdisjnt* *N M* $\Longrightarrow$ *fdisjnt* *N' M*
 $\langle$ proof $\rangle$

lemma *fdisjnt-union-right*: *fdisjnt* *M A* $\Longrightarrow$ *fdisjnt* *M B* $\Longrightarrow$ *fdisjnt* *M* (*A* $\mid\cup\mid$ *B*)
 $\langle$ proof $\rangle$

lemma *fdisjnt-union-left*: *fdisjnt* *A M* $\Longrightarrow$ *fdisjnt* *B M* $\Longrightarrow$ *fdisjnt* (*A* $\mid\cup\mid$ *B*) *M*
 $\langle$ proof $\rangle$

lemma *fdisjnt-swap*: *fdisjnt* *M N* $\Longrightarrow$ *fdisjnt* *N M*
including *fset.lifting* $\langle$ proof $\rangle$

lemma *distinct-append-fset*:

assumes *distinct xs distinct ys fdisjnt* (*fset-of-list* *xs*) (*fset-of-list* *ys*)

shows *distinct* (*xs* @ *ys*)

$\langle$ proof $\rangle$

lemma *fdisjnt-contrI*:

assumes $\bigwedge x. x \mid\in\mid M \Longrightarrow x \mid\in\mid N \Longrightarrow \text{False}$

shows *fdisjnt* *M N*

$\langle$ proof $\rangle$

lemma *fdisjnt-Union-left*: *fdisjnt* (*ffUnion* *S*) *T* $\longleftrightarrow$ *fBall* *S* ($\lambda S. \text{fdisjnt } S \text{ } T$)
 $\langle$ proof $\rangle$

lemma *fdisjnt-Union-right*: *fdisjnt* *T* (*ffUnion* *S*) $\longleftrightarrow$ *fBall* *S* ($\lambda S. \text{fdisjnt } T \text{ } S$)

<proof>

lemma *fdisjnt-ge-max*: $fBall\ X\ (\lambda x. x > fMax\ Y) \implies fdisjnt\ X\ Y$

<proof>

end

lemma *fmadd-disjnt*: $fdisjnt\ (fmdom\ m)\ (fmdom\ n) \implies m\ ++_f\ n = n\ ++_f\ m$

<proof>

including *fset.lifting 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* $\Rightarrow$ *'a dlist* **where** *abs-dlist* = *Abs-dlist o remdups*

definition *qcr-dlist* :: *'a list* $\Rightarrow$ *'a dlist* $\Rightarrow$ *bool* **where** *qcr-dlist* *x y* $\longleftrightarrow$ *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*:

$dxs = dys \longleftrightarrow \text{list-of-dlist } dxs = \text{list-of-dlist } dys$
 $\langle \text{proof} \rangle$

lemma *dlist-eqI*:

$\text{list-of-dlist } dxs = \text{list-of-dlist } dys \implies dxs = dys$
 $\langle \text{proof} \rangle$

Formal, totaled constructor for 'a dlist:

definition *Dlist* :: 'a list $\Rightarrow$ 'a dlist **where**

$Dlist\ xs = Abs\text{-}dlist\ (\text{remdups}\ xs)$

lemma *distinct-list-of-dlist* [*simp*, *intro*]:

$\text{distinct}\ (\text{list-of-dlist}\ dxs)$
 $\langle \text{proof} \rangle$

lemma *list-of-dlist-Dlist* [*simp*]:

$\text{list-of-dlist}\ (Dlist\ xs) = \text{remdups}\ xs$
 $\langle \text{proof} \rangle$

lemma *remdups-list-of-dlist* [*simp*]:

$\text{remdups}\ (\text{list-of-dlist}\ dxs) = \text{list-of-dlist}\ dxs$
 $\langle \text{proof} \rangle$

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:

$Dlist\ (\text{list-of-dlist}\ dxs) = dxs$
 $\langle \text{proof} \rangle$

Fundamental operations:

context

begin

qualified definition *empty* :: 'a dlist **where**

$\text{empty} = Dlist\ []$

qualified definition *insert* :: 'a $\Rightarrow$ 'a dlist $\Rightarrow$ 'a dlist **where**

$\text{insert}\ x\ dxs = Dlist\ (\text{List.insert}\ x\ (\text{list-of-dlist}\ dxs))$

qualified definition *remove* :: 'a $\Rightarrow$ 'a dlist $\Rightarrow$ 'a dlist **where**

$\text{remove}\ x\ dxs = Dlist\ (\text{remove1}\ x\ (\text{list-of-dlist}\ dxs))$

qualified definition *map* :: ('a $\Rightarrow$ 'b) $\Rightarrow$ 'a dlist $\Rightarrow$ 'b dlist **where**

$\text{map}\ f\ dxs = Dlist\ (\text{remdups}\ (\text{List.map}\ f\ (\text{list-of-dlist}\ dxs)))$

qualified definition *filter* :: ('a $\Rightarrow$ bool) $\Rightarrow$ 'a dlist $\Rightarrow$ 'a dlist **where**

$filter\ P\ dxs = Dlist\ (List.filter\ P\ (list-of-dlist\ dxs))$

qualified definition $rotate :: nat \Rightarrow 'a\ dlist \Rightarrow 'a\ dlist$ **where**
 $rotate\ n\ dxs = Dlist\ (List.rotate\ n\ (list-of-dlist\ dxs))$

end

Derived operations:

context
begin

qualified definition $null :: 'a\ dlist \Rightarrow bool$ **where**
 $null\ dxs = List.null\ (list-of-dlist\ dxs)$

qualified definition $member :: 'a\ dlist \Rightarrow 'a \Rightarrow bool$ **where**
 $member\ dxs = List.member\ (list-of-dlist\ dxs)$

qualified definition $length :: 'a\ dlist \Rightarrow nat$ **where**
 $length\ dxs = List.length\ (list-of-dlist\ dxs)$

qualified definition $fold :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ dlist \Rightarrow 'b \Rightarrow 'b$ **where**
 $fold\ f\ dxs = List.fold\ f\ (list-of-dlist\ dxs)$

qualified definition $foldr :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ dlist \Rightarrow 'b \Rightarrow 'b$ **where**
 $foldr\ f\ dxs = List.foldr\ f\ (list-of-dlist\ dxs)$

end

32.2 Executable version obeying invariant

lemma $list-of-dlist-empty$ $[simp, code\ abstract]$:
 $list-of-dlist\ Dlist.empty = []$
 $\langle proof \rangle$

lemma $list-of-dlist-insert$ $[simp, code\ abstract]$:
 $list-of-dlist\ (Dlist.insert\ x\ dxs) = List.insert\ x\ (list-of-dlist\ dxs)$
 $\langle proof \rangle$

lemma $list-of-dlist-remove$ $[simp, code\ abstract]$:
 $list-of-dlist\ (Dlist.remove\ x\ dxs) = remove1\ x\ (list-of-dlist\ dxs)$
 $\langle proof \rangle$

lemma $list-of-dlist-map$ $[simp, code\ abstract]$:
 $list-of-dlist\ (Dlist.map\ f\ dxs) = remdups\ (List.map\ f\ (list-of-dlist\ dxs))$
 $\langle proof \rangle$

lemma $list-of-dlist-filter$ $[simp, code\ abstract]$:
 $list-of-dlist\ (Dlist.filter\ P\ dxs) = List.filter\ P\ (list-of-dlist\ dxs)$
 $\langle proof \rangle$

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* $\longleftrightarrow$ *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* $\longleftrightarrow$ *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*: $\bigwedge x \text{ } dxs. \neg \text{Dlist.member } dxs \ x \implies P \ dxs \implies P \ (\text{Dlist.insert } x \ dxs)$
shows *P dxs*
 ⟨*proof*⟩

lemma *dlist-case* [*cases type: dlist*]:
obtains (*empty*) *dxs* = *Dlist.empty*
 | (*insert*) *x dys* **where** $\neg \text{Dlist.member } dys \ x$ **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 $\Rightarrow$ 'a list $\Rightarrow$ bool **where**

double (*xs* @ *ys*) (*xs* @ *x* # *ys*) **if** *x* $\in$ *set ys*

qualified lemma *strong-confluentp-double*: *strong-confluentp double*

$\langle$ proof $\rangle$ **lemma** *double-Cons1* [*simp*]: *double xs* (*x* # *xs*) **if** *x* $\in$ *set xs*

$\langle$ proof $\rangle$ **lemma** *double-Cons-same* [*simp*]: *double xs ys* $\Longrightarrow$ *double* (*x* # *xs*) (*x* # *ys*)

$\langle$ proof $\rangle$ **lemma** *doubles-Cons-same*: *double** xs ys* $\Longrightarrow$ *double*** (*x* # *xs*) (*x* # *ys*)

$\langle$ proof $\rangle$ **lemma** *remdups-into-doubles*: *double*** (*remdups xs*) *xs*

$\langle$ proof $\rangle$ **lemma** *dlist-eq-into-doubles*: *Dlist.dlist-eq* $\leq$ *equivclp double*

$\langle$ proof $\rangle$ **lemma** *factor-double-map*: *double* (*map f xs*) *ys* $\Longrightarrow$ $\exists$ *zs*. *Dlist.dlist-eq* *xs zs* $\wedge$ *ys* = *map f zs* $\wedge$ *set zs* $\subseteq$ *set xs*

$\langle$ proof $\rangle$ **lemma** *dlist-eq-set-eq*: *Dlist.dlist-eq xs ys* $\Longrightarrow$ *set xs* = *set ys*

$\langle$ proof $\rangle$ **lemma** *dlist-eq-map-respect*: *Dlist.dlist-eq xs ys* $\Longrightarrow$ *Dlist.dlist-eq* (*map f xs*) (*map f ys*)

$\langle$ proof $\rangle$ **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*

$\langle$ proof $\rangle$

lifting-update *dlist.lifting*

lifting-forget *dlist.lifting*

end

context begin

interpretation *Quotient-dlist*: *Quotient-dlist* $\langle$ proof $\rangle$

lift-bnf (*plugins del: code*) 'a *dlist*

$\langle$ proof $\rangle$ **lemma** *list-of-dlist-transfer*[*transfer-rule*]:

bi-unique R $\Longrightarrow$ (*rel-fun* (*Quotient-dlist.pcr-dlist R*) (*list-all2 R*)) *remdups list-of-dlist*

$\langle$ proof $\rangle$

lemma *list-of-dlist-map-dlist*[*simp*]:

list-of-dlist (*map-dlist f xs*) = *remdups* (*map f* (*list-of-dlist xs*))

$\langle$ proof $\rangle$

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  $\langle$ proof $\rangle$ 

```

```

setup-lifting type-definition-dual

```

```

code-datatype dual

```

```

lemma dual-eqI:
   $x = y$  if  $undual\ x = undual\ y$ 
   $\langle$ proof $\rangle$ 

```

```

lemma dual-eq-iff:
   $x = y \longleftrightarrow undual\ x = undual\ y$ 
   $\langle$ proof $\rangle$ 

```

```

lemma eq-dual-iff [iff]:
   $dual\ x = dual\ y \longleftrightarrow x = y$ 
   $\langle$ proof $\rangle$ 

```

```

lemma undual-dual [simp, code]:
   $undual\ (dual\ x) = x$ 
   $\langle$ proof $\rangle$ 

```

```

lemma dual-undual [simp]:
   $dual\ (undual\ x) = x$ 
   $\langle$ proof $\rangle$ 

```

```

lemma undual-comp-dual [simp]:
   $undual \circ dual = id$ 
   $\langle$ proof $\rangle$ 

```

```

lemma dual-comp-undual [simp]:
   $dual \circ undual = id$ 
   $\langle$ proof $\rangle$ 

```

```

lemma inj-dual:
  inj dual
   $\langle$ proof $\rangle$ 

```

lemma *inj-undual*:

inj undual
 $\langle \text{proof} \rangle$

lemma *surj-dual*:

surj dual
 $\langle \text{proof} \rangle$

lemma *surj-undual*:

surj undual
 $\langle \text{proof} \rangle$

lemma *bij-dual*:

bij dual
 $\langle \text{proof} \rangle$

lemma *bij-undual*:

bij undual
 $\langle \text{proof} \rangle$

instance *dual* :: (*finite*) *finite*

$\langle \text{proof} \rangle$

instantiation *dual* :: (*equal*) *equal*

begin

lift-definition *equal-dual* :: '*a dual* $\Rightarrow$ '*a dual* $\Rightarrow$ *bool*

is *HOL.equal* $\langle \text{proof} \rangle$

instance

$\langle \text{proof} \rangle$

end

33.1 Pointwise ordering

instantiation *dual* :: (*ord*) *ord*

begin

lift-definition *less-eq-dual* :: '*a dual* $\Rightarrow$ '*a dual* $\Rightarrow$ *bool*

is ($\geq$) $\langle \text{proof} \rangle$

lift-definition *less-dual* :: '*a dual* $\Rightarrow$ '*a dual* $\Rightarrow$ *bool*

is ($>$) $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *dual-less-eqI*:

$x \leq y$ **if** $\text{undual } y \leq \text{undual } x$
 $\langle \text{proof} \rangle$

lemma *dual-less-eq-iff*:

$x \leq y \iff \text{undual } y \leq \text{undual } x$
 $\langle \text{proof} \rangle$

lemma *less-eq-dual-iff* [iff]:

$\text{dual } x \leq \text{dual } y \iff y \leq x$
 $\langle \text{proof} \rangle$

lemma *dual-lessI*:

$x < y$ **if** $\text{undual } y < \text{undual } x$
 $\langle \text{proof} \rangle$

lemma *dual-less-iff*:

$x < y \iff \text{undual } y < \text{undual } x$
 $\langle \text{proof} \rangle$

lemma *less-dual-iff* [iff]:

$\text{dual } x < \text{dual } y \iff y < x$
 $\langle \text{proof} \rangle$

instance *dual* :: (preorder) preorder

$\langle \text{proof} \rangle$

instance *dual* :: (order) order

$\langle \text{proof} \rangle$

33.2 Binary infimum and supremum

instantiation *dual* :: (sup) inf

begin

lift-definition *inf-dual* :: 'a dual $\Rightarrow$ 'a dual $\Rightarrow$ 'a dual

is *sup* $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *undual-inf-eq* [simp]:

$\text{undual } (\text{inf } x \ y) = \text{sup } (\text{undual } x) \ (\text{undual } y)$
 $\langle \text{proof} \rangle$

lemma *dual-sup-eq* [simp]:

$\text{dual } (\text{sup } x \ y) = \text{inf } (\text{dual } x) \ (\text{dual } y)$


```

  <proof>

instantiation dual :: (inf) sup
begin

lift-definition sup-dual :: 'a dual  $\Rightarrow$  'a dual  $\Rightarrow$  '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  $\Rightarrow$  'a dual
  is uminus <proof>

instance <proof>

end

lemma undual-uminus-eq [simp]:
  undual (- x) = - undual x
  <proof>

lemma dual-uminus-eq [simp]:

```

$dual \ (- \ x) = - \ dual \ x$
 $\langle proof \rangle$

instantiation $dual :: (boolean-algebra) \ boolean-algebra$
begin

lift-definition $minus-dual :: 'a \ dual \Rightarrow 'a \ dual \Rightarrow 'a \ dual$
is $\lambda x \ y. - \ (y - x) \ \langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma $undual-minus-eq \ [simp]:$
 $undual \ (x - y) = - \ (undual \ y - undual \ x)$
 $\langle proof \rangle$

lemma $dual-minus-eq \ [simp]:$
 $dual \ (x - y) = - \ (dual \ y - dual \ x)$
 $\langle proof \rangle$

33.5 Complete lattice operations

The class of complete lattices is closed under formation of dual structures.

instantiation $dual :: (Sup) \ Inf$
begin

lift-definition $Inf-dual :: 'a \ dual \ set \Rightarrow 'a \ dual$
is $Sup \ \langle proof \rangle$

instance $\langle proof \rangle$

end

lemma $undual-Inf-eq \ [simp]:$
 $undual \ (Inf \ A) = Sup \ (undual \ 'A)$
 $\langle proof \rangle$

lemma $dual-Sup-eq \ [simp]:$
 $dual \ (Sup \ A) = Inf \ (dual \ 'A)$
 $\langle proof \rangle$

instantiation $dual :: (Inf) \ Sup$
begin

lift-definition $Sup-dual :: 'a \ dual \ set \Rightarrow 'a \ dual$
is $Inf \ \langle proof \rangle$

```

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
begin

```

34.1 Eqpoll

definition *eqpoll* :: 'a set $\Rightarrow$ 'b set $\Rightarrow$ bool (**infixl** $\approx$ 50)
where *eqpoll* A B $\equiv \exists f. \text{bij-betw } f \text{ A B}$

definition *lepoll* :: 'a set $\Rightarrow$ 'b set $\Rightarrow$ bool (**infixl** $\lesssim$ 50)
where *lepoll* A B $\equiv \exists f. \text{inj-on } f \text{ A} \wedge f ' A \subseteq B$

definition *lesspoll* :: 'a set $\Rightarrow$ 'b set $\Rightarrow$ bool (**infixl** $\prec\!\!\prec$ 50)
where $A \prec\!\!\prec B \equiv A \lesssim B \wedge \sim(A \approx B)$

lemma *lepoll-empty-iff-empty* [*simp*]: $A \lesssim \{\} \longleftrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *eqpoll-iff-card-of-ordIso*: $A \approx B \longleftrightarrow \text{ordIso2 } (\text{card-of } A) (\text{card-of } B)$
 $\langle \text{proof} \rangle$

lemma *eqpoll-refl* [*iff*]: $A \approx A$
 $\langle \text{proof} \rangle$

lemma *eqpoll-finite-iff*: $A \approx B \Longrightarrow \text{finite } A \longleftrightarrow \text{finite } B$
 $\langle \text{proof} \rangle$

lemma *eqpoll-iff-card*:
assumes *finite* A *finite* B
shows $A \approx B \longleftrightarrow \text{card } A = \text{card } B$
 $\langle \text{proof} \rangle$

lemma *lepoll-antisym*:
assumes $A \lesssim B$ $B \lesssim A$ **shows** $A \approx B$
 $\langle \text{proof} \rangle$

lemma *lepoll-trans* [*trans*]: $\llbracket A \lesssim B; B \lesssim C \rrbracket \Longrightarrow A \lesssim C$
 $\langle \text{proof} \rangle$

lemma *lepoll-trans1* [*trans*]: $\llbracket A \approx B; B \lesssim C \rrbracket \Longrightarrow A \lesssim C$
 $\langle \text{proof} \rangle$

lemma *lepoll-trans2* [*trans*]: $\llbracket A \lesssim B; B \approx C \rrbracket \Longrightarrow A \lesssim C$
 $\langle \text{proof} \rangle$

lemma *eqpoll-sym*: $A \approx B \Longrightarrow B \approx A$
 $\langle \text{proof} \rangle$

lemma *eqpoll-trans* [*trans*]: $\llbracket A \approx B; B \approx C \rrbracket \Longrightarrow A \approx C$
 $\langle \text{proof} \rangle$

lemma *eqpoll-imp-lepoll*: $A \approx B \Longrightarrow A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *subset-imp-lepoll*: $A \subseteq B \implies A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *lepoll-refl [iff]*: $A \lesssim A$
 $\langle \text{proof} \rangle$

lemma *lepoll-iff*: $A \lesssim B \iff (\exists g. A \subseteq g \text{ ‘ } B)$
 $\langle \text{proof} \rangle$

lemma *empty-lepoll [iff]*: $\{\} \lesssim A$
 $\langle \text{proof} \rangle$

lemma *subset-image-lepoll*: $B \subseteq f \text{ ‘ } A \implies B \lesssim A$
 $\langle \text{proof} \rangle$

lemma *image-lepoll*: $f \text{ ‘ } A \lesssim A$
 $\langle \text{proof} \rangle$

lemma *infinite-le-lepoll*: $\text{infinite } A \iff (\text{UNIV}::\text{nat set}) \lesssim A$
 $\langle \text{proof} \rangle$

lemma *lepoll-Pow-self*: $A \lesssim \text{Pow } A$
 $\langle \text{proof} \rangle$

lemma *eqpoll-iff-bijections*:
 $A \approx B \iff (\exists f g. (\forall x \in A. f x \in B \wedge g(f x) = x) \wedge (\forall y \in B. g y \in A \wedge f(g y) = y))$
 $\langle \text{proof} \rangle$

lemma *lepoll-restricted-funspace*:
 $\{f. f \text{ ‘ } A \subseteq B \wedge \{x. f x \neq k x\} \subseteq A \wedge \text{finite } \{x. f x \neq k x\}\} \lesssim \text{Fpow } (A \times B)$
 $\langle \text{proof} \rangle$

lemma *singleton-lepoll*: $\{x\} \lesssim \text{insert } y A$
 $\langle \text{proof} \rangle$

lemma *singleton-eqpoll*: $\{x\} \approx \{y\}$
 $\langle \text{proof} \rangle$

lemma *subset-singleton-iff-lepoll*: $(\exists x. S \subseteq \{x\}) \iff S \lesssim \{\{\}\}$
 $\langle \text{proof} \rangle$

lemma *infinite-insert-lepoll*:
assumes *infinite* *A* **shows** $\text{insert } a A \lesssim A$
 $\langle \text{proof} \rangle$

lemma *infinite-insert-eqpoll*: $\text{infinite } A \implies \text{insert } a A \approx A$
 $\langle \text{proof} \rangle$

lemma *finite-lepoll-infinite*:
assumes *infinite A finite B* **shows** $B \lesssim A$
 $\langle \text{proof} \rangle$

34.2 The strict relation

lemma *lesspoll-not-refl* [iff]: $\sim (i \prec i)$
 $\langle \text{proof} \rangle$

lemma *lesspoll-imp-lepoll*: $A \prec B \implies A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *lepoll-iff-leqpoll*: $A \lesssim B \longleftrightarrow A \prec B \mid A \approx B$
 $\langle \text{proof} \rangle$

lemma *lesspoll-trans* [trans]: $\llbracket X \prec Y; Y \prec Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-trans1* [trans]: $\llbracket X \lesssim Y; Y \prec Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-trans2* [trans]: $\llbracket X \prec Y; Y \lesssim Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *eq-lesspoll-trans* [trans]: $\llbracket X \approx Y; Y \prec Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-eq-trans* [trans]: $\llbracket X \prec Y; Y \approx Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-Pow-self*: $A \prec \text{Pow } A$
 $\langle \text{proof} \rangle$

lemma *finite-lesspoll-infinite*:
assumes *infinite A finite B* **shows** $B \prec A$
 $\langle \text{proof} \rangle$

34.3 Mapping by an injection

lemma *inj-on-image-epoll-self*: $\text{inj-on } f \ A \implies f \, ' \ A \approx A$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-lepoll-1* [simp]:
assumes *inj-on f A* **shows** $f \, ' \ A \lesssim B \longleftrightarrow A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-lepoll-2* [simp]:
assumes *inj-on f B* **shows** $A \lesssim f \, ' \ B \longleftrightarrow A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-lesspoll-1* [simp]:
 assumes *inj-on f A* shows $f \cdot A \prec B \longleftrightarrow A \prec B$
 ⟨proof⟩

lemma *inj-on-image-lesspoll-2* [simp]:
 assumes *inj-on f B* shows $A \prec f \cdot B \longleftrightarrow A \prec B$
 ⟨proof⟩

lemma *inj-on-image-epoll-1* [simp]:
 assumes *inj-on f A* shows $f \cdot A \approx B \longleftrightarrow A \approx B$
 ⟨proof⟩

lemma *inj-on-image-epoll-2* [simp]:
 assumes *inj-on f B* shows $A \approx f \cdot B \longleftrightarrow A \approx B$
 ⟨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-epoll-insertD*: $\llbracket \text{insert } u \text{ } A \approx \text{insert } v \text{ } B; u \notin A; v \notin B \rrbracket \implies A \approx B$
 ⟨proof⟩

lemma *insert-lepoll-cong*:
 assumes $A \lesssim B$ *b $\notin$ B* shows *insert a A $\lesssim$ insert b B*
 ⟨proof⟩

lemma *insert-epoll-cong*:
 $\llbracket A \approx B; a \notin A; b \notin B \rrbracket \implies \text{insert } a \text{ } A \approx \text{insert } b \text{ } B$
 ⟨proof⟩

lemma *insert-epoll-insert-iff*:
 $\llbracket a \notin A; b \notin B \rrbracket \implies \text{insert } a \text{ } A \approx \text{insert } b \text{ } B \longleftrightarrow A \approx B$
 ⟨proof⟩

lemma *insert-lepoll-insert-iff*:
 $\llbracket a \notin A; b \notin B \rrbracket \implies (\text{insert } a \text{ } A \lesssim \text{insert } b \text{ } B) \longleftrightarrow (A \lesssim B)$
 ⟨proof⟩

lemma *less-imp-insert-lepoll*:
 assumes $A \prec B$ shows *insert a A $\lesssim$ B*
 ⟨proof⟩

lemma *finite-insert-lepoll*: *finite A* $\implies (\text{insert } a \text{ } A \lesssim A) \longleftrightarrow (a \in A)$
 ⟨proof⟩

34.5 Binary sums and unions

lemma *Un-lepoll-mono*:

assumes $A \lesssim C$ $B \lesssim D$ *disjnt* C D **shows** $A \cup B \lesssim C \cup D$
 $\langle \text{proof} \rangle$

lemma *Un-epoll-cong*: $\llbracket A \approx C; B \approx D; \text{disjnt } A \ B; \text{disjnt } C \ D \rrbracket \implies A \cup B \approx C \cup D$
 $\langle \text{proof} \rangle$

lemma *sum-lepoll-mono*:

assumes $A \lesssim C$ $B \lesssim D$ **shows** $A <+> B \lesssim C <+> D$
 $\langle \text{proof} \rangle$

lemma *sum-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A <+> B \approx C <+> D$
 $\langle \text{proof} \rangle$

34.6 Binary Cartesian products

lemma *times-square-lepoll*: $A \lesssim A \times A$
 $\langle \text{proof} \rangle$

lemma *times-commute-epoll*: $A \times B \approx B \times A$
 $\langle \text{proof} \rangle$

lemma *times-assoc-epoll*: $(A \times B) \times C \approx A \times (B \times C)$
 $\langle \text{proof} \rangle$

lemma *times-singleton-epoll*: $\{a\} \times A \approx A$
 $\langle \text{proof} \rangle$

lemma *times-lepoll-mono*:

assumes $A \lesssim C$ $B \lesssim D$ **shows** $A \times B \lesssim C \times D$
 $\langle \text{proof} \rangle$

lemma *times-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A \times B \approx C \times D$
 $\langle \text{proof} \rangle$

lemma

assumes $B \neq \{\}$ **shows** *lepoll-times1*: $A \lesssim A \times B$ **and** *lepoll-times2*: $A \lesssim B \times A$
 $\langle \text{proof} \rangle$

lemma *times-0-epoll*: $\{\} \times A \approx \{\}$
 $\langle \text{proof} \rangle$

lemma *Sigma-lepoll-mono*:

assumes $A \subseteq C$ $\bigwedge x. x \in A \implies B \lesssim D \ x$ **shows** $\text{Sigma } A \ B \lesssim \text{Sigma } C \ D$
 $\langle \text{proof} \rangle$

lemma *sum-times-distrib-eqpoll*: $(A <+> B) \times C \approx (A \times C) <+> (B \times C)$
 $\langle \text{proof} \rangle$

lemma *prod-insert-eqpoll*:
 assumes $a \notin A$ shows $\text{insert } a \ A \times B \approx B <+> A \times B$
 $\langle \text{proof} \rangle$

34.7 General Unions

lemma *Union-eqpoll-Times*:
 assumes $B: \bigwedge x. x \in A \implies F x \approx B$ and *disj*: *pairwise* $(\lambda x y. \text{disjnt } (F x) (F y)) \ A$
 shows $(\bigcup_{x \in A} F x) \approx A \times B$
 $\langle \text{proof} \rangle$

lemma *UN-lepoll-UN*:
 assumes $A: \bigwedge x. x \in A \implies B x \lesssim C x$
 and *disj*: *pairwise* $(\lambda x y. \text{disjnt } (C x) (C y)) \ A$
 shows $\bigcup (B'A) \lesssim \bigcup (C'A)$
 $\langle \text{proof} \rangle$

lemma *UN-eqpoll-UN*:
 assumes $A: \bigwedge x. x \in A \implies B x \approx C x$
 and $B: \text{pairwise } (\lambda x y. \text{disjnt } (B x) (B y)) \ A$
 and $C: \text{pairwise } (\lambda x y. \text{disjnt } (C x) (C y)) \ A$
 shows $(\bigcup_{x \in A} B x) \approx (\bigcup_{x \in A} C x)$
 $\langle \text{proof} \rangle$

34.8 General Cartesian products (Pi)

lemma *PiE-sing-eqpoll-self*: $(\{a\} \rightarrow_E B) \approx B$
 $\langle \text{proof} \rangle$

lemma *lepoll-funcset-right*:
 $B \lesssim B' \implies A \rightarrow_E B \lesssim A \rightarrow_E B'$
 $\langle \text{proof} \rangle$

lemma *lepoll-funcset-left*:
 assumes $B \neq \{\}$ $A \lesssim A'$
 shows $A \rightarrow_E B \lesssim A' \rightarrow_E B$
 $\langle \text{proof} \rangle$

lemma *lepoll-funcset*:
 $\llbracket B \neq \{\}; A \lesssim A'; B \lesssim B' \rrbracket \implies A \rightarrow_E B \lesssim A' \rightarrow_E B'$
 $\langle \text{proof} \rangle$

lemma *lepoll-PiE*:
 assumes $\bigwedge i. i \in A \implies B i \lesssim C i$
 shows $\text{PiE } A \ B \lesssim \text{PiE } A \ C$
 $\langle \text{proof} \rangle$

lemma *card-le-PiE-subindex*:
assumes $A \subseteq A' \text{ Pi}_E A' B \neq \{\}$
shows $\text{Pi}_E A B \lesssim \text{Pi}_E A' B$
 $\langle \text{proof} \rangle$

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)*
 $\langle \text{proof} \rangle$

proposition *finite-PiE-iff*:
 $\text{finite}(\text{Pi}_E I S) \longleftrightarrow \text{Pi}_E I S = \{\} \vee \text{finite} \{i \in I. \sim(\exists a. S i \subseteq \{a\})\} \wedge (\forall i \in I. \text{finite}(S i))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

corollary *finite-funcset-iff*:
 $\text{finite}(I \rightarrow_E S) \longleftrightarrow (\exists a. S \subseteq \{a\}) \vee I = \{\} \vee \text{finite } I \wedge \text{finite } S$
 $\langle \text{proof} \rangle$

lemma *lists-lepoll-mono*:
assumes $A \lesssim B$ **shows** *lists A $\lesssim$ lists B*
 $\langle \text{proof} \rangle$

lemma *lepoll-lists*: $A \lesssim \text{lists } A$
 $\langle \text{proof} \rangle$

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
 $\langle ML \rangle$

end

theory *Extended*
imports *Simps-Case-Conv*
begin

datatype *'a extended* = *Fin 'a* | *Pinf* (∞) | *Minf* ($-\infty$)

instantiation *extended* :: (*order*)*order*
begin

fun *less-eq-extended* :: *'a extended* $\Rightarrow$ *'a extended* $\Rightarrow$ *bool* **where**
Fin $x \leq$ *Fin* $y = (x \leq y)$ |
 $- \leq$ *Pinf* = *True* |
Minf $\leq - =$ *True* |
 $(-::'a\ extended) \leq - =$ *False*

case-of-simps *less-eq-extended-case*: *less-eq-extended.simps*

definition *less-extended* :: *'a extended* $\Rightarrow$ *'a extended* $\Rightarrow$ *bool* **where**
 $((x::'a\ extended) < y) = (x \leq y \wedge \neg y \leq x)$

instance
 $\langle proof \rangle$

end

instance *extended* :: (*linorder*)*linorder*
 $\langle proof \rangle$

lemma *Minf-le[simp]*: *Minf* $\leq y$
 $\langle proof \rangle$

lemma *le-Pinf[simp]*: $x \leq$ *Pinf*
 $\langle proof \rangle$

lemma *le-Minf[simp]*: $x \leq$ *Minf* $\longleftrightarrow x =$ *Minf*
 $\langle proof \rangle$

lemma *Pinf-le[simp]*: *Pinf* $\leq x \longleftrightarrow x =$ *Pinf*
 $\langle proof \rangle$

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*

$\langle proof \rangle$

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]

```

```

instantiation extended :: (one)one
begin
definition 1 = Fin(1::'a)
instance <proof>
end

```

```

declare one-extended-def[symmetric, code-post]

```

```

instantiation extended :: (plus)plus
begin

```

The following definition 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 $x - y = x + -(y :: 'a \text{ extended})$
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*]: $Fin(\text{numeral } w) = \text{numeral } w$
 ⟨*proof*⟩

lemma *Fin-neg-numeral*[code-post]: $Fin (- numeral w) = - numeral w$
 $\langle proof \rangle$

instantiation *extended* :: (*lattice*) *bounded-lattice*
begin

definition *bot* = *Minf*

definition *top* = *Pinf*

fun *inf-extended* :: 'a *extended* $\Rightarrow$ 'a *extended* $\Rightarrow$ '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* $\Rightarrow$ 'a *extended* $\Rightarrow$ '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

$\langle proof \rangle$

end

end

35 Continuity and iterations

theory *Order-Continuity*

imports *Complex-Main Countable-Complete-Lattices*

begin

lemma *SUP-nat-binary*:

$(sup A (SUP x \in Collect ((<) (0::nat)). B)) = (sup A B::'a::countable-complete-lattice)$
 $\langle proof \rangle$

lemma *INF-nat-binary*:

$(inf A (INF x \in Collect ((<) (0::nat)). B)) = (inf A B::'a::countable-complete-lattice)$
 $\langle proof \rangle$

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 $\Rightarrow$ 'b::countable-complete-lattice) $\Rightarrow$ bool

where

sup-continuous $F \iff (\forall M::nat \Rightarrow 'a. \text{mono } M \longrightarrow F (SUP\ i. M\ i) = (SUP\ i. F\ (M\ i)))$

lemma *sup-continuousD*: *sup-continuous* $F \implies \text{mono } M \implies F (SUP\ i::nat. M\ i) = (SUP\ i. F\ (M\ i))$
 <proof>

lemma *sup-continuous-mono*:

mono F **if** *sup-continuous* F
 <proof>

lemma [*order-continuous-intros*]:

shows *sup-continuous-const*: *sup-continuous* $(\lambda x. c)$
and *sup-continuous-id*: *sup-continuous* $(\lambda x. x)$
and *sup-continuous-apply*: *sup-continuous* $(\lambda f. f\ x)$
and *sup-continuous-fun*: $(\bigwedge s. \text{sup-continuous } (\lambda x. P\ x\ s)) \implies \text{sup-continuous } P$
and *sup-continuous-If*: *sup-continuous* $F \implies \text{sup-continuous } G \implies \text{sup-continuous } (\lambda f. \text{if } C \text{ then } F\ f \text{ else } G\ f)$
 <proof>

lemma *sup-continuous-compose*:

assumes f : *sup-continuous* f **and** g : *sup-continuous* g
shows *sup-continuous* $(\lambda x. f\ (g\ x))$
 <proof>

lemma *sup-continuous-sup*[*order-continuous-intros*]:

sup-continuous $f \implies \text{sup-continuous } g \implies \text{sup-continuous } (\lambda x. \text{sup } (f\ x) (g\ x))$
 <proof>

lemma *sup-continuous-inf*[*order-continuous-intros*]:

fixes $P\ Q :: 'a :: \text{countable-complete-lattice} \Rightarrow 'b :: \text{countable-complete-distrib-lattice}$
assumes P : *sup-continuous* P **and** Q : *sup-continuous* Q
shows *sup-continuous* $(\lambda x. \text{inf } (P\ x) (Q\ x))$
 <proof>

lemma *sup-continuous-and*[*order-continuous-intros*]:

sup-continuous $P \implies \text{sup-continuous } Q \implies \text{sup-continuous } (\lambda x. P\ x \wedge Q\ x)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-or[order-continuous-intros]*:

sup-continuous $P \implies \text{sup-continuous } Q \implies \text{sup-continuous } (\lambda x. P \ x \vee Q \ x)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-lfp*:

assumes *sup-continuous* F **shows** $\text{lfp } F = (\text{SUP } i. (F \ \frown \ i) \ \text{bot})$ (**is** $\text{lfp } F = ?U$)

$\langle \text{proof} \rangle$

lemma *lfp-transfer-bounded*:

assumes $P: P \ \text{bot} \wedge x. P \ x \implies P \ (f \ x) \wedge M. (\wedge i. P \ (M \ i)) \implies P \ (\text{SUP } i::\text{nat}. M \ i)$

assumes $\alpha: \wedge M. \text{mono } M \implies (\wedge i::\text{nat}. P \ (M \ i)) \implies \alpha \ (\text{SUP } i. M \ i) = (\text{SUP } i. \alpha \ (M \ i))$

assumes $f: \text{sup-continuous } f$ **and** $g: \text{sup-continuous } g$

assumes *[simp]*: $\wedge x. P \ x \implies x \leq \text{lfp } f \implies \alpha \ (f \ x) = g \ (\alpha \ x)$

assumes *g-bound*: $\wedge x. \alpha \ \text{bot} \leq g \ x$

shows $\alpha \ (\text{lfp } f) = \text{lfp } g$

$\langle \text{proof} \rangle$

lemma *lfp-transfer*:

sup-continuous $\alpha \implies \text{sup-continuous } f \implies \text{sup-continuous } g \implies$

$(\wedge x. \alpha \ \text{bot} \leq g \ x) \implies (\wedge x. x \leq \text{lfp } f \implies \alpha \ (f \ x) = g \ (\alpha \ x)) \implies \alpha \ (\text{lfp } f) = \text{lfp } g$

$\langle \text{proof} \rangle$

definition

inf-continuous :: $('a::\text{countable-complete-lattice} \Rightarrow 'b::\text{countable-complete-lattice}) \Rightarrow \text{bool}$

where

inf-continuous $F \longleftrightarrow (\forall M::\text{nat} \Rightarrow 'a. \text{antimono } M \longrightarrow F \ (\text{INF } i. M \ i) = (\text{INF } i. F \ (M \ i)))$

lemma *inf-continuousD*: *inf-continuous* $F \implies \text{antimono } M \implies F \ (\text{INF } i::\text{nat}. M \ i) = (\text{INF } i. F \ (M \ i))$

$\langle \text{proof} \rangle$

lemma *inf-continuous-mono*:

mono F **if** *inf-continuous* F

$\langle \text{proof} \rangle$

lemma *[order-continuous-intros]*:

shows *inf-continuous-const*: *inf-continuous* $(\lambda x. c)$

and *inf-continuous-id*: *inf-continuous* $(\lambda x. x)$

and *inf-continuous-apply*: *inf-continuous* $(\lambda f. f \ x)$

and *inf-continuous-fun*: $(\wedge s. \text{inf-continuous } (\lambda x. P \ x \ s)) \implies \text{inf-continuous } P$

and *inf-continuous-If*: *inf-continuous* $F \implies \text{inf-continuous } G \implies \text{inf-continuous } (\lambda f. \text{if } C \text{ then } F \ f \text{ else } G \ f)$

$\langle \text{proof} \rangle$

lemma *inf-continuous-inf*[*order-continuous-intros*]:

inf-continuous $f \implies \text{inf-continuous } g \implies \text{inf-continuous } (\lambda x. \text{inf } (f \ x) \ (g \ x))$

$\langle \text{proof} \rangle$

lemma *inf-continuous-sup*[*order-continuous-intros*]:

fixes $P \ Q :: 'a :: \text{countable-complete-lattice} \Rightarrow 'b :: \text{countable-complete-distrib-lattice}$

assumes $P: \text{inf-continuous } P$ **and** $Q: \text{inf-continuous } Q$

shows $\text{inf-continuous } (\lambda x. \text{sup } (P \ x) \ (Q \ x))$

$\langle \text{proof} \rangle$

lemma *inf-continuous-and*[*order-continuous-intros*]:

$\text{inf-continuous } P \implies \text{inf-continuous } Q \implies \text{inf-continuous } (\lambda x. P \ x \wedge Q \ x)$

$\langle \text{proof} \rangle$

lemma *inf-continuous-or*[*order-continuous-intros*]:

$\text{inf-continuous } P \implies \text{inf-continuous } Q \implies \text{inf-continuous } (\lambda x. P \ x \vee Q \ x)$

$\langle \text{proof} \rangle$

lemma *inf-continuous-compose*:

assumes $f: \text{inf-continuous } f$ **and** $g: \text{inf-continuous } g$

shows $\text{inf-continuous } (\lambda x. f \ (g \ x))$

$\langle \text{proof} \rangle$

lemma *inf-continuous-gfp*:

assumes $\text{inf-continuous } F$ **shows** $\text{gfp } F = (\text{INF } i. (F \ \frown \ i) \ \text{top}) \ (\text{is } \text{gfp } F = ?U)$

$\langle \text{proof} \rangle$

lemma *gfp-transfer*:

assumes $\alpha: \text{inf-continuous } \alpha$ **and** $f: \text{inf-continuous } f$ **and** $g: \text{inf-continuous } g$

assumes [*simp*]: $\alpha \ \text{top} = \text{top} \wedge x. \alpha \ (f \ x) = g \ (\alpha \ x)$

shows $\alpha \ (\text{gfp } f) = \text{gfp } g$

$\langle \text{proof} \rangle$

lemma *gfp-transfer-bounded*:

assumes $P: P \ (f \ \text{top}) \wedge x. P \ x \implies P \ (f \ x) \wedge M. \text{antimono } M \implies (\bigwedge i. P \ (M \ i)) \implies P \ (\text{INF } i::\text{nat}. M \ i)$

assumes $\alpha: \bigwedge M. \text{antimono } M \implies (\bigwedge i::\text{nat}. P \ (M \ i)) \implies \alpha \ (\text{INF } i. M \ i) = (\text{INF } i. \alpha \ (M \ i))$

assumes $f: \text{inf-continuous } f$ **and** $g: \text{inf-continuous } g$

assumes [*simp*]: $\bigwedge x. P \ x \implies \alpha \ (f \ x) = g \ (\alpha \ x)$

assumes $g\text{-bound}: \bigwedge x. g \ x \leq \alpha \ (f \ \text{top})$

shows $\alpha \ (\text{gfp } f) = \text{gfp } g$

$\langle \text{proof} \rangle$

35.1.1 Least fixed points in countable complete lattices

definition (in *countable-complete-lattice*) $\text{cclfp} :: ('a \Rightarrow 'a) \Rightarrow 'a$

where $cclfp\ f = (SUP\ i.\ (f\ \frown\ i)\ bot)$

lemma *cclfp-unfold*:

assumes *sup-continuous F* **shows** $cclfp\ F = F\ (cclfp\ F)$
 $\langle proof \rangle$

lemma *cclfp-lowerbound*: **assumes** $f: mono\ f$ **and** $A: f\ A \leq A$ **shows** $cclfp\ f \leq A$
 $\langle proof \rangle$

lemma *cclfp-transfer*:

assumes *sup-continuous α mono f*
assumes $\alpha\ bot = bot \wedge x.\ \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (cclfp\ f) = cclfp\ g$
 $\langle proof \rangle$

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 $f :: nat \Rightarrow 'a::\{canonically-ordered-monoid-add,\ linorder-topology,\ complete-linorder\}$

begin

lemma *sums-SUP[simp, intro]*: $f\ sums\ (SUP\ n.\ \sum\ i<n.\ f\ i)$
 $\langle proof \rangle$

lemma *suminf-eq-SUP*: $suminf\ f = (SUP\ n.\ \sum\ i<n.\ f\ i)$
 $\langle proof \rangle$

end

36.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *enat* = *UNIV* :: *nat option set* $\langle proof \rangle$

TODO: introduce *enat* as coinductive datatype, *enat* is just *of-nat*

definition *enat* :: $nat \Rightarrow enat$ **where**
 $enat\ n = Abs-enat\ (Some\ n)$

```

instantiation enat :: infinity
begin

definition  $\infty = \text{Abs-enat } \text{None}$ 
instance  $\langle \text{proof} \rangle$ 

end

instance enat :: countable
 $\langle \text{proof} \rangle$ 

old-rep-datatype enat  $\infty :: \text{enat}$ 
 $\langle \text{proof} \rangle$ 

declare  $[[\text{coercion } \text{enat}::\text{nat} \Rightarrow \text{enat}]]$ 

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]:  $(x \neq \infty) = (\exists i. x = \text{enat } i)$ 
 $\langle \text{proof} \rangle$ 

lemma not-enat-eq [iff]:  $(\forall y. x \neq \text{enat } y) = (x = \infty)$ 
 $\langle \text{proof} \rangle$ 

lemma enat-ex-split:  $(\exists c::\text{enat}. P \ c) \longleftrightarrow P \ \infty \vee (\exists c::\text{nat}. P \ c)$ 
 $\langle \text{proof} \rangle$ 

primrec the-enat ::  $\text{enat} \Rightarrow \text{nat}$ 
  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
 $\langle \text{proof} \rangle$ 

end

definition eSuc ::  $\text{enat} \Rightarrow \text{enat}$  where

```

$$eSuc\ i = (case\ i\ of\ enat\ n \Rightarrow enat\ (Suc\ n) \mid \infty \Rightarrow \infty)$$

lemma *enat-0* [code-post]: $enat\ 0 = 0$
 ⟨proof⟩

lemma *enat-1* [code-post]: $enat\ 1 = 1$
 ⟨proof⟩

lemma *enat-0-iff*: $enat\ x = 0 \longleftrightarrow x = 0\ 0 = enat\ x \longleftrightarrow x = 0$
 ⟨proof⟩

lemma *enat-1-iff*: $enat\ x = 1 \longleftrightarrow x = 1\ 1 = enat\ x \longleftrightarrow x = 1$
 ⟨proof⟩

lemma *one-eSuc*: $1 = eSuc\ 0$
 ⟨proof⟩

lemma *infinity-ne-i0* [simp]: $(\infty::enat) \neq 0$
 ⟨proof⟩

lemma *i0-ne-infinity* [simp]: $0 \neq (\infty::enat)$
 ⟨proof⟩

lemma *zero-one-enat-neq*:
 $\neg 0 = (1::enat)$
 $\neg 1 = (0::enat)$
 ⟨proof⟩

lemma *infinity-ne-i1* [simp]: $(\infty::enat) \neq 1$
 ⟨proof⟩

lemma *i1-ne-infinity* [simp]: $1 \neq (\infty::enat)$
 ⟨proof⟩

lemma *eSuc-enat*: $eSuc\ (enat\ n) = enat\ (Suc\ n)$
 ⟨proof⟩

lemma *eSuc-infinity* [simp]: $eSuc\ \infty = \infty$
 ⟨proof⟩

lemma *eSuc-ne-0* [simp]: $eSuc\ n \neq 0$
 ⟨proof⟩

lemma *zero-ne-eSuc* [simp]: $0 \neq eSuc\ n$
 ⟨proof⟩

lemma *eSuc-inject* [simp]: $eSuc\ m = eSuc\ n \longleftrightarrow m = n$
 ⟨proof⟩

lemma *eSuc-enat-iff*: $eSuc\ x = enat\ y \longleftrightarrow (\exists n. y = Suc\ n \wedge x = enat\ n)$
 $\langle proof \rangle$

lemma *enat-eSuc-iff*: $enat\ y = eSuc\ x \longleftrightarrow (\exists n. y = Suc\ n \wedge enat\ n = x)$
 $\langle proof \rangle$

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 $q + \infty = \infty$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma *eSuc-plus-1*:
 $eSuc\ n = n + 1$
 $\langle proof \rangle$

lemma *plus-1-eSuc*:
 $1 + q = eSuc\ q$
 $q + 1 = eSuc\ q$
 $\langle proof \rangle$

lemma *iadd-Suc*: $eSuc\ m + n = eSuc\ (m + n)$
 $\langle proof \rangle$

lemma *iadd-Suc-right*: $m + eSuc\ n = eSuc\ (m + n)$
 $\langle proof \rangle$

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*]:
 $\text{enat } m * \text{enat } n = \text{enat } (m * n)$
 $\infty * \infty = (\infty :: \text{enat})$
 $\infty * \text{enat } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \infty)$
 $\text{enat } m * \infty = (\text{if } m = 0 \text{ then } 0 \text{ else } \infty)$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *mult-eSuc*: $e\text{Suc } m * n = n + m * n$
 $\langle \text{proof} \rangle$

lemma *mult-eSuc-right*: $m * e\text{Suc } n = m + m * n$
 $\langle \text{proof} \rangle$

lemma *of-nat-eq-enat*: $\text{of-nat } n = \text{enat } n$
 $\langle \text{proof} \rangle$

instance *enat :: semiring-char-0*
 $\langle \text{proof} \rangle$

lemma *imult-is-infinity*: $((a :: \text{enat}) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$
 $\langle \text{proof} \rangle$

36.5 Numerals

lemma *numeral-eq-enat*:
 $\text{numeral } k = \text{enat } (\text{numeral } k)$
 $\langle \text{proof} \rangle$

lemma *enat-numeral* [*code-abbrev*]:
 $\text{enat } (\text{numeral } k) = \text{numeral } k$
 $\langle \text{proof} \rangle$

lemma *infinity-ne-numeral* [*simp*]: $(\infty :: \text{enat}) \neq \text{numeral } k$
 $\langle \text{proof} \rangle$

lemma *numeral-ne-infinity* [*simp*]: $\text{numeral } k \neq (\infty :: \text{enat})$
 $\langle \text{proof} \rangle$

lemma *eSuc-numeral* [*simp*]: $e\text{Suc } (\text{numeral } k) = \text{numeral } (k + \text{Num.One})$
 $\langle \text{proof} \rangle$

36.6 Subtraction

instantiation *enat* :: *minus*
begin

definition *diff-enat-def*:

$$a - b = (\text{case } a \text{ of } (\text{enat } x) \Rightarrow (\text{case } b \text{ of } (\text{enat } y) \Rightarrow \text{enat } (x - y) \mid \infty \Rightarrow 0) \mid \infty \Rightarrow \infty)$$

instance $\langle \text{proof} \rangle$

end

lemma *idiff-enat-enat* [*simp*, *code*]: *enat* *a* − *enat* *b* = *enat* (*a* − *b*)
 $\langle \text{proof} \rangle$

lemma *idiff-infinity* [*simp*, *code*]: $\infty - n = (\infty :: \text{enat})$
 $\langle \text{proof} \rangle$

lemma *idiff-infinity-right* [*simp*, *code*]: *enat* *a* − $\infty = 0$
 $\langle \text{proof} \rangle$

lemma *idiff-0* [*simp*]: $(0 :: \text{enat}) - n = 0$
 $\langle \text{proof} \rangle$

lemmas *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

lemma *idiff-0-right* [*simp*]: $(n :: \text{enat}) - 0 = n$
 $\langle \text{proof} \rangle$

lemmas *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

lemma *idiff-self* [*simp*]: $n \neq \infty \implies (n :: \text{enat}) - n = 0$
 $\langle \text{proof} \rangle$

lemma *eSuc-minus-eSuc* [*simp*]: *eSuc* *n* − *eSuc* *m* = *n* − *m*
 $\langle \text{proof} \rangle$

lemma *eSuc-minus-1* [*simp*]: *eSuc* *n* − 1 = *n*
 $\langle \text{proof} \rangle$

36.7 Ordering

instantiation *enat* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:

$$m \leq n = (\text{case } n \text{ of } \text{enat } n1 \Rightarrow (\text{case } m \text{ of } \text{enat } m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow \text{False}) \mid \infty \Rightarrow \text{True})$$

definition *[nitpick-simp]*:

$m < n = (\text{case } m \text{ of } \text{enat } m1 \Rightarrow (\text{case } n \text{ of } \text{enat } n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow \text{True}) \mid \infty \Rightarrow \text{False})$

lemma *enat-ord-simps* *[simp]*:

$\text{enat } m \leq \text{enat } n \longleftrightarrow m \leq n$
 $\text{enat } m < \text{enat } n \longleftrightarrow m < n$
 $q \leq (\infty :: \text{enat})$
 $q < (\infty :: \text{enat}) \longleftrightarrow q \neq \infty$
 $(\infty :: \text{enat}) \leq q \longleftrightarrow q = \infty$
 $(\infty :: \text{enat}) < q \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *numeral-le-enat-iff* *[simp]*:

shows $\text{numeral } m \leq \text{enat } n \longleftrightarrow \text{numeral } m \leq n$
 $\langle \text{proof} \rangle$

lemma *numeral-less-enat-iff* *[simp]*:

shows $\text{numeral } m < \text{enat } n \longleftrightarrow \text{numeral } m < n$
 $\langle \text{proof} \rangle$

lemma *enat-ord-code* *[code]*:

$\text{enat } m \leq \text{enat } n \longleftrightarrow m \leq n$
 $\text{enat } m < \text{enat } n \longleftrightarrow m < n$
 $q \leq (\infty :: \text{enat}) \longleftrightarrow \text{True}$
 $\text{enat } m < \infty \longleftrightarrow \text{True}$
 $\infty \leq \text{enat } n \longleftrightarrow \text{False}$
 $(\infty :: \text{enat}) < q \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

instance

$\langle \text{proof} \rangle$

end

instance *enat* :: *dioid*

$\langle \text{proof} \rangle$

instance *enat* :: $\{\text{linordered-nonzero-semiring}, \text{strict-ordered-comm-monoid-add}\}$

$\langle \text{proof} \rangle$

lemma *add-diff-assoc-enat*: $z \leq y \implies x + (y - z) = x + y - (z :: \text{enat})$

$\langle \text{proof} \rangle$

lemma *enat-ord-number* *[simp]*:

$(\text{numeral } m :: \text{enat}) \leq \text{numeral } n \longleftrightarrow (\text{numeral } m :: \text{nat}) \leq \text{numeral } n$
 $(\text{numeral } m :: \text{enat}) < \text{numeral } n \longleftrightarrow (\text{numeral } m :: \text{nat}) < \text{numeral } n$

$\langle \text{proof} \rangle$

lemma *infinity-ileE* [elim!]: $\infty \leq \text{enat } m \implies R$
 $\langle \text{proof} \rangle$

lemma *infinity-ilessE* [elim!]: $\infty < \text{enat } m \implies R$
 $\langle \text{proof} \rangle$

lemma *eSuc-ile-mono* [simp]: $e\text{Suc } n \leq e\text{Suc } m \longleftrightarrow n \leq m$
 $\langle \text{proof} \rangle$

lemma *eSuc-mono* [simp]: $e\text{Suc } n < e\text{Suc } m \longleftrightarrow n < m$
 $\langle \text{proof} \rangle$

lemma *ile-eSuc* [simp]: $n \leq e\text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *not-eSuc-ilei0* [simp]: $\neg e\text{Suc } n \leq 0$
 $\langle \text{proof} \rangle$

lemma *i0-iless-eSuc* [simp]: $0 < e\text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *iless-eSuc0* [simp]: $(n < e\text{Suc } 0) = (n = 0)$
 $\langle \text{proof} \rangle$

lemma *ileI1*: $m < n \implies e\text{Suc } m \leq n$
 $\langle \text{proof} \rangle$

lemma *Suc-ile-eq*: $\text{enat } (\text{Suc } m) \leq n \longleftrightarrow \text{enat } m < n$
 $\langle \text{proof} \rangle$

lemma *iless-Suc-eq* [simp]: $\text{enat } m < e\text{Suc } n \longleftrightarrow \text{enat } m \leq n$
 $\langle \text{proof} \rangle$

lemma *imult-infinity*: $(0::\text{enat}) < n \implies \infty * n = \infty$
 $\langle \text{proof} \rangle$

lemma *imult-infinity-right*: $(0::\text{enat}) < n \implies n * \infty = \infty$
 $\langle \text{proof} \rangle$

lemma *enat-0-less-mult-iff*: $(0 < (m::\text{enat}) * n) = (0 < m \wedge 0 < n)$
 $\langle \text{proof} \rangle$

lemma *mono-eSuc*: *mono* *eSuc*
 $\langle \text{proof} \rangle$

lemma *min-enat-simps* [simp]:
 $\text{min } (\text{enat } m) (\text{enat } n) = \text{enat } (\text{min } m \ n)$

$\text{min } q \ 0 = 0$
 $\text{min } 0 \ q = 0$
 $\text{min } q \ (\infty::\text{enat}) = q$
 $\text{min } (\infty::\text{enat}) \ q = q$
 $\langle \text{proof} \rangle$

lemma *max-enat-simps* [simp]:
 $\text{max } (\text{enat } m) (\text{enat } n) = \text{enat } (\text{max } m \ n)$
 $\text{max } q \ 0 = q$
 $\text{max } 0 \ q = q$
 $\text{max } q \ \infty = (\infty::\text{enat})$
 $\text{max } \infty \ q = (\infty::\text{enat})$
 $\langle \text{proof} \rangle$

lemma *enat-ile*: $n \leq \text{enat } m \implies \exists k. n = \text{enat } k$
 $\langle \text{proof} \rangle$

lemma *enat-iless*: $n < \text{enat } m \implies \exists k. n = \text{enat } k$
 $\langle \text{proof} \rangle$

lemma *iadd-le-enat-iff*:
 $x + y \leq \text{enat } n \iff (\exists y' \ x'. x = \text{enat } x' \wedge y = \text{enat } y' \wedge x' + y' \leq n)$
 $\langle \text{proof} \rangle$

lemma *chain-incr*: $\forall i. \exists j. Y \ i < Y \ j \implies \exists j. \text{enat } k < Y \ j$
 $\langle \text{proof} \rangle$

lemma *eSuc-max*: $e\text{Suc } (\text{max } x \ y) = \text{max } (e\text{Suc } x) (e\text{Suc } y)$
 $\langle \text{proof} \rangle$

lemma *eSuc-Max*:
assumes *finite A* $A \neq \{\}$
shows $e\text{Suc } (\text{Max } A) = \text{Max } (e\text{Suc } ` A)$
 $\langle \text{proof} \rangle$

instantiation *enat* :: $\{\text{order-bot}, \text{order-top}\}$
begin

definition *bot-enat* :: *enat* **where** *bot-enat* = 0

definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
 $\langle \text{proof} \rangle$

end

lemma *finite-enat-bounded*:
assumes *le-fin*: $\bigwedge y. y \in A \implies y \leq \text{enat } n$
shows *finite A*

$\langle proof \rangle$

36.8 Cancellation simprocs

lemma *add-diff-cancel-enat[simp]*: $x \neq \infty \implies x + y - x = (y::enat)$
 $\langle proof \rangle$

lemma *enat-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::enat) \vee b = c$
 $\langle proof \rangle$

lemma *enat-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::enat) \vee b \leq c$
 $\langle proof \rangle$

lemma *enat-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::enat) \wedge b < c$
 $\langle proof \rangle$

lemma *plus-eq-infty-iff-enat*: $(m::enat) + n = \infty \longleftrightarrow m = \infty \vee n = \infty$
 $\langle proof \rangle$

$\langle ML \rangle$

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

36.9 Well-ordering

lemma *less-enatE*:
 $\llbracket n < enat\ m; !!k. n = enat\ k \implies k < m \implies P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *less-infinityE*:
 $\llbracket n < \infty; !!k. n = enat\ k \implies P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *enat-less-induct*:
assumes *prem*: $\bigwedge n. \forall m::enat. m < n \longrightarrow P\ m \implies P\ n$ **shows** $P\ n$
 $\langle proof \rangle$

instance *enat :: wellorder*
 $\langle proof \rangle$

36.10 Complete Lattice

instantiation *enat :: complete-lattice*
begin

definition *inf-enat* :: $enat \Rightarrow enat \Rightarrow enat$ **where**
 $inf-enat = min$

definition *sup-enat* :: *enat* $\Rightarrow$ *enat* $\Rightarrow$ *enat* **where**
sup-enat = *max*

definition *Inf-enat* :: *enat set* $\Rightarrow$ *enat* **where**
Inf-enat *A* = (if *A* = {} then ∞ else (LEAST *x*. *x* $\in$ *A*))

definition *Sup-enat* :: *enat set* $\Rightarrow$ *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*: $A \neq \{\}$ $\implies$ *eSuc* (*Sup* *A*) = *Sup* (*eSuc* ‘ *A*)
<proof>

lemma *sup-continuous-eSuc*: *sup-continuous* *f* $\implies$ *sup-continuous* ($\lambda x. \text{eSuc } (f x)$)
<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*: $(m + n = (0::\text{enat})) = (m = 0 \wedge n = 0)$
<proof>

lemma *i0-lb* : $(0::\text{enat}) \leq n$
<proof>

lemma *ile0-eq*: $n \leq (0::\text{enat}) \longleftrightarrow n = 0$
<proof>

lemma *not-iless0*: $\neg n < (0::\text{enat})$
<proof>

lemma *i0-less[simp]*: $(0::\text{enat}) < n \longleftrightarrow n \neq 0$
<proof>

lemma *imult-is-0*: $((m::\text{enat}) * n = 0) = (m = 0 \vee n = 0)$
<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 $x \leq \text{Sup } A \longleftrightarrow (\forall y < x. \exists a \in A. y < a)$

<proof>

lemma (in *conditionally-complete-linorder*) *le-cSUP-iff*:

$A \neq \{\} \implies \text{bdd-above } (f'A) \implies x \leq \text{Sup } (f' A) \longleftrightarrow (\forall y < x. \exists i \in A. y < f i)$

<proof>

lemma *le-cSup-iff-less*:

fixes $x :: 'a :: \{\text{conditionally-complete-linorder, dense-linorder}\}$

shows $A \neq \{\} \implies \text{bdd-above } (f'A) \implies x \leq (\text{SUP } i \in A. f i) \longleftrightarrow (\forall y < x. \exists i \in A. y \leq f i)$

<proof>

lemma *le-Sup-iff-less*:

fixes $x :: 'a :: \{\text{complete-linorder, dense-linorder}\}$

shows $x \leq (\text{SUP } i \in A. f i) \longleftrightarrow (\forall y < x. \exists i \in A. y \leq f i)$ (**is** ?lhs = ?rhs)

<proof>

lemma (in *conditionally-complete-linorder*) *cInf-le-iff*:

assumes $A \neq \{\}$ *bdd-below* A

shows $\text{Inf } A \leq x \longleftrightarrow (\forall y > x. \exists a \in A. y > a)$

<proof>

lemma (in *conditionally-complete-linorder*) *cINF-le-iff*:

$A \neq \{\} \implies \text{bdd-below } (f'A) \implies \text{Inf } (f' A) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. y > f i)$

<proof>

lemma *cInf-le-iff-less*:

fixes $x :: 'a :: \{\text{conditionally-complete-linorder, dense-linorder}\}$

shows $A \neq \{\} \implies \text{bdd-below } (f'A) \implies (\text{INF } i \in A. f i) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. f i \leq y)$

<proof>

lemma *Inf-le-iff-less*:

fixes $x :: 'a :: \{\text{complete-linorder, dense-linorder}\}$

shows $(\text{INF } i \in A. f i) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. f i \leq y)$

<proof>

lemma *SUP-pair*:

fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

shows $(\text{SUP } i \in A. \text{ SUP } j \in B. f \ i \ j) = (\text{SUP } p \in A \times B. f \ (\text{fst } p) \ (\text{snd } p))$
 $\langle \text{proof} \rangle$

lemma *INF-pair*:

fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

shows $(\text{INF } i \in A. \text{ INF } j \in B. f \ i \ j) = (\text{INF } p \in A \times B. f \ (\text{fst } p) \ (\text{snd } p))$
 $\langle \text{proof} \rangle$

lemma *INF-Sigma*:

fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

shows $(\text{INF } i \in A. \text{ INF } j \in B \ i. f \ i \ j) = (\text{INF } p \in \text{Sigma } A \ B. f \ (\text{fst } p) \ (\text{snd } p))$
 $\langle \text{proof} \rangle$

37.0.1 *Liminf and Limsup*

definition *Liminf* :: $'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$ **where**

$\text{Liminf } F \ f = (\text{SUP } P \in \{P. \text{eventually } P \ F\}. \text{ INF } x \in \{x. P \ x\}. f \ x)$

definition *Limsup* :: $'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$ **where**

$\text{Limsup } F \ f = (\text{INF } P \in \{P. \text{eventually } P \ F\}. \text{ SUP } x \in \{x. P \ x\}. f \ x)$

abbreviation *liminf* $\equiv \text{Liminf sequentially}$

abbreviation *limsup* $\equiv \text{Limsup sequentially}$

lemma *Liminf-eqI*:

$(\bigwedge P. \text{eventually } P \ F \Longrightarrow \text{Inf } (f \ ' (\text{Collect } P)) \leq x) \Longrightarrow$
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \Longrightarrow \text{Inf } (f \ ' (\text{Collect } P)) \leq y) \Longrightarrow x \leq y) \Longrightarrow \text{Liminf}$
 $F \ f = x$
 $\langle \text{proof} \rangle$

lemma *Limsup-eqI*:

$(\bigwedge P. \text{eventually } P \ F \Longrightarrow x \leq \text{Sup } (f \ ' (\text{Collect } P))) \Longrightarrow$
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \Longrightarrow y \leq \text{Sup } (f \ ' (\text{Collect } P))) \Longrightarrow y \leq x) \Longrightarrow$
 $\text{Limsup } F \ f = x$
 $\langle \text{proof} \rangle$

lemma *liminf-SUP-INF*: $\text{liminf } f = (\text{SUP } n. \text{ INF } m \in \{n..\}. f \ m)$

$\langle \text{proof} \rangle$

lemma *limsup-INF-SUP*: $\text{limsup } f = (\text{INF } n. \text{ SUP } m \in \{n..\}. f \ m)$

$\langle \text{proof} \rangle$

lemma *mem-limsup-iff*: $x \in \text{limsup } A \longleftrightarrow (\exists_F n \text{ in sequentially. } x \in A \ n)$

$\langle \text{proof} \rangle$

lemma *mem-liminf-iff*: $x \in \text{liminf } A \longleftrightarrow (\forall_F n \text{ in sequentially. } x \in A \ n)$

$\langle \text{proof} \rangle$

lemma *Limsup-const*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Limsup } F (\lambda x. c) = c$

$\langle \text{proof} \rangle$

lemma *Liminf-const*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Liminf } F (\lambda x. c) = c$

$\langle \text{proof} \rangle$

lemma *Liminf-mono*:

assumes *ev*: *eventually* $(\lambda x. f x \leq g x) F$

shows $\text{Liminf } F f \leq \text{Liminf } F g$

$\langle \text{proof} \rangle$

lemma *Liminf-eq*:

assumes *eventually* $(\lambda x. f x = g x) F$

shows $\text{Liminf } F f = \text{Liminf } F g$

$\langle \text{proof} \rangle$

lemma *Limsup-mono*:

assumes *ev*: *eventually* $(\lambda x. f x \leq g x) F$

shows $\text{Limsup } F f \leq \text{Limsup } F g$

$\langle \text{proof} \rangle$

lemma *Limsup-eq*:

assumes *eventually* $(\lambda x. f x = g x) \text{ net}$

shows $\text{Limsup } \text{net } f = \text{Limsup } \text{net } g$

$\langle \text{proof} \rangle$

lemma *Liminf-bot[simp]*: $\text{Liminf } \text{bot } f = \text{top}$

$\langle \text{proof} \rangle$

lemma *Limsup-bot[simp]*: $\text{Limsup } \text{bot } f = \text{bot}$

$\langle \text{proof} \rangle$

lemma *Liminf-le-Limsup*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Liminf } F f \leq \text{Limsup } F f$

$\langle \text{proof} \rangle$

lemma *Liminf-bounded*:

assumes *le*: *eventually* $(\lambda n. C \leq X n) F$

shows $C \leq \text{Liminf } F X$

$\langle \text{proof} \rangle$

lemma *Limsup-bounded*:

assumes *le*: *eventually* $(\lambda n. X n \leq C) F$

shows $\text{Limsup } F X \leq C$

$\langle \text{proof} \rangle$

lemma *le-Limsup*:

assumes $F: F \neq \text{bot}$ **and** $x: \forall_F x \text{ in } F. l \leq f x$

shows $l \leq \text{Limsup } F f$

$\langle \text{proof} \rangle$

lemma *Liminf-le*:

assumes $F: F \neq \text{bot}$ **and** $x: \forall_F x \text{ in } F. f x \leq l$

shows $\text{Liminf } F f \leq l$

$\langle \text{proof} \rangle$

lemma *le-Liminf-iff*:

fixes $X :: - \Rightarrow - :: \text{complete-linorder}$

shows $C \leq \text{Liminf } F X \longleftrightarrow (\forall y < C. \text{eventually } (\lambda x. y < X x) F)$

$\langle \text{proof} \rangle$

lemma *Limsup-le-iff*:

fixes $X :: - \Rightarrow - :: \text{complete-linorder}$

shows $C \geq \text{Limsup } F X \longleftrightarrow (\forall y > C. \text{eventually } (\lambda x. y > X x) F)$

$\langle \text{proof} \rangle$

lemma *less-LiminfD*:

$y < \text{Liminf } F (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f x > y) F$

$\langle \text{proof} \rangle$

lemma *Limsup-lessD*:

$y > \text{Limsup } F (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f x < y) F$

$\langle \text{proof} \rangle$

lemma *lim-imp-Liminf*:

fixes $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{ntriv}: \neg \text{trivial-limit } F$

assumes $\text{lim}: (f \longrightarrow f0) F$

shows $\text{Liminf } F f = f0$

$\langle \text{proof} \rangle$

lemma *lim-imp-Limsup*:

fixes $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{ntriv}: \neg \text{trivial-limit } F$

assumes $\text{lim}: (f \longrightarrow f0) F$

shows $\text{Limsup } F f = f0$

$\langle \text{proof} \rangle$

lemma *Liminf-eq-Limsup*:

fixes $f0 :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{ntriv}: \neg \text{trivial-limit } F$

and $\text{lim}: \text{Liminf } F f = f0 \text{ Limsup } F f = f0$

shows $(f \longrightarrow f0) F$

<proof>

lemma *tendsto-iff-Liminf-eq-Limsup*:

fixes $f0 :: 'a :: \{complete-linorder, linorder-topology\}$
shows $\neg trivial-limit\ F \implies (f \longrightarrow f0)\ F \longleftrightarrow (Liminf\ F\ f = f0 \wedge Limsup\ F\ f = f0)$
<proof>

lemma *liminf-subseq-mono*:

fixes $X :: nat \Rightarrow 'a :: complete-linorder$
assumes *strict-mono r*
shows $liminf\ X \leq liminf\ (X \circ r)$
<proof>

lemma *limsup-subseq-mono*:

fixes $X :: nat \Rightarrow 'a :: complete-linorder$
assumes *strict-mono r*
shows $limsup\ (X \circ r) \leq limsup\ X$
<proof>

lemma *continuous-on-imp-continuous-within*:

continuous-on s f $\implies t \subseteq s \implies x \in s \implies continuous\ (at\ x\ within\ t)\ f$
<proof>

lemma *Liminf-compose-continuous-mono*:

fixes $f :: 'a :: \{complete-linorder, linorder-topology\} \Rightarrow 'b :: \{complete-linorder, linorder-topology\}$
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 $f :: 'a :: \{complete-linorder, linorder-topology\} \Rightarrow 'b :: \{complete-linorder, linorder-topology\}$
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 $f :: 'a :: \{complete-linorder, linorder-topology\} \Rightarrow 'b :: \{complete-linorder, linorder-topology\}$
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 $f :: 'a :: \{complete-linorder, linorder-topology\} \Rightarrow 'b :: \{complete-linorder, linorder-topology\}$
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*: $\text{Liminf } (\text{filtermap } f \ F) \ g \leq \text{Liminf } F \ (\lambda x. \ g \ (f \ x))$
 ⟨proof⟩

lemma *Limsup-filtermap-ge*: $\text{Limsup } (\text{filtermap } f \ F) \ g \geq \text{Limsup } F \ (\lambda x. \ g \ (f \ x))$
 ⟨proof⟩

lemma *Liminf-least*: $(\bigwedge P. \text{eventually } P \ F \implies (\text{INF } x \in \text{Collect } P. \ f \ x) \leq x) \implies \text{Liminf } F \ f \leq x$
 ⟨proof⟩

lemma *Limsup-greatest*: $(\bigwedge P. \text{eventually } P \ F \implies x \leq (\text{SUP } x \in \text{Collect } P. \ f \ x)) \implies \text{Limsup } F \ f \geq x$
 ⟨proof⟩

lemma *Liminf-filtermap-ge*: $\text{inj } f \implies \text{Liminf } (\text{filtermap } f \ F) \ g \geq \text{Liminf } F \ (\lambda x. \ g \ (f \ x))$
 ⟨proof⟩

lemma *Limsup-filtermap-le*: $\text{inj } f \implies \text{Limsup } (\text{filtermap } f \ F) \ g \leq \text{Limsup } F \ (\lambda x. \ g \ (f \ x))$
 ⟨proof⟩

lemma *Liminf-filtermap-eq*: $\text{inj } f \implies \text{Liminf } (\text{filtermap } f \ F) \ g = \text{Liminf } F \ (\lambda x. \ g \ (f \ x))$
 ⟨proof⟩

lemma *Limsup-filtermap-eq*: $\text{inj } f \implies \text{Limsup } (\text{filtermap } f \ F) \ g = \text{Limsup } F \ (\lambda x. \ g \ (f \ x))$
 ⟨proof⟩

37.1 More Limits

lemma *convergent-limsup-cl*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
 shows $\text{convergent } X \implies \text{limsup } X = \lim X$
 ⟨proof⟩

lemma *convergent-liminf-cl*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
 shows $\text{convergent } X \implies \text{liminf } X = \lim X$
 ⟨proof⟩

lemma *lim-increasing-cl*:
 assumes $\bigwedge n \ m. \ n \geq m \implies f \ n \geq f \ m$
 obtains l where $f \longrightarrow (l :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\})$
 ⟨proof⟩

lemma *lim-decreasing-cl*:

assumes $\bigwedge n\ m. n \geq m \implies f\ n \leq f\ m$
obtains l **where** $f \longrightarrow (l :: 'a :: \{complete\_linorder, linorder\_topology\})$
 $\langle proof \rangle$

lemma *compact-complete-linorder*:
fixes $X :: nat \Rightarrow 'a :: \{complete\_linorder, linorder\_topology\}$
shows $\exists l\ r. strict\_mono\ r \wedge (X \circ r) \longrightarrow l$
 $\langle proof \rangle$

lemma *tendsto-Limsup*:
fixes $f :: - \Rightarrow 'a :: \{complete\_linorder, linorder\_topology\}$
shows $F \neq bot \implies Limsup\ F\ f = Liminf\ F\ f \implies (f \longrightarrow Limsup\ F\ f)\ F$
 $\langle proof \rangle$

lemma *tendsto-Liminf*:
fixes $f :: - \Rightarrow 'a :: \{complete\_linorder, linorder\_topology\}$
shows $F \neq bot \implies Limsup\ F\ f = Liminf\ F\ f \implies (f \longrightarrow Liminf\ F\ f)\ F$
 $\langle proof \rangle$

end

38 Extended real number line

theory *Extended-Real*
imports *Complex-Main Extended-Nat Liminf-Limsup*
begin

This should be part of *HOL-Library.Extended-Nat* or *HOL-Library.Order-Continuity*, but then the AFP-entry *Jinja-Thread* fails, as it does overload certain named from *Complex-Main*.

lemma *incseq-sumI2*:
fixes $f :: 'i \Rightarrow nat \Rightarrow 'a :: ordered\_comm\_monoid\_add$
shows $(\bigwedge n. n \in A \implies mono\ (f\ n)) \implies mono\ (\lambda i. \sum_{n \in A. f\ n\ i})$
 $\langle proof \rangle$

lemma *incseq-sumI*:
fixes $f :: nat \Rightarrow 'a :: ordered\_comm\_monoid\_add$
assumes $\bigwedge i. 0 \leq f\ i$
shows $incseq\ (\lambda i. sum\ f\ \{.. < i\})$
 $\langle proof \rangle$

lemma *continuous-at-left-imp-sup-continuous*:
fixes $f :: 'a :: \{complete\_linorder, linorder\_topology\} \Rightarrow 'b :: \{complete\_linorder, linorder\_topology\}$
assumes $mono\ f \wedge x. continuous\ (at\_left\ x)\ f$
shows $sup\_continuous\ f$
 $\langle proof \rangle$

lemma *sup-continuous-at-left*:
fixes $f :: 'a :: \{complete\_linorder, linorder\_topology, first\_countable\_topology\} \Rightarrow$

'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} $\Rightarrow$
'b::{complete-linorder, linorder-topology}
shows sup-continuous $f \longleftrightarrow (\forall x. \text{continuous (at-left } x) f) \wedge \text{mono } f$
 <proof>

lemma continuous-at-right-imp-inf-continuous:

fixes f :: 'a:: {complete-linorder, linorder-topology} $\Rightarrow$ 'b:: {complete-linorder, linorder-topology}
assumes mono $f \wedge x. \text{continuous (at-right } x) f$
shows inf-continuous f
 <proof>

lemma inf-continuous-at-right:

fixes f :: 'a:: {complete-linorder, linorder-topology, first-countable-topology} $\Rightarrow$
'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} $\Rightarrow$
'b::{complete-linorder, linorder-topology}
shows inf-continuous $f \longleftrightarrow (\forall x. \text{continuous (at-right } x) f) \wedge \text{mono } f$
 <proof>

instantiation enat :: linorder-topology

begin

definition open-enat :: enat set $\Rightarrow$ bool **where**

open-enat = generate-topology (range lessThan $\cup$ range greaterThan)

instance

<proof>

end

lemma open-enat: open {enat n }

<proof>

lemma open-enat-iff:

fixes A :: enat set

shows open $A \longleftrightarrow (\infty \in A \longrightarrow (\exists n::nat. \{n <..\} \subseteq A))$

<proof>

lemma *nhds-enat*: *nhds* $x = (\text{if } x = \infty \text{ then } \text{INF } i. \text{principal } \{\text{enat } i..\} \text{ else principal } \{x\})$
 $\langle \text{proof} \rangle$

instance *enat* :: *topological-comm-monoid-add*
 $\langle \text{proof} \rangle$

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*

lemma *ereal-cong*: $x = y \implies \text{ereal } x = \text{ereal } y$ $\langle \text{proof} \rangle$

instantiation *ereal* :: *uminus*
begin

fun *uminus-ereal* **where**
 $\quad - (\text{ereal } r) = \text{ereal } (- r)$
 $\quad | - \text{PInfty} = \text{MInfty}$
 $\quad | - \text{MInfty} = \text{PInfty}$

instance $\langle \text{proof} \rangle$

end

instantiation *ereal* :: *infinity*
begin

definition $(\infty :: \text{ereal}) = \text{PInfty}$
instance $\langle \text{proof} \rangle$

end

declare $[[\text{coercion } \text{ereal} :: \text{real} \Rightarrow \text{ereal}]]$

lemma *ereal-uminus-uminus*[*simp*]:
fixes $a :: \text{ereal}$
shows $- (- a) = a$
 $\langle \text{proof} \rangle$

lemma
shows *PInfty-eq-infinity*[*simp*]: $\text{PInfty} = \infty$
and *MInfty-eq-minfinity*[*simp*]: $\text{MInfty} = - \infty$
and *MInfty-neq-PInfty*[*simp*]: $\infty \neq - (\infty :: \text{ereal}) - \infty \neq (\infty :: \text{ereal})$
and *MInfty-neq-ereal*[*simp*]: $\text{ereal } r \neq - \infty - \infty \neq \text{ereal } r$
and *PInfty-neq-ereal*[*simp*]: $\text{ereal } r \neq \infty \infty \neq \text{ereal } r$

and $PInfty\text{-cases}[simp]: (case \ \infty \ of \ ereal \ r \Rightarrow f \ r \mid PInfty \Rightarrow y \mid MInfty \Rightarrow z)$
 $= y$
and $MInfty\text{-cases}[simp]: (case \ - \ \infty \ of \ ereal \ r \Rightarrow f \ r \mid PInfty \Rightarrow y \mid MInfty \Rightarrow z)$
 $= z$
 $\langle proof \rangle$

declare

$PInfty\text{-eq-infinity}[code-post]$
 $MInfty\text{-eq-minfinity}[code-post]$

lemma $[code-unfold]:$

$\infty = PInfty$
 $- PInfty = MInfty$
 $\langle proof \rangle$

lemma $inj\text{-ereal}[simp]: inj\text{-on } ereal \ A$

$\langle proof \rangle$

lemma $ereal\text{-cases}[cases \ type: \ ereal]:$

obtains $(real) \ r$ **where** $x = ereal \ r$
 $\mid (PInf) \ x = \infty$
 $\mid (MInf) \ x = -\infty$
 $\langle proof \rangle$

lemmas $ereal2\text{-cases} = ereal\text{-cases}[case\text{-product } ereal\text{-cases}]$

lemmas $ereal3\text{-cases} = ereal2\text{-cases}[case\text{-product } ereal\text{-cases}]$

lemma $ereal\text{-all-split}: \bigwedge P. (\forall x::ereal. P \ x) \longleftrightarrow P \ \infty \wedge (\forall x. P \ (ereal \ x)) \wedge P \ (-\infty)$

$\langle proof \rangle$

lemma $ereal\text{-ex-split}: \bigwedge P. (\exists x::ereal. P \ x) \longleftrightarrow P \ \infty \vee (\exists x. P \ (ereal \ x)) \vee P \ (-\infty)$

$\langle proof \rangle$

lemma $ereal\text{-uminus-eq-iff}[simp]:$

fixes $a \ b :: ereal$
shows $-a = -b \longleftrightarrow a = b$
 $\langle proof \rangle$

function $real\text{-of-ereal} :: ereal \Rightarrow real$ **where**

$real\text{-of-ereal} \ (ereal \ r) = r$
 $\mid real\text{-of-ereal} \ \infty = 0$
 $\mid real\text{-of-ereal} \ (-\infty) = 0$
 $\langle proof \rangle$

termination $\langle proof \rangle$

lemma $real\text{-of-ereal}[simp]:$

$real\text{-of-ereal} \ (- \ x :: ereal) = - \ (real\text{-of-ereal} \ x)$

$\langle proof \rangle$

lemma *range-ereal*[simp]: $range\ ereal = UNIV - \{\infty, -\infty\}$
 $\langle proof \rangle$

lemma *ereal-range-uminus*[simp]: $range\ uminus = (UNIV::ereal\ set)$
 $\langle proof \rangle$

instantiation *ereal* :: *abs*
begin

function *abs-ereal* **where**

$|ereal\ r| = ereal\ |r|$

$|-\infty| = (\infty::ereal)$

$|\infty| = (\infty::ereal)$

$\langle proof \rangle$

termination $\langle proof \rangle$

instance $\langle proof \rangle$

end

lemma *abs-eq-infinity-cases*[elim!]:

fixes $x :: ereal$

assumes $|x| = \infty$

obtains $x = \infty \mid x = -\infty$

$\langle proof \rangle$

lemma *abs-neq-infinity-cases*[elim!]:

fixes $x :: ereal$

assumes $|x| \neq \infty$

obtains r **where** $x = ereal\ r$

$\langle proof \rangle$

lemma *abs-ereal-uminus*[simp]:

fixes $x :: ereal$

shows $|-x| = |x|$

$\langle proof \rangle$

lemma *ereal-infinity-cases*:

fixes $a :: ereal$

shows $a \neq \infty \implies a \neq -\infty \implies |a| \neq \infty$

$\langle proof \rangle$

38.1.1 Addition

instantiation *ereal* :: $\{one, comm-monoid-add, zero-neq-one\}$
begin

definition $0 = \text{ereal } 0$

definition $1 = \text{ereal } 1$

function *plus-ereal* **where**
 $\text{ereal } r + \text{ereal } p = \text{ereal } (r + p)$
 $|\infty + a = (\infty::\text{ereal})$
 $|a + \infty = (\infty::\text{ereal})$
 $|\text{ereal } r + -\infty = -\infty$
 $|\text{ereal } r + -\infty = -(\infty::\text{ereal})$
 $|\text{ereal } r + -\infty = -(\infty::\text{ereal})$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

lemma *Infty-neq-0*[simp]:
 $(\infty::\text{ereal}) \neq 0 \quad 0 \neq (\infty::\text{ereal})$
 $-(\infty::\text{ereal}) \neq 0 \quad 0 \neq -(\infty::\text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-eq-0*[simp]:
 $\text{ereal } r = 0 \longleftrightarrow r = 0$
 $0 = \text{ereal } r \longleftrightarrow r = 0$
 $\langle \text{proof} \rangle$

lemma *ereal-eq-1*[simp]:
 $\text{ereal } r = 1 \longleftrightarrow r = 1$
 $1 = \text{ereal } r \longleftrightarrow r = 1$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *ereal-0-plus* [simp]: $\text{ereal } 0 + x = x$
and *plus-ereal-0* [simp]: $x + \text{ereal } 0 = x$
 $\langle \text{proof} \rangle$

instance *ereal* :: *numeral* $\langle \text{proof} \rangle$

lemma *real-of-ereal-0*[simp]: $\text{real-of-ereal } (0::\text{ereal}) = 0$
 $\langle \text{proof} \rangle$

lemma *abs-ereal-zero*[simp]: $|0| = (0::\text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-zero*[simp]: $- 0 = (0::\text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-zero-iff*[simp]:

```

fixes  $a :: \text{ereal}$ 
shows  $-a = 0 \longleftrightarrow a = 0$ 
 $\langle \text{proof} \rangle$ 

lemma ereal-plus-eq-PIfty[simp]:
  fixes  $a \ b :: \text{ereal}$ 
  shows  $a + b = \infty \longleftrightarrow a = \infty \vee b = \infty$ 
   $\langle \text{proof} \rangle$ 

lemma ereal-plus-eq-MInfty[simp]:
  fixes  $a \ b :: \text{ereal}$ 
  shows  $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$ 
   $\langle \text{proof} \rangle$ 

lemma ereal-add-cancel-left:
  fixes  $a \ b :: \text{ereal}$ 
  assumes  $a \neq -\infty$ 
  shows  $a + b = a + c \longleftrightarrow a = \infty \vee b = c$ 
   $\langle \text{proof} \rangle$ 

lemma ereal-add-cancel-right:
  fixes  $a \ b :: \text{ereal}$ 
  assumes  $a \neq -\infty$ 
  shows  $b + a = c + a \longleftrightarrow a = \infty \vee b = c$ 
   $\langle \text{proof} \rangle$ 

lemma ereal-real:  $\text{ereal} (\text{real-of-ereal } x) = (\text{if } |x| = \infty \text{ then } 0 \text{ else } x)$ 
 $\langle \text{proof} \rangle$ 

lemma real-of-ereal-add:
  fixes  $a \ b :: \text{ereal}$ 
  shows  $\text{real-of-ereal } (a + b) =$ 
     $(\text{if } (|a| = \infty) \wedge (|b| = \infty) \vee (|a| \neq \infty) \wedge (|b| \neq \infty) \text{ then } \text{real-of-ereal } a +$ 
     $\text{real-of-ereal } b \text{ else } 0)$ 
   $\langle \text{proof} \rangle$ 

38.1.2 Linear order on ereal

instantiation  $\text{ereal} :: \text{linorder}$ 
begin

function less-ereal
where
   $\text{ereal } x < \text{ereal } y \quad \longleftrightarrow x < y$ 
   $| (\infty :: \text{ereal}) < a \quad \longleftrightarrow \text{False}$ 
   $| a < -(\infty :: \text{ereal}) \quad \longleftrightarrow \text{False}$ 
   $| \text{ereal } x < \infty \quad \longleftrightarrow \text{True}$ 
   $| -\infty < \text{ereal } r \quad \longleftrightarrow \text{True}$ 
   $| -\infty < (\infty :: \text{ereal}) \quad \longleftrightarrow \text{True}$ 

```

$\langle proof \rangle$

termination $\langle proof \rangle$

definition $x \leq (y::ereal) \longleftrightarrow x < y \vee x = y$

lemma *ereal-infity-less[simp]*:

fixes $x :: ereal$

shows $x < \infty \longleftrightarrow (x \neq \infty)$

$-\infty < x \longleftrightarrow (x \neq -\infty)$

$\langle proof \rangle$

lemma *ereal-infity-less-eq[simp]*:

fixes $x :: ereal$

shows $\infty \leq x \longleftrightarrow x = \infty$

and $x \leq -\infty \longleftrightarrow x = -\infty$

$\langle proof \rangle$

lemma *ereal-less[simp]*:

$ereal\ r < 0 \longleftrightarrow (r < 0)$

$0 < ereal\ r \longleftrightarrow (0 < r)$

$ereal\ r < 1 \longleftrightarrow (r < 1)$

$1 < ereal\ r \longleftrightarrow (1 < r)$

$0 < (\infty::ereal)$

$-(\infty::ereal) < 0$

$\langle proof \rangle$

lemma *ereal-less-eq[simp]*:

$x \leq (\infty::ereal)$

$-(\infty::ereal) \leq x$

$ereal\ r \leq ereal\ p \longleftrightarrow r \leq p$

$ereal\ r \leq 0 \longleftrightarrow r \leq 0$

$0 \leq ereal\ r \longleftrightarrow 0 \leq r$

$ereal\ r \leq 1 \longleftrightarrow r \leq 1$

$1 \leq ereal\ r \longleftrightarrow 1 \leq r$

$\langle proof \rangle$

lemma *ereal-infity-less-eq2*:

$a \leq b \implies a = \infty \implies b = (\infty::ereal)$

$a \leq b \implies b = -\infty \implies a = -(\infty::ereal)$

$\langle proof \rangle$

instance

$\langle proof \rangle$

end

lemma *ereal-dense2*: $x < y \implies \exists z. x < ereal\ z \wedge ereal\ z < y$

$\langle proof \rangle$

instance *ereal* :: *dense-linorder*
 ⟨*proof*⟩

instance *ereal* :: *ordered-comm-monoid-add*
 ⟨*proof*⟩

lemma *ereal-one-not-less-zero-ereal*[*simp*]: $\neg 1 < (0 :: \text{ereal})$
 ⟨*proof*⟩

lemma *real-of-ereal-positive-mono*:
fixes $x\ y :: \text{ereal}$
shows $0 \leq x \implies x \leq y \implies y \neq \infty \implies \text{real-of-ereal } x \leq \text{real-of-ereal } y$
 ⟨*proof*⟩

lemma *ereal-MInfty-lessI*[*intro*, *simp*]:
fixes $a :: \text{ereal}$
shows $a \neq -\infty \implies -\infty < a$
 ⟨*proof*⟩

lemma *ereal-less-PInfty*[*intro*, *simp*]:
fixes $a :: \text{ereal}$
shows $a \neq \infty \implies a < \infty$
 ⟨*proof*⟩

lemma *ereal-less-ereal-Ex*:
fixes $a\ b :: \text{ereal}$
shows $x < \text{ereal } r \longleftrightarrow x = -\infty \vee (\exists p. p < r \wedge x = \text{ereal } p)$
 ⟨*proof*⟩

lemma *less-PInf-Ex-of-nat*: $x \neq \infty \longleftrightarrow (\exists n :: \text{nat}. x < \text{ereal } (\text{real } n))$
 ⟨*proof*⟩

lemma *ereal-add-strict-mono2*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $a < b$ **and** $c < d$
shows $a + c < b + d$
 ⟨*proof*⟩

lemma *ereal-minus-le-minus*[*simp*]:
fixes $a\ b :: \text{ereal}$
shows $-a \leq -b \longleftrightarrow b \leq a$
 ⟨*proof*⟩

lemma *ereal-minus-less-minus*[*simp*]:
fixes $a\ b :: \text{ereal}$
shows $-a < -b \longleftrightarrow b < a$
 ⟨*proof*⟩

lemma *ereal-le-real-iff*:

$$x \leq \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$$

<proof>

lemma *real-le-ereal-iff*:

$$\text{real-of-ereal } y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$$

<proof>

lemma *ereal-less-real-iff*:

$$x < \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x < y) \wedge (|y| = \infty \longrightarrow x < 0)$$

<proof>

lemma *real-less-ereal-iff*:

$$\text{real-of-ereal } y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y < \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 < x)$$

<proof>

To help with inferences like $\llbracket a < \text{ereal } x; x < y \rrbracket \Longrightarrow a < \text{ereal } y$, where x and y are real.

lemma *le-ereal-le*: $a \leq \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a \leq \text{ereal } y$

<proof>

lemma *le-ereal-less*: $a \leq \text{ereal } x \Longrightarrow x < y \Longrightarrow a < \text{ereal } y$

<proof>

lemma *less-ereal-le*: $a < \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a < \text{ereal } y$

<proof>

lemma *ereal-le-le*: $\text{ereal } y \leq a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x \leq a$

<proof>

lemma *ereal-le-less*: $\text{ereal } y \leq a \Longrightarrow x < y \Longrightarrow \text{ereal } x < a$

<proof>

lemma *ereal-less-le*: $\text{ereal } y < a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x < a$

<proof>

lemma *real-of-ereal-pos*:

fixes $x :: \text{ereal}$

shows $0 \leq x \Longrightarrow 0 \leq \text{real-of-ereal } x$ *<proof>*

lemmas *real-of-ereal-ord-simps* =

ereal-le-real-iff real-le-ereal-iffereal-less-real-iff real-less-ereal-iff

lemma *abs-ereal-ge0[simp]*: $0 \leq x \Longrightarrow |x :: \text{ereal}| = x$

<proof>

lemma *abs-ereal-less0[simp]*: $x < 0 \Longrightarrow |x :: \text{ereal}| = -x$

<proof>

lemma *abs-ereal-pos[simp]*: $0 \leq |x :: \text{ereal}|$

$\langle \text{proof} \rangle$

lemma *ereal-abs-leI*:

fixes $x\ y :: \text{ereal}$

shows $\llbracket x \leq y; -x \leq y \rrbracket \implies |x| \leq y$

$\langle \text{proof} \rangle$

lemma *ereal-abs-add*:

fixes $a\ b :: \text{ereal}$

shows $\text{abs}(a+b) \leq \text{abs}\ a + \text{abs}\ b$

$\langle \text{proof} \rangle$

lemma *real-of-ereal-le-0[simp]*: $\text{real-of-ereal}\ (x :: \text{ereal}) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$

$\langle \text{proof} \rangle$

lemma *abs-real-of-ereal[simp]*: $|\text{real-of-ereal}\ (x :: \text{ereal})| = \text{real-of-ereal}\ |x|$

$\langle \text{proof} \rangle$

lemma *zero-less-real-of-ereal*:

fixes $x :: \text{ereal}$

shows $0 < \text{real-of-ereal}\ x \longleftrightarrow 0 < x \wedge x \neq \infty$

$\langle \text{proof} \rangle$

lemma *ereal-0-le-uminus-iff[simp]*:

fixes $a :: \text{ereal}$

shows $0 \leq -a \longleftrightarrow a \leq 0$

$\langle \text{proof} \rangle$

lemma *ereal-uminus-le-0-iff[simp]*:

fixes $a :: \text{ereal}$

shows $-a \leq 0 \longleftrightarrow 0 \leq a$

$\langle \text{proof} \rangle$

lemma *ereal-add-strict-mono*:

fixes $a\ b\ c\ d :: \text{ereal}$

assumes $a \leq b$

and $0 \leq a$

and $a \neq \infty$

and $c < d$

shows $a + c < b + d$

$\langle \text{proof} \rangle$

lemma *ereal-less-add*:

fixes $a\ b\ c :: \text{ereal}$

shows $|a| \neq \infty \implies c < b \implies a + c < a + b$

$\langle \text{proof} \rangle$

lemma *ereal-add-nonneg-eq-0-iff*:

fixes $a\ b :: \text{ereal}$

shows $0 \leq a \implies 0 \leq b \implies a + b = 0 \longleftrightarrow a = 0 \wedge b = 0$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-eq-reorder*: $- a = b \longleftrightarrow a = (-b::ereal)$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-less-reorder*: $- a < b \longleftrightarrow -b < (a::ereal)$
 $\langle \text{proof} \rangle$

lemma *ereal-less-uminus-reorder*: $a < - b \longleftrightarrow b < - (a::ereal)$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-le-reorder*: $- a \leq b \longleftrightarrow -b \leq (a::ereal)$
 $\langle \text{proof} \rangle$

lemmas *ereal-uminus-reorder* =
ereal-uminus-eq-reorder *ereal-uminus-less-reorder* *ereal-uminus-le-reorder*

lemma *ereal-bot*:
fixes $x :: \text{ereal}$
assumes $\bigwedge B. x \leq \text{ereal } B$
shows $x = -\infty$
 $\langle \text{proof} \rangle$

lemma *ereal-top*:
fixes $x :: \text{ereal}$
assumes $\bigwedge B. x \geq \text{ereal } B$
shows $x = \infty$
 $\langle \text{proof} \rangle$

lemma
shows *ereal-max[simp]*: $\text{ereal } (\max x y) = \max (\text{ereal } x) (\text{ereal } y)$
and *ereal-min[simp]*: $\text{ereal } (\min x y) = \min (\text{ereal } x) (\text{ereal } y)$
 $\langle \text{proof} \rangle$

lemma *ereal-max-0*: $\max 0 (\text{ereal } r) = \text{ereal } (\max 0 r)$
 $\langle \text{proof} \rangle$

lemma
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
shows *ereal-incseq-uminus[simp]*: $\text{incseq } (\lambda x. - f x) \longleftrightarrow \text{decseq } f$
and *ereal-decseq-uminus[simp]*: $\text{decseq } (\lambda x. - f x) \longleftrightarrow \text{incseq } f$
 $\langle \text{proof} \rangle$

lemma *incseq-ereal*: $\text{incseq } f \implies \text{incseq } (\lambda x. \text{ereal } (f x))$
 $\langle \text{proof} \rangle$

lemma *sum-ereal[simp]*: $(\sum x \in A. \text{ereal } (f x)) = \text{ereal } (\sum x \in A. f x)$
 $\langle \text{proof} \rangle$

lemma *sum-list-ereal* [simp]: $\text{sum-list } (\text{map } (\lambda x. \text{ereal } (f x)) \text{ xs}) = \text{ereal } (\text{sum-list } (\text{map } f \text{ xs}))$
 ⟨proof⟩

lemma *sum-Pinfity*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 shows $(\sum_{x \in P}. f x) = \infty \longleftrightarrow \text{finite } P \wedge (\exists i \in P. f i = \infty)$
 ⟨proof⟩

lemma *sum-Inf*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 shows $|\text{sum } f A| = \infty \longleftrightarrow \text{finite } A \wedge (\exists i \in A. |f i| = \infty)$
 ⟨proof⟩

lemma *sum-real-of-ereal*:
 fixes $f :: 'i \Rightarrow \text{ereal}$
 assumes $\bigwedge x. x \in S \implies |f x| \neq \infty$
 shows $(\sum_{x \in S}. \text{real-of-ereal } (f x)) = \text{real-of-ereal } (\text{sum } f S)$
 ⟨proof⟩

lemma *sum-ereal-0*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 assumes *finite* A
 and $\bigwedge i. i \in A \implies 0 \leq f i$
 shows $(\sum_{x \in A}. f x) = 0 \longleftrightarrow (\forall i \in A. f i = 0)$
 ⟨proof⟩

38.1.3 Multiplication

instantiation *ereal* :: {*comm-monoid-mult*, *sgn*}
begin

function *sgn-ereal* :: *ereal* $\Rightarrow$ *ereal* **where**

$\text{sgn } (\text{ereal } r) = \text{ereal } (\text{sgn } r)$
 $|\text{sgn } (\infty :: \text{ereal}) = 1$
 $|\text{sgn } (-\infty :: \text{ereal}) = -1$
 ⟨proof⟩

termination ⟨proof⟩

function *times-ereal* **where**

$\text{ereal } r * \text{ereal } p = \text{ereal } (r * p)$
 $|\text{ereal } r * \infty = (\text{if } r = 0 \text{ then } 0 \text{ else if } r > 0 \text{ then } \infty \text{ else } -\infty)$
 $|\infty * \text{ereal } r = (\text{if } r = 0 \text{ then } 0 \text{ else if } r > 0 \text{ then } \infty \text{ else } -\infty)$
 $|\text{ereal } r * -\infty = (\text{if } r = 0 \text{ then } 0 \text{ else if } r > 0 \text{ then } -\infty \text{ else } \infty)$
 $|- \infty * \text{ereal } r = (\text{if } r = 0 \text{ then } 0 \text{ else if } r > 0 \text{ then } -\infty \text{ else } \infty)$
 $|\infty :: \text{ereal} * \infty = \infty$
 $|- \infty :: \text{ereal} * \infty = -\infty$
 $|\infty :: \text{ereal} * -\infty = -\infty$

| $-(\infty::ereal) * -\infty = \infty$

$\langle proof \rangle$

termination $\langle proof \rangle$

instance

$\langle proof \rangle$

end

lemma $[simp]$:

shows $ereal-1-times$: $ereal\ 1 * x = x$

and $times-ereal-1$: $x *ereal\ 1 = x$

$\langle proof \rangle$

lemma $one-not-le-zero-ereal[simp]$: $\neg (1 \leq (0::ereal))$

$\langle proof \rangle$

lemma $real-ereal-1[simp]$: $real-of-ereal\ (1::ereal) = 1$

$\langle proof \rangle$

lemma $real-of-ereal-le-1$:

fixes $a :: ereal$

shows $a \leq 1 \implies real-of-ereal\ a \leq 1$

$\langle proof \rangle$

lemma $abs-ereal-one[simp]$: $|1| = (1::ereal)$

$\langle proof \rangle$

lemma $ereal-mult-zero[simp]$:

fixes $a :: ereal$

shows $a * 0 = 0$

$\langle proof \rangle$

lemma $ereal-zero-mult[simp]$:

fixes $a :: ereal$

shows $0 * a = 0$

$\langle proof \rangle$

lemma $ereal-m1-less-0[simp]$: $-(1::ereal) < 0$

$\langle proof \rangle$

lemma $ereal-times[simp]$:

$1 \neq (\infty::ereal)\ (\infty::ereal) \neq 1$

$1 \neq -(\infty::ereal)\ -(\infty::ereal) \neq 1$

$\langle proof \rangle$

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]*:
 $a * (\infty::ereal) = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$
<proof>

lemma *ereal-infty-mult[simp]*:
 $(\infty::ereal) * a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$
<proof>

lemma *ereal-mult-strict-right-mono*:
assumes $a < b$
and $0 < c$
and $c < (\infty::ereal)$

shows $a * c < b * c$
 $\langle proof \rangle$

lemma *ereal-mult-strict-left-mono*:
 $a < b \implies 0 < c \implies c < (\infty :: \text{ereal}) \implies c * a < c * b$
 $\langle proof \rangle$

lemma *ereal-mult-right-mono*:
fixes $a\ b\ c :: \text{ereal}$
assumes $a \leq b\ 0 \leq c$
shows $a * c \leq b * c$
 $\langle proof \rangle$

lemma *ereal-mult-left-mono*:
fixes $a\ b\ c :: \text{ereal}$
shows $a \leq b \implies 0 \leq c \implies c * a \leq c * b$
 $\langle proof \rangle$

lemma *ereal-mult-mono*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $b \geq 0\ c \geq 0\ a \leq b\ c \leq d$
shows $a * c \leq b * d$
 $\langle proof \rangle$

lemma *ereal-mult-mono'*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $a \geq 0\ c \geq 0\ a \leq b\ c \leq d$
shows $a * c \leq b * d$
 $\langle proof \rangle$

lemma *ereal-mult-mono-strict*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $b > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
 $\langle proof \rangle$

lemma *ereal-mult-mono-strict'*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $a > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
 $\langle proof \rangle$

lemma *zero-less-one-ereal[simp]*: $0 \leq (1 :: \text{ereal})$
 $\langle proof \rangle$

lemma *ereal-0-le-mult[simp]*: $0 \leq a \implies 0 \leq b \implies 0 \leq a * (b :: \text{ereal})$
 $\langle proof \rangle$

lemma *ereal-right-distrib*:

fixes $r\ a\ b :: \text{ereal}$
shows $0 \leq a \implies 0 \leq b \implies r * (a + b) = r * a + r * b$
 $\langle \text{proof} \rangle$

lemma *ereal-left-distrib*:
fixes $r\ a\ b :: \text{ereal}$
shows $0 \leq a \implies 0 \leq b \implies (a + b) * r = a * r + b * r$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-le-0-iff*:
fixes $a\ b :: \text{ereal}$
shows $a * b \leq 0 \iff (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$
 $\langle \text{proof} \rangle$

lemma *ereal-zero-le-0-iff*:
fixes $a\ b :: \text{ereal}$
shows $0 \leq a * b \iff (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-less-0-iff*:
fixes $a\ b :: \text{ereal}$
shows $a * b < 0 \iff (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$
 $\langle \text{proof} \rangle$

lemma *ereal-zero-less-0-iff*:
fixes $a\ b :: \text{ereal}$
shows $0 < a * b \iff (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$
 $\langle \text{proof} \rangle$

lemma *ereal-left-mult-cong*:
fixes $a\ b\ c :: \text{ereal}$
shows $c = d \implies (d \neq 0 \implies a = b) \implies a * c = b * d$
 $\langle \text{proof} \rangle$

lemma *ereal-right-mult-cong*:
fixes $a\ b\ c :: \text{ereal}$
shows $c = d \implies (d \neq 0 \implies a = b) \implies c * a = d * b$
 $\langle \text{proof} \rangle$

lemma *ereal-distrib*:
fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$
and $|c| \neq \infty$
shows $(a + b) * c = a * c + b * c$
 $\langle \text{proof} \rangle$

lemma *numeral-eq-ereal* [simp]: *numeral* $w = \text{ereal}$ (*numeral* w)
 $\langle \text{proof} \rangle$

lemma *distrib-left-ereal-nn*:

$c \geq 0 \implies (x + y) * \text{ereal } c = x * \text{ereal } c + y * \text{ereal } c$
 $\langle \text{proof} \rangle$

lemma *sum-ereal-right-distrib*:

fixes $f :: 'a \Rightarrow \text{ereal}$
shows $(\bigwedge i. i \in A \implies 0 \leq f i) \implies r * \text{sum } f A = (\sum n \in A. r * f n)$
 $\langle \text{proof} \rangle$

lemma *sum-ereal-left-distrib*:

$(\bigwedge i. i \in A \implies 0 \leq f i) \implies \text{sum } f A * r = (\sum n \in A. f n * r :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *sum-distrib-right-ereal*:

$c \geq 0 \implies \text{sum } f A * \text{ereal } c = (\sum x \in A. f x * c :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-le-epsilon*:

fixes $x y :: \text{ereal}$
assumes $\bigwedge e. 0 < e \implies x \leq y + e$
shows $x \leq y$
 $\langle \text{proof} \rangle$

lemma *ereal-le-epsilon2*:

fixes $x y :: \text{ereal}$
assumes $\bigwedge e :: \text{real}. 0 < e \implies x \leq y + \text{ereal } e$
shows $x \leq y$
 $\langle \text{proof} \rangle$

lemma *ereal-le-real*:

fixes $x y :: \text{ereal}$
assumes $\bigwedge z. x \leq \text{ereal } z \implies y \leq \text{ereal } z$
shows $y \leq x$
 $\langle \text{proof} \rangle$

lemma *prod-ereal-0*:

fixes $f :: 'a \Rightarrow \text{ereal}$
shows $(\prod i \in A. f i) = 0 \longleftrightarrow \text{finite } A \wedge (\exists i \in A. f i = 0)$
 $\langle \text{proof} \rangle$

lemma *prod-ereal-pos*:

fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $\text{pos}: \bigwedge i. i \in I \implies 0 \leq f i$
shows $0 \leq (\prod i \in I. f i)$
 $\langle \text{proof} \rangle$

lemma *prod-PInf*:

fixes $f :: 'a \Rightarrow \text{ereal}$

assumes $\bigwedge i. i \in I \implies 0 \leq f\ i$
shows $(\prod_{i \in I}. f\ i) = \infty \iff \text{finite } I \wedge (\exists i \in I. f\ i = \infty) \wedge (\forall i \in I. f\ i \neq 0)$
 $\langle \text{proof} \rangle$

lemma *prod-ereal*: $(\prod_{i \in A}. \text{ereal } (f\ i)) = \text{ereal } (\text{prod } f\ A)$
 $\langle \text{proof} \rangle$

38.1.4 Power

lemma *ereal-power[simp]*: $(\text{ereal } x) ^ n = \text{ereal } (x ^ n)$
 $\langle \text{proof} \rangle$

lemma *ereal-power-PInf[simp]*: $(\infty :: \text{ereal}) ^ n = (\text{if } n = 0 \text{ then } 1 \text{ else } \infty)$
 $\langle \text{proof} \rangle$

lemma *ereal-power-uminus[simp]*:
fixes $x :: \text{ereal}$
shows $(- x) ^ n = (\text{if even } n \text{ then } x ^ n \text{ else } - (x ^ n))$
 $\langle \text{proof} \rangle$

lemma *ereal-power-numeral[simp]*:
 $(\text{numeral } num :: \text{ereal}) ^ n = \text{ereal } (\text{numeral } num ^ n)$
 $\langle \text{proof} \rangle$

lemma *zero-le-power-ereal[simp]*:
fixes $a :: \text{ereal}$
assumes $0 \leq a$
shows $0 \leq a ^ n$
 $\langle \text{proof} \rangle$

38.1.5 Subtraction

lemma *ereal-minus-minus-image[simp]*:
fixes $S :: \text{ereal set}$
shows $\text{uminus } ' \text{uminus } ' S = S$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-lessThan[simp]*:
fixes $a :: \text{ereal}$
shows $\text{uminus } ' \{.. < a\} = \{-a <..\}$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-greaterThan[simp]*: $\text{uminus } ' \{(a :: \text{ereal}) <..\} = \{.. < -a\}$
 $\langle \text{proof} \rangle$

instantiation *ereal* :: *minus*
begin

definition $x - y = x + -(y :: \text{ereal})$
instance $\langle \text{proof} \rangle$

end

lemma *ereal-minus[simp]*:
 $ereal\ r -ereal\ p =ereal\ (r - p)$
 $-\infty -ereal\ r =-\infty$
 $ereal\ r -\infty =-\infty$
 $(\infty::ereal) -x =\infty$
 $-(\infty::ereal) -\infty =-\infty$
 $x - -y =x + y$
 $x - 0 =x$
 $0 -x =-x$
 $\langle proof \rangle$

lemma *ereal-x-minus-x[simp]*: $x - x = (if\ |x| = \infty\ then\ \infty\ else\ 0::ereal)$
 $\langle proof \rangle$

lemma *ereal-eq-minus-iff*:
fixes $x\ y\ z ::ereal$
shows $x = z - y \longleftrightarrow$
 $(|y| \neq \infty \longrightarrow x + y = z) \wedge$
 $(y = -\infty \longrightarrow x = \infty) \wedge$
 $(y = \infty \longrightarrow z = \infty \longrightarrow x = \infty) \wedge$
 $(y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty)$
 $\langle proof \rangle$

lemma *ereal-eq-minus*:
fixes $x\ y\ z ::ereal$
shows $|y| \neq \infty \implies x = z - y \longleftrightarrow x + y = z$
 $\langle proof \rangle$

lemma *ereal-less-minus-iff*:
fixes $x\ y\ z ::ereal$
shows $x < z - y \longleftrightarrow$
 $(y = \infty \longrightarrow z = \infty \wedge x \neq \infty) \wedge$
 $(y = -\infty \longrightarrow x \neq \infty) \wedge$
 $(|y| \neq \infty \longrightarrow x + y < z)$
 $\langle proof \rangle$

lemma *ereal-less-minus*:
fixes $x\ y\ z ::ereal$
shows $|y| \neq \infty \implies x < z - y \longleftrightarrow x + y < z$
 $\langle proof \rangle$

lemma *ereal-le-minus-iff*:
fixes $x\ y\ z ::ereal$
shows $x \leq z - y \longleftrightarrow (y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty) \wedge (|y| \neq \infty \longrightarrow x + y \leq z)$
 $\langle proof \rangle$

lemma *ereal-le-minus:*

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x \leq z - y \longleftrightarrow x + y \leq z$

$\langle \text{proof} \rangle$

lemma *ereal-minus-less-iff:*

fixes $x\ y\ z :: \text{ereal}$

shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$

$\langle \text{proof} \rangle$

lemma *ereal-minus-less:*

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x - y < z \longleftrightarrow x < z + y$

$\langle \text{proof} \rangle$

lemma *ereal-minus-le-iff:*

fixes $x\ y\ z :: \text{ereal}$

shows $x - y \leq z \longleftrightarrow$

$(y = -\infty \longrightarrow z = \infty) \wedge$

$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$

$(|y| \neq \infty \longrightarrow x \leq z + y)$

$\langle \text{proof} \rangle$

lemma *ereal-minus-le:*

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x - y \leq z \longleftrightarrow x \leq z + y$

$\langle \text{proof} \rangle$

lemma *ereal-minus-eq-minus-iff:*

fixes $a\ b\ c :: \text{ereal}$

shows $a - b = a - c \longleftrightarrow$

$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$

$\langle \text{proof} \rangle$

lemma *ereal-add-le-add-iff:*

fixes $a\ b\ c :: \text{ereal}$

shows $c + a \leq c + b \longleftrightarrow$

$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

$\langle \text{proof} \rangle$

lemma *ereal-add-le-add-iff2:*

fixes $a\ b\ c :: \text{ereal}$

shows $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

$\langle \text{proof} \rangle$

lemma *ereal-mult-le-mult-iff:*

fixes $a\ b\ c :: \text{ereal}$

shows $|c| \neq \infty \implies c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
 ⟨proof⟩

lemma *ereal-minus-mono*:

fixes $A B C D :: \text{ereal}$ **assumes** $A \leq B \ D \leq C$
shows $A - C \leq B - D$
 ⟨proof⟩

lemma *ereal-mono-minus-cancel*:

fixes $a b c :: \text{ereal}$
shows $c - a \leq c - b \implies 0 \leq c \implies c < \infty \implies b \leq a$
 ⟨proof⟩

lemma *real-of-ereal-minus*:

fixes $a b :: \text{ereal}$
shows $\text{real-of-ereal } (a - b) = (\text{if } |a| = \infty \vee |b| = \infty \text{ then } 0 \text{ else } \text{real-of-ereal } a - \text{real-of-ereal } b)$
 ⟨proof⟩

lemma *real-of-ereal-minus'*: $|x| = \infty \longleftrightarrow |y| = \infty \implies \text{real-of-ereal } x - \text{real-of-ereal } y = \text{real-of-ereal } (x - y :: \text{ereal})$
 ⟨proof⟩

lemma *ereal-diff-positive*:

fixes $a b :: \text{ereal}$ **shows** $a \leq b \implies 0 \leq b - a$
 ⟨proof⟩

lemma *ereal-between*:

fixes $x e :: \text{ereal}$
assumes $|x| \neq \infty$
and $0 < e$
shows $x - e < x$
and $x < x + e$
 ⟨proof⟩

lemma *ereal-minus-eq-PIfty-iff*:

fixes $x y :: \text{ereal}$
shows $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$
 ⟨proof⟩

lemma *ereal-diff-add-eq-diff-diff-swap*:

fixes $x y z :: \text{ereal}$
shows $|y| \neq \infty \implies x - (y + z) = x - y - z$
 ⟨proof⟩

lemma *ereal-diff-add-assoc2*:

fixes $x y z :: \text{ereal}$
shows $x + y - z = x - z + y$
 ⟨proof⟩

lemma *ereal-add-uminus-conv-diff*: **fixes** $x\ y\ z :: \text{ereal}$ **shows** $-x + y = y - x$
 $\langle \text{proof} \rangle$

lemma *ereal-minus-diff-eq*:
fixes $x\ y :: \text{ereal}$
shows $\llbracket x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq -\infty \rrbracket \Longrightarrow -(x - y) = y - x$
 $\langle \text{proof} \rangle$

lemma *ediff-le-self* [simp]: $x - y \leq (x :: \text{enat})$
 $\langle \text{proof} \rangle$

lemma *ereal-abs-diff*:
fixes $a\ b :: \text{ereal}$
shows $\text{abs}(a - b) \leq \text{abs}\ a + \text{abs}\ b$
 $\langle \text{proof} \rangle$

38.1.6 Division

instantiation *ereal* :: *inverse*
begin

function *inverse-ereal* **where**
 $\text{inverse}(\text{ereal}\ r) = (\text{if } r = 0 \text{ then } \infty \text{ else } \text{ereal}(\text{inverse}\ r))$
 $\mid \text{inverse}(\infty :: \text{ereal}) = 0$
 $\mid \text{inverse}(-\infty :: \text{ereal}) = 0$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

definition $x \text{ div } y = x * \text{inverse}(y :: \text{ereal})$

instance $\langle \text{proof} \rangle$

end

lemma *real-of-ereal-inverse*[simp]:
fixes $a :: \text{ereal}$
shows $\text{real-of-ereal}(\text{inverse}\ a) = 1 / \text{real-of-ereal}\ a$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse*[simp]:
 $\text{inverse}(0 :: \text{ereal}) = \infty$
 $\text{inverse}(1 :: \text{ereal}) = 1$
 $\langle \text{proof} \rangle$

lemma *ereal-divide*[simp]:
 $\text{ereal}\ r / \text{ereal}\ p = (\text{if } p = 0 \text{ then } \text{ereal}\ r * \infty \text{ else } \text{ereal}(r / p))$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-same*[simp]:

fixes $x :: \text{ereal}$

shows $x / x = (\text{if } |x| = \infty \vee x = 0 \text{ then } 0 \text{ else } 1)$

$\langle \text{proof} \rangle$

lemma *ereal-inv-inv*[simp]:

fixes $x :: \text{ereal}$

shows $\text{inverse} (\text{inverse } x) = (\text{if } x \neq -\infty \text{ then } x \text{ else } \infty)$

$\langle \text{proof} \rangle$

lemma *ereal-inverse-minus*[simp]:

fixes $x :: \text{ereal}$

shows $\text{inverse} (-x) = (\text{if } x = 0 \text{ then } \infty \text{ else } -\text{inverse } x)$

$\langle \text{proof} \rangle$

lemma *ereal-uminus-divide*[simp]:

fixes $x \ y :: \text{ereal}$

shows $-x / y = -(x / y)$

$\langle \text{proof} \rangle$

lemma *ereal-divide-Infty*[simp]:

fixes $x :: \text{ereal}$

shows $x / \infty = 0 \ x / -\infty = 0$

$\langle \text{proof} \rangle$

lemma *ereal-divide-one*[simp]: $x / 1 = (x :: \text{ereal})$

$\langle \text{proof} \rangle$

lemma *ereal-divide-ereal*[simp]: $\infty / \text{ereal } r = (\text{if } 0 \leq r \text{ then } \infty \text{ else } -\infty)$

$\langle \text{proof} \rangle$

lemma *ereal-inverse-nonneg-iff*: $0 \leq \text{inverse } (x :: \text{ereal}) \longleftrightarrow 0 \leq x \vee x = -\infty$

$\langle \text{proof} \rangle$

lemma *inverse-ereal-ge0I*: $0 \leq (x :: \text{ereal}) \implies 0 \leq \text{inverse } x$

$\langle \text{proof} \rangle$

lemma *zero-le-divide-ereal*[simp]:

fixes $a :: \text{ereal}$

assumes $0 \leq a$

and $0 \leq b$

shows $0 \leq a / b$

$\langle \text{proof} \rangle$

lemma *ereal-le-divide-pos*:

fixes $x \ y \ z :: \text{ereal}$

shows $x > 0 \implies x \neq \infty \implies y \leq z / x \longleftrightarrow x * y \leq z$

$\langle \text{proof} \rangle$

lemma *ereal-divide-le-pos*:

fixes $x\ y\ z :: \text{ereal}$

shows $x > 0 \implies x \neq \infty \implies z / x \leq y \longleftrightarrow z \leq x * y$
 $\langle \text{proof} \rangle$

lemma *ereal-le-divide-neg*:

fixes $x\ y\ z :: \text{ereal}$

shows $x < 0 \implies x \neq -\infty \implies y \leq z / x \longleftrightarrow z \leq x * y$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-le-neg*:

fixes $x\ y\ z :: \text{ereal}$

shows $x < 0 \implies x \neq -\infty \implies z / x \leq y \longleftrightarrow x * y \leq z$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-antimono-strict*:

fixes $x\ y :: \text{ereal}$

shows $0 \leq x \implies x < y \implies \text{inverse } y < \text{inverse } x$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-antimono*:

fixes $x\ y :: \text{ereal}$

shows $0 \leq x \implies x \leq y \implies \text{inverse } y \leq \text{inverse } x$
 $\langle \text{proof} \rangle$

lemma *inverse-inverse-Pinfity-iff[simp]*:

fixes $x :: \text{ereal}$

shows $\text{inverse } x = \infty \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-eq-0*:

fixes $x :: \text{ereal}$

shows $\text{inverse } x = 0 \longleftrightarrow x = \infty \vee x = -\infty$
 $\langle \text{proof} \rangle$

lemma *ereal-0-gt-inverse*:

fixes $x :: \text{ereal}$

shows $0 < \text{inverse } x \longleftrightarrow x \neq \infty \wedge 0 \leq x$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-le-0-iff*:

fixes $x :: \text{ereal}$

shows $\text{inverse } x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-eq-0-iff*: $x / y = 0 \longleftrightarrow x = 0 \vee |y :: \text{ereal}| = \infty$

$\langle \text{proof} \rangle$

lemma *ereal-mult-less-right*:

fixes $a\ b\ c :: \text{ereal}$
assumes $b * a < c * a$
and $0 < a$
and $a < \infty$
shows $b < c$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-divide*: **fixes** $a\ b :: \text{ereal}$ **shows** $0 < b \implies b < \infty \implies b * (a / b) = a$
 $\langle \text{proof} \rangle$

lemma *ereal-power-divide*:
fixes $x\ y :: \text{ereal}$
shows $y \neq 0 \implies (x / y) ^ n = x ^ n / y ^ n$
 $\langle \text{proof} \rangle$

lemma *ereal-le-mult-one-interval*:
fixes $x\ y :: \text{ereal}$
assumes $y: y \neq -\infty$
assumes $z: \bigwedge z. 0 < z \implies z < 1 \implies z * x \leq y$
shows $x \leq y$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-right-mono[simp]*:
fixes $x\ y\ z :: \text{ereal}$
assumes $x \leq y$
and $0 < z$
shows $x / z \leq y / z$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-left-mono[simp]*:
fixes $x\ y\ z :: \text{ereal}$
assumes $y \leq x$
and $0 < z$
and $0 < x * y$
shows $z / x \leq z / y$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-zero-left[simp]*:
fixes $a :: \text{ereal}$
shows $0 / a = 0$
 $\langle \text{proof} \rangle$

lemma *ereal-times-divide-eq-left[simp]*:
fixes $a\ b\ c :: \text{ereal}$
shows $b / c * a = b * a / c$
 $\langle \text{proof} \rangle$

lemma *ereal-times-divide-eq*: $a * (b / c :: \text{ereal}) = a * b / c$

$\langle proof \rangle$

lemma *ereal-inverse-real* [simp]: $|z| \neq \infty \implies z \neq 0 \implies \text{ereal} (\text{inverse} (\text{real-of-ereal } z)) = \text{inverse } z$
 $\langle proof \rangle$

lemma *ereal-inverse-mult*:
 $a \neq 0 \implies b \neq 0 \implies \text{inverse} (a * (b::ereal)) = \text{inverse } a * \text{inverse } b$
 $\langle proof \rangle$

lemma *inverse-eq-infinity-iff-eq-zero* [simp]:
 $1/(x::ereal) = \infty \iff x = 0$
 $\langle proof \rangle$

lemma *ereal-distrib-left*:
fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$
and $|c| \neq \infty$
shows $c * (a + b) = c * a + c * b$
 $\langle proof \rangle$

lemma *ereal-distrib-minus-left*:
fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq \infty$
and $a \neq -\infty \vee b \neq -\infty$
and $|c| \neq \infty$
shows $c * (a - b) = c * a - c * b$
 $\langle proof \rangle$

lemma *ereal-distrib-minus-right*:
fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq \infty$
and $a \neq -\infty \vee b \neq -\infty$
and $|c| \neq \infty$
shows $(a - b) * c = a * c - b * c$
 $\langle proof \rangle$

38.2 Complete lattice

instantiation *ereal* :: *lattice*
begin

definition [simp]: $\text{sup } x\ y = (\text{max } x\ y :: \text{ereal})$

definition [simp]: $\text{inf } x\ y = (\text{min } x\ y :: \text{ereal})$

instance $\langle proof \rangle$

end

instantiation *ereal* :: *complete-lattice*
begin

definition *bot* = $(-\infty :: \text{ereal})$

definition *top* = $(\infty :: \text{ereal})$

definition *Sup* *S* = $(\text{SOME } x :: \text{ereal}. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z))$

definition *Inf* *S* = $(\text{SOME } x :: \text{ereal}. (\forall y \in S. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x))$

lemma *ereal-complete-Sup*:

fixes *S* :: *ereal set*

shows $\exists x. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z)$

<proof>

lemma *ereal-complete-uminus-eq*:

fixes *S* :: *ereal set*

shows $(\forall y \in \text{uminus}'S. y \leq x) \wedge (\forall z. (\forall y \in \text{uminus}'S. y \leq z) \longrightarrow x \leq z)$

$\longleftrightarrow (\forall y \in S. -x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq -x)$

<proof>

lemma *ereal-complete-Inf*:

$\exists x. (\forall y \in S :: \text{ereal set}. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x)$

<proof>

instance

<proof>

end

instance *ereal* :: *complete-linorder* *<proof>*

instance *ereal* :: *linear-continuum*

<proof>

lemma *min-PInf* [*simp*]: $\min (\infty :: \text{ereal}) x = x$

<proof>

lemma *min-PInf2* [*simp*]: $\min x (\infty :: \text{ereal}) = x$

<proof>

lemma *max-PInf* [*simp*]: $\max (\infty :: \text{ereal}) x = \infty$

<proof>

lemma *max-PInf2* [*simp*]: $\max x (\infty :: \text{ereal}) = \infty$

<proof>

lemma *min-MInf* [*simp*]: $\min (-\infty :: \text{ereal}) x = -\infty$

$\langle \text{proof} \rangle$

lemma *min-MInf2* [simp]: $\min x \ (-\infty::ereal) = -\infty$
 $\langle \text{proof} \rangle$

lemma *max-MInf* [simp]: $\max \ (-\infty::ereal) \ x = x$
 $\langle \text{proof} \rangle$

lemma *max-MInf2* [simp]: $\max x \ (-\infty::ereal) = x$
 $\langle \text{proof} \rangle$

38.3 Extended real intervals

lemma *real-greaterThanLessThan-infinity-eq*:
 $\text{real-of-ereal} \ \{ N::ereal <..
 $(\text{if } N = \infty \text{ then } \{ \} \text{ else if } N = -\infty \text{ then UNIV else } \{ \text{real-of-ereal } N <.. \})$
 $\langle \text{proof} \rangle$$

lemma *real-greaterThanLessThan-minus-infinity-eq*:
 $\text{real-of-ereal} \ \{ -\infty <..
 $(\text{if } N = \infty \text{ then UNIV else if } N = -\infty \text{ then } \{ \} \text{ else } \{ .. < \text{real-of-ereal } N \})$
 $\langle \text{proof} \rangle$$

lemma *real-greaterThanLessThan-inter*:
 $\text{real-of-ereal} \ \{ N <..
 $\langle \text{proof} \rangle$$

lemma *real-atLeastGreaterThan-eq*: $\text{real-of-ereal} \ \{ N <..
 $(\text{if } N = \infty \text{ then } \{ \} \text{ else}$
 $\text{if } N = -\infty \text{ then}$
 $(\text{if } M = \infty \text{ then UNIV}$
 $\text{else if } M = -\infty \text{ then } \{ \}$
 $\text{else } \{ .. < \text{real-of-ereal } M \})$
 $\text{else if } M = -\infty \text{ then } \{ \}$
 $\text{else if } M = \infty \text{ then } \{ \text{real-of-ereal } N <.. \}$
 $\text{else } \{ \text{real-of-ereal } N <..
 $\langle \text{proof} \rangle$$$

lemma *real-image-ereal-ivl*:
fixes $a \ b::ereal$
shows
 $\text{real-of-ereal} \ \{ a <..
 $(\text{if } a < b \text{ then } (\text{if } a = -\infty \text{ then if } b = \infty \text{ then UNIV else } \{ .. < \text{real-of-ereal } b \})$
 $\text{else if } b = \infty \text{ then } \{ \text{real-of-ereal } a <.. \} \text{ else } \{ \text{real-of-ereal } a <..
 $\text{else } \{ \}$
 $\langle \text{proof} \rangle$$$

lemma **fixes** $a \ b \ c::ereal$

shows *not-inftyI*: $a < b \implies b < c \implies \text{abs } b \neq \infty$
 ⟨*proof*⟩

lemma

interval-neqs:

fixes $r \ s \ t :: \text{real}$

shows $\{r < .. < s\} \neq \{t < ..\}$
and $\{r < .. < s\} \neq \{.. < t\}$
and $\{r < .. < ra\} \neq \text{UNIV}$
and $\{r < ..\} \neq \{.. < s\}$
and $\{r < ..\} \neq \text{UNIV}$
and $\{.. < r\} \neq \text{UNIV}$
and $\{\} \neq \{r < ..\}$
and $\{\} \neq \{.. < r\}$
 ⟨*proof*⟩

lemma *greaterThanLessThan-eq-iff*:

fixes $r \ s \ t \ u :: \text{real}$

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 ‘ $\{a < .. < b\} = \text{real-of-ereal}$ ‘ $\{c < .. < d\} \longleftrightarrow (a \geq b \wedge c \geq d \vee a = c \wedge b = d)$
 ⟨*proof*⟩

lemma *uminus-image-real-of-ereal-image-greaterThanLessThan*:

uminus ‘ *real-of-ereal* ‘ $\{l < .. < u\} = \text{real-of-ereal}$ ‘ $\{-u < .. < -l\}$
 ⟨*proof*⟩

lemma *add-image-real-of-ereal-image-greaterThanLessThan*:

$(+)$ *c* ‘ *real-of-ereal* ‘ $\{l < .. < u\} = \text{real-of-ereal}$ ‘ $\{c + l < .. < c + u\}$
 ⟨*proof*⟩

lemma *add2-image-real-of-ereal-image-greaterThanLessThan*:

$(\lambda x. x + c)$ ‘ *real-of-ereal* ‘ $\{l < .. < u\} = \text{real-of-ereal}$ ‘ $\{l + c < .. < u + c\}$
 ⟨*proof*⟩

lemma *minus-image-real-of-ereal-image-greaterThanLessThan*:

$(-)$ *c* ‘ *real-of-ereal* ‘ $\{l < .. < u\} = \text{real-of-ereal}$ ‘ $\{c - u < .. < c - l\}$
 (**is** ?*l* = ?*r*)
 ⟨*proof*⟩

lemma *real-ereal-bound-lemma-up*:

assumes $s \in \text{real-of-ereal}$ ‘ $\{a < .. < b\}$
assumes $t \notin \text{real-of-ereal}$ ‘ $\{a < .. < b\}$
assumes $s \leq t$
shows $b \neq \infty$
 ⟨*proof*⟩

lemma *real-ereal-bound-lemma-down*:
assumes $s: s \in \text{real-of-ereal } \{a < .. < b\}$
and $t: t \notin \text{real-of-ereal } \{a < .. < b\}$
and $t \leq s$
shows $a \neq -\infty$
 $\langle \text{proof} \rangle$

38.4 Topological space

instantiation *ereal* :: *linear-continuum-topology*
begin

definition *open-ereal* :: *ereal set* $\Rightarrow$ *bool* **where**
open-ereal-generated: *open-ereal* = *generate-topology* (*range lessThan* $\cup$ *range greaterThan*)

instance
 $\langle \text{proof} \rangle$

end

lemma *continuous-on-ereal*[*continuous-intros*]:
assumes $f: \text{continuous-on } s \text{ } f$ **shows** *continuous-on* $s \text{ } (\lambda x. \text{ereal } (f \ x))$
 $\langle \text{proof} \rangle$

lemma *tendsto-ereal*[*tendsto-intros*, *simp*, *intro*]: $(f \longrightarrow x) \ F \Longrightarrow ((\lambda x. \text{ereal } (f \ x)) \longrightarrow \text{ereal } x) \ F$
 $\langle \text{proof} \rangle$

lemma *tendsto-uminus-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes $(f \longrightarrow x) \ F$
shows $((\lambda x. - f \ x :: \text{ereal}) \longrightarrow - x) \ F$
 $\langle \text{proof} \rangle$

lemma *at-infty-ereal-eq-at-top*: $\text{at } \infty = \text{filtermap } \text{ereal } \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *ereal-Lim-uminus*: $(f \longrightarrow f0) \ \text{net} \longleftrightarrow ((\lambda x. - f \ x :: \text{ereal}) \longrightarrow - f0) \ \text{net}$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-less-iff*: $0 < (c :: \text{ereal}) \Longrightarrow c < \infty \Longrightarrow a / c < b \longleftrightarrow a < b * c$
 $\langle \text{proof} \rangle$

lemma *ereal-less-divide-iff*: $0 < (c :: \text{ereal}) \Longrightarrow c < \infty \Longrightarrow a < b / c \longleftrightarrow a * c < b$
 $\langle \text{proof} \rangle$

lemma *tendsto-cmult-ereal*[*tendsto-intros*, *simp*, *intro*]:

assumes $c: |c| \neq \infty$ **and** $f: (f \longrightarrow x) F$ **shows** $((\lambda x. c * f x :: \text{ereal}) \longrightarrow c * x) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-cmult-ereal-not-0*[*tendsto-intros*, *simp*, *intro*]:

assumes $x \neq 0$ **and** $f: (f \longrightarrow x) F$ **shows** $((\lambda x. c * f x :: \text{ereal}) \longrightarrow c * x) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-cadd-ereal*[*tendsto-intros*, *simp*, *intro*]:

assumes $c: y \neq -\infty$ $x \neq -\infty$ **and** $f: (f \longrightarrow x) F$ **shows** $((\lambda x. f x + y :: \text{ereal}) \longrightarrow x + y) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-add-left-ereal*[*tendsto-intros*, *simp*, *intro*]:

assumes $c: |y| \neq \infty$ **and** $f: (f \longrightarrow x) F$ **shows** $((\lambda x. f x + y :: \text{ereal}) \longrightarrow x + y) F$
 $\langle \text{proof} \rangle$

lemma *continuous-at-ereal*[*continuous-intros*]: *continuous* $F f \implies \text{continuous } F (\lambda x. \text{ereal } (f x))$
 $\langle \text{proof} \rangle$

lemma *ereal-Sup*:

assumes $*$: $|SUP a \in A. \text{ereal } a| \neq \infty$
shows $\text{ereal } (Sup A) = (SUP a \in A. \text{ereal } a)$
 $\langle \text{proof} \rangle$

lemma *ereal-SUP*: $|SUP a \in A. \text{ereal } (f a)| \neq \infty \implies \text{ereal } (SUP a \in A. f a) = (SUP a \in A. \text{ereal } (f a))$
 $\langle \text{proof} \rangle$

lemma *ereal-Inf*:

assumes $*$: $|INF a \in A. \text{ereal } a| \neq \infty$
shows $\text{ereal } (Inf A) = (INF a \in A. \text{ereal } a)$
 $\langle \text{proof} \rangle$

lemma *ereal-Inf'*:

assumes $*$: *bdd-below* A $A \neq \{\}$
shows $\text{ereal } (Inf A) = (INF a \in A. \text{ereal } a)$
 $\langle \text{proof} \rangle$

lemma *ereal-INF*: $|INF a \in A. \text{ereal } (f a)| \neq \infty \implies \text{ereal } (INF a \in A. f a) = (INF a \in A. \text{ereal } (f a))$
 $\langle \text{proof} \rangle$

lemma *ereal-Sup-uminus-image-eq*: $Sup (\text{uminus } 'S :: \text{ereal set}) = - Inf S$
 $\langle \text{proof} \rangle$

lemma *ereal-SUP-uminus-eq*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $(\text{SUP } x \in S. \text{uminus } (f x)) = - (\text{INF } x \in S. f x)$

$\langle \text{proof} \rangle$

lemma *ereal-inj-on-uminus*[*intro, simp*]: *inj-on uminus* ($A :: \text{ereal set}$)

$\langle \text{proof} \rangle$

lemma *ereal-Inf-uminus-image-eq*: $\text{Inf } (\text{uminus } ` S :: \text{ereal set}) = - \text{Sup } S$

$\langle \text{proof} \rangle$

lemma *ereal-INF-uminus-eq*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $(\text{INF } x \in S. - f x) = - (\text{SUP } x \in S. f x)$

$\langle \text{proof} \rangle$

lemma *ereal-SUP-uminus*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $(\text{SUP } i \in R. - f i) = - (\text{INF } i \in R. f i)$

$\langle \text{proof} \rangle$

lemma *ereal-SUP-not-infty*:

fixes $f :: - \Rightarrow \text{ereal}$

shows $A \neq \{\} \implies l \neq -\infty \implies u \neq \infty \implies \forall a \in A. l \leq f a \wedge f a \leq u \implies |\text{Sup } (f ` A)| \neq \infty$

$\langle \text{proof} \rangle$

lemma *ereal-INF-not-infty*:

fixes $f :: - \Rightarrow \text{ereal}$

shows $A \neq \{\} \implies l \neq -\infty \implies u \neq \infty \implies \forall a \in A. l \leq f a \wedge f a \leq u \implies |\text{Inf } (f ` A)| \neq \infty$

$\langle \text{proof} \rangle$

lemma *ereal-image-uminus-shift*:

fixes $X Y :: \text{ereal set}$

shows $\text{uminus } ` X = Y \longleftrightarrow X = \text{uminus } ` Y$

$\langle \text{proof} \rangle$

lemma *Sup-eq-MInfty*:

fixes $S :: \text{ereal set}$

shows $\text{Sup } S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$

$\langle \text{proof} \rangle$

lemma *Inf-eq-PInfty*:

fixes $S :: \text{ereal set}$

shows $\text{Inf } S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$

$\langle \text{proof} \rangle$

lemma *Inf-eq-MInfty*:

fixes $S :: \text{ereal set}$

shows $-\infty \in S \implies \text{Inf } S = -\infty$

$\langle \text{proof} \rangle$

lemma *Sup-eq-PInfty*:

fixes $S :: \text{ereal set}$

shows $\infty \in S \implies \text{Sup } S = \infty$

$\langle \text{proof} \rangle$

lemma *not-MInfty-nonneg[simp]*: $0 \leq (x :: \text{ereal}) \implies x \neq -\infty$

$\langle \text{proof} \rangle$

lemma *Sup-ereal-close*:

fixes $e :: \text{ereal}$

assumes $0 < e$

and $S: |\text{Sup } S| \neq \infty \ S \neq \{\}$

shows $\exists x \in S. \text{Sup } S - e < x$

$\langle \text{proof} \rangle$

lemma *Inf-ereal-close*:

fixes $e :: \text{ereal}$

assumes $|\text{Inf } X| \neq \infty$

and $0 < e$

shows $\exists x \in X. x < \text{Inf } X + e$

$\langle \text{proof} \rangle$

lemma *SUP-PInfty*:

$(\bigwedge n :: \text{nat}. \exists i \in A. \text{ereal } (\text{real } n) \leq f\ i) \implies (\text{SUP } i \in A. f\ i :: \text{ereal}) = \infty$

$\langle \text{proof} \rangle$

lemma *SUP-nat-Infty*: $(\text{SUP } i. \text{ereal } (\text{real } i)) = \infty$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-add-left*:

assumes $I \neq \{\}$ $c \neq -\infty$

shows $(\text{SUP } i \in I. f\ i + c :: \text{ereal}) = (\text{SUP } i \in I. f\ i) + c$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-add-right*:

fixes $c :: \text{ereal}$

shows $I \neq \{\} \implies c \neq -\infty \implies (\text{SUP } i \in I. c + f\ i) = c + (\text{SUP } i \in I. f\ i)$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-minus-right*:

assumes $I \neq \{\}$ $c \neq -\infty$

shows $(\text{SUP } i \in I. c - f\ i :: \text{ereal}) = c - (\text{INF } i \in I. f\ i)$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-minus-left*:

assumes $I \neq \{\}$ $c \neq \infty$

shows $(\text{SUP } i \in I. f\ i - c :: \text{ereal}) = (\text{SUP } i \in I. f\ i) - c$

<proof>

lemma *INF-ereal-minus-right*:

assumes $I \neq \{\}$ **and** $|c| \neq \infty$

shows $(\text{INF } i \in I. c - f\ i) = c - (\text{SUP } i \in I. f\ i :: \text{ereal})$

<proof>

lemma *SUP-ereal-le-addI*:

fixes $f :: 'i \Rightarrow \text{ereal}$

assumes $\bigwedge i. f\ i + y \leq z$ **and** $y \neq -\infty$

shows $\text{Sup } (f \text{ ' UNIV}) + y \leq z$

<proof>

lemma *SUP-combine*:

fixes $f :: 'a :: \text{semilattice-sup} \Rightarrow 'a :: \text{semilattice-sup} \Rightarrow 'b :: \text{complete-lattice}$

assumes *mono*: $\bigwedge a\ b\ c\ d. a \leq b \implies c \leq d \implies f\ a\ c \leq f\ b\ d$

shows $(\text{SUP } i \in \text{UNIV}. \text{SUP } j \in \text{UNIV}. f\ i\ j) = (\text{SUP } i. f\ i\ i)$

<proof>

lemma *SUP-ereal-add*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ereal}$

assumes *inc*: $\text{incseq } f\ \text{incseq } g$

and *pos*: $\bigwedge i. f\ i \neq -\infty \bigwedge i. g\ i \neq -\infty$

shows $(\text{SUP } i. f\ i + g\ i) = \text{Sup } (f \text{ ' UNIV}) + \text{Sup } (g \text{ ' UNIV})$

<proof>

lemma *INF-eq-minf*: $(\text{INF } i \in I. f\ i :: \text{ereal}) \neq -\infty \longleftrightarrow (\exists b > -\infty. \forall i \in I. b \leq f\ i)$

<proof>

lemma *INF-ereal-add-left*:

assumes $I \neq \{\}$ $c \neq -\infty \bigwedge x. x \in I \implies 0 \leq f\ x$

shows $(\text{INF } i \in I. f\ i + c :: \text{ereal}) = (\text{INF } i \in I. f\ i) + c$

<proof>

lemma *INF-ereal-add-right*:

assumes $I \neq \{\}$ $c \neq -\infty \bigwedge x. x \in I \implies 0 \leq f\ x$

shows $(\text{INF } i \in I. c + f\ i :: \text{ereal}) = c + (\text{INF } i \in I. f\ i)$

<proof>

lemma *INF-ereal-add-directed*:

fixes $f\ g :: 'a \Rightarrow \text{ereal}$

assumes *nonneg*: $\bigwedge i. i \in I \implies 0 \leq f\ i \bigwedge i. i \in I \implies 0 \leq g\ i$

assumes *directed*: $\bigwedge i\ j. i \in I \implies j \in I \implies \exists k \in I. f\ i + g\ j \geq f\ k + g\ k$

shows $(\text{INF } i \in I. f\ i + g\ i) = (\text{INF } i \in I. f\ i) + (\text{INF } i \in I. g\ i)$

<proof>

lemma *INF-ereal-add:*

fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\text{decseq } f \text{ decseq } g$
and $\text{fin}: \bigwedge i. f \ i \neq \infty \ \bigwedge i. g \ i \neq \infty$
shows $(\text{INF } i. f \ i + g \ i) = \text{Inf } (f \text{ ‘ UNIV}) + \text{Inf } (g \text{ ‘ UNIV})$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-add-pos:*

fixes $f \ g :: \text{nat} \Rightarrow \text{ereal}$
assumes $\text{inc}: \text{incseq } f \text{ incseq } g$
and $\text{pos}: \bigwedge i. 0 \leq f \ i \ \bigwedge i. 0 \leq g \ i$
shows $(\text{SUP } i. f \ i + g \ i) = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-sum:*

fixes $f \ g :: 'a \Rightarrow \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge n. n \in A \implies \text{incseq } (f \ n)$
and $\text{pos}: \bigwedge n \ i. n \in A \implies 0 \leq f \ n \ i$
shows $(\text{SUP } i. \sum_{n \in A} f \ n \ i) = (\sum_{n \in A} \text{Sup } ((f \ n) \text{ ‘ UNIV}))$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-mult-left:*

fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $I \neq \{\}$
assumes $f: \bigwedge i. i \in I \implies 0 \leq f \ i$ **and** $c: 0 \leq c$
shows $(\text{SUP } i \in I. c * f \ i) = c * (\text{SUP } i \in I. f \ i)$
 $\langle \text{proof} \rangle$

lemma *countable-approach:*

fixes $x :: \text{ereal}$
assumes $x \neq -\infty$
shows $\exists f. \text{incseq } f \wedge (\forall i :: \text{nat}. f \ i < x) \wedge (f \longrightarrow x)$
 $\langle \text{proof} \rangle$

lemma *Sup-countable-SUP:*

assumes $A \neq \{\}$
shows $\exists f :: \text{nat} \Rightarrow \text{ereal}. \text{incseq } f \wedge \text{range } f \subseteq A \wedge \text{Sup } A = (\text{SUP } i. f \ i)$
 $\langle \text{proof} \rangle$

lemma *Inf-countable-INF:*

assumes $A \neq \{\}$ **shows** $\exists f :: \text{nat} \Rightarrow \text{ereal}. \text{decseq } f \wedge \text{range } f \subseteq A \wedge \text{Inf } A = (\text{INF } i. f \ i)$
 $\langle \text{proof} \rangle$

lemma *SUP-countable-SUP:*

$A \neq \{\} \implies \exists f :: \text{nat} \Rightarrow \text{ereal}. \text{range } f \subseteq g \text{ ‘ } A \wedge \text{Sup } (g \text{ ‘ } A) = \text{Sup } (f \text{ ‘ UNIV})$
 $\langle \text{proof} \rangle$

38.5 Relation to *enat*

definition *ereal-of-enat* $n = (\text{case } n \text{ of } \text{enat } n \Rightarrow \text{ereal } (\text{real } n) \mid \infty \Rightarrow \infty)$

declare $[[\text{coercion } \text{ereal-of-enat} :: \text{enat} \Rightarrow \text{ereal}]]$

declare $[[\text{coercion } (\lambda n. \text{ereal } (\text{real } n)) :: \text{nat} \Rightarrow \text{ereal}]]$

lemma *ereal-of-enat-simps*[*simp*]:

ereal-of-enat (*enat* n) = *ereal* n

ereal-of-enat $\infty = \infty$

<proof>

lemma *ereal-of-enat-le-iff*[*simp*]: *ereal-of-enat* $m \leq \text{ereal-of-enat } n \longleftrightarrow m \leq n$

<proof>

lemma *ereal-of-enat-less-iff*[*simp*]: *ereal-of-enat* $m < \text{ereal-of-enat } n \longleftrightarrow m < n$

<proof>

lemma *numeral-le-ereal-of-enat-iff*[*simp*]: *numeral* $m \leq \text{ereal-of-enat } n \longleftrightarrow \text{numeral } m \leq n$

<proof>

lemma *numeral-less-ereal-of-enat-iff*[*simp*]: *numeral* $m < \text{ereal-of-enat } n \longleftrightarrow \text{numeral } m < n$

<proof>

lemma *ereal-of-enat-ge-zero-cancel-iff*[*simp*]: $0 \leq \text{ereal-of-enat } n \longleftrightarrow 0 \leq n$

<proof>

lemma *ereal-of-enat-gt-zero-cancel-iff*[*simp*]: $0 < \text{ereal-of-enat } n \longleftrightarrow 0 < n$

<proof>

lemma *ereal-of-enat-zero*[*simp*]: *ereal-of-enat* $0 = 0$

<proof>

lemma *ereal-of-enat-inf*[*simp*]: *ereal-of-enat* $n = \infty \longleftrightarrow n = \infty$

<proof>

lemma *ereal-of-enat-add*: *ereal-of-enat* ($m + n$) = *ereal-of-enat* $m + \text{ereal-of-enat } n$

<proof>

lemma *ereal-of-enat-sub*:

assumes $n \leq m$

shows *ereal-of-enat* ($m - n$) = *ereal-of-enat* $m - \text{ereal-of-enat } n$

<proof>

lemma *ereal-of-enat-mult*:

ereal-of-enat ($m * n$) = *ereal-of-enat* $m * \text{ereal-of-enat } n$

<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* $n \geq 0$

<proof>

lemma *ereal-of-enat-Sup*:

assumes $A \neq \{\}$ **shows** *ereal-of-enat* ($\text{Sup } A$) = ($\text{SUP } a \in A. \text{ereal-of-enat } a$)

<proof>

lemma *ereal-of-enat-SUP*:

$A \neq \{\} \implies \text{ereal-of-enat } (\text{SUP } a \in A. f a) = (\text{SUP } a \in A. \text{ereal-of-enat } (f a))$

<proof>

38.6 Limits on *ereal*

lemma *open-PInfy*: *open* $A \implies \infty \in A \implies (\exists x. \{\text{ereal } x < ..\} \subseteq A)$

<proof>

lemma *open-MInfy*: *open* $A \implies -\infty \in A \implies (\exists x. \{.. < \text{ereal } x\} \subseteq A)$

<proof>

lemma *open-ereal-vimage*: *open* $S \implies \text{open } (\text{ereal } - ' S)$

<proof>

lemma *open-ereal*: *open* $S \implies \text{open } (\text{ereal } ' S)$

<proof>

lemma *open-image-real-of-ereal*:

fixes $X :: \text{ereal set}$

assumes *open* X

assumes *infy*: $\infty \notin X \text{ } -\infty \notin X$

shows *open* (*real-of-ereal* ' X)

<proof>

lemma *eventually-finite*:

fixes $x :: \text{ereal}$

assumes $|x| \neq \infty$ ($f \longrightarrow x$) F

shows *eventually* ($\lambda x. |f x| \neq \infty$) F

<proof>

lemma *open-ereal-def*:

open $A \longleftrightarrow \text{open } (\text{ereal } - ' A) \wedge (\infty \in A \longrightarrow (\exists x. \{\text{ereal } x < ..\} \subseteq A)) \wedge (-\infty \in A \longrightarrow (\exists x. \{.. < \text{ereal } x\} \subseteq A))$

(**is** *open* $A \longleftrightarrow ?rhs$)

<proof>

lemma *open-PInfy2*:
assumes *open A*
and $\infty \in A$
obtains *x* **where** $\{ereal\ x < ..\} \subseteq A$
 $\langle proof \rangle$

lemma *open-MInfy2*:
assumes *open A*
and $-\infty \in A$
obtains *x* **where** $\{.. <ereal\ x\} \subseteq A$
 $\langle proof \rangle$

lemma *ereal-openE*:
assumes *open A*
obtains *x y* **where** *open (ereal - 'A)*
and $\infty \in A \implies \{ereal\ x < ..\} \subseteq A$
and $-\infty \in A \implies \{.. <ereal\ y\} \subseteq A$
 $\langle proof \rangle$

lemmas *open-ereal-lessThan* = *open-lessThan*[**where** '*a=ereal*]
lemmas *open-ereal-greaterThan* = *open-greaterThan*[**where** '*a=ereal*]
lemmas *ereal-open-greaterThanLessThan* = *open-greaterThanLessThan*[**where** '*a=ereal*]
lemmas *closed-ereal-atLeast* = *closed-atLeast*[**where** '*a=ereal*]
lemmas *closed-ereal-atMost* = *closed-atMost*[**where** '*a=ereal*]
lemmas *closed-ereal-atLeastAtMost* = *closed-atLeastAtMost*[**where** '*a=ereal*]
lemmas *closed-ereal-singleton* = *closed-singleton*[**where** '*a=ereal*]

lemma *ereal-open-cont-interval*:
fixes *S :: ereal set*
assumes *open S*
and $x \in S$
and $|x| \neq \infty$
obtains *e* **where** $e > 0$ **and** $\{x-e < .. < x+e\} \subseteq S$
 $\langle proof \rangle$

lemma *ereal-open-cont-interval2*:
fixes *S :: ereal set*
assumes *open S*
and $x \in S$
and $x: |x| \neq \infty$
obtains *a b* **where** $a < x$ **and** $x < b$ **and** $\{a < .. < b\} \subseteq S$
 $\langle proof \rangle$

38.6.1 Convergent sequences

lemma *lim-real-of-ereal[simp]*:
assumes *lim: (f $\longrightarrow$ ereal x) net*
shows $((\lambda x. real-of-ereal\ (f\ x)) \longrightarrow x)$ *net*
 $\langle proof \rangle$

lemma *lim-ereal[simp]*: $((\lambda n. \text{ereal } (f\ n)) \longrightarrow \text{ereal } x) \text{ net} \longleftrightarrow (f \longrightarrow x) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *convergent-real-imp-convergent-ereal*:

assumes *convergent* a

shows *convergent* $(\lambda n. \text{ereal } (a\ n))$ **and** $\lim (\lambda n. \text{ereal } (a\ n)) = \text{ereal } (\lim a)$
 $\langle \text{proof} \rangle$

lemma *tendsto-PIInfty*: $(f \longrightarrow \infty) F \longleftrightarrow (\forall r. \text{eventually } (\lambda x. \text{ereal } r < f\ x) F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-PIInfty'*: $(f \longrightarrow \infty) F = (\forall r > c. \text{eventually } (\lambda x. \text{ereal } r < f\ x) F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-PIInfty-eq-at-top*:

$((\lambda z. \text{ereal } (f\ z)) \longrightarrow \infty) F \longleftrightarrow (LIM\ z\ F. f\ z :> \text{at-top})$
 $\langle \text{proof} \rangle$

lemma *tendsto-MInfty*: $(f \longrightarrow -\infty) F \longleftrightarrow (\forall r. \text{eventually } (\lambda x. f\ x < \text{ereal } r) F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-MInfty'*: $(f \longrightarrow -\infty) F = (\forall r < c. \text{eventually } (\lambda x. \text{ereal } r > f\ x) F)$
 $\langle \text{proof} \rangle$

lemma *Lim-PIInfty*: $f \longrightarrow \infty \longleftrightarrow (\forall B. \exists N. \forall n \geq N. f\ n \geq \text{ereal } B)$
 $\langle \text{proof} \rangle$

lemma *Lim-MInfty*: $f \longrightarrow -\infty \longleftrightarrow (\forall B. \exists N. \forall n \geq N. \text{ereal } B \geq f\ n)$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded-PIInfty*: $f \longrightarrow l \implies (\bigwedge n. f\ n \leq \text{ereal } B) \implies l \neq \infty$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded-MInfty*: $f \longrightarrow l \implies (\bigwedge n. \text{ereal } B \leq f\ n) \implies l \neq -\infty$
 $\langle \text{proof} \rangle$

lemma *tendsto-zero-erealI*:

assumes $\bigwedge e. e > 0 \implies \text{eventually } (\lambda x. |f\ x| < \text{ereal } e) F$

shows $(f \longrightarrow 0) F$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded-PIInfty2*: $f \longrightarrow l \implies \forall n \geq N. f\ n \leq \text{ereal } B \implies l \neq \infty$
 $\langle \text{proof} \rangle$

lemma *real-of-ereal-mult[simp]*:

fixes $a\ b :: \text{ereal}$
shows $\text{real-of-ereal } (a * b) = \text{real-of-ereal } a * \text{real-of-ereal } b$
 $\langle \text{proof} \rangle$

lemma $\text{real-of-ereal-eq-0}$:
fixes $x :: \text{ereal}$
shows $\text{real-of-ereal } x = 0 \longleftrightarrow x = \infty \vee x = -\infty \vee x = 0$
 $\langle \text{proof} \rangle$

lemma $\text{tendsto-ereal-realD}$:
fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $x \neq 0$
and $\text{tendsto}: ((\lambda x. \text{ereal } (\text{real-of-ereal } (f\ x))) \longrightarrow x) \text{ net}$
shows $(f \longrightarrow x) \text{ net}$
 $\langle \text{proof} \rangle$

lemma $\text{tendsto-ereal-realI}$:
fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $x: |x| \neq \infty$ **and** $\text{tendsto}: (f \longrightarrow x) \text{ net}$
shows $((\lambda x. \text{ereal } (\text{real-of-ereal } (f\ x))) \longrightarrow x) \text{ net}$
 $\langle \text{proof} \rangle$

lemma $\text{ereal-mult-cancel-left}$:
fixes $a\ b\ c :: \text{ereal}$
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
 $\langle \text{proof} \rangle$

lemma tendsto-add-ereal :
fixes $x\ y :: \text{ereal}$
assumes $x: |x| \neq \infty$ **and** $y: |y| \neq \infty$
assumes $f: (f \longrightarrow x) F$ **and** $g: (g \longrightarrow y) F$
shows $((\lambda x. f\ x + g\ x) \longrightarrow x + y) F$
 $\langle \text{proof} \rangle$

lemma $\text{tendsto-add-ereal-nonneg}$:
fixes $x\ y :: \text{ereal}$
assumes $x \neq -\infty$ $y \neq -\infty$ $(f \longrightarrow x) F$ $(g \longrightarrow y) F$
shows $((\lambda x. f\ x + g\ x) \longrightarrow x + y) F$
 $\langle \text{proof} \rangle$

lemma $\text{ereal-inj-affinity}$:
fixes $m\ t :: \text{ereal}$
assumes $|m| \neq \infty$
and $m \neq 0$
and $|t| \neq \infty$
shows $\text{inj-on } (\lambda x. m * x + t) A$
 $\langle \text{proof} \rangle$

lemma $\text{ereal-PInfy-eq-plus[simp]}$:

fixes $a\ b :: \text{ereal}$
shows $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$
 $\langle \text{proof} \rangle$

lemma *ereal-MInfty-eq-plus[simp]*:
fixes $a\ b :: \text{ereal}$
shows $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$
 $\langle \text{proof} \rangle$

lemma *ereal-less-divide-pos*:
fixes $x\ y :: \text{ereal}$
shows $x > 0 \implies x \neq \infty \implies y < z / x \longleftrightarrow x * y < z$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-less-pos*:
fixes $x\ y\ z :: \text{ereal}$
shows $x > 0 \implies x \neq \infty \implies y / x < z \longleftrightarrow y < x * z$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-eq*:
fixes $a\ b\ c :: \text{ereal}$
shows $b \neq 0 \implies |b| \neq \infty \implies a / b = c \longleftrightarrow a = b * c$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-not-MInfty[simp]*: *inverse* $(a :: \text{ereal}) \neq -\infty$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-m1[simp]*: $x * \text{ereal } (-1) = -x$
 $\langle \text{proof} \rangle$

lemma *ereal-real'*:
assumes $|x| \neq \infty$
shows $\text{ereal } (\text{real-of-ereal } x) = x$
 $\langle \text{proof} \rangle$

lemma *real-ereal-id*: $\text{real-of-ereal} \circ \text{ereal} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *open-image-ereal*: $\text{open}(UNIV - \{ \infty, (-\infty :: \text{ereal}) \})$
 $\langle \text{proof} \rangle$

lemma *ereal-le-distrib*:
fixes $a\ b\ c :: \text{ereal}$
shows $c * (a + b) \leq c * a + c * b$
 $\langle \text{proof} \rangle$

lemma *ereal-pos-distrib*:
fixes $a\ b\ c :: \text{ereal}$
assumes $0 \leq c$

and $c \neq \infty$
shows $c * (a + b) = c * a + c * b$
 $\langle \text{proof} \rangle$

lemma *ereal-LimI-finite*:

fixes $x :: \text{ereal}$
assumes $|x| \neq \infty$
and $\bigwedge r. 0 < r \implies \exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r$
shows $u \longrightarrow x$
 $\langle \text{proof} \rangle$

lemma *tendsto-obtains-N*:

assumes $f \longrightarrow f0$
assumes *open* S
and $f0 \in S$
obtains N **where** $\forall n \geq N. f\ n \in S$
 $\langle \text{proof} \rangle$

lemma *ereal-LimI-finite-iff*:

fixes $x :: \text{ereal}$
assumes $|x| \neq \infty$
shows $u \longrightarrow x \longleftrightarrow (\forall r. 0 < r \longrightarrow (\exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r))$
(is ?lhs $\longleftrightarrow$?rhs)
 $\langle \text{proof} \rangle$

lemma *ereal-Limsup-uminus*:

fixes $f :: 'a \Rightarrow \text{ereal}$
shows $\text{Limsup}\ \text{net}\ (\lambda x. - (f\ x)) = -\ \text{Liminf}\ \text{net}\ f$
 $\langle \text{proof} \rangle$

lemma *liminf-bounded-iff*:

fixes $x :: \text{nat} \Rightarrow \text{ereal}$
shows $C \leq \text{liminf}\ x \longleftrightarrow (\forall B < C. \exists N. \forall n \geq N. B < x\ n)$
(is ?lhs $\longleftrightarrow$?rhs)
 $\langle \text{proof} \rangle$

lemma *Liminf-add-le*:

fixes $f\ g :: - \Rightarrow \text{ereal}$
assumes $F: F \neq \text{bot}$
assumes *ev*: *eventually* $(\lambda x. 0 \leq f\ x)\ F$ *eventually* $(\lambda x. 0 \leq g\ x)\ F$
shows $\text{Liminf}\ F\ f + \text{Liminf}\ F\ g \leq \text{Liminf}\ F\ (\lambda x. f\ x + g\ x)$
 $\langle \text{proof} \rangle$

lemma *Sup-ereal-mult-right'*:

assumes *nonempty*: $Y \neq \{\}$
and $x: x \geq 0$
shows $(\text{SUP}\ i \in Y. f\ i) * \text{ereal}\ x = (\text{SUP}\ i \in Y. f\ i * \text{ereal}\ x)$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

lemma *Sup-ereal-mult-left'*:

$\llbracket Y \neq \{\}; x \geq 0 \rrbracket \implies \text{ereal } x * (\text{SUP } i \in Y. f i) = (\text{SUP } i \in Y. \text{ereal } x * f i)$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-add[order-continuous-intros]*:

fixes $f g :: 'a :: \text{complete-lattice} \Rightarrow \text{ereal}$
assumes $nn: \bigwedge x. 0 \leq f x \bigwedge x. 0 \leq g x$ **and** $cont: \text{sup-continuous } f \text{ sup-continuous } g$
shows $\text{sup-continuous } (\lambda x. f x + g x)$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-mult-right[order-continuous-intros]*:

$0 \leq c \implies c < \infty \implies \text{sup-continuous } f \implies \text{sup-continuous } (\lambda x. f x * c :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-mult-left[order-continuous-intros]*:

$0 \leq c \implies c < \infty \implies \text{sup-continuous } f \implies \text{sup-continuous } (\lambda x. c * f x :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-ereal-of-enat[order-continuous-intros]*:

assumes $f: \text{sup-continuous } f$ **shows** $\text{sup-continuous } (\lambda x. \text{ereal-of-enat } (f x))$
 $\langle \text{proof} \rangle$

38.6.2 Sums

lemma *sums-ereal-positive*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. 0 \leq f i$
shows $f \text{ sums } (\text{SUP } n. \sum i < n. f i)$
 $\langle \text{proof} \rangle$

lemma *summable-ereal-pos*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. 0 \leq f i$
shows $\text{summable } f$
 $\langle \text{proof} \rangle$

lemma *sums-ereal*: $(\lambda x. \text{ereal } (f x)) \text{ sums } \text{ereal } x \longleftrightarrow f \text{ sums } x$

$\langle \text{proof} \rangle$

lemma *suminf-ereal-eq-SUP*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. 0 \leq f i$
shows $(\sum x. f x) = (\text{SUP } n. \sum i < n. f i)$
 $\langle \text{proof} \rangle$

lemma *suminf-bound*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes $\forall N. (\sum n < N. f\ n) \leq x$
and $pos: \bigwedge n. 0 \leq f\ n$
shows $\text{suminf } f \leq x$
 $\langle proof \rangle$

lemma *suminf-bound-add*:
fixes $f :: nat \Rightarrow ereal$
assumes $\forall N. (\sum n < N. f\ n) + y \leq x$
and $pos: \bigwedge n. 0 \leq f\ n$
and $y \neq -\infty$
shows $\text{suminf } f + y \leq x$
 $\langle proof \rangle$

lemma *suminf-upper*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n. 0 \leq f\ n$
shows $(\sum n < N. f\ n) \leq (\sum n. f\ n)$
 $\langle proof \rangle$

lemma *suminf-0-le*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n. 0 \leq f\ n$
shows $0 \leq (\sum n. f\ n)$
 $\langle proof \rangle$

lemma *suminf-le-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge N. f\ N \leq g\ N$
and $\bigwedge N. 0 \leq f\ N$
shows $\text{suminf } f \leq \text{suminf } g$
 $\langle proof \rangle$

lemma *suminf-half-series-ereal*: $(\sum n. (1/2 :: ereal) \wedge \text{Suc } n) = 1$
 $\langle proof \rangle$

lemma *suminf-add-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i. 0 \leq f\ i \wedge i. 0 \leq g\ i$
shows $(\sum i. f\ i + g\ i) = \text{suminf } f + \text{suminf } g$
 $\langle proof \rangle$

lemma *suminf-cmult-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i. 0 \leq f\ i$
and $0 \leq a$
shows $(\sum i. a * f\ i) = a * \text{suminf } f$
 $\langle proof \rangle$

lemma *suminf-PInf*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. 0 \leq f\ i$
and $\text{suminf } f \neq \infty$
shows $f\ i \neq \infty$
 $\langle \text{proof} \rangle$

lemma *suminf-PInfy-fun*:
assumes $\bigwedge i. 0 \leq f\ i$
and $\text{suminf } f \neq \infty$
shows $\exists f'. f = (\lambda x. \text{ereal } (f'\ x))$
 $\langle \text{proof} \rangle$

lemma *summable-ereal*:
assumes $\bigwedge i. 0 \leq f\ i$
and $(\sum i. \text{ereal } (f\ i)) \neq \infty$
shows *summable* f
 $\langle \text{proof} \rangle$

lemma *suminf-ereal*:
assumes $\bigwedge i. 0 \leq f\ i$
and $(\sum i. \text{ereal } (f\ i)) \neq \infty$
shows $(\sum i. \text{ereal } (f\ i)) = \text{ereal } (\text{suminf } f)$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal-minus*:
fixes $f\ g :: \text{nat} \Rightarrow \text{ereal}$
assumes *ord*: $\bigwedge i. g\ i \leq f\ i \wedge i. 0 \leq g\ i$
and *fin*: $\text{suminf } f \neq \infty \wedge \text{suminf } g \neq \infty$
shows $(\sum i. f\ i - g\ i) = \text{suminf } f - \text{suminf } g$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal-PInf [simp]*: $(\sum x. \infty :: \text{ereal}) = \infty$
 $\langle \text{proof} \rangle$

lemma *summable-real-of-ereal*:
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes *f*: $\bigwedge i. 0 \leq f\ i$
and *fin*: $(\sum i. f\ i) \neq \infty$
shows *summable* $(\lambda i. \text{real-of-ereal } (f\ i))$
 $\langle \text{proof} \rangle$

lemma *suminf-SUP-eq*:
fixes $f :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. \text{incseq } (\lambda n. f\ n\ i)$
and $\bigwedge n\ i. 0 \leq f\ n\ i$
shows $(\sum i. \text{SUP } n. f\ n\ i) = (\text{SUP } n. \sum i. f\ n\ i)$
 $\langle \text{proof} \rangle$

lemma *suminf-sum-ereal*:

fixes $f :: - \Rightarrow - \Rightarrow \text{ereal}$
assumes *nonneg*: $\bigwedge i \ a. \ a \in A \implies 0 \leq f \ i \ a$
shows $(\sum i. \sum_{a \in A}. f \ i \ a) = (\sum_{a \in A}. \sum i. f \ i \ a)$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal-eq-0*:
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes *nneg*: $\bigwedge i. \ 0 \leq f \ i$
shows $(\sum i. f \ i) = 0 \iff (\forall i. f \ i = 0)$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal-offset-le*:
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes *f*: $\bigwedge i. \ 0 \leq f \ i$
shows $(\sum i. f \ (i + k)) \leq \text{suminf } f$
 $\langle \text{proof} \rangle$

lemma *sums-suminf-ereal*: $f \text{ sums } x \implies (\sum i. \text{ereal } (f \ i)) = \text{ereal } x$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal'*: $\text{summable } f \implies (\sum i. \text{ereal } (f \ i)) = \text{ereal } (\sum i. f \ i)$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal-finite*: $\text{summable } f \implies (\sum i. \text{ereal } (f \ i)) \neq \infty$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal-finite-neg*:
assumes *summable* f
shows $(\sum x. \text{ereal } (f \ x)) \neq -\infty$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-add-directed*:
fixes $f \ g :: 'a \Rightarrow \text{ereal}$
assumes *nonneg*: $\bigwedge i. \ i \in I \implies 0 \leq f \ i \ \wedge \ i \in I \implies 0 \leq g \ i$
assumes *directed*: $\bigwedge i \ j. \ i \in I \implies j \in I \implies \exists k \in I. \ f \ i + g \ j \leq f \ k + g \ k$
shows $(\text{SUP } i \in I. f \ i + g \ i) = (\text{SUP } i \in I. f \ i) + (\text{SUP } i \in I. g \ i)$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-sum-directed*:
fixes $f \ g :: 'a \Rightarrow 'b \Rightarrow \text{ereal}$
assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N \ i \ j. \ N \subseteq A \implies i \in I \implies j \in I \implies \exists k \in I. \ \forall n \in N. \ f \ n \ i \leq f \ n \ k \ \wedge \ f \ n \ j \leq f \ n \ k$
assumes *nonneg*: $\bigwedge n \ i. \ i \in I \implies n \in A \implies 0 \leq f \ n \ i$
shows $(\text{SUP } i \in I. \sum_{n \in A}. f \ n \ i) = (\sum_{n \in A}. \text{SUP } i \in I. f \ n \ i)$
 $\langle \text{proof} \rangle$

lemma *suminf-SUP-eq-directed*:
fixes $f :: - \Rightarrow \text{nat} \Rightarrow \text{ereal}$

assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N \ i \ j. i \in I \implies j \in I \implies \text{finite } N \implies \exists k \in I. \forall n \in N. f \ i \ n \leq f \ k \ n \wedge f \ j \ n \leq f \ k \ n$
assumes *nonneg*: $\bigwedge n \ i. 0 \leq f \ n \ i$
shows $(\sum i. \text{SUP } n \in I. f \ n \ i) = (\text{SUP } n \in I. \sum i. f \ n \ i)$
 <proof>

lemma *ereal-dense3*:

fixes $x \ y :: \text{ereal}$
shows $x < y \implies \exists r :: \text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$
 <proof>

lemma *continuous-within-ereal*[*intro, simp*]: $x \in A \implies \text{continuous (at } x \text{ within } A)$
ereal

<proof>

lemma *ereal-open-uminus*:

fixes $S :: \text{ereal set}$
assumes *open* S
shows *open* $(\text{uminus } S)$
 <proof>

lemma *ereal-uminus-complement*:

fixes $S :: \text{ereal set}$
shows $\text{uminus } (- S) = - \text{uminus } S$
 <proof>

lemma *ereal-closed-uminus*:

fixes $S :: \text{ereal set}$
assumes *closed* S
shows *closed* $(\text{uminus } S)$
 <proof>

lemma *ereal-open-affinity-pos*:

fixes $S :: \text{ereal set}$
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 $S :: \text{ereal set}$
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 $S :: \text{ereal set}$

shows $\text{open } (\text{uminus } S) \longleftrightarrow \text{open } S$

<proof>

lemma *ereal-Liminf-uminus*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $\text{Liminf } \text{net } (\lambda x. - (f x)) = - \text{Limsup } \text{net } f$

<proof>

lemma *Liminf-PInfy*:

fixes $f :: 'a \Rightarrow \text{ereal}$

assumes $\neg \text{trivial-limit } \text{net}$

shows $(f \longrightarrow \infty) \text{ net} \longleftrightarrow \text{Liminf } \text{net } f = \infty$

<proof>

lemma *Limsup-MInfy*:

fixes $f :: 'a \Rightarrow \text{ereal}$

assumes $\neg \text{trivial-limit } \text{net}$

shows $(f \longrightarrow -\infty) \text{ net} \longleftrightarrow \text{Limsup } \text{net } f = -\infty$

<proof>

lemma *convergent-ereal*: — RENAME

fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

shows $\text{convergent } X \longleftrightarrow \text{limsup } X = \text{liminf } X$

<proof>

lemma *limsup-le-liminf-real*:

fixes $X :: \text{nat} \Rightarrow \text{real}$ **and** $L :: \text{real}$

assumes 1: $\text{limsup } X \leq L$ **and** 2: $L \leq \text{liminf } X$

shows $X \longrightarrow L$

<proof>

lemma *liminf-PInfy*:

fixes $X :: \text{nat} \Rightarrow \text{ereal}$

shows $X \longrightarrow \infty \longleftrightarrow \text{liminf } X = \infty$

<proof>

lemma *limsup-MInfy*:

fixes $X :: \text{nat} \Rightarrow \text{ereal}$

shows $X \longrightarrow -\infty \longleftrightarrow \text{limsup } X = -\infty$

<proof>

lemma *SUP-eq-LIMSEQ*:

assumes *mono* f

shows $(\text{SUP } n. \text{ereal } (f n)) = \text{ereal } x \longleftrightarrow f \longrightarrow x$

<proof>

lemma *liminf-ereal-cminus*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $c \neq -\infty$
shows $\liminf (\lambda x. c - f x) = c - \limsup f$
 $\langle \text{proof} \rangle$

38.6.3 Continuity

lemma *continuous-at-of-ereal*:
 $|x0 :: \text{ereal}| \neq \infty \implies \text{continuous } (\text{at } x0) \text{ real-of-ereal}$
 $\langle \text{proof} \rangle$

lemma *nhds-ereal*: $\text{nhds } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{nhds } r)$
 $\langle \text{proof} \rangle$

lemma *at-ereal*: $\text{at } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at } r)$
 $\langle \text{proof} \rangle$

lemma *at-left-ereal*: $\text{at-left } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at-left } r)$
 $\langle \text{proof} \rangle$

lemma *at-right-ereal*: $\text{at-right } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at-right } r)$
 $\langle \text{proof} \rangle$

lemma
shows *at-left-PInf*: $\text{at-left } \infty = \text{filtermap } \text{ereal } \text{at-top}$
and *at-right-MInf*: $\text{at-right } (-\infty) = \text{filtermap } \text{ereal } \text{at-bot}$
 $\langle \text{proof} \rangle$

lemma *ereal-tendsto-simps1*:
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-left } (\text{ereal } x)) \longleftrightarrow (f \longrightarrow y) (\text{at-left } x)$
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-right } (\text{ereal } x)) \longleftrightarrow (f \longrightarrow y) (\text{at-right } x)$
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-left } (\infty :: \text{ereal})) \longleftrightarrow (f \longrightarrow y) \text{at-top}$
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-right } (-\infty :: \text{ereal})) \longleftrightarrow (f \longrightarrow y) \text{at-bot}$
 $\langle \text{proof} \rangle$

lemma *ereal-tendsto-simps2*:
 $((\text{ereal} \circ f) \longrightarrow \text{ereal } a) F \longleftrightarrow (f \longrightarrow a) F$
 $((\text{ereal} \circ f) \longrightarrow \infty) F \longleftrightarrow (\text{LIM } x F. f x :> \text{at-top})$
 $((\text{ereal} \circ f) \longrightarrow -\infty) F \longleftrightarrow (\text{LIM } x F. f x :> \text{at-bot})$
 $\langle \text{proof} \rangle$

lemma *inverse-infty-ereal-tendsto-0*: $\text{inverse } -\infty \rightarrow (0 :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *inverse-ereal-tendsto-pos*:
fixes $x :: \text{ereal}$ **assumes** $0 < x$
shows $\text{inverse } -x \rightarrow \text{inverse } x$
 $\langle \text{proof} \rangle$

lemma *inverse-ereal-tendsto-at-right-0*: $(\text{inverse} \longrightarrow \infty) (\text{at-right } (0::\text{ereal}))$
 $\langle \text{proof} \rangle$

lemmas *ereal-tendsto-simps* = *ereal-tendsto-simps1* *ereal-tendsto-simps2*

lemma *continuous-at-iff-ereal*:
fixes $f :: 'a::t2\text{-space} \Rightarrow \text{real}$
shows $\text{continuous } (\text{at } x0 \text{ within } s) f \longleftrightarrow \text{continuous } (\text{at } x0 \text{ within } s) (\text{ereal} \circ f)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-iff-ereal*:
fixes $f :: 'a::t2\text{-space} \Rightarrow \text{real}$
assumes $\text{open } A$
shows $\text{continuous-on } A f \longleftrightarrow \text{continuous-on } A (\text{ereal} \circ f)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-real*: $\text{continuous-on } (\text{UNIV} - \{\infty, -\infty::\text{ereal}\}) \text{real-of-ereal}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-iff-real*:
fixes $f :: 'a::t2\text{-space} \Rightarrow \text{ereal}$
assumes $\bigwedge x. x \in A \implies |f x| \neq \infty$
shows $\text{continuous-on } A f \longleftrightarrow \text{continuous-on } A (\text{real-of-ereal} \circ f)$
 $\langle \text{proof} \rangle$

lemma *continuous-uminus-ereal* [*continuous-intros*]: $\text{continuous-on } (A :: \text{ereal set})$
 uminus
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-atMost* [*simp*]: $\text{uminus } ' \{..(a::\text{ereal})\} = \{-a.. \}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-inverse-ereal* [*continuous-intros*]:
 $\text{continuous-on } \{0::\text{ereal} ..\} \text{inverse}$
 $\langle \text{proof} \rangle$

lemma *continuous-inverse-ereal-nonpos*: $\text{continuous-on } (\{..<0\} :: \text{ereal set}) \text{inverse}$
 $\langle \text{proof} \rangle$

lemma *tendsto-inverse-ereal*:
assumes $(f \longrightarrow (c :: \text{ereal})) F$
assumes $\text{eventually } (\lambda x. f x \geq 0) F$
shows $((\lambda x. \text{inverse } (f x)) \longrightarrow \text{inverse } c) F$
 $\langle \text{proof} \rangle$

38.6.4 liminf and limsup

lemma *Limsup-ereal-mult-right*:

assumes $F \neq \text{bot } (c::\text{real}) \geq 0$
shows $\text{Limsup } F (\lambda n. f n * \text{ereal } c) = \text{Limsup } F f * \text{ereal } c$
 $\langle \text{proof} \rangle$

lemma *Liminf-ereal-mult-right:*
assumes $F \neq \text{bot } (c::\text{real}) \geq 0$
shows $\text{Liminf } F (\lambda n. f n * \text{ereal } c) = \text{Liminf } F f * \text{ereal } c$
 $\langle \text{proof} \rangle$

lemma *Liminf-ereal-mult-left:*
assumes $F \neq \text{bot } (c::\text{real}) \geq 0$
shows $\text{Liminf } F (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{Liminf } F f$
 $\langle \text{proof} \rangle$

lemma *Limsup-ereal-mult-left:*
assumes $F \neq \text{bot } (c::\text{real}) \geq 0$
shows $\text{Limsup } F (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{Limsup } F f$
 $\langle \text{proof} \rangle$

lemma *limsup-ereal-mult-right:*
 $(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. f n * \text{ereal } c) = \text{limsup } f * \text{ereal } c$
 $\langle \text{proof} \rangle$

lemma *limsup-ereal-mult-left:*
 $(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{limsup } f$
 $\langle \text{proof} \rangle$

lemma *Limsup-add-ereal-right:*
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. g n + (c :: \text{ereal})) = \text{Limsup } F g + c$
 $\langle \text{proof} \rangle$

lemma *Limsup-add-ereal-left:*
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Limsup } F g$
 $\langle \text{proof} \rangle$

lemma *Liminf-add-ereal-right:*
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. g n + (c :: \text{ereal})) = \text{Liminf } F g + c$
 $\langle \text{proof} \rangle$

lemma *Liminf-add-ereal-left:*
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Liminf } F g$
 $\langle \text{proof} \rangle$

lemma
assumes $F \neq \text{bot}$
assumes *nonneg: eventually* $(\lambda x. f x \geq (0::\text{ereal})) F$
shows *Liminf-inverse-ereal:* $\text{Liminf } F (\lambda x. \text{inverse } (f x)) = \text{inverse } (\text{Limsup } F)$

$f)$
and *Limsup-inverse-ereal*: $\text{Limsup } F (\lambda x. \text{inverse } (f x)) = \text{inverse } (\text{Liminf } F f)$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-le-mono-left*: $\llbracket x \leq z; 0 \leq y \rrbracket \implies x - y \leq (z :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *neg-0-less-iff-less-erea [simp]*: $0 < -a \longleftrightarrow (a :: \text{ereal}) < 0$
 $\langle \text{proof} \rangle$

lemma *not-infity-ereal*: $|x| \neq \infty \longleftrightarrow (\exists x'. x = \text{ereal } x')$
 $\langle \text{proof} \rangle$

lemma *neg-PInf-trans*: **fixes** $x y :: \text{ereal}$ **shows** $\llbracket y \neq \infty; x \leq y \rrbracket \implies x \neq \infty$
 $\langle \text{proof} \rangle$

lemma *mult-2-ereal*: $\text{ereal } 2 * x = x + x$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-le-self*: $0 \leq y \implies x - y \leq (x :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-le-add-self*: $0 \leq y \implies x \leq x + (y :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-le-add-self2*: $0 \leq y \implies x \leq y + (x :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-le-add-mono1*: $\llbracket x \leq y; 0 \leq (z :: \text{ereal}) \rrbracket \implies x \leq y + z$
 $\langle \text{proof} \rangle$

lemma *ereal-le-add-mono2*: $\llbracket x \leq z; 0 \leq (y :: \text{ereal}) \rrbracket \implies x \leq y + z$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-nonpos*:
fixes $a b :: \text{ereal}$ **shows** $\llbracket a \leq b; a = \infty \implies b \neq \infty; a = -\infty \implies b \neq -\infty \rrbracket$
 $\implies a - b \leq 0$
 $\langle \text{proof} \rangle$

lemma *minus-ereal-0 [simp]*: $x - \text{ereal } 0 = x$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-eq-0-iff*: **fixes** $a b :: \text{ereal}$
shows $(|a| = \infty \implies |b| \neq \infty) \implies a - b = 0 \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-eq-0-iff-nonneg*:
fixes $f :: - \Rightarrow \text{ereal}$ **and** A

assumes *nonneg*: $\forall x \in A. f\ x \geq 0$
and $A:A \neq \{\}$
shows $(\text{SUP } x \in A. f\ x) = 0 \longleftrightarrow (\forall x \in A. f\ x = 0)$ (**is** $?lhs \longleftrightarrow ?rhs$)
 $\langle \text{proof} \rangle$

lemma *ereal-divide-le-posI*:
fixes $x\ y\ z :: \text{ereal}$
shows $x > 0 \implies z \neq -\infty \implies z \leq x * y \implies z / x \leq y$
 $\langle \text{proof} \rangle$

lemma *add-diff-eq-ereal*: **fixes** $x\ y\ z :: \text{ereal}$
shows $x + (y - z) = x + y - z$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-gr0*:
fixes $a\ b :: \text{ereal}$ **shows** $a < b \implies 0 < b - a$
 $\langle \text{proof} \rangle$

lemma *ereal-minus-minus*: **fixes** $x\ y\ z :: \text{ereal}$ **shows**
 $(|y| = \infty \implies |z| \neq \infty) \implies x - (y - z) = x + z - y$
 $\langle \text{proof} \rangle$

lemma *diff-add-eq-ereal*: **fixes** $a\ b\ c :: \text{ereal}$ **shows** $a - b + c = a + c - b$
 $\langle \text{proof} \rangle$

lemma *diff-diff-commute-ereal*: **fixes** $x\ y\ z :: \text{ereal}$ **shows** $x - y - z = x - z - y$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-eq-MInfty-iff*: **fixes** $x\ y :: \text{ereal}$ **shows** $x - y = -\infty \longleftrightarrow x = -\infty \wedge y \neq -\infty \vee y = \infty \wedge |x| \neq \infty$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-add-inverse*: **fixes** $x\ y :: \text{ereal}$ **shows** $|x| \neq \infty \implies x + y - x = y$
 $\langle \text{proof} \rangle$

lemma *tendsto-diff-ereal*:
fixes $x\ y :: \text{ereal}$
assumes $x: |x| \neq \infty$ **and** $y: |y| \neq \infty$
assumes $f: (f \longrightarrow x)\ F$ **and** $g: (g \longrightarrow y)\ F$
shows $((\lambda x. f\ x - g\ x) \longrightarrow x - y)\ F$
 $\langle \text{proof} \rangle$

lemma *continuous-on-diff-ereal*:
 $\text{continuous-on } A\ f \implies \text{continuous-on } A\ g \implies (\bigwedge x. x \in A \implies |f\ x| \neq \infty) \implies$
 $(\bigwedge x. x \in A \implies |g\ x| \neq \infty) \implies \text{continuous-on } A\ (\lambda z. f\ z - g\ z :: \text{ereal})$
 $\langle \text{proof} \rangle$

38.6.5 Tests for code generator

A small list of simple arithmetic expressions.

```

value  $-\infty :: \text{ereal}$ 
value  $|\infty| :: \text{ereal}$ 
value  $4 + 5 / 4 - \text{ereal } 2 :: \text{ereal}$ 
value  $\text{ereal } 3 < \infty$ 
value  $\text{real-of-ereal } (\infty :: \text{ereal}) = 0$ 

```

end

39 Indicator Function

```

theory Indicator-Function
imports Complex-Main Disjoint-Sets
begin

```

definition $\text{indicator } S \ x = \text{of-bool } (x \in S)$

Type constrained version

abbreviation $\text{indicat-real} :: 'a \text{ set} \Rightarrow 'a \Rightarrow \text{real}$ **where** $\text{indicat-real } S \equiv \text{indicator } S$

lemma $\text{indicator-simps}[\text{simp}]$:
 $x \in S \Longrightarrow \text{indicator } S \ x = 1$
 $x \notin S \Longrightarrow \text{indicator } S \ x = 0$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-pos-le}[\text{intro}, \text{simp}]$: $(0 :: 'a :: \text{linordered-semidom}) \leq \text{indicator } S \ x$
and $\text{indicator-le-1}[\text{intro}, \text{simp}]$: $\text{indicator } S \ x \leq (1 :: 'a :: \text{linordered-semidom})$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-abs-le-1}$: $|\text{indicator } S \ x| \leq (1 :: 'a :: \text{linordered-idom})$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-eq-0-iff}$: $\text{indicator } A \ x = (0 :: 'a :: \text{zero-neq-one}) \longleftrightarrow x \notin A$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-eq-1-iff}$: $\text{indicator } A \ x = (1 :: 'a :: \text{zero-neq-one}) \longleftrightarrow x \in A$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-UNIV} [\text{simp}]$: $\text{indicator } \text{UNIV} = (\lambda x. 1)$
 $\langle \text{proof} \rangle$

lemma indicator-leI :
 $(x \in A \Longrightarrow y \in B) \Longrightarrow (\text{indicator } A \ x :: 'a :: \text{linordered-nonzero-semiring}) \leq \text{indicator } B \ y$
 $\langle \text{proof} \rangle$

lemma *split-indicator*: $P \text{ (indicator } S \text{ } x) \longleftrightarrow ((x \in S \longrightarrow P \text{ } 1) \wedge (x \notin S \longrightarrow P \text{ } 0))$

<proof>

lemma *split-indicator-asm*: $P \text{ (indicator } S \text{ } x) \longleftrightarrow (\neg (x \in S \wedge \neg P \text{ } 1 \vee x \notin S \wedge \neg P \text{ } 0))$

<proof>

lemma *indicator-inter-arith*: $\text{indicator } (A \cap B) \text{ } x = \text{indicator } A \text{ } x * (\text{indicator } B \text{ } x :: 'a::\text{semiring-1})$

<proof>

lemma *indicator-union-arith*:

$\text{indicator } (A \cup B) \text{ } x = \text{indicator } A \text{ } x + \text{indicator } B \text{ } x - \text{indicator } A \text{ } x * (\text{indicator } B \text{ } x :: 'a::\text{ring-1})$

<proof>

lemma *indicator-inter-min*: $\text{indicator } (A \cap B) \text{ } x = \min (\text{indicator } A \text{ } x) (\text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

and *indicator-union-max*: $\text{indicator } (A \cup B) \text{ } x = \max (\text{indicator } A \text{ } x) (\text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

<proof>

lemma *indicator-disj-union*:

$A \cap B = \{\} \implies \text{indicator } (A \cup B) \text{ } x = (\text{indicator } A \text{ } x + \text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

<proof>

lemma *indicator-compl*: $\text{indicator } (- A) \text{ } x = 1 - (\text{indicator } A \text{ } x :: 'a::\text{ring-1})$

and *indicator-diff*: $\text{indicator } (A - B) \text{ } x = \text{indicator } A \text{ } x * (1 - \text{indicator } B \text{ } x :: 'a::\text{ring-1})$

<proof>

lemma *indicator-times*:

$\text{indicator } (A \times B) \text{ } x = \text{indicator } A \text{ } (\text{fst } x) * (\text{indicator } B \text{ } (\text{snd } x) :: 'a::\text{semiring-1})$

<proof>

lemma *indicator-sum*:

$\text{indicator } (A <+> B) \text{ } x = (\text{case } x \text{ of } \text{Inl } x \Rightarrow \text{indicator } A \text{ } x \mid \text{Inr } x \Rightarrow \text{indicator } B \text{ } x)$

<proof>

lemma *indicator-image*: $\text{inj } f \implies \text{indicator } (f ' X) \text{ } (f \text{ } x) = (\text{indicator } X \text{ } x :: \text{zero-neq-one})$

<proof>

lemma *indicator-vimage*: $\text{indicator } (f - ' A) \text{ } x = \text{indicator } A \text{ } (f \text{ } x)$

<proof>

lemma *mult-indicator-cong*:

fixes $f\ g :: - \Rightarrow 'a :: \text{semiring-1}$
shows $(\bigwedge x. x \in A \implies f\ x = g\ x) \implies \text{indicator } A\ x * f\ x = \text{indicator } A\ x * g\ x$
 $\langle \text{proof} \rangle$

lemma

fixes $f :: 'a \Rightarrow 'b :: \text{semiring-1}$
assumes $\text{finite } A$
shows $\text{sum-mult-indicator}[simp]: (\sum x \in A. f\ x * \text{indicator } B\ x) = (\sum x \in A \cap B. f\ x)$
and $\text{sum-indicator-mult}[simp]: (\sum x \in A. \text{indicator } B\ x * f\ x) = (\sum x \in A \cap B. f\ x)$
 $\langle \text{proof} \rangle$

lemma $\text{sum-indicator-eq-card}:$

assumes $\text{finite } A$
shows $(\sum x \in A. \text{indicator } B\ x) = \text{card } (A \text{ Int } B)$
 $\langle \text{proof} \rangle$

lemma $\text{sum-indicator-scaleR}[simp]:$

$\text{finite } A \implies$
 $(\sum x \in A. \text{indicator } (B\ x)\ (g\ x) *_{\mathbb{R}} f\ x) = (\sum x \in \{x \in A. g\ x \in B\ x\}. f\ x :: 'a :: \text{real-vector})$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-incseq}:$

assumes $\text{incseq } A$
shows $(\lambda i. \text{indicator } (A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcup i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-UN}:$

$(\lambda k. \text{indicator } (\bigcup i < k. A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcup i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-decseq}:$

assumes $\text{decseq } A$
shows $(\lambda i. \text{indicator } (A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcap i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-INT}:$

$(\lambda k. \text{indicator } (\bigcap i < k. A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcap i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-add}:$

$A \cap B = \{\} \implies (\text{indicator } A\ x :: \text{monoid-add}) + \text{indicator } B\ x = \text{indicator } (A \cup B)\ x$

$\langle \text{proof} \rangle$

lemma *of-real-indicator*: $\text{of-real } (\text{indicator } A \ x) = \text{indicator } A \ x$
 $\langle \text{proof} \rangle$

lemma *real-of-nat-indicator*: $\text{real } (\text{indicator } A \ x :: \text{nat}) = \text{indicator } A \ x$
 $\langle \text{proof} \rangle$

lemma *abs-indicator*: $|\text{indicator } A \ x :: 'a::\text{linordered-idom}| = \text{indicator } A \ x$
 $\langle \text{proof} \rangle$

lemma *mult-indicator-subset*:
 $A \subseteq B \implies \text{indicator } A \ x * \text{indicator } B \ x = (\text{indicator } A \ x :: 'a::\text{comm-semiring-1})$
 $\langle \text{proof} \rangle$

lemma *indicator-times-eq-if*:
fixes $f :: 'a \Rightarrow 'b::\text{comm-ring-1}$
shows $\text{indicator } S \ x * f \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0) \ f \ x * \text{indicator } S \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *indicator-scaleR-eq-if*:
fixes $f :: 'a \Rightarrow 'b::\text{real-vector}$
shows $\text{indicator } S \ x *_R f \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *indicator-sums*:
assumes $\bigwedge i \ j. i \neq j \implies A \ i \cap A \ j = \{\}$
shows $(\lambda i. \text{indicator } (A \ i) \ x::\text{real}) \text{ sums } \text{indicator } (\bigcup i. A \ i) \ x$
 $\langle \text{proof} \rangle$

The indicator function of the union of a disjoint family of sets is the sum over all the individual indicators.

lemma *indicator-UN-disjoint*:
 $\text{finite } A \implies \text{disjoint-family-on } f \ A \implies \text{indicator } (\bigcup (f \ ` \ A)) \ x = (\sum y \in A. \text{indicator } (f \ y) \ x)$
 $\langle \text{proof} \rangle$

end

39.1 The type of non-negative extended real numbers

theory *Extended-Nonnegative-Real*
imports *Extended-Real Indicator-Function*
begin

lemma *ereal-ineq-diff-add*:
assumes $b \neq (-\infty::\text{ereal}) \ a \geq b$
shows $a = b + (a - b)$

<proof>

lemma *Limsup-const-add*:

fixes $c :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}, \text{topological-monoid-add}, \text{ordered-ab-semigroup-add}\}$

shows $F \neq \text{bot} \implies \text{Limsup } F (\lambda x. c + f x) = c + \text{Limsup } F f$

<proof>

lemma *Liminf-const-add*:

fixes $c :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}, \text{topological-monoid-add}, \text{ordered-ab-semigroup-add}\}$

shows $F \neq \text{bot} \implies \text{Liminf } F (\lambda x. c + f x) = c + \text{Liminf } F f$

<proof>

lemma *Liminf-add-const*:

fixes $c :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}, \text{topological-monoid-add}, \text{ordered-ab-semigroup-add}\}$

shows $F \neq \text{bot} \implies \text{Liminf } F (\lambda x. f x + c) = \text{Liminf } F f + c$

<proof>

lemma *sums-offset*:

fixes $f g :: \text{nat} \Rightarrow 'a :: \{\text{t2-space}, \text{topological-comm-monoid-add}\}$

assumes $(\lambda n. f (n + i)) \text{ sums } l$ **shows** $f \text{ sums } (l + (\sum j < i. f j))$

<proof>

lemma *suminf-offset*:

fixes $f g :: \text{nat} \Rightarrow 'a :: \{\text{t2-space}, \text{topological-comm-monoid-add}\}$

shows $\text{summable } (\lambda j. f (j + i)) \implies \text{suminf } f = (\sum j. f (j + i)) + (\sum j < i. f j)$

<proof>

lemma *eventually-at-left-1*: $(\bigwedge z :: \text{real}. 0 < z \implies z < 1 \implies P z) \implies \text{eventually } P (\text{at-left } 1)$

<proof>

lemma *mult-eq-1*:

fixes $a b :: 'a :: \{\text{ordered-semiring}, \text{comm-monoid-mult}\}$

shows $0 \leq a \implies a \leq 1 \implies b \leq 1 \implies a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$

<proof>

lemma *ereal-add-diff-cancel*:

fixes $a b :: \text{ereal}$

shows $|b| \neq \infty \implies (a + b) - b = a$

<proof>

lemma *add-top*:

fixes $x :: 'a :: \{\text{order-top}, \text{ordered-comm-monoid-add}\}$

shows $0 \leq x \implies x + \text{top} = \text{top}$

<proof>

lemma *top-add*:

fixes $x :: 'a::\{\text{order-top, ordered-comm-monoid-add}\}$

shows $0 \leq x \implies \text{top} + x = \text{top}$

$\langle \text{proof} \rangle$

lemma *le-lfp*: $\text{mono } f \implies x \leq \text{lfp } f \implies f x \leq \text{lfp } f$

$\langle \text{proof} \rangle$

lemma *lfp-transfer*:

assumes α : *sup-continuous* α **and** f : *sup-continuous* f **and** mg : *mono* g

assumes bot : $\alpha \text{ bot} \leq \text{lfp } g$ **and** eq : $\bigwedge x. x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)$

shows $\alpha (\text{lfp } f) = \text{lfp } g$

$\langle \text{proof} \rangle$

lemma *sup-continuous-applyD*: *sup-continuous* $f \implies \text{sup-continuous } (\lambda x. f x h)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-SUP*[*order-continuous-intros*]:

fixes $M :: - \Rightarrow - \Rightarrow 'a::\text{complete-lattice}$

assumes M : $\bigwedge i. i \in I \implies \text{sup-continuous } (M i)$

shows *sup-continuous* $(\text{SUP } i \in I. M i)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-apply-SUP*[*order-continuous-intros*]:

fixes $M :: - \Rightarrow - \Rightarrow 'a::\text{complete-lattice}$

shows $(\bigwedge i. i \in I \implies \text{sup-continuous } (M i)) \implies \text{sup-continuous } (\lambda x. \text{SUP } i \in I. M i x)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-lfp'*[*order-continuous-intros*]:

assumes 1 : *sup-continuous* f

assumes 2 : $\bigwedge g. \text{sup-continuous } g \implies \text{sup-continuous } (f g)$

shows *sup-continuous* $(\text{lfp } f)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-lfp''*[*order-continuous-intros*]:

assumes 1 : $\bigwedge s. \text{sup-continuous } (f s)$

assumes 2 : $\bigwedge g. \text{sup-continuous } g \implies \text{sup-continuous } (\lambda s. f s (g s))$

shows *sup-continuous* $(\lambda x. \text{lfp } (f x))$

$\langle \text{proof} \rangle$

lemma *mono-INF-fun*:

$(\bigwedge x y. \text{mono } (F x y)) \implies \text{mono } (\lambda z x. \text{INF } y \in X x. F x y z :: 'a :: \text{complete-lattice})$

$\langle \text{proof} \rangle$

lemma *continuous-on-cmult-ereal*:

$|c::\text{ereal}| \neq \infty \implies \text{continuous-on } A f \implies \text{continuous-on } A (\lambda x. c * f x)$

$\langle \text{proof} \rangle$

lemma *real-of-nat-Sup*:
 assumes $A \neq \{\}$ *bdd-above* A
 shows $\text{of-nat } (\text{Sup } A) = (\text{SUP } a \in A. \text{ of-nat } a :: \text{real})$
 $\langle \text{proof} \rangle$

lemma (*in complete-lattice*) *SUP-sup-const1*:
 $I \neq \{\} \implies (\text{SUP } i \in I. \text{ sup } c (f i)) = \text{sup } c (\text{SUP } i \in I. f i)$
 $\langle \text{proof} \rangle$

lemma (*in complete-lattice*) *SUP-sup-const2*:
 $I \neq \{\} \implies (\text{SUP } i \in I. \text{ sup } (f i) c) = \text{sup } (\text{SUP } i \in I. f i) c$
 $\langle \text{proof} \rangle$

lemma *one-less-of-natD*:
 assumes $(1 :: 'a :: \text{linordered-semidom}) < \text{of-nat } n$ **shows** $1 < n$
 $\langle \text{proof} \rangle$

39.2 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = $\{x :: \text{ereal}. 0 \leq x\}$
morphisms *enn2ereal* *e2ennreal'*
 $\langle \text{proof} \rangle$

definition *e2ennreal* $x = e2ennreal' (\max 0 x)$

lemma *enn2ereal-range*: *e2ennreal* ‘ $\{0..\}$ ’ = *UNIV*
 $\langle \text{proof} \rangle$

lemma *type-definition-ennreal'*: *type-definition* *enn2ereal* *e2ennreal* $\{x. 0 \leq x\}$
 $\langle \text{proof} \rangle$

setup-lifting *type-definition-ennreal'*

declare $[[\text{coercion } e2ennreal]]$

instantiation *ennreal* :: *complete-linorder*
begin

lift-definition *top-ennreal* :: *ennreal* **is** *top* $\langle \text{proof} \rangle$
lift-definition *bot-ennreal* :: *ennreal* **is** *0* $\langle \text{proof} \rangle$
lift-definition *sup-ennreal* :: *ennreal* $\Rightarrow$ *ennreal* $\Rightarrow$ *ennreal* **is** *sup* $\langle \text{proof} \rangle$
lift-definition *inf-ennreal* :: *ennreal* $\Rightarrow$ *ennreal* $\Rightarrow$ *ennreal* **is** *inf* $\langle \text{proof} \rangle$

lift-definition *Inf-ennreal* :: *ennreal* *set* $\Rightarrow$ *ennreal* **is** *Inf*
 $\langle \text{proof} \rangle$

lift-definition *Sup-ennreal* :: *ennreal* *set* $\Rightarrow$ *ennreal* **is** *sup 0* $\circ$ *Sup*


```

    <proof>

lift-definition less-eq-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  bool is ( $\leq$ ) <proof>
lift-definition less-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  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  $\longleftrightarrow$  f = enn2ereal  $\circ$  g
  <proof>

instantiation ennreal :: infinity
begin

definition infinity-ennreal :: ennreal
where
  [simp]:  $\infty = (top::ennreal)$ 

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  $\Rightarrow$  ennreal  $\Rightarrow$  ennreal is (+) <proof>
lift-definition times-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  ennreal is (*) <proof>

instance
  <proof>

end

instantiation ennreal :: minus
begin

lift-definition minus-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  ennreal is  $\lambda a\ b. \max\ 0\ (a - b)$ 
  <proof>

instance <proof>

```

end

instance *ennreal* :: *numeral* $\langle \text{proof} \rangle$

instantiation *ennreal* :: *inverse*

begin

lift-definition *inverse-ennreal* :: *ennreal* $\Rightarrow$ *ennreal* **is** *inverse*
 $\langle \text{proof} \rangle$

definition *divide-ennreal* :: *ennreal* $\Rightarrow$ *ennreal* $\Rightarrow$ *ennreal*
where $x \text{ div } y = x * \text{inverse } (y :: \text{ennreal})$

instance $\langle \text{proof} \rangle$

end

lemma *ennreal-zero-less-one*: $0 < (1 :: \text{ennreal})$ — **TODO**: remove
 $\langle \text{proof} \rangle$

instance *ennreal* :: *diod*
 $\langle \text{proof} \rangle$

instance *ennreal* :: *ordered-comm-semiring*
 $\langle \text{proof} \rangle$

instance *ennreal* :: *linordered-nonzero-semiring*
 $\langle \text{proof} \rangle$

instance *ennreal* :: *strict-ordered-ab-semigroup-add*
 $\langle \text{proof} \rangle$

declare $[[\text{coercion of-nat} :: \text{nat} \Rightarrow \text{ennreal}]]$

lemma *e2ennreal-neg*: $x \leq 0 \Longrightarrow e2ennreal\ x = 0$
 $\langle \text{proof} \rangle$

lemma *e2ennreal-mono*: $x \leq y \Longrightarrow e2ennreal\ x \leq e2ennreal\ y$
 $\langle \text{proof} \rangle$

lemma *enn2ereal-nonneg[simp]*: $0 \leq \text{enn2ereal } x$
 $\langle \text{proof} \rangle$

lemma *ereal-ennreal-cases*:
obtains b **where** $0 \leq a$ $a = \text{enn2ereal } b$ $|$ $a < 0$
 $\langle \text{proof} \rangle$

lemma *rel-fun-liminf[transfer-rule]*: *rel-fun* (*rel-fun* $(=)$ *pcr-ennreal*) *pcr-ennreal*
liminf *liminf*

$\langle \text{proof} \rangle$

lemma *rel-fun-limsup[transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal*
limsup *limsup*
 $\langle \text{proof} \rangle$

lemma *sum-enn2ereal[simp]*: $(\bigwedge i. i \in I \implies 0 \leq f i) \implies (\sum_{i \in I}. \text{enn2ereal } (f i))$
 $= \text{enn2ereal } (\text{sum } f I)$
 $\langle \text{proof} \rangle$

lemma *transfer-e2ennreal-sum [transfer-rule]*:
rel-fun (*rel-fun* (=) *pcr-ennreal*) (*rel-fun* (=) *pcr-ennreal*) *sum* *sum*
 $\langle \text{proof} \rangle$

lemma *enn2ereal-of-nat[simp]*: *enn2ereal* (*of-nat* *n*) = *ereal* *n*
 $\langle \text{proof} \rangle$

lemma *enn2ereal-numeral[simp]*: *enn2ereal* (*numeral* *a*) = *numeral* *a*
 $\langle \text{proof} \rangle$

lemma *transfer-numeral[transfer-rule]*: *pcr-ennreal* (*numeral* *a*) (*numeral* *a*)
 $\langle \text{proof} \rangle$

39.3 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty :: \text{ennreal}) \vee b = c$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty :: \text{ennreal}) \vee b \leq c$
 $\langle \text{proof} \rangle$

lemma *ereal-add-left-cancel-less*:
fixes *a b c* :: *ereal*
shows $0 \leq a \implies 0 \leq b \implies a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty :: \text{ennreal}) \wedge b < c$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

39.4 Order with top

lemma *ennreal-zero-less-top[simp]*: $0 < (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-one-less-top[simp]*: $1 < (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-zero-neq-top[simp]*: $0 \neq (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-top-neq-zero[simp]*: $(\text{top}::\text{ennreal}) \neq 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-top-neq-one[simp]*: $\text{top} \neq (1::\text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-one-neq-top[simp]*: $1 \neq (\text{top}::\text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-less-top[simp]*:
fixes $a\ b :: \text{ennreal}$
shows $a + b < \text{top} \longleftrightarrow a < \text{top} \wedge b < \text{top}$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-eq-top[simp]*:
fixes $a\ b :: \text{ennreal}$
shows $a + b = \text{top} \longleftrightarrow a = \text{top} \vee b = \text{top}$
 $\langle \text{proof} \rangle$

lemma *ennreal-sum-less-top[simp]*:
fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $\text{finite } I \implies (\sum i \in I. f\ i) < \text{top} \longleftrightarrow (\forall i \in I. f\ i < \text{top})$
 $\langle \text{proof} \rangle$

lemma *ennreal-sum-eq-top[simp]*:
fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $\text{finite } I \implies (\sum i \in I. f\ i) = \text{top} \longleftrightarrow (\exists i \in I. f\ i = \text{top})$
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-eq-top-iff*:
fixes $a\ b :: \text{ennreal}$
shows $a * b = \text{top} \longleftrightarrow (a = \text{top} \wedge b \neq 0) \vee (b = \text{top} \wedge a \neq 0)$
 $\langle \text{proof} \rangle$

lemma *ennreal-top-eq-mult-iff*:
fixes $a\ b :: \text{ennreal}$
shows $\text{top} = a * b \longleftrightarrow (a = \text{top} \wedge b \neq 0) \vee (b = \text{top} \wedge a \neq 0)$
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-less-top*:
fixes $a\ b :: \text{ennreal}$
shows $a * b < \text{top} \longleftrightarrow (a = 0 \vee b = 0 \vee (a < \text{top} \wedge b < \text{top}))$
 $\langle \text{proof} \rangle$

lemma *top-power-ennreal*: $\text{top} \wedge^n = (\text{if } n = 0 \text{ then } 1 \text{ else } \text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-prod-eq-0*[simp]:

fixes $f :: 'a \Rightarrow \text{ennreal}$

shows $(\text{prod } f \ A = 0) = (\text{finite } A \wedge (\exists i \in A. f \ i = 0))$

$\langle \text{proof} \rangle$

lemma *ennreal-prod-eq-top*:

fixes $f :: 'a \Rightarrow \text{ennreal}$

shows $(\prod i \in I. f \ i) = \text{top} \longleftrightarrow (\text{finite } I \wedge ((\forall i \in I. f \ i \neq 0) \wedge (\exists i \in I. f \ i = \text{top})))$

$\langle \text{proof} \rangle$

lemma *ennreal-top-mult*: $\text{top} * a = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-mult-top*: $a * \text{top} = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *enn2ereal-eq-top-iff*[simp]: $\text{enn2ereal } x = \infty \longleftrightarrow x = \text{top}$

$\langle \text{proof} \rangle$

lemma *enn2ereal-top*[simp]: $\text{enn2ereal } \text{top} = \infty$

$\langle \text{proof} \rangle$

lemma *e2ennreal-infty*[simp]: $e2ennreal \ \infty = \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-top-minus*[simp]: $\text{top} - x = (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *minus-top-ennreal*: $x - \text{top} = (\text{if } x = \text{top} \text{ then } \text{top} \text{ else } 0 :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *bot-ennreal*: $\text{bot} = (0 :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-of-nat-neq-top*[simp]: $\text{of-nat } i \neq (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *numeral-eq-of-nat*: $(\text{numeral } a :: \text{ennreal}) = \text{of-nat } (\text{numeral } a)$

$\langle \text{proof} \rangle$

lemma *of-nat-less-top*: $\text{of-nat } i < (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *top-neq-numeral*[simp]: $\text{top} \neq (\text{numeral } i :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-numeral-less-top*[simp]: $\text{numeral } i < (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-add-bot[simp]*: $\text{bot} + x = (x :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *add-top-right-ennreal [simp]*: $x + \text{top} = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *add-top-left-ennreal [simp]*: $\text{top} + x = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-top-mult-left [simp]*: $x \neq 0 \implies x * \text{top} = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-top-mult-right [simp]*: $x \neq 0 \implies \text{top} * x = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *power-top-ennreal [simp]*: $n > 0 \implies \text{top} ^ n = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *power-eq-top-ennreal-iff*: $x ^ n = \text{top} \longleftrightarrow x = (\text{top} :: \text{ennreal}) \wedge n > 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-le-mult-iff*: $c \neq 0 \implies c \neq \text{top} \implies c * a \leq c * b \longleftrightarrow a \leq (b :: \text{ennreal})$
including *ennreal.lifting*
 $\langle \text{proof} \rangle$

lemma *power-mono-ennreal*: $x \leq y \implies x ^ n \leq (y ^ n :: \text{ennreal})$
 $\langle \text{proof} \rangle$

instance *ennreal :: semiring-char-0*
 $\langle \text{proof} \rangle$

39.5 Arithmetic

lemma *ennreal-minus-zero[simp]*: $a - (0 :: \text{ennreal}) = a$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-diff-cancel-right[simp]*:
fixes $x \ y \ z :: \text{ennreal}$ **shows** $y \neq \text{top} \implies (x + y) - y = x$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-diff-cancel-left[simp]*:
fixes $x \ y \ z :: \text{ennreal}$ **shows** $y \neq \text{top} \implies (y + x) - y = x$
 $\langle \text{proof} \rangle$

lemma
fixes $a \ b :: \text{ennreal}$
shows $a - b = 0 \implies a \leq b$

$\langle proof \rangle$

lemma *ennreal-minus-cancel*:

fixes $a\ b\ c :: ennreal$

shows $c \neq top \implies a \leq c \implies b \leq c \implies c - a = c - b \implies a = b$

$\langle proof \rangle$

lemma *sup-const-add-ennreal*:

fixes $a\ b\ c :: ennreal$

shows $sup\ (c + a)\ (c + b) = c + sup\ a\ b$

$\langle proof \rangle$

lemma *ennreal-diff-add-assoc*:

fixes $a\ b\ c :: ennreal$

shows $a \leq b \implies c + b - a = c + (b - a)$

$\langle proof \rangle$

lemma *mult-divide-eq-ennreal*:

fixes $a\ b :: ennreal$

shows $b \neq 0 \implies b \neq top \implies (a * b) / b = a$

$\langle proof \rangle$

lemma *divide-mult-eq*: $a \neq 0 \implies a \neq \infty \implies x * a / (b * a) = x / (b :: ennreal)$

$\langle proof \rangle$

lemma *ennreal-mult-divide-eq*:

fixes $a\ b :: ennreal$

shows $b \neq 0 \implies b \neq top \implies (a * b) / b = a$

$\langle proof \rangle$

lemma *ennreal-add-diff-cancel*:

fixes $a\ b :: ennreal$

shows $b \neq \infty \implies (a + b) - b = a$

$\langle proof \rangle$

lemma *ennreal-minus-eq-0*:

$a - b = 0 \implies a \leq (b :: ennreal)$

$\langle proof \rangle$

lemma *ennreal-mono-minus-cancel*:

fixes $a\ b\ c :: ennreal$

shows $a - b \leq a - c \implies a < top \implies b \leq a \implies c \leq a \implies c \leq b$

$\langle proof \rangle$

lemma *ennreal-mono-minus*:

fixes $a\ b\ c :: ennreal$

shows $c \leq b \implies a - b \leq a - c$

$\langle proof \rangle$

lemma *ennreal-minus-pos-iff*:

fixes $a\ b :: \text{ennreal}$

shows $a < \text{top} \vee b < \text{top} \implies 0 < a - b \implies b < a$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-top[simp]*: $\text{inverse } \text{top} = (0 :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-zero[simp]*: $\text{inverse } 0 = (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-top-divide*: $\text{top} / (x :: \text{ennreal}) = (\text{if } x = \text{top} \text{ then } 0 \text{ else } \text{top})$

$\langle \text{proof} \rangle$

lemma *ennreal-zero-divide[simp]*: $0 / (x :: \text{ennreal}) = 0$

$\langle \text{proof} \rangle$

lemma *ennreal-divide-zero[simp]*: $x / (0 :: \text{ennreal}) = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{top})$

$\langle \text{proof} \rangle$

lemma *ennreal-divide-top[simp]*: $x / (\text{top} :: \text{ennreal}) = 0$

$\langle \text{proof} \rangle$

lemma *ennreal-times-divide*: $a * (b / c) = a * b / (c :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-zero-less-divide*: $0 < a / b \longleftrightarrow (0 < a \wedge b < (\text{top} :: \text{ennreal}))$

$\langle \text{proof} \rangle$

lemma *add-divide-distrib-ennreal*: $(a + b) / c = a / c + b / (c :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *divide-right-mono-ennreal*:

fixes $a\ b\ c :: \text{ennreal}$

shows $a \leq b \implies a / c \leq b / c$

$\langle \text{proof} \rangle$

lemma *ennreal-mult-strict-right-mono*: $(a :: \text{ennreal}) < c \implies 0 < b \implies b < \text{top}$

$\implies a * b < c * b$

$\langle \text{proof} \rangle$

lemma *ennreal-indicator-less[simp]*:

$\text{indicator } A\ x \leq (\text{indicator } B\ x :: \text{ennreal}) \longleftrightarrow (x \in A \longrightarrow x \in B)$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-positive*: $0 < \text{inverse } x \longleftrightarrow (x :: \text{ennreal}) \neq \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-mult'*: $((0 < b \vee a < \text{top}) \wedge (0 < a \vee b < \text{top})) \implies$

inverse ($a * b :: \text{ennreal}$) = *inverse* $a * \text{inverse } b$
 ⟨proof⟩

lemma *ennreal-inverse-mult*: $a < \text{top} \implies b < \text{top} \implies \text{inverse } (a * b :: \text{ennreal}) = \text{inverse } a * \text{inverse } b$
 ⟨proof⟩

lemma *ennreal-inverse-1[simp]*: $\text{inverse } (1 :: \text{ennreal}) = 1$
 ⟨proof⟩

lemma *ennreal-inverse-eq-0-iff[simp]*: $\text{inverse } (a :: \text{ennreal}) = 0 \longleftrightarrow a = \text{top}$
 ⟨proof⟩

lemma *ennreal-inverse-eq-top-iff[simp]*: $\text{inverse } (a :: \text{ennreal}) = \text{top} \longleftrightarrow a = 0$
 ⟨proof⟩

lemma *ennreal-divide-eq-0-iff[simp]*: $(a :: \text{ennreal}) / b = 0 \longleftrightarrow (a = 0 \vee b = \text{top})$
 ⟨proof⟩

lemma *ennreal-divide-eq-top-iff*: $(a :: \text{ennreal}) / b = \text{top} \longleftrightarrow ((a \neq 0 \wedge b = 0) \vee (a = \text{top} \wedge b \neq \text{top}))$
 ⟨proof⟩

lemma *one-divide-one-divide-ennreal[simp]*: $1 / (1 / c) = (c :: \text{ennreal})$
including *ennreal.lifting*
 ⟨proof⟩

lemma *ennreal-mult-left-cong*:
 $((a :: \text{ennreal}) \neq 0 \implies b = c) \implies a * b = a * c$
 ⟨proof⟩

lemma *ennreal-mult-right-cong*:
 $((a :: \text{ennreal}) \neq 0 \implies b = c) \implies b * a = c * a$
 ⟨proof⟩

lemma *ennreal-zero-less-mult-iff*: $0 < a * b \longleftrightarrow 0 < a \wedge 0 < (b :: \text{ennreal})$
 ⟨proof⟩

lemma *less-diff-eq-ennreal*:
fixes $a \ b \ c :: \text{ennreal}$
shows $b < \text{top} \vee c < \text{top} \implies a < b - c \longleftrightarrow a + c < b$
 ⟨proof⟩

lemma *diff-add-cancel-ennreal*:
fixes $a \ b :: \text{ennreal}$ **shows** $a \leq b \implies b - a + a = b$
 ⟨proof⟩

lemma *ennreal-diff-self[simp]*: $a \neq \text{top} \implies a - a = (0 :: \text{ennreal})$
 ⟨proof⟩

lemma *ennreal-minus-mono*:

fixes $a\ b\ c :: \text{ennreal}$

shows $a \leq c \implies d \leq b \implies a - b \leq c - d$

$\langle \text{proof} \rangle$

lemma *ennreal-minus-eq-top[simp]*: $a - (b :: \text{ennreal}) = \text{top} \longleftrightarrow a = \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-divide-self[simp]*: $a \neq 0 \implies a < \text{top} \implies a / a = (1 :: \text{ennreal})$

$\langle \text{proof} \rangle$

39.6 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: $\text{real} \Rightarrow \text{ennreal}$ **is** $\text{sup } 0 \circ \text{ereal}$

$\langle \text{proof} \rangle$

declare $[[\text{coercion } \text{ennreal}]]$

lemma *ennreal-cong*: $x = y \implies \text{ennreal } x = \text{ennreal } y$ $\langle \text{proof} \rangle$

lemma *ennreal-cases*[*cases type: ennreal*]:

fixes $x :: \text{ennreal}$

obtains $(\text{real})\ r :: \text{real}$ **where** $0 \leq r$ $x = \text{ennreal } r \mid (\text{top})\ x = \text{top}$

$\langle \text{proof} \rangle$

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]

lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neg-top[simp]*: $\text{ennreal } r \neq \text{top}$

$\langle \text{proof} \rangle$

lemma *top-neg-ennreal[simp]*: $\text{top} \neq \text{ennreal } r$

$\langle \text{proof} \rangle$

lemma *ennreal-less-top[simp]*: $\text{ennreal } x < \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-neg*: $x \leq 0 \implies \text{ennreal } x = 0$

$\langle \text{proof} \rangle$

lemma *ennreal-inj[simp]*:

$0 \leq a \implies 0 \leq b \implies \text{ennreal } a = \text{ennreal } b \longleftrightarrow a = b$

$\langle \text{proof} \rangle$

lemma *ennreal-le-iff[simp]*: $0 \leq y \implies \text{ennreal } x \leq \text{ennreal } y \longleftrightarrow x \leq y$

$\langle \text{proof} \rangle$

lemma *le-ennreal-iff*: $0 \leq r \implies x \leq \text{ennreal } r \longleftrightarrow (\exists q \geq 0. x = \text{ennreal } q \wedge q \leq$

$r)$
 $\langle proof \rangle$

lemma *ennreal-less-iff*: $0 \leq r \implies \text{ennreal } r < \text{ennreal } q \longleftrightarrow r < q$
 $\langle proof \rangle$

lemma *ennreal-eq-zero-iff[simp]*: $0 \leq x \implies \text{ennreal } x = 0 \longleftrightarrow x = 0$
 $\langle proof \rangle$

lemma *ennreal-less-zero-iff[simp]*: $0 < \text{ennreal } x \longleftrightarrow 0 < x$
 $\langle proof \rangle$

lemma *ennreal-lessI*: $0 < q \implies r < q \implies \text{ennreal } r < \text{ennreal } q$
 $\langle proof \rangle$

lemma *ennreal-leI*: $x \leq y \implies \text{ennreal } x \leq \text{ennreal } y$
 $\langle proof \rangle$

lemma *enn2ereal-ennreal[simp]*: $0 \leq x \implies \text{enn2ereal } (\text{ennreal } x) = x$
 $\langle proof \rangle$

lemma *e2ennreal-enn2ereal[simp]*: $e2ennreal (\text{enn2ereal } x) = x$
 $\langle proof \rangle$

lemma *enn2ereal-e2ennreal*: $x \geq 0 \implies \text{enn2ereal } (e2ennreal x) = x$
 $\langle proof \rangle$

lemma *e2ennreal-ereal [simp]*: $e2ennreal (\text{ereal } x) = \text{ennreal } x$
 $\langle proof \rangle$

lemma *ennreal-0[simp]*: $\text{ennreal } 0 = 0$
 $\langle proof \rangle$

lemma *ennreal-1[simp]*: $\text{ennreal } 1 = 1$
 $\langle proof \rangle$

lemma *ennreal-eq-0-iff*: $\text{ennreal } x = 0 \longleftrightarrow x \leq 0$
 $\langle proof \rangle$

lemma *ennreal-le-iff2*: $\text{ennreal } x \leq \text{ennreal } y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge y \leq 0))$
 $\langle proof \rangle$

lemma *ennreal-eq-1[simp]*: $\text{ennreal } x = 1 \longleftrightarrow x = 1$
 $\langle proof \rangle$

lemma *ennreal-le-1[simp]*: $\text{ennreal } x \leq 1 \longleftrightarrow x \leq 1$
 $\langle proof \rangle$

lemma *ennreal-ge-1[simp]*: $\text{ennreal } x \geq 1 \longleftrightarrow x \geq 1$
 $\langle \text{proof} \rangle$

lemma *one-less-ennreal[simp]*: $1 < \text{ennreal } x \longleftrightarrow 1 < x$
 $\langle \text{proof} \rangle$

lemma *ennreal-plus[simp]*:
 $0 \leq a \implies 0 \leq b \implies \text{ennreal } (a + b) = \text{ennreal } a + \text{ennreal } b$
 $\langle \text{proof} \rangle$

lemma *add-mono-ennreal*: $x < \text{ennreal } y \implies x' < \text{ennreal } y' \implies x + x' < \text{ennreal } (y + y')$
 $\langle \text{proof} \rangle$

lemma *sum-ennreal[simp]*: $(\bigwedge i. i \in I \implies 0 \leq f i) \implies (\sum_{i \in I} \text{ennreal } (f i)) = \text{ennreal } (\sum f I)$
 $\langle \text{proof} \rangle$

lemma *sum-list-ennreal[simp]*:
assumes $\bigwedge x. x \in \text{set } xs \implies f x \geq 0$
shows $\text{sum-list } (\text{map } (\lambda x. \text{ennreal } (f x)) xs) = \text{ennreal } (\text{sum-list } (\text{map } f xs))$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-nat-eq-real-of-nat*: $\text{of-nat } i = \text{ennreal } (\text{of-nat } i)$
 $\langle \text{proof} \rangle$

lemma *of-nat-le-ennreal-iff[simp]*: $0 \leq r \implies \text{of-nat } i \leq \text{ennreal } r \longleftrightarrow \text{of-nat } i \leq r$
 $\langle \text{proof} \rangle$

lemma *ennreal-le-of-nat-iff[simp]*: $\text{ennreal } r \leq \text{of-nat } i \longleftrightarrow r \leq \text{of-nat } i$
 $\langle \text{proof} \rangle$

lemma *ennreal-indicator*: $\text{ennreal } (\text{indicator } A x) = \text{indicator } A x$
 $\langle \text{proof} \rangle$

lemma *ennreal-numeral[simp]*: $\text{ennreal } (\text{numeral } n) = \text{numeral } n$
 $\langle \text{proof} \rangle$

lemma *ennreal-less-numeral-iff [simp]*: $\text{ennreal } n < \text{numeral } w \longleftrightarrow n < \text{numeral } w$
 $\langle \text{proof} \rangle$

lemma *numeral-less-ennreal-iff [simp]*: $\text{numeral } w < \text{ennreal } n \longleftrightarrow \text{numeral } w < n$
 $\langle \text{proof} \rangle$

lemma *numeral-le-ennreal-iff [simp]*: $\text{numeral } n \leq \text{ennreal } m \longleftrightarrow \text{numeral } n \leq m$
 $\langle \text{proof} \rangle$

lemma *min-ennreal*: $0 \leq x \implies 0 \leq y \implies \min (\text{ennreal } x) (\text{ennreal } y) = \text{ennreal } (\min x y)$
 ⟨proof⟩

lemma *ennreal-half[simp]*: $\text{ennreal } (1/2) = \text{inverse } 2$
 ⟨proof⟩

lemma *ennreal-minus*: $0 \leq q \implies \text{ennreal } r - \text{ennreal } q = \text{ennreal } (r - q)$
 ⟨proof⟩

lemma *ennreal-minus-top[simp]*: $\text{ennreal } a - \text{top} = 0$
 ⟨proof⟩

lemma *e2eenreal-enn2ereal-diff [simp]*:
 $e2\text{ennreal}(\text{enn2ereal } x - \text{enn2ereal } y) = x - y$ **for** $x y$
 ⟨proof⟩

lemma *ennreal-mult*: $0 \leq a \implies 0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
 ⟨proof⟩

lemma *ennreal-mult'*: $0 \leq a \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
 ⟨proof⟩

lemma *indicator-mult-ennreal*: $\text{indicator } A x * \text{ennreal } r = \text{ennreal } (\text{indicator } A x * r)$
 ⟨proof⟩

lemma *ennreal-mult''*: $0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
 ⟨proof⟩

lemma *numeral-mult-ennreal*: $0 \leq x \implies \text{numeral } b * \text{ennreal } x = \text{ennreal } (\text{numeral } b * x)$
 ⟨proof⟩

lemma *ennreal-power*: $0 \leq r \implies \text{ennreal } r ^ n = \text{ennreal } (r ^ n)$
 ⟨proof⟩

lemma *power-eq-top-ennreal*: $x ^ n = \text{top} \longleftrightarrow (n \neq 0 \wedge (x::\text{ennreal}) = \text{top})$
 ⟨proof⟩

lemma *inverse-ennreal*: $0 < r \implies \text{inverse } (\text{ennreal } r) = \text{ennreal } (\text{inverse } r)$
 ⟨proof⟩

lemma *divide-ennreal*: $0 \leq r \implies 0 < q \implies \text{ennreal } r / \text{ennreal } q = \text{ennreal } (r / q)$
 ⟨proof⟩

lemma *ennreal-inverse-power*: $\text{inverse } (x \wedge n :: \text{ennreal}) = \text{inverse } x \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-divide-distrib-ennreal* [algebra-simps]:
 $(x / y) \wedge n = x \wedge n / (y \wedge n :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-divide-numeral*: $0 \leq x \implies \text{ennreal } x / \text{numeral } b = \text{ennreal } (x / \text{numeral } b)$
 $\langle \text{proof} \rangle$

lemma *prod-ennreal*: $(\bigwedge i. i \in A \implies 0 \leq f i) \implies (\prod_{i \in A. \text{ennreal } (f i)} = \text{ennreal } (\text{prod } f A))$
 $\langle \text{proof} \rangle$

lemma *prod-mono-ennreal*:
assumes $\bigwedge x. x \in A \implies f x \leq (g x :: \text{ennreal})$
shows $\text{prod } f A \leq \text{prod } g A$
 $\langle \text{proof} \rangle$

lemma *mult-right-ennreal-cancel*: $a * \text{ennreal } c = b * \text{ennreal } c \longleftrightarrow (a = b \vee c \leq 0)$
 $\langle \text{proof} \rangle$

lemma *ennreal-le-epsilon*:
 $(\bigwedge e :: \text{real}. y < \text{top} \implies 0 < e \implies x \leq y + \text{ennreal } e) \implies x \leq y$
 $\langle \text{proof} \rangle$

lemma *ennreal-rat-dense*:
fixes $x y :: \text{ennreal}$
shows $x < y \implies \exists r :: \text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$
 $\langle \text{proof} \rangle$

lemma *ennreal-Ex-less-of-nat*: $(x :: \text{ennreal}) < \text{top} \implies \exists n. x < \text{of-nat } n$
 $\langle \text{proof} \rangle$

39.7 Coercion from *ennreal* to *real*

definition $\text{enn2real } x = \text{real-of-ereal } (\text{enn2ereal } x)$

lemma *enn2real-nonneg[simp]*: $0 \leq \text{enn2real } x$
 $\langle \text{proof} \rangle$

lemma *enn2real-mono*: $a \leq b \implies b < \text{top} \implies \text{enn2real } a \leq \text{enn2real } b$
 $\langle \text{proof} \rangle$

lemma *enn2real-of-nat[simp]*: $\text{enn2real } (\text{of-nat } n) = n$
 $\langle \text{proof} \rangle$

lemma *enn2real-ennreal[simp]*: $0 \leq r \implies \text{enn2real } (\text{ennreal } r) = r$
 ⟨proof⟩

lemma *ennreal-enn2real[simp]*: $r < \text{top} \implies \text{ennreal } (\text{enn2real } r) = r$
 ⟨proof⟩

lemma *real-of-ereal-enn2ereal[simp]*: $\text{real-of-ereal } (\text{enn2ereal } x) = \text{enn2real } x$
 ⟨proof⟩

lemma *enn2real-top[simp]*: $\text{enn2real } \text{top} = 0$
 ⟨proof⟩

lemma *enn2real-0[simp]*: $\text{enn2real } 0 = 0$
 ⟨proof⟩

lemma *enn2real-1[simp]*: $\text{enn2real } 1 = 1$
 ⟨proof⟩

lemma *enn2real-numeral[simp]*: $\text{enn2real } (\text{numeral } n) = (\text{numeral } n)$
 ⟨proof⟩

lemma *enn2real-mult*: $\text{enn2real } (a * b) = \text{enn2real } a * \text{enn2real } b$
 ⟨proof⟩

lemma *enn2real-leI*: $0 \leq B \implies x \leq \text{ennreal } B \implies \text{enn2real } x \leq B$
 ⟨proof⟩

lemma *enn2real-positive-iff*: $0 < \text{enn2real } x \longleftrightarrow (0 < x \wedge x < \text{top})$
 ⟨proof⟩

lemma *enn2real-eq-posreal-iff[simp]*: $c > 0 \implies \text{enn2real } x = c \longleftrightarrow x = c$
 ⟨proof⟩

lemma *ennreal-enn2real-if*: $\text{ennreal } (\text{enn2real } r) = (\text{if } r = \text{top} \text{ then } 0 \text{ else } r)$
 ⟨proof⟩

39.8 Coercion from *enat* to *ennreal*

definition *ennreal-of-enat* :: $\text{enat} \Rightarrow \text{ennreal}$

where

ennreal-of-enat $n = (\text{case } n \text{ of } \infty \Rightarrow \text{top} \mid \text{enat } n \Rightarrow \text{of-nat } n)$

declare [[*coercion* *ennreal-of-enat*]]

declare [[*coercion* *of-nat* :: $\text{nat} \Rightarrow \text{ennreal}$]]

lemma *ennreal-of-enat-infty[simp]*: $\text{ennreal-of-enat } \infty = \infty$
 ⟨proof⟩

lemma *ennreal-of-enat-enat[simp]*: $\text{ennreal-of-enat } (\text{enat } n) = \text{of-nat } n$

$\langle \text{proof} \rangle$

lemma *ennreal-of-enat-0[simp]*: *ennreal-of-enat 0 = 0*
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-1[simp]*: *ennreal-of-enat 1 = 1*
 $\langle \text{proof} \rangle$

lemma *ennreal-top-neq-of-nat[simp]*: *(top::ennreal) $\neq$ of-nat i*
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-inj[simp]*: *ennreal-of-enat i = ennreal-of-enat j $\longleftrightarrow$ i = j*
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-le-iff[simp]*: *ennreal-of-enat m $\leq$ ennreal-of-enat n $\longleftrightarrow$ m $\leq$ n*
 $\langle \text{proof} \rangle$

lemma *of-nat-less-ennreal-of-nat[simp]*: *of-nat n $\leq$ ennreal-of-enat x $\longleftrightarrow$ of-nat n $\leq$ x*
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-Sup*: *ennreal-of-enat (Sup X) = (SUP x $\in$ X. ennreal-of-enat x)*
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-eSuc[simp]*: *ennreal-of-enat (eSuc x) = 1 + ennreal-of-enat x*
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-plus[simp]*: *ennreal-of-enat (a+b) = ennreal-of-enat a + ennreal-of-enat b*
 $\langle \text{proof} \rangle$

lemma *sum-ennreal-of-enat[simp]*: *($\sum i \in I. \text{ennreal-of-enat } (f i)$) = ennreal-of-enat ($\text{sum } f I$)*
 $\langle \text{proof} \rangle$

39.9 Topology on *ennreal*

lemma *enn2ereal-Iio*: *enn2ereal - ‘{.. a } = (if 0 $\leq$ a then {.. $e2ennreal a$ } else {})*
 $\langle \text{proof} \rangle$

lemma *enn2ereal-Ioi*: *enn2ereal - ‘{a <.. $\}$ = (if 0 $\leq$ a then { $e2ennreal a$ <.. $\}$ else UNIV)*
 $\langle \text{proof} \rangle$

instantiation *ennreal* :: *linear-continuum-topology*
begin

definition *open-ennreal* :: *ennreal set* $\Rightarrow$ *bool*
where (*open* :: *ennreal set* $\Rightarrow$ *bool*) = *generate-topology* (*range lessThan* $\cup$ *range greaterThan*)

instance
 $\langle$ *proof* $\rangle$

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*
 $\langle$ *proof* $\rangle$

lemma *continuous-at-e2ennreal*: *continuous (at x within A) e2ennreal*
 $\langle$ *proof* $\rangle$

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*
 $\langle$ *proof* $\rangle$

lemma *continuous-at-enn2ereal*: *continuous (at x within A) enn2ereal*
 $\langle$ *proof* $\rangle$

lemma *sup-continuous-e2ennreal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* ($\lambda x. e2ennreal (f x)$)
 $\langle$ *proof* $\rangle$

lemma *sup-continuous-enn2ereal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* ($\lambda x. enn2ereal (f x)$)
 $\langle$ *proof* $\rangle$

lemma *sup-continuous-mult-left-ennreal'*:
fixes *c* :: *ennreal*
shows *sup-continuous* ($\lambda x. c * x$)
 $\langle$ *proof* $\rangle$

lemma *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:
sup-continuous f $\Longrightarrow$ *sup-continuous* ($\lambda x. c * f x :: ennreal$)
 $\langle$ *proof* $\rangle$

lemma *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:
sup-continuous f $\Longrightarrow$ *sup-continuous* ($\lambda x. f x * c :: ennreal$)
 $\langle$ *proof* $\rangle$

lemma *sup-continuous-divide-ennreal*[*order-continuous-intros*]:
fixes *f g* :: '*a::complete-lattice* $\Rightarrow$ *ennreal*
shows *sup-continuous f* $\Longrightarrow$ *sup-continuous* ($\lambda x. f x / c$)

$\langle \text{proof} \rangle$

lemma *transfer-enn2ereal-continuous-on* [*transfer-rule*]:

rel-fun (=) (*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *continuous-on* *continuous-on*
 $\langle \text{proof} \rangle$

lemma *transfer-sup-continuous*[*transfer-rule*]:

(*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *sup-continuous* *sup-continuous*
 $\langle \text{proof} \rangle$

lemma *continuous-on-ennreal*[*tendsto-intros*]:

continuous-on *A* *f* $\implies$ *continuous-on* *A* ($\lambda x. \text{ennreal } (f x)$)
 $\langle \text{proof} \rangle$

lemma *tendsto-ennrealD*:

assumes *lim*: ($\lambda x. \text{ennreal } (f x) \longrightarrow \text{ennreal } x$) *F*
assumes *: $\forall_F x \text{ in } F. 0 \leq f x$ **and** *x*: $0 \leq x$
shows ($f \longrightarrow x$) *F*
 $\langle \text{proof} \rangle$

lemma *tendsto-ennreal-iff* [*simp*]:

$\langle (\lambda x. \text{ennreal } (f x) \longrightarrow \text{ennreal } x) F \longleftrightarrow (f \longrightarrow x) F \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)
if $\langle \forall_F x \text{ in } F. 0 \leq f x \rangle \langle 0 \leq x \rangle$
 $\langle \text{proof} \rangle$

lemma *tendsto-enn2ereal-iff*[*simp*]: ($\lambda i. \text{enn2ereal } (f i) \longrightarrow \text{enn2ereal } x$) *F* $\longleftrightarrow$
 $(f \longrightarrow x) F$
 $\langle \text{proof} \rangle$

lemma *ennreal-tendsto-0-iff*: ($\bigwedge n. f n \geq 0$) $\implies$ ($\lambda n. \text{ennreal } (f n) \longrightarrow 0$)
 $\longleftrightarrow (f \longrightarrow 0)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-add-ennreal*:

fixes *f g* :: 'a::topological-space $\Rightarrow$ *ennreal*
shows *continuous-on* *A* *f* $\implies$ *continuous-on* *A* *g* $\implies$ *continuous-on* *A* ($\lambda x. f x$
 $+ g x$)
 $\langle \text{proof} \rangle$

lemma *continuous-on-inverse-ennreal*[*continuous-intros*]:

fixes *f* :: 'a::topological-space $\Rightarrow$ *ennreal*
shows *continuous-on* *A* *f* $\implies$ *continuous-on* *A* ($\lambda x. \text{inverse } (f x)$)
 $\langle \text{proof} \rangle$

instance *ennreal* :: *topological-comm-monoid-add*

$\langle \text{proof} \rangle$

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:

fixes *f g* :: 'a::complete-lattice $\Rightarrow$ *ennreal*

shows $\text{sup-continuous } f \implies \text{sup-continuous } g \implies \text{sup-continuous } (\lambda x. f\ x + g\ x)$
 ⟨proof⟩

lemma $\text{ennreal-suminf-lessD}$: $(\sum i. f\ i :: \text{ennreal}) < x \implies f\ i < x$
 ⟨proof⟩

lemma $\text{sums-ennreal[simp]}$: $(\bigwedge i. 0 \leq f\ i) \implies 0 \leq x \implies (\lambda i. \text{ennreal } (f\ i)) \text{ sums } \text{ennreal } x \longleftrightarrow f \text{ sums } x$
 ⟨proof⟩

lemma $\text{summable-suminf-not-top}$: $(\bigwedge i. 0 \leq f\ i) \implies (\sum i. \text{ennreal } (f\ i)) \neq \text{top} \implies \text{summable } f$
 ⟨proof⟩

lemma $\text{suminf-ennreal[simp]}$:
 $(\bigwedge i. 0 \leq f\ i) \implies (\sum i. \text{ennreal } (f\ i)) \neq \text{top} \implies (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } (\sum i. f\ i)$
 ⟨proof⟩

lemma $\text{sums-enn2ereal[simp]}$: $(\lambda i. \text{enn2ereal } (f\ i)) \text{ sums } \text{enn2ereal } x \longleftrightarrow f \text{ sums } x$
 ⟨proof⟩

lemma $\text{suminf-enn2ereal[simp]}$: $(\sum i. \text{enn2ereal } (f\ i)) = \text{enn2ereal } (\text{suminf } f)$
 ⟨proof⟩

lemma $\text{transfer-e2ennreal-suminf [transfer-rule]}$: $\text{rel-fun } (\text{rel-fun } (=) \text{ pcr-ennreal}) \text{ pcr-ennreal suminf suminf}$
 ⟨proof⟩

lemma $\text{ennreal-suminf-cmult[simp]}$: $(\sum i. r * f\ i) = r * (\sum i. f\ i :: \text{ennreal})$
 ⟨proof⟩

lemma $\text{ennreal-suminf-multc[simp]}$: $(\sum i. f\ i * r) = (\sum i. f\ i :: \text{ennreal}) * r$
 ⟨proof⟩

lemma $\text{ennreal-suminf-divide[simp]}$: $(\sum i. f\ i / r) = (\sum i. f\ i :: \text{ennreal}) / r$
 ⟨proof⟩

lemma $\text{ennreal-suminf-neq-top}$: $\text{summable } f \implies (\bigwedge i. 0 \leq f\ i) \implies (\sum i. \text{ennreal } (f\ i)) \neq \text{top}$
 ⟨proof⟩

lemma suminf-ennreal-eq :
 $(\bigwedge i. 0 \leq f\ i) \implies f \text{ sums } x \implies (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } x$
 ⟨proof⟩

lemma $\text{ennreal-suminf-bound-add}$:

fixes $f :: \text{nat} \Rightarrow \text{ennreal}$
shows $(\bigwedge N. (\sum n < N. f\ n) + y \leq x) \Longrightarrow \text{suminf } f + y \leq x$
 $\langle \text{proof} \rangle$

lemma *ennreal-suminf-SUP-eq-directed*:

fixes $f :: 'a \Rightarrow \text{nat} \Rightarrow \text{ennreal}$
assumes $*$: $\bigwedge N\ i\ j. i \in I \Longrightarrow j \in I \Longrightarrow \text{finite } N \Longrightarrow \exists k \in I. \forall n \in N. f\ i\ n \leq f\ k\ n \wedge f\ j\ n \leq f\ k\ n$
shows $(\sum n. \text{SUP } i \in I. f\ i\ n) = (\text{SUP } i \in I. \sum n. f\ i\ n)$
 $\langle \text{proof} \rangle$

lemma *INF-ennreal-add-const*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$
shows $(\text{INF } i. f\ i + c) = (\text{INF } i. f\ i) + c$
 $\langle \text{proof} \rangle$

lemma *INF-ennreal-const-add*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$
shows $(\text{INF } i. c + f\ i) = c + (\text{INF } i. f\ i)$
 $\langle \text{proof} \rangle$

lemma *SUP-mult-left-ennreal*: $c * (\text{SUP } i \in I. f\ i) = (\text{SUP } i \in I. c * f\ i :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *SUP-mult-right-ennreal*: $(\text{SUP } i \in I. f\ i) * c = (\text{SUP } i \in I. f\ i * c :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *SUP-divide-ennreal*: $(\text{SUP } i \in I. f\ i) / c = (\text{SUP } i \in I. f\ i / c :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-of-nat-eq-top*: $(\text{SUP } x. \text{of-nat } x :: \text{ennreal}) = \text{top}$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-eq-top*:

fixes $f :: 'a \Rightarrow \text{ennreal}$
assumes $\bigwedge n. \exists i \in I. \text{of-nat } n \leq f\ i$
shows $(\text{SUP } i \in I. f\ i) = \text{top}$
 $\langle \text{proof} \rangle$

lemma *ennreal-INF-const-minus*:

fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $I \neq \{\} \Longrightarrow (\text{SUP } x \in I. c - f\ x) = c - (\text{INF } x \in I. f\ x)$
 $\langle \text{proof} \rangle$

lemma *of-nat-Sup-ennreal*:

assumes $A \neq \{\}$ *bdd-above* A
shows $\text{of-nat } (\text{Sup } A) = (\text{SUP } a \in A. \text{of-nat } a :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-tendsto-const-minus*:

fixes $g :: 'a \Rightarrow \text{ennreal}$
assumes $ae: \forall_F x \text{ in } F. g\ x \leq c$
assumes $g: ((\lambda x. c - g\ x) \longrightarrow 0) F$
shows $(g \longrightarrow c) F$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-add*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$
shows $\text{incseq } f \Longrightarrow \text{incseq } g \Longrightarrow (\text{SUP } i. f\ i + g\ i) = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-sum*:

fixes $f :: 'a \Rightarrow \text{nat} \Rightarrow \text{ennreal}$
shows $(\bigwedge i. i \in I \Longrightarrow \text{incseq } (f\ i)) \Longrightarrow (\text{SUP } n. \sum_{i \in I} f\ i\ n) = (\sum_{i \in I} \text{SUP } n. f\ i\ n)$
 $\langle \text{proof} \rangle$

lemma *ennreal-liminf-minus*:

fixes $f :: \text{nat} \Rightarrow \text{ennreal}$
shows $(\bigwedge n. f\ n \leq c) \Longrightarrow \text{liminf } (\lambda n. c - f\ n) = c - \text{limsup } f$
 $\langle \text{proof} \rangle$

lemma *ennreal-continuous-on-cmult*:

$(c :: \text{ennreal}) < \text{top} \Longrightarrow \text{continuous-on } A\ f \Longrightarrow \text{continuous-on } A\ (\lambda x. c * f\ x)$
 $\langle \text{proof} \rangle$

lemma *ennreal-tendsto-cmult*:

$(c :: \text{ennreal}) < \text{top} \Longrightarrow (f \longrightarrow x) F \Longrightarrow ((\lambda x. c * f\ x) \longrightarrow c * x) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-ennrealI*[*intro, simp, tendsto-intros*]:

$(f \longrightarrow x) F \Longrightarrow ((\lambda x. \text{ennreal } (f\ x)) \longrightarrow \text{ennreal } x) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-enn2erealI* [*tendsto-intros*]:

assumes $(f \longrightarrow l) F$
shows $((\lambda i. \text{enn2ereal } (f\ i)) \longrightarrow \text{enn2ereal } l) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-e2ennrealI* [*tendsto-intros*]:

assumes $(f \longrightarrow l) F$
shows $((\lambda i. \text{e2ennreal } (f\ i)) \longrightarrow \text{e2ennreal } l) F$
 $\langle \text{proof} \rangle$

lemma *ennreal-suminf-minus*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$
shows $(\bigwedge i. g\ i \leq f\ i) \Longrightarrow \text{suminf } f \neq \text{top} \Longrightarrow \text{suminf } g \neq \text{top} \Longrightarrow (\sum i. f\ i - g\ i)$

$i) = \text{suminf } f - \text{suminf } g$
 $\langle \text{proof} \rangle$

lemma *ennreal-Sup-countable-SUP*:

$A \neq \{\}$ $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{incseq } f \wedge \text{range } f \subseteq A \wedge \text{Sup } A = (\text{SUP } i. f \ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-Inf-countable-INF*:

$A \neq \{\}$ $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{decseq } f \wedge \text{range } f \subseteq A \wedge \text{Inf } A = (\text{INF } i. f \ i)$
including *ennreal.lifting*
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-countable-SUP*:

$A \neq \{\}$ $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{range } f \subseteq g'A \wedge \text{Sup } (g' A) = \text{Sup } (f' \text{ UNIV})$
 $\langle \text{proof} \rangle$

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n :: \text{nat}. \text{of-nat } n :: \text{ennreal}) \longrightarrow \text{top}$
 $\langle \text{proof} \rangle$

lemma *SUP-sup-continuous-ennreal*:

fixes $f :: \text{ennreal} \Rightarrow 'a :: \text{complete-lattice}$
assumes f : *sup-continuous* f **and** $I \neq \{\}$
shows $(\text{SUP } i \in I. f \ (g \ i)) = f \ (\text{SUP } i \in I. g \ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-suminf-SUP-eq*:

fixes $f :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ennreal}$
shows $(\bigwedge i. \text{incseq } (\lambda n. f \ n \ i)) \implies (\sum i. \text{SUP } n. f \ n \ i) = (\text{SUP } n. \sum i. f \ n \ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-add-left*:

fixes $c :: \text{ennreal}$
shows $I \neq \{\} \implies (\text{SUP } i \in I. f \ i + c) = (\text{SUP } i \in I. f \ i) + c$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-const-minus*:

fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $I \neq \{\} \implies c < \text{top} \implies (\text{INF } x \in I. c - f \ x) = c - (\text{SUP } x \in I. f \ x)$
 $\langle \text{proof} \rangle$

lemma *isCont-ennreal[simp]*: $\langle \text{isCont } \text{ennreal } x \rangle$
 $\langle \text{proof} \rangle$

lemma *isCont-ennreal-of-enat[simp]*: $\langle \text{isCont } \text{ennreal-of-enat } x \rangle$
 $\langle \text{proof} \rangle$

39.10 Approximation lemmas

lemma *INF-approx-ennreal*:

fixes $x::ennreal$ **and** $e::real$

assumes $e > 0$

assumes *INF*: $x = (INF\ i \in A. f\ i)$

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*: $\bigwedge i. i \in A \implies f\ i \leq x$

assumes *approx*: $\bigwedge e. (e::real) > 0 \implies \exists i \in A. x \leq f\ i + e$

shows $x = (SUP\ i \in A. f\ i)$

<proof>

lemma *ennreal-approx-INF*:

fixes $x::ennreal$

assumes *f-bound*: $\bigwedge i. i \in A \implies x \leq f\ i$

assumes *approx*: $\bigwedge e. (e::real) > 0 \implies \exists i \in A. f\ i \leq x + e$

shows $x = (INF\ i \in A. f\ i)$

<proof>

lemma *ennreal-approx-unit*:

$(\bigwedge a::ennreal. 0 < a \implies a < 1 \implies a * z \leq y) \implies z \leq y$

<proof>

lemma *suminf-ennreal2*:

$(\bigwedge i. 0 \leq f\ i) \implies \text{summable } f \implies (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } (\sum i. f\ i)$

<proof>

lemma *less-top-ennreal*: $x < \text{top} \longleftrightarrow (\exists r \geq 0. x = \text{ennreal } r)$

<proof>

lemma *enn2real-less-iff[simp]*: $x < \text{top} \implies \text{enn2real } x < c \longleftrightarrow x < c$

<proof>

lemma *enn2real-le-iff[simp]*: $\llbracket x < \text{top}; c > 0 \rrbracket \implies \text{enn2real } x \leq c \longleftrightarrow x \leq c$

<proof>

lemma *enn2real-less*:

assumes $enn2real\ e < r\ e \neq top$ **shows** $e < ennreal\ r$
 $\langle proof \rangle$

lemma *enn2real-le*:

assumes $enn2real\ e \leq r\ e \neq top$ **shows** $e \leq ennreal\ r$
 $\langle proof \rangle$

lemma *tendsto-top-iff-ennreal*:

fixes $f :: 'a \Rightarrow ennreal$
shows $(f \longrightarrow top)\ F \longleftrightarrow (\forall l \geq 0. eventually\ (\lambda x. ennreal\ l < f\ x)\ F)$
 $\langle proof \rangle$

lemma *ennreal-tendsto-top-eq-at-top*:

$((\lambda z. ennreal\ (f\ z)) \longrightarrow top)\ F \longleftrightarrow (LIM\ z\ F. f\ z :> at-top)$
 $\langle proof \rangle$

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:

fixes $f :: - \Rightarrow ennreal$
shows $Limsup\ F\ f = 0 \implies (f \longrightarrow 0)\ F$
 $\langle proof \rangle$

lemma *diff-le-self-ennreal[simp]*: $a - b \leq (a::ennreal)$

$\langle proof \rangle$

lemma *ennreal-ineq-diff-add*: $b \leq a \implies a = b + (a - b::ennreal)$

$\langle proof \rangle$

lemma *ennreal-mult-strict-left-mono*: $(a::ennreal) < c \implies 0 < b \implies b < top \implies$

$b * a < b * c$

$\langle proof \rangle$

lemma *ennreal-between*: $0 < e \implies 0 < x \implies x < top \implies x - e < (x::ennreal)$

$\langle proof \rangle$

lemma *minus-less-iff-ennreal*: $b < top \implies b \leq a \implies a - b < c \longleftrightarrow a < c +$

$(b::ennreal)$

$\langle proof \rangle$

lemma *tendsto-zero-ennreal*:

assumes $ev: \bigwedge r. 0 < r \implies \forall_F\ x\ in\ F. f\ x < ennreal\ r$

shows $(f \longrightarrow 0)\ F$

$\langle proof \rangle$

lifting-update *ennreal.lifting*

lifting-forget *ennreal.lifting*

39.11 *ennreal* theorems

lemma *neg-top-trans*: **fixes** $x\ y :: ennreal$ **shows** $\llbracket y \neq top; x \leq y \rrbracket \implies x \neq top$

$\langle proof \rangle$

lemma *diff-diff-ennreal*: **fixes** $a\ b :: \text{ennreal}$ **shows** $a \leq b \implies b \neq \infty \implies b - (b - a) = a$
 $\langle proof \rangle$

lemma *ennreal-less-one-iff[simp]*: $\text{ennreal } x < 1 \longleftrightarrow x < 1$
 $\langle proof \rangle$

lemma *SUP-const-minus-ennreal*:
fixes $f :: 'a \Rightarrow \text{ennreal}$ **shows** $I \neq \{\} \implies (\text{SUP } x \in I. c - f\ x) = c - (\text{INF } x \in I. f\ x)$
including *ennreal.lifting*
 $\langle proof \rangle$

lemma *zero-minus-ennreal[simp]*: $0 - (a :: \text{ennreal}) = 0$
including *ennreal.lifting*
 $\langle proof \rangle$

lemma *diff-diff-commute-ennreal*:
fixes $a\ b\ c :: \text{ennreal}$ **shows** $a - b - c = a - c - b$
 $\langle proof \rangle$

lemma *diff-gr0-ennreal*: $b < (a :: \text{ennreal}) \implies 0 < a - b$
including *ennreal.lifting* $\langle proof \rangle$

lemma *divide-le-posI-ennreal*:
fixes $x\ y\ z :: \text{ennreal}$
shows $x > 0 \implies z \leq x * y \implies z / x \leq y$
 $\langle proof \rangle$

lemma *add-diff-eq-ennreal*:
fixes $x\ y\ z :: \text{ennreal}$
shows $z \leq y \implies x + (y - z) = x + y - z$
including *ennreal.lifting*
 $\langle proof \rangle$

lemma *add-diff-inverse-ennreal*:
fixes $x\ y :: \text{ennreal}$ **shows** $x \leq y \implies x + (y - x) = y$
 $\langle proof \rangle$

lemma *add-diff-eq-iff-ennreal[simp]*:
fixes $x\ y :: \text{ennreal}$ **shows** $x + (y - x) = y \longleftrightarrow x \leq y$
 $\langle proof \rangle$

lemma *add-diff-le-ennreal*: $a + b - c \leq a + (b - c :: \text{ennreal})$
 $\langle proof \rangle$

lemma *diff-eq-0-ennreal*: $a < \text{top} \implies a \leq b \implies a - b = (0 :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *diff-diff-ennreal'*: **fixes** $x\ y\ z :: \text{ennreal}$ **shows** $z \leq y \implies y - z \leq x \implies x - (y - z) = x + z - y$
 $\langle \text{proof} \rangle$

lemma *diff-diff-ennreal''*: **fixes** $x\ y\ z :: \text{ennreal}$
shows $z \leq y \implies x - (y - z) = (\text{if } y - z \leq x \text{ then } x + z - y \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *power-less-top-ennreal*: **fixes** $x :: \text{ennreal}$ **shows** $x \wedge n < \text{top} \longleftrightarrow x < \text{top} \vee n = 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-divide-times*: $(a / b) * c = a * (c / b :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *diff-less-top-ennreal*: $a - b < \text{top} \longleftrightarrow a < (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *divide-less-ennreal*: $b \neq 0 \implies b < \text{top} \implies a / b < c \longleftrightarrow a < (c * b :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *one-less-numeral[simp]*: $1 < (\text{numeral } n :: \text{ennreal}) \longleftrightarrow (\text{num.One} < n)$
 $\langle \text{proof} \rangle$

lemma *divide-eq-1-ennreal*: $a / b = (1 :: \text{ennreal}) \longleftrightarrow (b \neq \text{top} \wedge b \neq 0 \wedge b = a)$
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-cancel-left*: $(a * b = a * c) = (a = \text{top} \wedge b \neq 0 \wedge c \neq 0 \vee a = 0 \vee b = (c :: \text{ennreal}))$
 $\langle \text{proof} \rangle$

lemma *ennreal-minus-if*: $\text{ennreal } a - \text{ennreal } b = \text{ennreal } (\text{if } 0 \leq b \text{ then } (\text{if } b \leq a \text{ then } a - b \text{ else } 0) \text{ else } a)$
 $\langle \text{proof} \rangle$

lemma *ennreal-plus-if*: $\text{ennreal } a + \text{ennreal } b = \text{ennreal } (\text{if } 0 \leq a \text{ then } (\text{if } 0 \leq b \text{ then } a + b \text{ else } a) \text{ else } b)$
 $\langle \text{proof} \rangle$

lemma *power-le-one-iff*: $0 \leq (a :: \text{real}) \implies a \wedge n \leq 1 \longleftrightarrow (n = 0 \vee a \leq 1)$
 $\langle \text{proof} \rangle$

lemma *ennreal-diff-le-mono-left*: $a \leq b \implies a - c \leq (b :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-minus-le-iff*: $a - b \leq c \longleftrightarrow (a \leq b + (c :: \text{ennreal}) \wedge (a = \text{top} \wedge$

$b = \text{top} \longrightarrow c = \text{top}))$
 $\langle \text{proof} \rangle$

lemma *ennreal-le-minus-iff*: $a \leq b - c \longleftrightarrow (a + c \leq (b::\text{ennreal}) \vee (a = 0 \wedge b \leq c))$
 $\langle \text{proof} \rangle$

lemma *diff-add-eq-diff-diff-swap-ennreal*: $x - (y + z :: \text{ennreal}) = x - y - z$
 $\langle \text{proof} \rangle$

lemma *diff-add-assoc2-ennreal*: $b \leq a \implies (a - b + c::\text{ennreal}) = a + c - b$
 $\langle \text{proof} \rangle$

lemma *diff-gt-0-iff-gt-ennreal*: $0 < a - b \longleftrightarrow (a = \text{top} \wedge b = \text{top} \vee b < (a::\text{ennreal}))$
 $\langle \text{proof} \rangle$

lemma *diff-eq-0-iff-ennreal*: $(a - b::\text{ennreal}) = 0 \longleftrightarrow (a < \text{top} \wedge a \leq b)$
 $\langle \text{proof} \rangle$

lemma *add-diff-self-ennreal*: $a + (b - a::\text{ennreal}) = (\text{if } a \leq b \text{ then } b \text{ else } a)$
 $\langle \text{proof} \rangle$

lemma *diff-add-self-ennreal*: $(b - a + a::\text{ennreal}) = (\text{if } a \leq b \text{ then } b \text{ else } a)$
 $\langle \text{proof} \rangle$

lemma *ennreal-minus-cancel-iff*:
fixes $a \ b \ c :: \text{ennreal}$
shows $a - b = a - c \longleftrightarrow (b = c \vee (a \leq b \wedge a \leq c) \vee a = \text{top})$
 $\langle \text{proof} \rangle$

The next lemma is wrong for $a = \text{top}$, for $b = c = 1$ for instance.

lemma *ennreal-right-diff-distrib*:
fixes $a \ b \ c :: \text{ennreal}$
assumes $a \neq \text{top}$
shows $a * (b - c) = a * b - a * c$
 $\langle \text{proof} \rangle$

lemma *SUP-diff-ennreal*:
 $c < \text{top} \implies (\text{SUP } i \in I. f \ i - c :: \text{ennreal}) = (\text{SUP } i \in I. f \ i) - c$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-add-right*:
fixes $c :: \text{ennreal}$ **shows** $I \neq \{\} \implies c + (\text{SUP } i \in I. f \ i) = (\text{SUP } i \in I. c + f \ i)$
 $\langle \text{proof} \rangle$

lemma *SUP-add-directed-ennreal*:
fixes $f \ g :: - \Rightarrow \text{ennreal}$
assumes *directed*: $\bigwedge i \ j. i \in I \implies j \in I \implies \exists k \in I. f \ i + g \ j \leq f \ k + g \ k$
shows $(\text{SUP } i \in I. f \ i + g \ i) = (\text{SUP } i \in I. f \ i) + (\text{SUP } i \in I. g \ i)$

<proof>

lemma *enn2real-eq-0-iff*: $\text{enn2real } x = 0 \longleftrightarrow x = 0 \vee x = \text{top}$
<proof>

lemma *continuous-on-diff-ennreal*:

continuous-on A f $\implies$ continuous-on A g $\implies$ ($\bigwedge x. x \in A \implies f x \neq \text{top}$) $\implies$
($\bigwedge x. x \in A \implies g x \neq \text{top}$) $\implies$ continuous-on A ($\lambda z. f z - g z :: \text{ennreal}$)
including *ennreal.lifting*
<proof>

lemma *tendsto-diff-ennreal*:

(f $\longrightarrow$ x) F $\implies$ (g $\longrightarrow$ y) F $\implies$ $x \neq \text{top} \implies y \neq \text{top} \implies$ ($\lambda z. f z - g z :: \text{ennreal}$) $\longrightarrow$ x - y) F
<proof>

declare *lim-real-of-ereal* [*tendsto-intros*]

lemma *tendsto-enn2real* [*tendsto-intros*]:

assumes *(u $\longrightarrow$ ennreal l) F l ≥ 0*
shows *(($\lambda n. \text{enn2real } (u n)$) $\longrightarrow$ l) F*
<proof>

end

40 Logarithm of Natural Numbers

theory *Log-Nat*

imports *Complex-Main*

begin

40.1 Preliminaries

lemma *divide-nat-diff-div-nat-less-one*:

real x / real b - real (x div b) < 1 **for** *x b :: nat*
<proof>

40.2 Floorlog

definition *floorlog* :: *nat $\Rightarrow$ nat $\Rightarrow$ nat*

where *floorlog b a = (if a > 0 $\wedge$ b > 1 then nat $\lfloor \log b a \rfloor$ + 1 else 0)*

lemma *floorlog-mono*: *x $\leq$ y $\implies$ floorlog b x $\leq$ floorlog b y*

<proof>

lemma *floorlog-bounds*:

*b $\wedge$ (floorlog b x - 1) $\leq$ x $\wedge$ x < b $\wedge$ (floorlog b x) **if** x > 0 b > 1*
<proof>

lemma *floorlog-power* [simp]:

$\text{floorlog } b \ (a * b^c) = \text{floorlog } b \ a + c$ **if** $a > 0 \ b > 1$
 ⟨proof⟩

lemma *floor-log-add-eqI*:

$\lfloor \log b \ (a + r) \rfloor = \lfloor \log b \ a \rfloor$ **if** $b > 1 \ a \geq 1 \ 0 \leq r \ r < 1$
for $a \ b :: \text{nat}$ **and** $r :: \text{real}$
 ⟨proof⟩

lemma *floor-log-div*:

$\lfloor \log b \ x \rfloor = \lfloor \log b \ (x \text{ div } b) \rfloor + 1$ **if** $b > 1 \ x > 0 \ x \text{ div } b > 0$
for $b \ x :: \text{nat}$
 ⟨proof⟩

lemma *compute-floorlog* [code]:

$\text{floorlog } b \ x = (\text{if } x > 0 \wedge b > 1 \text{ then } \text{floorlog } b \ (x \text{ div } b) + 1 \text{ else } 0)$
 ⟨proof⟩

lemma *floor-log-eq-if*:

$\lfloor \log b \ x \rfloor = \lfloor \log b \ y \rfloor$ **if** $x \text{ div } b = y \text{ div } b \ b > 1 \ x > 0 \ x \text{ div } b \geq 1$
for $b \ x \ y :: \text{nat}$
 ⟨proof⟩

lemma *floorlog-eq-if*:

$\text{floorlog } b \ x = \text{floorlog } b \ y$ **if** $x \text{ div } b = y \text{ div } b \ b > 1 \ x > 0 \ x \text{ div } b \geq 1$
for $b \ x \ y :: \text{nat}$
 ⟨proof⟩

lemma *floorlog-leD*:

$\text{floorlog } b \ x \leq w \implies b > 1 \implies x < b^w$
 ⟨proof⟩

lemma *floorlog-leI*:

$x < b^w \implies 0 \leq w \implies b > 1 \implies \text{floorlog } b \ x \leq w$
 ⟨proof⟩

lemma *floorlog-eq-zero-iff*:

$\text{floorlog } b \ x = 0 \iff b \leq 1 \vee x \leq 0$
 ⟨proof⟩

lemma *floorlog-le-iff*:

$\text{floorlog } b \ x \leq w \iff b \leq 1 \vee b > 1 \wedge 0 \leq w \wedge x < b^w$
 ⟨proof⟩

lemma *floorlog-ge-SucI*:

$\text{Suc } w \leq \text{floorlog } b \ x$ **if** $b^w \leq x \ b > 1$
 ⟨proof⟩

lemma *floorlog-geI*:

$w \leq \text{floorlog } b \ x \text{ if } b \wedge (w - 1) \leq x \ b > 1$
 $\langle \text{proof} \rangle$

lemma *floorlog-geD*:
 $b \wedge (w - 1) \leq x \text{ if } w \leq \text{floorlog } b \ x \ w > 0$
 $\langle \text{proof} \rangle$

40.3 Bitlen

definition *bitlen* :: $\text{int} \Rightarrow \text{int}$
where *bitlen* $a = \text{floorlog } 2 \ (\text{nat } a)$

lemma *bitlen-alt-def*:
 $\text{bitlen } a = (\text{if } a > 0 \text{ then } \lfloor \log 2 \ a \rfloor + 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *bitlen-zero* [*simp*]:
 $\text{bitlen } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *bitlen-nonneg*:
 $0 \leq \text{bitlen } x$
 $\langle \text{proof} \rangle$

lemma *bitlen-bounds*:
 $2^{\text{nat } (\text{bitlen } x - 1)} \leq x \wedge x < 2^{\text{nat } (\text{bitlen } x)} \text{ if } x > 0$
 $\langle \text{proof} \rangle$

lemma *bitlen-pow2* [*simp*]:
 $\text{bitlen } (b * 2^c) = \text{bitlen } b + c \text{ if } b > 0$
 $\langle \text{proof} \rangle$

lemma *compute-bitlen* [*code*]:
 $\text{bitlen } x = (\text{if } x > 0 \text{ then } \text{bitlen } (x \text{ div } 2) + 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *bitlen-eq-zero-iff*:
 $\text{bitlen } x = 0 \iff x \leq 0$
 $\langle \text{proof} \rangle$

lemma *bitlen-div*:
 $1 \leq \text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)}$
and $\text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)} < 2 \text{ if } 0 < m$
 $\langle \text{proof} \rangle$

lemma *bitlen-le-iff-floorlog*:
 $\text{bitlen } x \leq w \iff w \geq 0 \wedge \text{floorlog } 2 \ (\text{nat } x) \leq \text{nat } w$
 $\langle \text{proof} \rangle$

lemma *bitlen-le-iff-power*:
 $\text{bitlen } x \leq w \longleftrightarrow w \geq 0 \wedge x < 2^{\wedge \text{nat } w}$
 $\langle \text{proof} \rangle$

lemma *less-power-nat-iff-bitlen*:
 $x < 2^{\wedge w} \longleftrightarrow \text{bitlen } (\text{int } x) \leq w$
 $\langle \text{proof} \rangle$

lemma *bitlen-ge-iff-power*:
 $w \leq \text{bitlen } x \longleftrightarrow w \leq 0 \vee 2^{\wedge (\text{nat } w - 1)} \leq x$
 $\langle \text{proof} \rangle$

lemma *bitlen-twopow-add-eq*:
 $\text{bitlen } (2^{\wedge w} + b) = w + 1 \text{ if } 0 \leq b < 2^{\wedge w}$
 $\langle \text{proof} \rangle$

end

41 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*: $a + \inf b \ c = \inf (a + b) \ (a + c)$
 $\langle \text{proof} \rangle$

lemma *add-inf-distrib-right*: $\inf a \ b + c = \inf (a + c) \ (b + c)$
 $\langle \text{proof} \rangle$

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*: $a + \sup b \ c = \sup (a + b) \ (a + c)$
 $\langle \text{proof} \rangle$

lemma *add-sup-distrib-right*: $\sup a \ b + c = \sup (a + c) \ (b + c)$
 $\langle \text{proof} \rangle$

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*

begin

subclass *semilattice-inf-ab-group-add* $\langle \text{proof} \rangle$

subclass *semilattice-sup-ab-group-add* $\langle \text{proof} \rangle$

lemmas *add-sup-inf-distrib* =

add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

lemma *inf-eq-neg-sup*: $\inf a\ b = -\ \sup (-\ a)\ (-\ b)$

$\langle \text{proof} \rangle$

lemma *sup-eq-neg-inf*: $\sup a\ b = -\ \inf (-\ a)\ (-\ b)$

$\langle \text{proof} \rangle$

lemma *neg-inf-eq-sup*: $-\ \inf a\ b = \sup (-\ a)\ (-\ b)$

$\langle \text{proof} \rangle$

lemma *diff-inf-eq-sup*: $a - \inf b\ c = a + \sup (-\ b)\ (-\ c)$

$\langle \text{proof} \rangle$

lemma *neg-sup-eq-inf*: $-\ \sup a\ b = \inf (-\ a)\ (-\ b)$

$\langle \text{proof} \rangle$

lemma *diff-sup-eq-inf*: $a - \sup b\ c = a + \inf (-\ b)\ (-\ c)$

$\langle \text{proof} \rangle$

lemma *add-eq-inf-sup*: $a + b = \sup a\ b + \inf a\ b$

$\langle \text{proof} \rangle$

41.1 Positive Part, Negative Part, Absolute Value

definition *nppt* :: $'a \Rightarrow 'a$

where *nppt* $x = \inf x\ 0$

definition *pprt* :: $'a \Rightarrow 'a$

where *pprt* $x = \sup x\ 0$

lemma *pprt-neg*: $\text{pprt}\ (-\ x) = -\ \text{nppt}\ x$

$\langle \text{proof} \rangle$

lemma *nppt-neg*: $\text{nppt}\ (-\ x) = -\ \text{pprt}\ x$

$\langle \text{proof} \rangle$

lemma *prts*: $a = \text{pprt}\ a + \text{nppt}\ a$

$\langle \text{proof} \rangle$

lemma *zero-le-pprt[simp]*: $0 \leq \text{pprt}\ a$

$\langle \text{proof} \rangle$

lemma *nprrt-le-zero*[simp]: $\text{nprrt } a \leq 0$
 $\langle \text{proof} \rangle$

lemma *le-eq-neg*: $a \leq -b \longleftrightarrow a + b \leq 0$
 (is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *pprrt-0*[simp]: $\text{pprrt } 0 = 0$ $\langle \text{proof} \rangle$
lemma *nprrt-0*[simp]: $\text{nprrt } 0 = 0$ $\langle \text{proof} \rangle$

lemma *pprrt-eq-id* [simp, no-atp]: $0 \leq x \implies \text{pprrt } x = x$
 $\langle \text{proof} \rangle$

lemma *nprrt-eq-id* [simp, no-atp]: $x \leq 0 \implies \text{nprrt } x = x$
 $\langle \text{proof} \rangle$

lemma *pprrt-eq-0* [simp, no-atp]: $x \leq 0 \implies \text{pprrt } x = 0$
 $\langle \text{proof} \rangle$

lemma *nprrt-eq-0* [simp, no-atp]: $0 \leq x \implies \text{nprrt } x = 0$
 $\langle \text{proof} \rangle$

lemma *sup-0-imp-0*:
 assumes $\text{sup } a (-a) = 0$
 shows $a = 0$
 $\langle \text{proof} \rangle$

lemma *inf-0-imp-0*: $\text{inf } a (-a) = 0 \implies a = 0$
 $\langle \text{proof} \rangle$

lemma *inf-0-eq-0* [simp, no-atp]: $\text{inf } a (-a) = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *sup-0-eq-0* [simp, no-atp]: $\text{sup } a (-a) = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *zero-le-double-add-iff-zero-le-single-add* [simp]: $0 \leq a + a \longleftrightarrow 0 \leq a$
 (is ?lhs $\longleftrightarrow$?rhs)
 $\langle \text{proof} \rangle$

lemma *double-zero* [simp]: $a + a = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *zero-less-double-add-iff-zero-less-single-add* [simp]: $0 < a + a \longleftrightarrow 0 < a$
 $\langle \text{proof} \rangle$

lemma *double-add-le-zero-iff-single-add-le-zero* [simp]: $a + a \leq 0 \longleftrightarrow a \leq 0$
 $\langle \text{proof} \rangle$

lemma *double-add-less-zero-iff-single-less-zero* [simp]: $a + a < 0 \longleftrightarrow a < 0$
 ⟨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*: $0 \leq a \longleftrightarrow \text{nprt } a = 0$
 ⟨proof⟩

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow \text{pprt } a = 0$
 ⟨proof⟩

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow \text{pprt } a = a$
 ⟨proof⟩

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow \text{nprt } a = a$
 ⟨proof⟩

lemma *pprt-mono* [simp, no-atp]: $a \leq b \implies \text{pprt } a \leq \text{pprt } b$
 ⟨proof⟩

lemma *nprt-mono* [simp, no-atp]: $a \leq b \implies \text{nprt } a \leq \text{nprt } b$
 ⟨proof⟩

end

lemmas *add-sup-inf-distrib* =
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*: $|a| = \text{pprt } a - \text{nprt } a$
 ⟨proof⟩

subclass *ordered-ab-group-add-abs*
 ⟨proof⟩

end

lemma *sup-eq-if*:

fixes $a :: 'a::\{\text{lattice-ab-group-add}, \text{linorder}\}$
shows $\text{sup } a \ (-\ a) = (\text{if } a < 0 \text{ then } -\ a \text{ else } a)$
 $\langle \text{proof} \rangle$

lemma *abs-if-lattice*:

fixes $a :: 'a::\{\text{lattice-ab-group-add-abs}, \text{linorder}\}$
shows $|a| = (\text{if } a < 0 \text{ then } -\ a \text{ else } a)$
 $\langle \text{proof} \rangle$

lemma *estimate-by-abs*:

fixes $a\ b\ c :: 'a::\text{lattice-ab-group-add-abs}$
assumes $a + b \leq c$
shows $a \leq c + |b|$
 $\langle \text{proof} \rangle$

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
begin

subclass *semilattice-inf-ab-group-add* $\langle \text{proof} \rangle$

subclass *semilattice-sup-ab-group-add* $\langle \text{proof} \rangle$

end

lemma *abs-le-mult*:

fixes $a\ b :: 'a::\text{lattice-ring}$
shows $|a * b| \leq |a| * |b|$
 $\langle \text{proof} \rangle$

instance *lattice-ring* $\subseteq$ *ordered-ring-abs*

$\langle \text{proof} \rangle$

lemma *mult-le-prts*:

fixes $a\ b :: 'a::\text{lattice-ring}$
assumes $a1 \leq a$
and $a \leq a2$
and $b1 \leq b$
and $b \leq b2$
shows $a * b \leq$
 $\text{pprt } a2 * \text{pprt } b2 + \text{pprt } a1 * \text{nprt } b2 + \text{nprt } a2 * \text{pprt } b1 + \text{nprt } a1 * \text{nprt } b1$
 $\langle \text{proof} \rangle$

lemma *mult-ge-prts*:

fixes $a\ b :: 'a::\text{lattice-ring}$
assumes $a1 \leq a$
and $a \leq a2$
and $b1 \leq b$

```

    and  $b \leq b2$ 
  shows  $a * b \geq$ 
     $nprt\ a1 * ppert\ b2 + nprt\ a2 * nprt\ b2 + ppert\ a1 * ppert\ b1 + ppert\ a2 * nprt$ 
 $b1$ 
   $\langle proof \rangle$ 

```

```

instance int :: lattice-ring
   $\langle proof \rangle$ 

```

```

instance real :: lattice-ring
   $\langle proof \rangle$ 

```

```

end

```

42 Floating-Point Numbers

```

theory Float
imports Log-Nat Lattice-Algebras
begin

```

```

definition float =  $\{m * 2^{\text{powr } e} \mid (m :: \text{int}) (e :: \text{int}). \text{True}\}$ 

```

```

typedef float = float
  morphisms real-of-float float-of
   $\langle proof \rangle$ 

```

```

setup-lifting type-definition-float

```

```

declare real-of-float [code-unfold]

```

```

lemmas float-of-inject[simp]

```

```

declare [[coercion real-of-float :: float  $\Rightarrow$  real]]

```

```

lemma real-of-float-eq:  $f1 = f2 \longleftrightarrow \text{real-of-float } f1 = \text{real-of-float } f2$  for  $f1\ f2 ::$ 
float
   $\langle proof \rangle$ 

```

```

declare real-of-float-inverse[simp] float-of-inverse [simp]

```

```

declare real-of-float [simp]

```

42.1 Real operations preserving the representation as floating point number

```

lemma floatI:  $m * 2^{\text{powr } e} = x \implies x \in \text{float}$  for  $m\ e :: \text{int}$ 
   $\langle proof \rangle$ 

```

```

lemma zero-float[simp]:  $0 \in \text{float}$ 
   $\langle proof \rangle$ 

```

lemma *one-float[simp]*: $1 \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *numeral-float[simp]*: $\text{numeral } i \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *neg-numeral-float[simp]*: $-\text{numeral } i \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *real-of-int-float[simp]*: $\text{real-of-int } x \in \text{float}$ **for** $x :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *real-of-nat-float[simp]*: $\text{real } x \in \text{float}$ **for** $x :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *two-powr-int-float[simp]*: $2^{\text{powr } (\text{real-of-int } i)} \in \text{float}$ **for** $i :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *two-powr-nat-float[simp]*: $2^{\text{powr } (\text{real } i)} \in \text{float}$ **for** $i :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *two-powr-minus-int-float[simp]*: $2^{\text{powr } - (\text{real-of-int } i)} \in \text{float}$ **for** $i :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *two-powr-minus-nat-float[simp]*: $2^{\text{powr } - (\text{real } i)} \in \text{float}$ **for** $i :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *two-powr-numeral-float[simp]*: $2^{\text{powr } \text{numeral } i} \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *two-powr-neg-numeral-float[simp]*: $2^{\text{powr } - \text{numeral } i} \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *two-pow-float[simp]*: $2^n \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *plus-float[simp]*: $r \in \text{float} \implies p \in \text{float} \implies r + p \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *uminus-float[simp]*: $x \in \text{float} \implies -x \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *times-float[simp]*: $x \in \text{float} \implies y \in \text{float} \implies x * y \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *minus-float[simp]*: $x \in \text{float} \implies y \in \text{float} \implies x - y \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *abs-float[simp]*: $x \in \text{float} \implies |x| \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *sgn-of-float[simp]*: $x \in \text{float} \implies \text{sgn } x \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *div-power-2-float[simp]*: $x \in \text{float} \implies x / 2^d \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *div-power-2-int-float[simp]*: $x \in \text{float} \implies x / (2::\text{int})^d \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *div-numeral-Bit0-float[simp]*:
assumes $x / \text{numeral } n \in \text{float}$
shows $x / (\text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *div-neg-numeral-Bit0-float[simp]*:
assumes $x / \text{numeral } n \in \text{float}$
shows $x / (- \text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *power-float[simp]*:
assumes $a \in \text{float}$
shows $a ^ b \in \text{float}$
 $\langle \text{proof} \rangle$

lift-definition *Float* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{float}$ **is** $\lambda(m::\text{int}) (e::\text{int}). m * 2^{\text{powr } e}$
 $\langle \text{proof} \rangle$

declare *Float.rep-eq[simp]*

code-datatype *Float*

lemma *compute-real-of-float[code]*:
 $\text{real-of-float } (\text{Float } m \ e) = (\text{if } e \geq 0 \text{ then } m * 2^{\text{nat } e} \text{ else } m / 2^{\text{nat } (-e)})$
 $\langle \text{proof} \rangle$

42.2 Arithmetic operations on floating point numbers

instantiation *float* :: $\{\text{ring-1}, \text{linorder}, \text{linordered-ring}, \text{linordered-idom}, \text{numeral}, \text{equal}\}$
begin

lift-definition *zero-float* :: *float* **is** 0 $\langle \text{proof} \rangle$
declare *zero-float.rep-eq[simp]*

lift-definition *one-float* :: *float* **is** 1 $\langle \text{proof} \rangle$
declare *one-float.rep-eq[simp]*

lift-definition *plus-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *float* **is** (+) $\langle$ *proof* $\rangle$
declare *plus-float.rep-eq*[*simp*]

lift-definition *times-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *float* **is** (*) $\langle$ *proof* $\rangle$
declare *times-float.rep-eq*[*simp*]

lift-definition *minus-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *float* **is** (−) $\langle$ *proof* $\rangle$
declare *minus-float.rep-eq*[*simp*]

lift-definition *uminus-float* :: *float* $\Rightarrow$ *float* **is** *uminus* $\langle$ *proof* $\rangle$
declare *uminus-float.rep-eq*[*simp*]

lift-definition *abs-float* :: *float* $\Rightarrow$ *float* **is** *abs* $\langle$ *proof* $\rangle$
declare *abs-float.rep-eq*[*simp*]

lift-definition *sgn-float* :: *float* $\Rightarrow$ *float* **is** *sgn* $\langle$ *proof* $\rangle$
declare *sgn-float.rep-eq*[*simp*]

lift-definition *equal-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *bool* **is** (=) :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool* $\langle$ *proof* $\rangle$

lift-definition *less-eq-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *bool* **is** ($\leq$) $\langle$ *proof* $\rangle$
declare *less-eq-float.rep-eq*[*simp*]

lift-definition *less-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *bool* **is** (<) $\langle$ *proof* $\rangle$
declare *less-float.rep-eq*[*simp*]

instance
 $\langle$ *proof* $\rangle$

end

lemma *real-of-float* [*simp*]: *real-of-float* (*of-nat* *n*) = *of-nat* *n* $\langle$ *proof* $\rangle$

lemma *real-of-float-of-int-eq* [*simp*]: *real-of-float* (*of-int* *z*) = *of-int* *z* $\langle$ *proof* $\rangle$

lemma *Float-0-eq-0*[*simp*]: *Float* 0 *e* = 0 $\langle$ *proof* $\rangle$

lemma *real-of-float-power*[*simp*]: *real-of-float* ($f^{\wedge}n$) = *real-of-float* $f^{\wedge}n$ **for** *f* :: *float* $\langle$ *proof* $\rangle$

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*

$\langle \text{proof} \rangle$

instance *float* :: *unbounded-dense-linorder*
 $\langle \text{proof} \rangle$

instantiation *float* :: *lattice-ab-group-add*
begin

definition *inf-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *float*
where *inf-float* *a b* = *min a b*

definition *sup-float* :: *float* $\Rightarrow$ *float* $\Rightarrow$ *float*
where *sup-float* *a b* = *max a b*

instance
 $\langle \text{proof} \rangle$

end

lemma *float-numeral[simp]*: *real-of-float* (*numeral x* :: *float*) = *numeral x*
 $\langle \text{proof} \rangle$

lemma *transfer-numeral [transfer-rule]*:
rel-fun (=) *pcr-float* (*numeral* :: - $\Rightarrow$ *real*) (*numeral* :: - $\Rightarrow$ *float*)
 $\langle \text{proof} \rangle$

lemma *float-neg-numeral[simp]*: *real-of-float* ($-$ *numeral x* :: *float*) = $-$ *numeral x*
 $\langle \text{proof} \rangle$

lemma *transfer-neg-numeral [transfer-rule]*:
rel-fun (=) *pcr-float* ($-$ *numeral* :: - $\Rightarrow$ *real*) ($-$ *numeral* :: - $\Rightarrow$ *float*)
 $\langle \text{proof} \rangle$

lemma *float-of-numeral*: *numeral k* = *float-of* (*numeral k*)
and *float-of-neg-numeral*: $-$ *numeral k* = *float-of* ($-$ *numeral k*)
 $\langle \text{proof} \rangle$

42.3 Quickcheck

instantiation *float* :: *exhaustive*
begin

definition *exhaustive-float* **where**
exhaustive-float *f d* =
Quickcheck-Exhaustive.exhaustive ($\lambda x.$ *Quickcheck-Exhaustive.exhaustive* ($\lambda y.$ *f*
(*Float x y*)) *d*) *d*

instance $\langle \text{proof} \rangle$

end

context

includes *term-syntax*

begin

definition [*code-unfold*]:

valtermify-float $x\ y = \text{Code-Evaluation.valtermify Float } \{\cdot\}\ x\ \{\cdot\}\ y$

end

instantiation *float* :: *full-exhaustive*

begin

definition

full-exhaustive-float $f\ d =$

Quickcheck-Exhaustive.full-exhaustive

$(\lambda x. \text{Quickcheck-Exhaustive.full-exhaustive } (\lambda y. f\ (\text{valtermify-float } x\ y))\ d)\ d$

instance $\langle \text{proof} \rangle$

end

instantiation *float* :: *random*

begin

definition *Quickcheck-Random.random* $i =$

scomp (*Quickcheck-Random.random* $(2^{\wedge} \text{nat-of-natural } i)$)

$(\lambda \text{man. scomp } (\text{Quickcheck-Random.random } i)\ (\lambda \text{exp. Pair } (\text{valtermify-float } \text{man } \text{exp})))$

instance $\langle \text{proof} \rangle$

end

42.4 Represent floats as unique mantissa and exponent

lemma *int-induct-abs*[*case-names less*]:

fixes $j :: \text{int}$

assumes $H: \bigwedge n. (\bigwedge i. |i| < |n| \implies P\ i) \implies P\ n$

shows $P\ j$

$\langle \text{proof} \rangle$

lemma *int-cancel-factors*:

fixes $n :: \text{int}$

assumes $1 < r$

shows $n = 0 \vee (\exists k\ i. n = k * r^{\wedge} i \wedge \neg r\ \text{dvd } k)$

$\langle \text{proof} \rangle$

lemma *mult-powr-eq-mult-powr-iff-asym*:

fixes $m1\ m2\ e1\ e2 :: int$
assumes $m1: \neg 2\ dvd\ m1$
and $e1 \leq e2$
shows $m1 * 2\ powr\ e1 = m2 * 2\ powr\ e2 \longleftrightarrow m1 = m2 \wedge e1 = e2$
(is ?lhs $\longleftrightarrow$?rhs)
 $\langle proof \rangle$

lemma *mult-powr-eq-mult-powr-iff*:

$\neg 2\ dvd\ m1 \implies \neg 2\ dvd\ m2 \implies m1 * 2\ powr\ e1 = m2 * 2\ powr\ e2 \longleftrightarrow m1 = m2 \wedge e1 = e2$
for $m1\ m2\ e1\ e2 :: int$
 $\langle proof \rangle$

lemma *floatE-normed*:

assumes $x: x \in float$
obtains $(zero)\ x = 0$
 $| (powr)\ m\ e :: int$ **where** $x = m * 2\ powr\ e \neg 2\ dvd\ m\ x \neq 0$
 $\langle proof \rangle$

lemma *float-normed-cases*:

fixes $f :: float$
obtains $(zero)\ f = 0$
 $| (powr)\ m\ e :: int$ **where** $real-of-float\ f = m * 2\ powr\ e \neg 2\ dvd\ m\ f \neq 0$
 $\langle proof \rangle$

definition *mantissa* $:: float \Rightarrow int$

where $mantissa\ f =$
 $fst\ (SOME\ p::int \times int. (f = 0 \wedge fst\ p = 0 \wedge snd\ p = 0) \vee$
 $(f \neq 0 \wedge real-of-float\ f = real-of-int\ (fst\ p) * 2\ powr\ real-of-int\ (snd\ p) \wedge \neg$
 $2\ dvd\ fst\ p))$

definition *exponent* $:: float \Rightarrow int$

where $exponent\ f =$
 $snd\ (SOME\ p::int \times int. (f = 0 \wedge fst\ p = 0 \wedge snd\ p = 0) \vee$
 $(f \neq 0 \wedge real-of-float\ f = real-of-int\ (fst\ p) * 2\ powr\ real-of-int\ (snd\ p) \wedge \neg$
 $2\ dvd\ fst\ p))$

lemma *exponent-0[simp]*: $exponent\ 0 = 0$ **(is ?E)**

and *mantissa-0[simp]*: $mantissa\ 0 = 0$ **(is ?M)**
 $\langle proof \rangle$

lemma *mantissa-exponent*: $real-of-float\ f = mantissa\ f * 2\ powr\ exponent\ f$ **(is ?E)**

and *mantissa-not-dvd*: $f \neq 0 \implies \neg 2\ dvd\ mantissa\ f$ **(is - $\implies ?D$)**
 $\langle proof \rangle$

lemma *mantissa-noteq-0*: $f \neq 0 \implies mantissa\ f \neq 0$

<proof>

lemma *mantissa-eq-zero-iff*: $\text{mantissa } x = 0 \longleftrightarrow x = 0$
 (is ?lhs $\longleftrightarrow$?rhs)
<proof>

lemma *mantissa-pos-iff*: $0 < \text{mantissa } x \longleftrightarrow 0 < x$
<proof>

lemma *mantissa-nonneg-iff*: $0 \leq \text{mantissa } x \longleftrightarrow 0 \leq x$
<proof>

lemma *mantissa-neg-iff*: $0 > \text{mantissa } x \longleftrightarrow 0 > x$
<proof>

lemma
 fixes $m \ e :: \text{int}$
 defines $f \equiv \text{float-of } (m * 2^{\text{powr } e})$
 assumes *dvd*: $\neg 2 \text{ dvd } m$
 shows *mantissa-float*: $\text{mantissa } f = m$ (is ?M)
 and *exponent-float*: $m \neq 0 \implies \text{exponent } f = e$ (is - $\implies$?E)
<proof>

42.5 Compute arithmetic operations

lemma *Float-mantissa-exponent*: $\text{Float } (\text{mantissa } f) (\text{exponent } f) = f$
<proof>

lemma *Float-cases* [*cases type: float*]:
 fixes $f :: \text{float}$
 obtains $(\text{Float}) \ m \ e :: \text{int}$ **where** $f = \text{Float } m \ e$
<proof>

lemma *denormalize-shift*:
 assumes *f-def*: $f = \text{Float } m \ e$
 and *not-0*: $f \neq 0$
 obtains i **where** $m = \text{mantissa } f * 2^i$ $e = \text{exponent } f - i$
<proof>

context
begin

qualified lemma *compute-float-zero*[*code-unfold, code*]: $0 = \text{Float } 0 \ 0$
<proof> **lemma** *compute-float-one*[*code-unfold, code*]: $1 = \text{Float } 1 \ 0$
<proof>

lift-definition *normfloat* :: $\text{float} \Rightarrow \text{float}$ **is** $\lambda x. x$ *<proof>*
lemma *normfloat-id*[*simp*]: $\text{normfloat } x = x$ *<proof>* **lemma** *compute-normfloat*[*code*]:
 $\text{normfloat } (\text{Float } m \ e) =$

(if $m \bmod 2 = 0 \wedge m \neq 0$ then $\text{normfloat } (\text{Float } (m \text{ div } 2) (e + 1))$
 else if $m = 0$ then 0 else $\text{Float } m \ e$)
 ⟨proof⟩ **lemma** *compute-float-numeral*[code-abbrev]: $\text{Float } (\text{numeral } k) \ 0 = \text{numeral } k$
 ⟨proof⟩ **lemma** *compute-float-neg-numeral*[code-abbrev]: $\text{Float } (- \text{numeral } k) \ 0 = - \text{numeral } k$
 ⟨proof⟩ **lemma** *compute-float-uminus*[code]: $- \text{Float } m1 \ e1 = \text{Float } (- m1) \ e1$
 ⟨proof⟩ **lemma** *compute-float-times*[code]: $\text{Float } m1 \ e1 * \text{Float } m2 \ e2 = \text{Float } (m1 * m2) (e1 + e2)$
 ⟨proof⟩ **lemma** *compute-float-plus*[code]:
 $\text{Float } m1 \ e1 + \text{Float } m2 \ e2 =$
 (if $m1 = 0$ then $\text{Float } m2 \ e2$
 else if $m2 = 0$ then $\text{Float } m1 \ e1$
 else if $e1 \leq e2$ then $\text{Float } (m1 + m2 * 2^{\text{nat } (e2 - e1)}) \ e1$
 else $\text{Float } (m2 + m1 * 2^{\text{nat } (e1 - e2)}) \ e2$)
 ⟨proof⟩ **lemma** *compute-float-minus*[code]: $f - g = f + (-g)$ **for** $f \ g :: \text{float}$
 ⟨proof⟩ **lemma** *compute-float-sgn*[code]:
 $\text{sgn } (\text{Float } m1 \ e1) = (\text{if } 0 < m1 \text{ then } 1 \text{ else if } m1 < 0 \text{ then } -1 \text{ else } 0)$
 ⟨proof⟩

lift-definition *is-float-pos* :: $\text{float} \Rightarrow \text{bool}$ **is** $(<) \ 0 :: \text{real} \Rightarrow \text{bool}$ ⟨proof⟩ **lemma**
compute-is-float-pos[code]: $\text{is-float-pos } (\text{Float } m \ e) \longleftrightarrow 0 < m$
 ⟨proof⟩

lift-definition *is-float-nonneg* :: $\text{float} \Rightarrow \text{bool}$ **is** $(\leq) \ 0 :: \text{real} \Rightarrow \text{bool}$ ⟨proof⟩ **lemma**
compute-is-float-nonneg[code]: $\text{is-float-nonneg } (\text{Float } m \ e) \longleftrightarrow 0 \leq m$
 ⟨proof⟩

lift-definition *is-float-zero* :: $\text{float} \Rightarrow \text{bool}$ **is** $(=) \ 0 :: \text{real} \Rightarrow \text{bool}$ ⟨proof⟩ **lemma**
compute-is-float-zero[code]: $\text{is-float-zero } (\text{Float } m \ e) \longleftrightarrow 0 = m$
 ⟨proof⟩ **lemma** *compute-float-abs*[code]: $|\text{Float } m \ e| = \text{Float } |m| \ e$
 ⟨proof⟩ **lemma** *compute-float-eq*[code]: $\text{equal-class.equal } f \ g = \text{is-float-zero } (f - g)$
 ⟨proof⟩

end

42.6 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]

42.7 Rounding Real Numbers

definition *round-down* :: *int* $\Rightarrow$ *real* $\Rightarrow$ *real*
where *round-down* *prec* *x* = $\lfloor x * 2^{\text{powr } \text{prec}} \rfloor * 2^{\text{powr } -\text{prec}}$

definition *round-up* :: *int* $\Rightarrow$ *real* $\Rightarrow$ *real*
where *round-up* *prec* *x* = $\lceil x * 2^{\text{powr } \text{prec}} \rceil * 2^{\text{powr } -\text{prec}}$

lemma *round-down-float[simp]*: *round-down* *prec* *x* $\in$ *float*
 $\langle \text{proof} \rangle$

lemma *round-up-float[simp]*: *round-up* *prec* *x* $\in$ *float*
 $\langle \text{proof} \rangle$

lemma *round-up*: $x \leq \text{round-up } \text{prec } x$
 $\langle \text{proof} \rangle$

lemma *round-down*: *round-down* *prec* *x* $\leq x$
 $\langle \text{proof} \rangle$

lemma *round-up-0[simp]*: *round-up* *p* 0 = 0
 $\langle \text{proof} \rangle$

lemma *round-down-0[simp]*: *round-down* *p* 0 = 0
 $\langle \text{proof} \rangle$

lemma *round-up-diff-round-down*: *round-up* *prec* *x* - *round-down* *prec* *x* $\leq 2^{\text{powr } -\text{prec}}$
 $\langle \text{proof} \rangle$

lemma *round-down-shift*: *round-down* *p* ($x * 2^{\text{powr } k}$) = $2^{\text{powr } k} * \text{round-down } (p + k) x$
 $\langle \text{proof} \rangle$

lemma *round-up-shift*: *round-up* *p* ($x * 2^{\text{powr } k}$) = $2^{\text{powr } k} * \text{round-up } (p + k) x$
 $\langle \text{proof} \rangle$

lemma *round-up-uminus-eq*: *round-up* *p* ($-x$) = $- \text{round-down } p x$
and *round-down-uminus-eq*: *round-down* *p* ($-x$) = $- \text{round-up } p x$
 $\langle \text{proof} \rangle$

lemma *round-up-mono*: $x \leq y \implies \text{round-up } p x \leq \text{round-up } p y$
 $\langle \text{proof} \rangle$

lemma *round-up-le1*:
assumes $x \leq 1$ $\text{prec} \geq 0$
shows *round-up* *prec* *x* ≤ 1
 $\langle \text{proof} \rangle$

lemma *round-up-less1*:
 assumes $x < 1 \mid 2 \mid p > 0$
 shows *round-up* $p \ x < 1$
 $\langle \text{proof} \rangle$

lemma *round-down-ge1*:
 assumes $x: x \geq 1$
 assumes *prec*: $p \geq -\log 2 \ x$
 shows $1 \leq \text{round-down } p \ x$
 $\langle \text{proof} \rangle$

lemma *round-up-le0*: $x \leq 0 \implies \text{round-up } p \ x \leq 0$
 $\langle \text{proof} \rangle$

42.8 Rounding Floats

definition *div-twoPow* :: $\text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$
 where [*simp*]: $\text{div-twoPow } x \ n = x \text{ div } (2 \wedge n)$

definition *mod-twoPow* :: $\text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$
 where [*simp*]: $\text{mod-twoPow } x \ n = x \text{ mod } (2 \wedge n)$

lemma *compute-div-twoPow*[*code*]:
 $\text{div-twoPow } x \ n = (\text{if } x = 0 \vee x = -1 \vee n = 0 \text{ then } x \text{ else } \text{div-twoPow } (x \text{ div } 2) \ (n - 1))$
 $\langle \text{proof} \rangle$

lemma *compute-mod-twoPow*[*code*]:
 $\text{mod-twoPow } x \ n = (\text{if } n = 0 \text{ then } 0 \text{ else } x \text{ mod } 2 + 2 * \text{mod-twoPow } (x \text{ div } 2) \ (n - 1))$
 $\langle \text{proof} \rangle$

lift-definition *float-up* :: $\text{int} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *round-up* $\langle \text{proof} \rangle$
declare *float-up.rep-eq*[*simp*]

lemma *round-up-correct*: $\text{round-up } e \ f - f \in \{0..2 \text{ powr } -e\}$
 $\langle \text{proof} \rangle$

lemma *float-up-correct*: $\text{real-of-float } (\text{float-up } e \ f) - \text{real-of-float } f \in \{0..2 \text{ powr } -e\}$
 $\langle \text{proof} \rangle$

lift-definition *float-down* :: $\text{int} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *round-down* $\langle \text{proof} \rangle$
declare *float-down.rep-eq*[*simp*]

lemma *round-down-correct*: $f - (\text{round-down } e \ f) \in \{0..2 \text{ powr } -e\}$
 $\langle \text{proof} \rangle$

lemma *float-down-correct*: $\text{real-of-float } f - \text{real-of-float } (\text{float-down } e \ f) \in \{0..2 \text{ powr } -e\}$

$\text{powr } -e\}$
 $\langle \text{proof} \rangle$

context
begin

qualified lemma *compute-float-down*[code]:
 $\text{float-down } p \text{ (Float } m \text{ } e) =$
 $(\text{if } p + e < 0 \text{ then Float (div-twopow } m \text{ (nat } (-(p + e)))) (-p) \text{ else Float } m \text{ } e)$
 $\langle \text{proof} \rangle$

lemma *abs-round-down-le*: $|f - (\text{round-down } e \text{ } f)| \leq 2 \text{ powr } -e$
 $\langle \text{proof} \rangle$

lemma *abs-round-up-le*: $|f - (\text{round-up } e \text{ } f)| \leq 2 \text{ powr } -e$
 $\langle \text{proof} \rangle$

lemma *round-down-nonneg*: $0 \leq s \implies 0 \leq \text{round-down } p \text{ } s$
 $\langle \text{proof} \rangle$

lemma *ceil-divide-floor-conv*:
assumes $b \neq 0$
shows $\lceil \text{real-of-int } a / \text{real-of-int } b \rceil =$
 $(\text{if } b \text{ dvd } a \text{ then } a \text{ div } b \text{ else } \lfloor \text{real-of-int } a / \text{real-of-int } b \rfloor + 1)$
 $\langle \text{proof} \rangle$ **lemma** *compute-float-up*[code]: $\text{float-up } p \text{ } x = - \text{float-down } p \text{ } (-x)$
 $\langle \text{proof} \rangle$

end

lemma *bitlen-Float*:
fixes $m \text{ } e$
defines $[\text{THEN meta-eq-to-obj-eq}]: f \equiv \text{Float } m \text{ } e$
shows $\text{bitlen } |\text{mantissa } f| + \text{exponent } f = (\text{if } m = 0 \text{ then } 0 \text{ else bitlen } |m| + e)$
 $\langle \text{proof} \rangle$

lemma *float-gt1-scale*:
assumes $1 \leq \text{Float } m \text{ } e$
shows $0 \leq e + (\text{bitlen } m - 1)$
 $\langle \text{proof} \rangle$

42.9 Truncating Real Numbers

definition *truncate-down::nat $\Rightarrow$ real $\Rightarrow$ real*
where $\text{truncate-down } \text{prec } x = \text{round-down } (\text{prec} - \lfloor \log 2 |x| \rfloor) \text{ } x$

lemma *truncate-down*: $\text{truncate-down } \text{prec } x \leq x$
 $\langle \text{proof} \rangle$

lemma *truncate-down-le*: $x \leq y \implies \text{truncate-down } \text{prec } x \leq y$
 ⟨proof⟩

lemma *truncate-down-zero[simp]*: $\text{truncate-down } \text{prec } 0 = 0$
 ⟨proof⟩

lemma *truncate-down-float[simp]*: $\text{truncate-down } p \ x \in \text{float}$
 ⟨proof⟩

definition *truncate-up::nat $\Rightarrow$ real $\Rightarrow$ real*
 where $\text{truncate-up } \text{prec } x = \text{round-up } (\text{prec} - \lfloor \log 2 \ |x| \rfloor) \ x$

lemma *truncate-up*: $x \leq \text{truncate-up } \text{prec } x$
 ⟨proof⟩

lemma *truncate-up-le*: $x \leq y \implies x \leq \text{truncate-up } \text{prec } y$
 ⟨proof⟩

lemma *truncate-up-zero[simp]*: $\text{truncate-up } \text{prec } 0 = 0$
 ⟨proof⟩

lemma *truncate-up-uminus-eq*: $\text{truncate-up } \text{prec } (-x) = - \text{truncate-down } \text{prec } x$
 and *truncate-down-uminus-eq*: $\text{truncate-down } \text{prec } (-x) = - \text{truncate-up } \text{prec } x$
 ⟨proof⟩

lemma *truncate-up-float[simp]*: $\text{truncate-up } p \ x \in \text{float}$
 ⟨proof⟩

lemma *mult-powr-eq*: $0 < b \implies b \neq 1 \implies 0 < x \implies x * b^{\text{powr } y} = b^{\text{powr } (y + \log b \ x)}$
 ⟨proof⟩

lemma *truncate-down-pos*:
 assumes $x > 0$
 shows $\text{truncate-down } p \ x > 0$
 ⟨proof⟩

lemma *truncate-down-nonneg*: $0 \leq y \implies 0 \leq \text{truncate-down } \text{prec } y$
 ⟨proof⟩

lemma *truncate-down-ge1*: $1 \leq x \implies 1 \leq \text{truncate-down } p \ x$
 ⟨proof⟩

lemma *truncate-up-nonpos*: $x \leq 0 \implies \text{truncate-up } \text{prec } x \leq 0$
 ⟨proof⟩

lemma *truncate-up-le1*:
 assumes $x \leq 1$
 shows $\text{truncate-up } p \ x \leq 1$

$\langle \text{proof} \rangle$

lemma *truncate-down-shift-int*:

truncate-down p ($x * 2^{\text{powr real-of-int } k}$) = *truncate-down* p $x * 2^{\text{powr } k}$

$\langle \text{proof} \rangle$

lemma *truncate-down-shift-nat*: *truncate-down* p ($x * 2^{\text{powr real } k}$) = *truncate-down* p $x * 2^{\text{powr } k}$

$\langle \text{proof} \rangle$

lemma *truncate-up-shift-int*: *truncate-up* p ($x * 2^{\text{powr real-of-int } k}$) = *truncate-up* p $x * 2^{\text{powr } k}$

$\langle \text{proof} \rangle$

lemma *truncate-up-shift-nat*: *truncate-up* p ($x * 2^{\text{powr real } k}$) = *truncate-up* p $x * 2^{\text{powr } k}$

$\langle \text{proof} \rangle$

42.10 Truncating Floats

lift-definition *float-round-up* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *truncate-up*

$\langle \text{proof} \rangle$

lemma *float-round-up*: *real-of-float* $x \leq \text{real-of-float}$ (*float-round-up* *prec* x)

$\langle \text{proof} \rangle$

lemma *float-round-up-zero[simp]*: *float-round-up* *prec* 0 = 0

$\langle \text{proof} \rangle$

lift-definition *float-round-down* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *truncate-down*

$\langle \text{proof} \rangle$

lemma *float-round-down*: *real-of-float* (*float-round-down* *prec* x) $\leq \text{real-of-float } x$

$\langle \text{proof} \rangle$

lemma *float-round-down-zero[simp]*: *float-round-down* *prec* 0 = 0

$\langle \text{proof} \rangle$

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*: $- \text{float-round-up } \text{prec } x = \text{float-round-down } \text{prec } (- x)$

and *minus-float-round-down-eq*: $- \text{float-round-down } \text{prec } x = \text{float-round-up } \text{prec } (- x)$

$\langle \text{proof} \rangle$

context

begin

qualified lemma *compute-float-round-down*[code]:
 $\text{float-round-down } \text{prec } (\text{Float } m \ e) =$
 (let $d = \text{bitlen } |m| - \text{int } \text{prec} - 1$ in
 if $0 < d$ then $\text{Float } (\text{div-two } m \ (\text{nat } d)) \ (e + d)$
 else $\text{Float } m \ e$)
 ⟨proof⟩ **lemma** *compute-float-round-up*[code]:
 $\text{float-round-up } \text{prec } x = - \text{float-round-down } \text{prec } (-x)$
 ⟨proof⟩

end

lemma *truncate-up-nonneg-mono*:
 assumes $0 \leq x \ x \leq y$
 shows $\text{truncate-up } \text{prec } x \leq \text{truncate-up } \text{prec } y$
 ⟨proof⟩

lemma *truncate-up-switch-sign-mono*:
 assumes $x \leq 0 \ 0 \leq y$
 shows $\text{truncate-up } \text{prec } x \leq \text{truncate-up } \text{prec } y$
 ⟨proof⟩

lemma *truncate-down-switch-sign-mono*:
 assumes $x \leq 0$
 and $0 \leq y$
 and $x \leq y$
 shows $\text{truncate-down } \text{prec } x \leq \text{truncate-down } \text{prec } y$
 ⟨proof⟩

lemma *truncate-down-nonneg-mono*:
 assumes $0 \leq x \ x \leq y$
 shows $\text{truncate-down } \text{prec } x \leq \text{truncate-down } \text{prec } y$
 ⟨proof⟩

lemma *truncate-down-eq-truncate-up*: $\text{truncate-down } p \ x = - \text{truncate-up } p \ (-x)$
 and *truncate-up-eq-truncate-down*: $\text{truncate-up } p \ x = - \text{truncate-down } p \ (-x)$
 ⟨proof⟩

lemma *truncate-down-mono*: $x \leq y \implies \text{truncate-down } p \ x \leq \text{truncate-down } p \ y$
 ⟨proof⟩

lemma *truncate-up-mono*: $x \leq y \implies \text{truncate-up } p \ x \leq \text{truncate-up } p \ y$
 ⟨proof⟩

lemma *truncate-up-nonneg*: $0 \leq \text{truncate-up } p \ x$ if $0 \leq x$
 ⟨proof⟩

lemma *truncate-up-pos*: $0 < \text{truncate-up } p \ x$ if $0 < x$
 ⟨proof⟩

lemma *truncate-up-less-zero-iff*[simp]: *truncate-up p x < 0 $\longleftrightarrow$ x < 0*
 ⟨proof⟩

lemma *truncate-up-nonneg-iff*[simp]: *truncate-up p x $\geq$ 0 $\longleftrightarrow$ x $\geq$ 0*
 ⟨proof⟩

lemma *truncate-down-less-zero-iff*[simp]: *truncate-down p x < 0 $\longleftrightarrow$ x < 0*
 ⟨proof⟩

lemma *truncate-down-nonneg-iff*[simp]: *truncate-down p x $\geq$ 0 $\longleftrightarrow$ x $\geq$ 0*
 ⟨proof⟩

lemma *truncate-down-eq-zero-iff*[simp]: *truncate-down prec x = 0 $\longleftrightarrow$ x = 0*
 ⟨proof⟩

lemma *truncate-up-eq-zero-iff*[simp]: *truncate-up prec x = 0 $\longleftrightarrow$ x = 0*
 ⟨proof⟩

42.11 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 $\lfloor$ a / b $\rfloor$ 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 $\lfloor \log p (i / j) \rfloor = \text{floor } (\log p i) - \text{floor } (\log p j) -$
 (if $i \geq j * p^{\text{power } (\text{floor } (\log p i) - \text{floor } (\log p j))}$ 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 $\Rightarrow$ nat $\Rightarrow$ nat $\Rightarrow$ float
is $\lambda \text{prec } (x::\text{nat}) (y::\text{nat}). \text{truncate-down prec } (x / y)$
 ⟨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**

λprec (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

42.12 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* $\Rightarrow$ *float* $\Rightarrow$ *float* $\Rightarrow$ *float* **is** *real-divr*

<proof> **lemma** *compute-float-divr*[*code*]:

float-divr prec x y = - *float-divl prec (-x) y*

<proof>

end

42.13 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* $\Rightarrow$ *float* $\Rightarrow$ *float* $\Rightarrow$ *float* **is** *plus-down* *<proof>*

lift-definition *float-plus-up* :: *nat* $\Rightarrow$ *float* $\Rightarrow$ *float* $\Rightarrow$ *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 $\lfloor x * 2^{\text{powr } (p - \lfloor \log 2 |x| \rfloor)} \rfloor = \lfloor y * 2^{\text{powr } (p - \lfloor \log 2 |x| \rfloor)} \rfloor$

shows *truncate-down p x* = *truncate-down p y*

<proof>

lemma *sum-neq-zeroI*:

$|a| \geq k \implies |b| < k \implies a + b \neq 0$

$|a| > k \implies |b| \leq k \implies a + b \neq 0$

for $a\ k :: \text{real}$

$\langle \text{proof} \rangle$

lemma *abs-real-le-2-powr-bitlen[simp]*: $| \text{real-of-int } m2 | < 2^{\text{powr } \text{real-of-int } (\text{bitlen } |m2|)}$

$\langle \text{proof} \rangle$

lemma *floor-sum-times-2-powr-sgn-eq*:

fixes $ai\ p\ q :: \text{int}$

and $a\ b :: \text{real}$

assumes $a * 2^{\text{powr } p} = ai$

and $b\text{-le-1}$: $|b * 2^{\text{powr } (p + 1)}| \leq 1$

and $leqp$: $q \leq p$

shows $\lfloor (a + b) * 2^{\text{powr } q} \rfloor = \lfloor (2 * ai + \text{sgn } b) * 2^{\text{powr } (q - p - 1)} \rfloor$

$\langle \text{proof} \rangle$

lemma *log2-abs-int-add-less-half-sgn-eq*:

fixes $ai :: \text{int}$

and $b :: \text{real}$

assumes $|b| \leq 1/2$

and $ai \neq 0$

shows $\lfloor \log 2\ |\text{real-of-int } ai + b| \rfloor = \lfloor \log 2\ |ai + \text{sgn } b / 2| \rfloor$

$\langle \text{proof} \rangle$

context

begin

qualified lemma *compute-far-float-plus-down*:

fixes $m1\ e1\ m2\ e2 :: \text{int}$

and $p :: \text{nat}$

defines $k1 \equiv \text{Suc } p - \text{nat } (\text{bitlen } |m1|)$

assumes H : $\text{bitlen } |m2| \leq e1 - e2 - k1 - 2$ $m1 \neq 0$ $m2 \neq 0$ $e1 \geq e2$

shows $\text{float-plus-down } p\ (\text{Float } m1\ e1)\ (\text{Float } m2\ e2) =$

$\text{float-round-down } p\ (\text{Float } (m1 * 2^{\wedge} (\text{Suc } (\text{Suc } k1)) + \text{sgn } m2)\ (e1 - \text{int } k1 - 2))$

$\langle \text{proof} \rangle$

lemma *compute-float-plus-down-naive[code]*: $\text{float-plus-down } p\ x\ y = \text{float-round-down } p\ (x + y)$

$\langle \text{proof} \rangle$ **lemma** *compute-float-plus-down[code]*:

fixes $p :: \text{nat}$ **and** $m1\ e1\ m2\ e2 :: \text{int}$

shows $\text{float-plus-down } p\ (\text{Float } m1\ e1)\ (\text{Float } m2\ e2) =$

$(\text{if } m1 = 0 \text{ then } \text{float-round-down } p\ (\text{Float } m2\ e2)$

$\text{else if } m2 = 0 \text{ then } \text{float-round-down } p\ (\text{Float } m1\ e1)$

else

$(\text{if } e1 \geq e2 \text{ then}$

```

    (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 ⟷ is-float-pos (float-plus-down
0 b (- a))
⟨proof⟩ lemma compute-float-le[code]: a ≤ b ⟷ 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 ≤ c + d
⟨proof⟩

```

```

lemma plus-up-mono: plus-up p a b ≤ plus-up p c d if a + b ≤ c + d
⟨proof⟩

```

42.14 Approximate Multiplication

```

lemma mult-mono-nonpos-nonneg: a * b ≤ c * d
if a ≤ c a ≤ 0 0 ≤ d d ≤ b for a b c d::'a::ordered-ring
⟨proof⟩

```

```

lemma mult-mono-nonneg-nonpos: b * a ≤ d * c
if a ≤ c c ≤ 0 0 ≤ d d ≤ b for a b c d::'a::ordered-ring
⟨proof⟩

```

```

lemma mult-mono-nonpos-nonpos: a * b ≤ c * d
if a ≥ c a ≤ 0 b ≥ d d ≤ 0 for a b c d::real
⟨proof⟩

```

lemma *mult-float-mono1*:

```

notes mono-rules = plus-down-mono add-mono nprt-mono nprt-le-zero zero-le-pprt
pprt-mono

```

```

shows a ≤ b ⟹ ab ≤ bb ⟹
  aa ≤ a ⟹
  b ≤ ba ⟹
  ac ≤ ab ⟹
  bb ≤ bc ⟹
  plus-down prec (nprt aa * ppert bc)
  (plus-down prec (nprt ba * npert bc)

```


$$\begin{aligned}
& (plus\text{-}down\ prec\ (pprt\ aa * pprt\ ac) \\
& \quad (pprt\ ba * npert\ ac))) \\
\leq & plus\text{-}down\ prec\ (npert\ a * pprt\ bb) \\
& (plus\text{-}down\ prec\ (npert\ b * npert\ bb) \\
& \quad (plus\text{-}down\ prec\ (pprt\ a * pprt\ ab) \\
& \quad \quad (pprt\ b * npert\ ab))) \\
\langle proof \rangle
\end{aligned}$$

lemma *mult-float-mono2*:

notes *mono-rules* = *plus-up-mono add-mono npert-mono npert-le-zero zero-le-pprt pprt-mono*

shows $a \leq b \implies$
 $ab \leq bb \implies$
 $aa \leq a \implies$
 $b \leq ba \implies$
 $ac \leq ab \implies$
 $bb \leq bc \implies$
 $plus\text{-}up\ prec\ (pprt\ b * pprt\ bb)$
 $(plus\text{-}up\ prec\ (pprt\ a * npert\ bb))$
 $(plus\text{-}up\ prec\ (npert\ b * pprt\ ab)$
 $\quad (npert\ a * npert\ ab)))$
 $\leq plus\text{-}up\ prec\ (pprt\ ba * pprt\ bc)$
 $(plus\text{-}up\ prec\ (pprt\ aa * npert\ bc)$
 $\quad (plus\text{-}up\ prec\ (npert\ ba * pprt\ ac)$
 $\quad \quad (npert\ aa * npert\ ac)))$
 $\langle proof \rangle$

42.15 Approximate Power

lemma *div2-less-self*[*termination-simp*]: $odd\ n \implies n\ div\ 2 < n$ **for** $n :: nat$
 $\langle proof \rangle$

fun *power-down* :: $nat \Rightarrow real \Rightarrow nat \Rightarrow real$

where

$power\text{-}down\ p\ x\ 0 = 1$
 $| power\text{-}down\ p\ x\ (Suc\ n) =$
 $\quad (if\ odd\ n\ then\ truncate\text{-}down\ (Suc\ p)\ ((power\text{-}down\ p\ x\ (Suc\ n\ div\ 2))^2)$
 $\quad \quad else\ truncate\text{-}down\ (Suc\ p)\ (x * power\text{-}down\ p\ x\ n))$

fun *power-up* :: $nat \Rightarrow real \Rightarrow nat \Rightarrow real$

where

$power\text{-}up\ p\ x\ 0 = 1$
 $| power\text{-}up\ p\ x\ (Suc\ n) =$
 $\quad (if\ odd\ n\ then\ truncate\text{-}up\ p\ ((power\text{-}up\ p\ x\ (Suc\ n\ div\ 2))^2)$
 $\quad \quad else\ truncate\text{-}up\ p\ (x * power\text{-}up\ p\ x\ n))$

lift-definition *power-up-fl* :: $nat \Rightarrow float \Rightarrow nat \Rightarrow float$ **is** *power-up*
 $\langle proof \rangle$

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*)) ($\frown$) ($\frown$)
 ⟨*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* $\implies$ 0 < *power-down* *p* *x* *n*
 ⟨*proof*⟩

lemma *power-down-nonneg*: 0 $\leq$ *x* $\implies$ 0 $\leq$ *power-down* *p* *x* *n*
 ⟨*proof*⟩

lemma *power-down*: 0 $\leq$ *x* $\implies$ *power-down* *p* *x* *n* $\leq$ *x* ^{*n*}
 ⟨*proof*⟩

lemma *power-up*: 0 $\leq$ *x* $\implies$ *x* ^{*n*} $\leq$ *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 $\leq$ *x* $\implies$ *power-down-fl* *p* *x* *n* $\leq$ *x* ^{*n*}
 ⟨*proof*⟩

lemma *power-up-fl*: 0 $\leq$ *x* $\implies$ *x* ^{*n*} $\leq$ *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*: $\text{even } n \implies 0 \leq \text{power-down } p \ x \ n$
 ⟨proof⟩

lemma *power-down-eq-zero-iff*[*simp*]: $\text{power-down } \text{prec } b \ n = 0 \longleftrightarrow b = 0 \wedge n \neq 0$
 ⟨proof⟩

lemma *power-down-nonneg-iff*[*simp*]:
 $\text{power-down } \text{prec } b \ n \geq 0 \longleftrightarrow \text{even } n \vee b \geq 0$
 ⟨proof⟩

lemma *power-down-neg-iff*[*simp*]:
 $\text{power-down } \text{prec } b \ n < 0 \longleftrightarrow$
 $b < 0 \wedge \text{odd } n$
 ⟨proof⟩

lemma *power-down-nonpos-iff*[*simp*]:
notes [*simp del*] = *power-down-neg-iff power-down-eq-zero-iff*
shows $\text{power-down } \text{prec } b \ n \leq 0 \longleftrightarrow b < 0 \wedge \text{odd } n \vee b = 0 \wedge n \neq 0$
 ⟨proof⟩

lemma *power-down-mono*:
 $\text{power-down } \text{prec } a \ n \leq \text{power-down } \text{prec } b \ n$
if $((0 \leq a \wedge a \leq b) \vee (\text{odd } n \wedge a \leq b) \vee (\text{even } n \wedge a \leq 0 \wedge b \leq a))$
 ⟨proof⟩

lemma *power-up-even-nonneg*: $\text{even } n \implies 0 \leq \text{power-up } p \ x \ n$
 ⟨proof⟩

lemma *power-up-eq-zero-iff*[*simp*]: $\text{power-up } \text{prec } b \ n = 0 \longleftrightarrow b = 0 \wedge n \neq 0$
 ⟨proof⟩

lemma *power-up-nonneg-iff*[*simp*]:
 $\text{power-up } \text{prec } b \ n \geq 0 \longleftrightarrow \text{even } n \vee b \geq 0$
 ⟨proof⟩

lemma *power-up-neg-iff*[*simp*]:
 $\text{power-up } \text{prec } b \ n < 0 \longleftrightarrow b < 0 \wedge \text{odd } n$
 ⟨proof⟩

lemma *power-up-nonpos-iff*[*simp*]:
notes [*simp del*] = *power-up-neg-iff power-up-eq-zero-iff*
shows $\text{power-up } \text{prec } b \ n \leq 0 \longleftrightarrow b < 0 \wedge \text{odd } n \vee b = 0 \wedge n \neq 0$
 ⟨proof⟩

lemma *power-up-mono*:

power-up prec a n $\leq$ *power-up prec b n*
if $((0 \leq a \wedge a \leq b) \vee (\text{odd } n \wedge a \leq b) \vee (\text{even } n \wedge a \leq 0 \wedge b \leq a))$
 $\langle \text{proof} \rangle$

42.16 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
 $\langle \text{proof} \rangle$

lemma *real-of-Float-int[simp]*: *real-of-float (Float n 0)* = *real n*
 $\langle \text{proof} \rangle$

lemma *float-zero[simp]*: *real-of-float (Float 0 e)* = 0
 $\langle \text{proof} \rangle$

lemma *abs-div-2-less*: $a \neq 0 \implies a \neq -1 \implies |(a::\text{int}) \text{ div } 2| < |a|$
 $\langle \text{proof} \rangle$

lemma *lapprox-rat*: *real-of-float (lapprox-rat prec x y)* $\leq$ *real-of-int x / real-of-int y*
 $\langle \text{proof} \rangle$

lemma *mult-div-le*:

fixes $a \ b :: \text{int}$
assumes $b > 0$
shows $a \geq b * (a \text{ div } b)$
 $\langle \text{proof} \rangle$

lemma *lapprox-rat-nonneg*:

assumes $0 \leq x$ **and** $0 \leq y$
shows $0 \leq \text{real-of-float (lapprox-rat } n \ x \ y)$
 $\langle \text{proof} \rangle$

lemma *rapprox-rat*: *real-of-int x / real-of-int y* $\leq$ *real-of-float (rapprox-rat prec x y)*
 $\langle \text{proof} \rangle$

lemma *rapprox-rat-le1*:

assumes $0 \leq x$ $0 < y$ $x \leq y$

shows *real-of-float* (*rapprox-rat* *n* *x* *y*) ≤ 1
 ⟨*proof*⟩

lemma *rapprox-rat-nonneg-nonpos*: $0 \leq x \implies y \leq 0 \implies \text{real-of-float } (\text{rapprox-rat } n \ x \ y) \leq 0$
 ⟨*proof*⟩

lemma *rapprox-rat-nonpos-nonneg*: $x \leq 0 \implies 0 \leq y \implies \text{real-of-float } (\text{rapprox-rat } n \ x \ y) \leq 0$
 ⟨*proof*⟩

lemma *real-divl*: $\text{real-divl } \text{prec } x \ y \leq x / y$
 ⟨*proof*⟩

lemma *real-divr*: $x / y \leq \text{real-divr } \text{prec } x \ y$
 ⟨*proof*⟩

lemma *float-divl*: $\text{real-of-float } (\text{float-divl } \text{prec } x \ y) \leq x / y$
 ⟨*proof*⟩

lemma *real-divl-lower-bound*: $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-divl } \text{prec } x \ y$
 ⟨*proof*⟩

lemma *float-divl-lower-bound*: $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-of-float } (\text{float-divl } \text{prec } x \ y)$
 ⟨*proof*⟩

lemma *exponent-1*: $\text{exponent } 1 = 0$
 ⟨*proof*⟩

lemma *mantissa-1*: $\text{mantissa } 1 = 1$
 ⟨*proof*⟩

lemma *bitlen-1*: $\text{bitlen } 1 = 1$
 ⟨*proof*⟩

lemma *float-upper-bound*: $x \leq 2^{\text{powr } (\text{bitlen } |\text{mantissa } x| + \text{exponent } x)}$
 ⟨*proof*⟩

lemma *real-divl-pos-less1-bound*:
assumes $0 < x \leq 1$
shows $1 \leq \text{real-divl } \text{prec } 1 \ x$
 ⟨*proof*⟩

lemma *float-divl-pos-less1-bound*:
 $0 < \text{real-of-float } x \implies \text{real-of-float } x \leq 1 \implies \text{prec} \geq 1 \implies$
 $1 \leq \text{real-of-float } (\text{float-divl } \text{prec } 1 \ x)$
 ⟨*proof*⟩

lemma *float-divr*: *real-of-float* x / *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*: $0 < x \implies x \leq 1 \implies 1 \leq$ *float-divr prec* 1 x
 ⟨*proof*⟩

lemma *real-divr-nonpos-pos-upper-bound*: $x \leq 0 \implies 0 \leq y \implies$ *real-divr prec* x $y \leq 0$
 ⟨*proof*⟩

lemma *float-divr-nonpos-pos-upper-bound*:
real-of-float $x \leq 0 \implies 0 \leq$ *real-of-float* $y \implies$ *real-of-float* (*float-divr prec* x y) ≤ 0
 ⟨*proof*⟩

lemma *real-divr-nonneg-neg-upper-bound*: $0 \leq x \implies y \leq 0 \implies$ *real-divr prec* x $y \leq 0$
 ⟨*proof*⟩

lemma *float-divr-nonneg-neg-upper-bound*:
 $0 \leq$ *real-of-float* $x \implies$ *real-of-float* $y \leq 0 \implies$ *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 $a :: \text{float}$
 shows *real-of-float* (*pprt* a) = *pprt* (*real-of-float* a)
 ⟨*proof*⟩

lemma *real-of-float-nprt[simp]*:
 fixes $a :: \text{float}$
 shows *real-of-float* (*nprt* a) = *nprt* (*real-of-float* a)
 ⟨*proof*⟩

context
begin

lift-definition *int-floor-fl* :: *float* $\Rightarrow$ *int* **is** *floor* ⟨*proof*⟩ **lemma** *compute-int-floor-fl[code]*:

int-floor-fl (*Float m e*) = (if $0 \leq e$ then $m * 2^{\wedge} \text{nat } e$ else $m \text{ div } (2^{\wedge} (\text{nat } (-e)))$)
 ⟨proof⟩

lift-definition *floor-fl* :: *float* $\Rightarrow$ *float* **is** $\lambda x. \text{real-of-int } \lfloor x \rfloor$

⟨proof⟩ **lemma** *compute-floor-fl*[code]:
floor-fl (*Float m e*) = (if $0 \leq e$ then *Float m e* else *Float* ($m \text{ div } (2^{\wedge} (\text{nat } (-e)))$)
 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*) $\leq$ *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

43 Pointwise instantiation of functions to division

theory *Function-Division*
imports *Function-Algebras*
begin

43.1 Syntactic with division

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

definition *inverse f* = *inverse* $\circ$ *f*

definition $f \text{ div } g = (\lambda x. f \ x \ / \ g \ x)$

instance $\langle \text{proof} \rangle$

end

lemma *inverse-fun-apply* [simp]:
 $\text{inverse } f \ x = \text{inverse } (f \ x)$
 $\langle \text{proof} \rangle$

lemma *divide-fun-apply* [simp]:
 $(f \ / \ g) \ x = f \ x \ / \ g \ x$
 $\langle \text{proof} \rangle$

Unfortunately, we cannot lift this operations to algebraic type classes for division: being different from the constant zero function $f \neq (0 :: 'a)$ is too weak as precondition. So we must introduce our own set of lemmas.

abbreviation *zero-free* :: $('b \Rightarrow 'a::\text{field}) \Rightarrow \text{bool}$ **where**
 $\text{zero-free } f \equiv \neg (\exists x. f \ x = 0)$

lemma *fun-left-inverse*:
fixes $f :: 'b \Rightarrow 'a::\text{field}$
shows $\text{zero-free } f \Longrightarrow \text{inverse } f * f = 1$
 $\langle \text{proof} \rangle$

lemma *fun-right-inverse*:
fixes $f :: 'b \Rightarrow 'a::\text{field}$
shows $\text{zero-free } f \Longrightarrow f * \text{inverse } f = 1$
 $\langle \text{proof} \rangle$

lemma *fun-divide-inverse*:
fixes $f \ g :: 'b \Rightarrow 'a::\text{field}$
shows $f \ / \ g = f * \text{inverse } g$
 $\langle \text{proof} \rangle$

Feel free to extend this.

Another possibility would be a reformulation of the division type classes to use a *zero-free* predicate rather than a direct $a \neq (0 :: 'a)$ condition.

end

44 Lexicographic order on functions

theory *Fun-Lexorder*
imports *Main*
begin

definition *less-fun* :: $('a::\text{linorder} \Rightarrow 'b::\text{linorder}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$
where

$$\text{less-fun } f \ g \longleftrightarrow (\exists k. f \ k < g \ k \wedge (\forall k' < k. f \ k' = g \ k'))$$

lemma *less-funI*:

assumes $\exists k. f \ k < g \ k \wedge (\forall k' < k. f \ k' = g \ k')$

shows *less-fun* $f \ g$

$\langle \text{proof} \rangle$

lemma *less-funE*:

assumes *less-fun* $f \ g$

obtains k **where** $f \ k < g \ k$ **and** $\bigwedge k'. k' < k \implies f \ k' = g \ k'$

$\langle \text{proof} \rangle$

lemma *less-fun-asym*:

assumes *less-fun* $f \ g$

shows $\neg \text{less-fun } g \ f$

$\langle \text{proof} \rangle$

lemma *less-fun-irrefl*:

$\neg \text{less-fun } f \ f$

$\langle \text{proof} \rangle$

lemma *less-fun-trans*:

assumes *less-fun* $f \ g$ **and** *less-fun* $g \ h$

shows *less-fun* $f \ h$

$\langle \text{proof} \rangle$

lemma *order-less-fun*:

class.order $(\lambda f \ g. \text{less-fun } f \ g \vee f = g) \text{ less-fun}$

$\langle \text{proof} \rangle$

lemma *less-fun-trichotomy*:

assumes *finite* $\{k. f \ k \neq g \ k\}$

shows *less-fun* $f \ g \vee f = g \vee \text{less-fun } g \ f$

$\langle \text{proof} \rangle$

end

45 The *going-to* filter

theory *Going-To-Filter*

imports *Complex-Main*

begin

definition *going-to-within* :: $('a \Rightarrow 'b) \Rightarrow 'b \text{ filter} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ filter}$

$(\langle (-) / \text{going}'\text{-to } (-) / \text{within } (-) \rangle [1000, 60, 60] \ 60) \text{ where}$

$f \text{ going-to } F \text{ within } A = \inf (\text{filtercomap } f \ F) (\text{principal } A)$

abbreviation *going-to* :: $('a \Rightarrow 'b) \Rightarrow 'b \text{ filter} \Rightarrow 'a \text{ filter}$

$(\text{infix } \langle \text{going}'\text{-to} \rangle \ 60)$

where f going-to $F \equiv f$ 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 $f : A \rightarrow B$ 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 neighbourhood 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 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 $F = \text{filtercomap } f \ F$
 ⟨proof⟩

lemma *eventually-going-toI* [intro]:
assumes *eventually* $P \ F$
shows *eventually* $(\lambda x. P \ (f \ x)) \ (f \text{ going-to } F)$
 ⟨proof⟩

lemma *filterlim-going-toI-weak* [intro]: $\text{filterlim } f \ F \ (f \text{ going-to } F \text{ within } A)$
 ⟨proof⟩

lemma *going-to-mono*: $F \leq G \implies A \subseteq B \implies f \text{ going-to } F \text{ within } A \leq f \text{ going-to } G \text{ within } B$
 ⟨proof⟩

lemma *going-to-inf*:
 $f \text{ going-to } (\inf F \ G) \text{ within } A = \inf (f \text{ going-to } F \text{ within } A) \ (f \text{ going-to } G \text{ within } A)$
 ⟨proof⟩

lemma *going-to-sup*:
 $f \text{ going-to } (\sup F \ G) \text{ within } A \geq \sup (f \text{ going-to } F \text{ within } A) \ (f \text{ going-to } G \text{ within } A)$
 ⟨proof⟩

lemma *going-to-top* [simp]: $f \text{ going-to top within } A = \text{principal } A$
 ⟨proof⟩

lemma *going-to-bot* [simp]: f going-to bot within $A = \text{bot}$
 ⟨proof⟩

lemma *going-to-principal*:
 f going-to principal A within $B = \text{principal } (f \text{ -- } A \cap B)$
 ⟨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 $(A \cup B) = \text{sup } (f \text{ going-to } F \text{ within } A) (f \text{ going-to } F \text{ within } B)$
 ⟨proof⟩

lemma *eventually-going-to-at-top-linorder*:
 fixes $f :: 'a \Rightarrow 'b :: \text{linorder}$
 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 $f :: 'a \Rightarrow 'b :: \text{linorder}$
 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 $f :: 'a \Rightarrow 'b :: \{\text{linorder}, \text{no-top}\}$
 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 $f :: 'a \Rightarrow 'b :: \{\text{linorder}, \text{no-bot}\}$
 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) $\longleftrightarrow$
 $(\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f x \in S \longrightarrow P x))$
 ⟨proof⟩

lemma *eventually-going-to-at*:
 eventually P (f going-to (a within B) within A) $\longleftrightarrow$
 $(\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f x \in B \cap S - \{a\} \longrightarrow P x))$
 ⟨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

```

46 Big sum and product over function bodies

```

theory Groups-Big-Fun
imports
  Main
begin

```

46.1 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 1 g {a. g a  $\neq$  1}

```

```

interpretation F: comm-monoid-set f 1
  ⟨proof⟩

```

```

lemma expand-superset:
  assumes finite A and {a. g a  $\neq$  1}  $\subseteq$  A
  shows G g = F.F g A
  ⟨proof⟩

```

```

lemma conditionalize:
  assumes finite A
  shows F.F g A = G ( $\lambda a.$  if a  $\in$  A then g a else 1)
  ⟨proof⟩

```

```

lemma neutral [simp]:
  G ( $\lambda a.$  1) = 1
  ⟨proof⟩

```

```

lemma update [simp]:
  assumes finite {a. g a  $\neq$  1}
  assumes g a = 1
  shows G (g(a := b)) = b * G g
  ⟨proof⟩

```

```

lemma infinite [simp]:
   $\neg$  finite {a. g a  $\neq$  1}  $\implies$  G g = 1

```

$\langle \text{proof} \rangle$

lemma *cong* [*cong*]:
 assumes $\bigwedge a. g\ a = h\ a$
 shows $G\ g = G\ h$
 $\langle \text{proof} \rangle$

lemma *not-neutral-obtains-not-neutral*:
 assumes $G\ g \neq \mathbf{1}$
 obtains a where $g\ a \neq \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *reindex-cong*:
 assumes *bij* l
 assumes $g \circ l = h$
 shows $G\ g = G\ h$
 $\langle \text{proof} \rangle$

lemma *distrib*:
 assumes *finite* $\{a. g\ a \neq \mathbf{1}\}$ and *finite* $\{a. h\ a \neq \mathbf{1}\}$
 shows $G\ (\lambda a. g\ a * h\ a) = G\ g * G\ h$
 $\langle \text{proof} \rangle$

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 $?A \times ?B \subseteq C$)
 shows $G\ (\lambda a. G\ (g\ a)) = G\ (\lambda b. G\ (\lambda a. g\ a\ b))$
 $\langle \text{proof} \rangle$

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 $?A \times ?B \subseteq C$)
 shows $G\ (\lambda a. G\ (g\ a)) = G\ (\lambda(a, b). g\ a\ b)$
 $\langle \text{proof} \rangle$

lemma *cartesian-product2*:
 assumes *fin*: *finite* D
 assumes *subset*: $\{(a, b). \exists c. g\ a\ b\ c \neq \mathbf{1}\} \times \{c. \exists a\ b. g\ a\ b\ c \neq \mathbf{1}\} \subseteq D$ (is $?AB \times ?C \subseteq D$)
 shows $G\ (\lambda(a, b). G\ (g\ a\ b)) = G\ (\lambda(a, b, c). g\ a\ b\ c)$
 $\langle \text{proof} \rangle$

lemma *delta* [*simp*]:
 $G\ (\lambda b. \text{if } b = a \text{ then } g\ b \text{ else } \mathbf{1}) = g\ a$
 $\langle \text{proof} \rangle$

lemma *delta'* [*simp*]:

$G (\lambda b. \text{ if } a = b \text{ then } g \ b \text{ else } \mathbf{1}) = g \ a$
 $\langle \text{proof} \rangle$

end

46.2 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*
 $\langle \text{proof} \rangle$

end

syntax (*ASCII*)
 $\text{-Sum-any} :: \text{pttrn} \Rightarrow 'a \Rightarrow 'a :: \text{comm-monoid-add} \quad ((\mathcal{S} \text{SUM } -. \text{ }) [0, 10] \ 10)$

syntax
 $\text{-Sum-any} :: \text{pttrn} \Rightarrow 'a \Rightarrow 'a :: \text{comm-monoid-add} \quad ((\mathcal{S} \sum -. \text{ }) [0, 10] \ 10)$

translations
 $\sum a. b \Rightarrow \text{CONST } \text{Sum-any } (\lambda a. \ b)$

lemma *Sum-any-left-distrib:*
fixes $r :: 'a :: \text{semiring-0}$
assumes $\text{finite } \{a. \ g \ a \neq 0\}$
shows $\text{Sum-any } g * r = (\sum n. \ g \ n * r)$
 $\langle \text{proof} \rangle$

lemma *Sum-any-right-distrib:*
fixes $r :: 'a :: \text{semiring-0}$
assumes $\text{finite } \{a. \ g \ a \neq 0\}$
shows $r * \text{Sum-any } g = (\sum n. \ r * g \ n)$
 $\langle \text{proof} \rangle$

lemma *Sum-any-product:*
fixes $f \ g :: 'b \Rightarrow 'a :: \text{semiring-0}$
assumes $\text{finite } \{a. \ f \ a \neq 0\}$ **and** $\text{finite } \{b. \ g \ b \neq 0\}$
shows $\text{Sum-any } f * \text{Sum-any } g = (\sum a. \ \sum b. \ f \ a * g \ b)$
 $\langle \text{proof} \rangle$

lemma *Sum-any-eq-zero-iff [simp]:*
fixes $f :: 'a \Rightarrow \text{nat}$
assumes $\text{finite } \{a. \ f \ a \neq 0\}$
shows $\text{Sum-any } f = 0 \longleftrightarrow f = (\lambda -. \ 0)$
 $\langle \text{proof} \rangle$

46.3 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 $\Rightarrow$ 'a $\Rightarrow$ 'a::comm-monoid-mult (($\exists$ PROD -. -) [0, 10] 10)

syntax
 -Prod-any :: pttrn $\Rightarrow$ 'a $\Rightarrow$ 'a::comm-monoid-mult (($\exists$ $\prod$ -. -) [0, 10] 10)

translations
 $\prod a. b == \text{CONST } \text{Prod-any } (\lambda a. b)$

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-semiring-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 $c \wedge (\sum a. f a) = (\prod a. c \wedge f a)$
<proof>

end

47 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 ≤ 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 ⟷ lower a = lower b ∧ upper a = upper b
  ⟨proof⟩

lemma interval-eqI: lower a = lower b ⟹ upper a = upper b ⟹ a = b
  ⟨proof⟩

lemma lower-le-upper[simp]: lower i ≤ upper i
  ⟨proof⟩

lift-definition set-of :: 'a::preorder interval ⇒ 'a set is λx. {fst x .. snd x} ⟨proof⟩

lemma set-of-eq: set-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 λa b. if a ≤ b then Some (a, b) else None
  ⟨proof⟩

lemma Interval'-split:
  P (Interval' a b) ⟷
    (∀ ivl. a ≤ b ⟶ lower ivl = a ⟶ upper ivl = b ⟶ P (Some ivl)) ∧ (¬a ≤ b
    ⟶ P None)
  ⟨proof⟩

lemma Interval'-split-asm:
  P (Interval' a b) ⟷
    ¬((∃ ivl. a ≤ b ∧ lower ivl = a ∧ upper ivl = b ∧ ¬P (Some ivl)) ∨ (¬a ≤ b ∧
    ¬P None))
  ⟨proof⟩

lemmas Interval'-splits = Interval'-split Interval'-split-asm

lemma Interval'-eq-Some: Interval' a b = Some i ⟹ lower i = a ∧ upper i = b
  ⟨proof⟩

end

```


instantiation *interval* :: (*{preorder,equal}*) *equal*
begin

definition *equal-class.equal* *a b* $\equiv$ (*lower a = lower b*) $\wedge$ (*upper a = upper b*)

instance *<proof>*
end

instantiation *interval* :: (*preorder*) *ord* **begin**

definition *less-eq-interval* :: '*a interval* $\Rightarrow$ '*a interval* $\Rightarrow$ *bool*
where *less-eq-interval* *a b* $\longleftrightarrow$ *lower b* $\leq$ *lower a* $\wedge$ *upper a* $\leq$ *upper b*

definition *less-interval* :: '*a interval* $\Rightarrow$ '*a interval* $\Rightarrow$ *bool*
where *less-interval* *x y* = (*x* $\leq$ *y* $\wedge$ $\neg$ *y* $\leq$ *x*)

instance *<proof>*
end

instantiation *interval* :: (*lattice*) *semilattice-sup*
begin

lift-definition *sup-interval* :: '*a interval* $\Rightarrow$ '*a interval* $\Rightarrow$ '*a interval*
is $\lambda(a, b) (c, d). (inf\ a\ c, sup\ b\ d)$
<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* $\cup$ *set-of B* $\subseteq$ *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* $\subseteq$ *set-of (sup a A)* **for** *A :: 'a::lattice interval*
<proof>

lemma *interval-union-mono2*: *set-of A* $\subseteq$ *set-of (sup a A)* **for** *A :: 'a::lattice interval*

<proof>

lift-definition *interval-of* :: 'a::preorder $\Rightarrow$ 'a interval **is** $\lambda x. (x, x)$
<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 $\Rightarrow$ 'a
 where *width* i = upper i – lower i

instantiation *interval* :: (ordered-ab-semigroup-add) ab-semigroup-add
begin

lift-definition *plus-interval*::'a interval $\Rightarrow$ 'a interval $\Rightarrow$ 'a interval
is $\lambda(a, b). \lambda(c, d). (a + c, b + d)$
<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* $\Rightarrow$ 'a *interval* **is** $\lambda(a, b). (-b, -a)$
 $\langle proof \rangle$

lemma *lower-uminus[simp]*: *lower* ($- A$) = $-$ *upper* *A*
 $\langle proof \rangle$

lemma *upper-uminus[simp]*: *upper* ($- A$) = $-$ *lower* *A*
 $\langle proof \rangle$

instance $\langle proof \rangle$
end

instantiation *interval* :: (*{ordered-ab-group-add}*) *minus*
begin

definition *minus-interval*::'a *interval* $\Rightarrow$ 'a *interval* $\Rightarrow$ 'a *interval*
where *minus-interval* *a b* = *a* + $-$ *b*

lemma *lower-minus[simp]*: *lower* (*minus* *A B*) = *minus* (*lower* *A*) (*upper* *B*)
 $\langle proof \rangle$

lemma *upper-minus[simp]*: *upper* (*minus* *A B*) = *minus* (*upper* *A*) (*lower* *B*)
 $\langle proof \rangle$

instance $\langle proof \rangle$
end

instantiation *interval* :: (*linordered-semiring*) *times*
begin

lift-definition *times-interval* :: 'a *interval* $\Rightarrow$ 'a *interval* $\Rightarrow$ 'a *interval*
is $\lambda(a1, a2). \lambda(b1, b2).$
 $(let\ x1 = a1 * b1; x2 = a1 * b2; x3 = a2 * b1; x4 = a2 * b2$
 $in\ (min\ x1\ (min\ x2\ (min\ x3\ x4)),\ max\ x1\ (max\ x2\ (max\ x3\ x4))))$
 $\langle proof \rangle$

lemma *lower-times*:

lower (*times* *A B*) = *Min* {*lower* *A* * *lower* *B*, *lower* *A* * *upper* *B*, *upper* *A* *
lower *B*, *upper* *A* * *upper* *B*}
 $\langle proof \rangle$

lemma *upper-times*:

upper (*times* *A B*) = *Max* {*lower* *A* * *lower* *B*, *lower* *A* * *upper* *B*, *upper* *A* *
lower *B*, *upper* *A* * *upper* *B*}
 $\langle proof \rangle$

instance $\langle proof \rangle$
end

lemma *interval-eq-set-of-iff*: *X* = *Y* $\longleftrightarrow$ *set-of* *X* = *set-of* *Y* **for** *X Y*::'a::*order*
interval

$\langle \text{proof} \rangle$

47.1 Membership

abbreviation (in *preorder*) *in-interval* $((-/ \in_i -) [51, 51] 50)$
 where *in-interval* $x X \equiv x \in \text{set-of } X$

lemma *in-interval-to-interval*[intro!]: $a \in_i \text{interval-of } a$
 $\langle \text{proof} \rangle$

lemma *plus-in-intervalI*:
fixes $x y :: 'a :: \text{ordered-ab-semigroup-add}$
shows $x \in_i X \implies y \in_i Y \implies x + y \in_i X + Y$
 $\langle \text{proof} \rangle$

lemma *connected-set-of*[intro, simp]:
connected (*set-of* X) **for** $X :: 'a :: \text{linear-continuum-topology interval}$
 $\langle \text{proof} \rangle$

lemma *ex-sum-in-interval-lemma*: $\exists xa \in \{la .. ua\}. \exists xb \in \{lb .. ub\}. x = xa + xb$
if $la \leq ua \text{ } lb \leq ub \text{ } la + lb \leq x \text{ } x \leq ua + ub$
 $ua - la \leq ub - lb$
for $la \text{ } b \text{ } c \text{ } d :: 'a :: \text{linordered-ab-group-add}$
 $\langle \text{proof} \rangle$

lemma *ex-sum-in-interval*: $\exists xa \geq la. xa \leq ua \wedge (\exists xb \geq lb. xb \leq ub \wedge x = xa + xb)$
if $a: la \leq ua$ **and** $b: lb \leq ub$ **and** $x: la + lb \leq x \text{ } x \leq ua + ub$
for $la \text{ } b \text{ } c \text{ } d :: 'a :: \text{linordered-ab-group-add}$
 $\langle \text{proof} \rangle$

lemma *Icc-plus-Icc*:
 $\{a .. b\} + \{c .. d\} = \{a + c .. b + d\}$
if $a \leq b \text{ } c \leq d$
for $a \text{ } b \text{ } c \text{ } d :: 'a :: \text{linordered-ab-group-add}$
 $\langle \text{proof} \rangle$

lemma *set-of-plus*:
fixes $A :: 'a :: \text{linordered-ab-group-add interval}$
shows *set-of* $(A + B) = \text{set-of } A + \text{set-of } B$
 $\langle \text{proof} \rangle$

lemma *plus-in-intervalE*:
fixes $xy :: 'a :: \text{linordered-ab-group-add}$
assumes $xy \in_i X + Y$
obtains $x \text{ } y$ **where** $xy = x + y \text{ } x \in_i X \text{ } y \in_i Y$
 $\langle \text{proof} \rangle$

lemma *set-of-uminus*: *set-of* $(-X) = \{-x \mid x. x \in \text{set-of } X\}$

for $X :: 'a :: \text{ordered-ab-group-add interval}$
 $\langle \text{proof} \rangle$

lemma *uminus-in-intervalI*:
fixes $x :: 'a :: \text{ordered-ab-group-add}$
shows $x \in_i X \implies -x \in_i -X$
 $\langle \text{proof} \rangle$

lemma *uminus-in-intervalD*:
fixes $x :: 'a :: \text{ordered-ab-group-add}$
shows $x \in_i -X \implies -x \in_i X$
 $\langle \text{proof} \rangle$

lemma *minus-in-intervalI*:
fixes $x y :: 'a :: \text{ordered-ab-group-add}$
shows $x \in_i X \implies y \in_i Y \implies x - y \in_i X - Y$
 $\langle \text{proof} \rangle$

lemma *set-of-minus*: $\text{set-of } (X - Y) = \{x - y \mid x y . x \in \text{set-of } X \wedge y \in \text{set-of } Y\}$
for $X Y :: 'a :: \text{linordered-ab-group-add interval}$
 $\langle \text{proof} \rangle$

lemma *times-in-intervalI*:
fixes $x y :: 'a :: \text{linordered-ring}$
assumes $x \in_i X \ y \in_i Y$
shows $x * y \in_i X * Y$
 $\langle \text{proof} \rangle$

lemma *times-in-intervalE*:
fixes $xy :: 'a :: \{\text{linordered-semiring, real-normed-algebra, linear-continuum-topology}\}$
— TODO: linear continuum topology is pretty strong
assumes $xy \in_i X * Y$
obtains $x y$ **where** $xy = x * y \ x \in_i X \ y \in_i Y$
 $\langle \text{proof} \rangle$

lemma *set-of-times*: $\text{set-of } (X * Y) = \{x * y \mid x y . x \in \text{set-of } X \wedge y \in \text{set-of } Y\}$
for $X Y :: 'a :: \{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$
 interval
 $\langle \text{proof} \rangle$

instance *interval* :: $(\text{linordered-idom}) \text{ cancel-semigroup-add}$
 $\langle \text{proof} \rangle$

lemma *interval-mul-commute*: $A * B = B * A$ **for** $A B :: 'a :: \text{linordered-idom interval}$
 $\langle \text{proof} \rangle$

lemma *interval-times-zero-right[simp]*: $A * 0 = 0$ **for** $A :: 'a :: \text{linordered-ring interval}$

terval
 $\langle \text{proof} \rangle$

lemma *interval-times-zero-left*[simp]:
 $0 * A = 0$ **for** $A :: 'a::\text{linordered-ring interval}$
 $\langle \text{proof} \rangle$

instantiation *interval* :: ($\{\text{preorder}, \text{one}\}$) *one*
begin

lift-definition *one-interval*:: $'a \text{ interval}$ **is** $(1, 1)$ $\langle \text{proof} \rangle$

lemma *lower-one*[simp]: $\text{lower } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *upper-one*[simp]: $\text{upper } 1 = 1$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$
end

instance *interval* :: ($\{\text{one}, \text{preorder}, \text{linordered-semiring}\}$) *power*
 $\langle \text{proof} \rangle$

lemma *set-of-one*[simp]: $\text{set-of } (1::'a::\{\text{one}, \text{order}\} \text{ interval}) = \{1\}$
 $\langle \text{proof} \rangle$

instance *interval* ::
 $(\{\text{linordered-idom}, \text{linordered-ring}, \text{real-normed-algebra}, \text{linear-continuum-topology}\})$
monoid-mult
 $\langle \text{proof} \rangle$

lemma *one-times-ivl-left*[simp]: $1 * A = A$ **for** $A :: 'a::\text{linordered-idom interval}$
 $\langle \text{proof} \rangle$

lemma *one-times-ivl-right*[simp]: $A * 1 = A$ **for** $A :: 'a::\text{linordered-idom interval}$
 $\langle \text{proof} \rangle$

lemma *set-of-power-mono*: $a^{\wedge} n \in \text{set-of } (A^{\wedge} n)$ **if** $a \in \text{set-of } A$
for $a :: 'a::\text{linordered-idom}$
 $\langle \text{proof} \rangle$

lemma *set-of-add-cong*:
 $\text{set-of } (A + B) = \text{set-of } (A' + B')$
if $\text{set-of } A = \text{set-of } A'$ $\text{set-of } B = \text{set-of } B'$
for $A :: 'a::\text{linordered-ab-group-add interval}$
 $\langle \text{proof} \rangle$

lemma *set-of-add-inc-left*:
 $\text{set-of } (A + B) \subseteq \text{set-of } (A' + B)$
if $\text{set-of } A \subseteq \text{set-of } A'$
for $A :: 'a::\text{linordered-ab-group-add interval}$

$\langle proof \rangle$

lemma *set-of-add-inc-right*:
 $set-of (A + B) \subseteq set-of (A + B')$
if $set-of B \subseteq set-of B'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-add-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 $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-neg-inc*:
 $set-of (-A) \subseteq set-of (-A')$
if $set-of A \subseteq set-of A'$
for $A :: 'a::ordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-sub-inc-left*:
 $set-of (A - B) \subseteq set-of (A' - B)$
if $set-of A \subseteq set-of A'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-sub-inc-right*:
 $set-of (A - B) \subseteq set-of (A - B')$
if $set-of B \subseteq set-of B'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

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 $A :: 'a::linordered-idom interval$
 $\langle proof \rangle$

lemma *set-of-mul-inc-right*:
 $set-of (A * B) \subseteq set-of (A * B')$
if $set-of B \subseteq set-of B'$
for $A :: 'a::linordered-ring interval$
 $\langle proof \rangle$

lemma *set-of-distrib-left*:
 $set-of (B * (A1 + A2)) \subseteq set-of (B * A1 + B * A2)$
for $A1 :: 'a::linordered-ring interval$
 $\langle proof \rangle$

lemma *set-of-distrib-right:*

set-of $((A1 + A2) * B) \subseteq \text{set-of } (A1 * B + A2 * B)$
for $A1\ A2\ B :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$
interval
<proof>

lemma *set-of-mul-inc-left:*

set-of $(A * B) \subseteq \text{set-of } (A' * B)$
if *set-of* $A \subseteq \text{set-of } A'$
for $A :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *set-of-mul-inc:*

set-of $(A * B) \subseteq \text{set-of } (A' * B')$
if *set-of* $A \subseteq \text{set-of } A'$ *set-of* $B \subseteq \text{set-of } B'$
for $A :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *set-of-pow-inc:*

set-of $(A^{\wedge}n) \subseteq \text{set-of } (A'^{\wedge}n)$
if *set-of* $A \subseteq \text{set-of } A'$
for $A :: 'a::\{\text{linordered-idom, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *set-of-distrib-right-left:*

set-of $((A1 + A2) * (B1 + B2)) \subseteq \text{set-of } (A1 * B1 + A1 * B2 + A2 * B1 + A2 * B2)$
for $A1 :: 'a::\{\text{linordered-idom, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *mult-bounds-enclose-zero1:*

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$
 $0 \leq \max (la * lb) (\max (la * ub) (\max (lb * ua) (ua * ub)))$
if $la \leq 0\ 0 \leq ua$
for $la\ lb\ ua\ ub:: 'a::\text{linordered-idom}$
<proof>

lemma *mult-bounds-enclose-zero2:*

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$
 $0 \leq \max (la * lb) (\max (la * ub) (\max (lb * ua) (ua * ub)))$
if $lb \leq 0\ 0 \leq ub$
for $la\ lb\ ua\ ub:: 'a::\text{linordered-idom}$
<proof>

lemma *set-of-mul-contains-zero:*

$0 \in \text{set-of } (A * B)$
if $0 \in \text{set-of } A \vee 0 \in \text{set-of } B$
for $A :: 'a::\text{linordered-idom interval}$
 $\langle \text{proof} \rangle$

instance $\text{interval} :: (\text{linordered-semiring}) \text{ mult-zero}$
 $\langle \text{proof} \rangle$

lift-definition $\text{min-interval}::'a::\text{linorder interval} \Rightarrow 'a \text{ interval} \Rightarrow 'a \text{ interval}$ **is**
 $\lambda(l1, u1). \lambda(l2, u2). (\text{min } l1 \ l2, \text{min } u1 \ u2)$
 $\langle \text{proof} \rangle$

lemma $\text{lower-min-interval[simp]}: \text{lower } (\text{min-interval } x \ y) = \text{min } (\text{lower } x) (\text{lower } y)$
 $\langle \text{proof} \rangle$

lemma $\text{upper-min-interval[simp]}: \text{upper } (\text{min-interval } x \ y) = \text{min } (\text{upper } x) (\text{upper } y)$
 $\langle \text{proof} \rangle$

lemma $\text{min-intervalI}:$
 $a \in_i A \Longrightarrow b \in_i B \Longrightarrow \text{min } a \ b \in_i \text{min-interval } A \ B$
 $\langle \text{proof} \rangle$

lift-definition $\text{max-interval}::'a::\text{linorder interval} \Rightarrow 'a \text{ interval} \Rightarrow 'a \text{ interval}$ **is**
 $\lambda(l1, u1). \lambda(l2, u2). (\text{max } l1 \ l2, \text{max } u1 \ u2)$
 $\langle \text{proof} \rangle$

lemma $\text{lower-max-interval[simp]}: \text{lower } (\text{max-interval } x \ y) = \text{max } (\text{lower } x) (\text{lower } y)$
 $\langle \text{proof} \rangle$

lemma $\text{upper-max-interval[simp]}: \text{upper } (\text{max-interval } x \ y) = \text{max } (\text{upper } x) (\text{upper } y)$
 $\langle \text{proof} \rangle$

lemma $\text{max-intervalI}:$
 $a \in_i A \Longrightarrow b \in_i B \Longrightarrow \text{max } a \ b \in_i \text{max-interval } A \ B$
 $\langle \text{proof} \rangle$

lift-definition $\text{abs-interval}::'a::\text{linordered-idom interval} \Rightarrow 'a \text{ interval}$ **is**
 $(\lambda(l, u). (\text{if } l < 0 \wedge 0 < u \text{ then } 0 \text{ else } \text{min } |l| \ |u|, \text{max } |l| \ |u|))$
 $\langle \text{proof} \rangle$

lemma $\text{lower-abs-interval[simp]}:$
 $\text{lower } (\text{abs-interval } x) = (\text{if } \text{lower } x < 0 \wedge 0 < \text{upper } x \text{ then } 0 \text{ else } \text{min } |\text{lower } x| \ |\text{upper } x|)$
 $\langle \text{proof} \rangle$

lemma $\text{upper-abs-interval[simp]}: \text{upper } (\text{abs-interval } x) = \text{max } |\text{lower } x| \ |\text{upper } x|$
 $\langle \text{proof} \rangle$

lemma $\text{in-abs-intervalI1}:$
 $lx < 0 \Longrightarrow 0 < ux \Longrightarrow 0 \leq xa \Longrightarrow xa \leq \text{max } (- \ lx) \ (ux) \Longrightarrow xa \in \text{abs } ' \{lx..ux\}$

for $xa::'a::linordered-idom$
 $\langle proof \rangle$

lemma *in-abs-intervalI2*:
 $min (|lx|) |ux| \leq xa \implies xa \leq max |lx| |ux| \implies lx \leq ux \implies 0 \leq lx \vee ux \leq 0$
 $\implies$
 $xa \in abs \text{ ‘ } \{lx..ux\}$
for $xa::'a::linordered-idom$
 $\langle proof \rangle$

lemma *set-of-abs-interval*: $set-of (abs-interval x) = abs \text{ ‘ } set-of x$
 $\langle proof \rangle$

fun *split-domain* :: $('a::preorder \text{ interval} \Rightarrow 'a \text{ interval list}) \Rightarrow 'a \text{ interval list} \Rightarrow$
 $'a \text{ interval list list}$
where *split-domain split* [] = []
 $| \text{ split-domain split } (I \# Is) = ($
 $\quad let S = split I;$
 $\quad \quad D = split-domain split Is$
 $\quad in concat (map (\lambda d. map (\lambda s. s \# d) S) D)$
 $\quad)$

context notes $[[typedef-overloaded]]$ **begin**

lift-definition(*code-dt*) *split-interval*:: $'a::linorder \text{ interval} \Rightarrow 'a \Rightarrow ('a \text{ interval} \times$
 $'a \text{ interval})$
is $\lambda(l, u) x. ((min l x, max l x), (min u x, max u x))$
 $\langle proof \rangle$
end

lemma *split-domain-nonempty*:
assumes $\bigwedge I. split I \neq []$
shows $split-domain split I \neq []$
 $\langle proof \rangle$

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*: $upper (snd (split-interval X m)) = max (upper X) m$
 $\langle proof \rangle$

lemma *split-intervalD*: $split-interval X x = (A, B) \implies set-of X \subseteq set-of A \cup set-of B$
 $\langle proof \rangle$

instantiation *interval* :: $(\{topological-space, preorder\}) \text{ topological-space}$
begin

definition *open-interval-def*[*code del*]: $open (X::'a \text{ interval set}) =$
 $(\forall x \in X.$

$\exists A B.$
 $\text{open } A \wedge$
 $\text{open } B \wedge$
 $\text{lower } x \in A \wedge \text{upper } x \in B \wedge \text{Interval } (A \times B) \subseteq X$

instance
 $\langle \text{proof} \rangle$

end

47.2 Quickcheck

lift-definition $\text{Ivl}::'a \Rightarrow 'a::\text{preorder} \Rightarrow 'a \text{ interval}$ **is** $\lambda a b. (\text{min } a b, b)$
 $\langle \text{proof} \rangle$

instantiation $\text{interval} :: (\{\text{exhaustive}, \text{preorder}\}) \text{ exhaustive}$
begin

definition $\text{exhaustive-interval}::('a \text{ interval} \Rightarrow (\text{bool} \times \text{term list}) \text{ option})$
 $\Rightarrow \text{natural} \Rightarrow (\text{bool} \times \text{term list}) \text{ option}$
where
 $\text{exhaustive-interval } f d =$
 $\text{Quickcheck-Exhaustive.exhaustive } (\lambda x. \text{Quickcheck-Exhaustive.exhaustive } (\lambda y. f$
 $(\text{Ivl } x y)) d) d$

instance $\langle \text{proof} \rangle$

end

context
includes term-syntax
begin

definition $[\text{code-unfold}]$:
 $\text{valtermify-interval } x y = \text{Code-Evaluation.valtermify } (\text{Ivl}::'a::\{\text{preorder}, \text{typerep}\} \Rightarrow -)$
 $\{\cdot\} x \{\cdot\} y$

end

instantiation $\text{interval} :: (\{\text{full-exhaustive}, \text{preorder}, \text{typerep}\}) \text{ full-exhaustive}$
begin

definition $\text{full-exhaustive-interval}::$
 $('a \text{ interval} \times (\text{unit} \Rightarrow \text{term}) \Rightarrow (\text{bool} \times \text{term list}) \text{ option})$
 $\Rightarrow \text{natural} \Rightarrow (\text{bool} \times \text{term list}) \text{ option}$ **where**
 $\text{full-exhaustive-interval } f d =$
 $\text{Quickcheck-Exhaustive.full-exhaustive}$
 $(\lambda x. \text{Quickcheck-Exhaustive.full-exhaustive } (\lambda y. f (\text{valtermify-interval } x y)) d)$
 d

```

instance  $\langle proof \rangle$ 

end

instantiation interval :: ( $\{random, preorder, typerep\}$ ) random
begin

definition random-interval ::
  natural
 $\Rightarrow$  natural  $\times$  natural
   $\Rightarrow$  (a interval  $\times$  (unit  $\Rightarrow$  term))  $\times$  natural  $\times$  natural where
  random-interval i =
    scomp (Quickcheck-Random.random i)
    ( $\lambda man.$  scomp (Quickcheck-Random.random i) ( $\lambda exp.$  Pair (valtermify-interval
      man exp))))

instance  $\langle proof \rangle$ 

end

lifting-update interval.lifting
lifting-forget interval.lifting

end

```

48 Approximate Operations on Intervals of Floating Point Numbers

```

theory Interval-Float
  imports
    Interval
    Float
begin

definition mid :: float interval  $\Rightarrow$  float
  where mid i = (lower i + upper i) * Float 1 (-1)

lemma mid-in-interval: mid i  $\in_i$  i
   $\langle proof \rangle$ 

lemma mid-le: lower i  $\leq$  mid i mid i  $\leq$  upper i
   $\langle proof \rangle$ 

definition centered :: float interval  $\Rightarrow$  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*: $\text{split-float-interval } X = (A, B) \implies \text{set-of } X \subseteq \text{set-of } A \cup \text{set-of } B$
 $\langle \text{proof} \rangle$

lemma *split-float-interval-bounds*:
shows

lower-split-float-interval1: $\text{lower } (\text{fst } (\text{split-float-interval } X)) = \text{lower } X$
and *lower-split-float-interval2*: $\text{lower } (\text{snd } (\text{split-float-interval } X)) = \text{mid } X$
and *upper-split-float-interval1*: $\text{upper } (\text{fst } (\text{split-float-interval } X)) = \text{mid } X$
and *upper-split-float-interval2*: $\text{upper } (\text{snd } (\text{split-float-interval } X)) = \text{upper } X$
 $\langle \text{proof} \rangle$

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*).

48.1 Intervals with Floating Point Bounds

context includes *interval.lifting* **begin**

lift-definition *round-interval* :: $\text{nat} \Rightarrow \text{float interval} \Rightarrow \text{float interval}$
is $\lambda p. \lambda(l, u). (\text{float-round-down } p \ l, \text{float-round-up } p \ u)$
 $\langle \text{proof} \rangle$

lemma *lower-round-ivl[simp]*: $\text{lower } (\text{round-interval } p \ x) = \text{float-round-down } p \ (\text{lower } x)$
 $\langle \text{proof} \rangle$

lemma *upper-round-ivl[simp]*: $\text{upper } (\text{round-interval } p \ x) = \text{float-round-up } p \ (\text{upper } x)$
 $\langle \text{proof} \rangle$

lemma *round-ivl-correct*: $\text{set-of } A \subseteq \text{set-of } (\text{round-interval } \text{prec } A)$
 $\langle \text{proof} \rangle$

lift-definition *truncate-ivl* :: $\text{nat} \Rightarrow \text{real interval} \Rightarrow \text{real interval}$
is $\lambda p. \lambda(l, u). (\text{truncate-down } p \ l, \text{truncate-up } p \ u)$
 $\langle \text{proof} \rangle$

lemma *lower-truncate-ivl[simp]*: $\text{lower } (\text{truncate-ivl } p \ x) = \text{truncate-down } p \ (\text{lower } x)$
 $\langle \text{proof} \rangle$

lemma *upper-truncate-ivl[simp]*: $\text{upper } (\text{truncate-ivl } p \ x) = \text{truncate-up } p \ (\text{upper } x)$
 $\langle \text{proof} \rangle$

lemma *truncate-ivl-correct*: $\text{set-of } A \subseteq \text{set-of } (\text{truncate-ivl prec } A)$
 $\langle \text{proof} \rangle$

lift-definition *real-interval::float interval* $\Rightarrow$ *real interval*
is $\lambda(l, u). (\text{real-of-float } l, \text{real-of-float } u)$
 $\langle \text{proof} \rangle$

lemma *lower-real-interval[simp]*: $\text{lower } (\text{real-interval } x) = \text{lower } x$
 $\langle \text{proof} \rangle$

lemma *upper-real-interval[simp]*: $\text{upper } (\text{real-interval } x) = \text{upper } x$
 $\langle \text{proof} \rangle$

definition *set-of'* $x = (\text{case } x \text{ of } \text{None} \Rightarrow \text{UNIV} \mid \text{Some } i \Rightarrow \text{set-of } (\text{real-interval } i))$

lemma *real-interval-min-interval[simp]*:
 $\text{real-interval } (\text{min-interval } a \ b) = \text{min-interval } (\text{real-interval } a) (\text{real-interval } b)$
 $\langle \text{proof} \rangle$

lemma *real-interval-max-interval[simp]*:
 $\text{real-interval } (\text{max-interval } a \ b) = \text{max-interval } (\text{real-interval } a) (\text{real-interval } b)$
 $\langle \text{proof} \rangle$

lemma *in-intervalI*:
 $x \in_i X$ **if** $\text{lower } X \leq x \leq \text{upper } X$
 $\langle \text{proof} \rangle$

abbreviation *in-real-interval* $((-/ \in_r -) [51, 51] 50)$ **where**
 $x \in_r X \equiv x \in_i \text{real-interval } X$

lemma *in-real-intervalI*:
 $x \in_r X$ **if** $\text{lower } X \leq x \leq \text{upper } X$ **for** $x::\text{real}$ **and** $X::\text{float interval}$
 $\langle \text{proof} \rangle$

48.2 intros for *real-interval*

lemma *in-round-intervalI*: $x \in_r A \implies x \in_r (\text{round-interval prec } A)$
 $\langle \text{proof} \rangle$

lemma *zero-in-float-intervalI*: $0 \in_r 0$
 $\langle \text{proof} \rangle$

lemma *plus-in-float-intervalI*: $a + b \in_r A + B$ **if** $a \in_r A$ $b \in_r B$
 $\langle \text{proof} \rangle$

lemma *minus-in-float-intervalI*: $a - b \in_r A - B$ **if** $a \in_r A$ $b \in_r B$
 $\langle \text{proof} \rangle$

lemma *uminus-in-float-intervalI*: $-a \in_r -A$ **if** $a \in_r A$

$\langle \text{proof} \rangle$

lemma *real-interval-times*: $\text{real-interval } (A * B) = \text{real-interval } A * \text{real-interval } B$
 $\langle \text{proof} \rangle$

lemma *times-in-float-intervalI*: $a * b \in_r A * B$ **if** $a \in_r A$ $b \in_r B$
 $\langle \text{proof} \rangle$

lemma *real-interval-abs*: $\text{real-interval } (\text{abs-interval } A) = \text{abs-interval } (\text{real-interval } A)$
 $\langle \text{proof} \rangle$

lemma *abs-in-float-intervalI*: $\text{abs } a \in_r \text{abs-interval } A$ **if** $a \in_r A$
 $\langle \text{proof} \rangle$

lemma *interval-of[intro,simp]*: $x \in_r \text{interval-of } x$
 $\langle \text{proof} \rangle$

lemma *split-float-interval-realD*: $\text{split-float-interval } X = (A, B) \implies x \in_r X \implies x \in_r A \vee x \in_r B$
 $\langle \text{proof} \rangle$

48.3 bounds for lists

lemma *lower-Interval*: $\text{lower } (\text{Interval } x) = \text{fst } x$
and *upper-Interval*: $\text{upper } (\text{Interval } x) = \text{snd } x$
if $\text{fst } x \leq \text{snd } x$
 $\langle \text{proof} \rangle$

definition *all-in-i* :: $'a::\text{preorder list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool}$
 $(\text{infix } (\text{all}'\text{-in}_i) \ 50)$
where $x \text{ all-in}_i I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_i I ! i))$

definition *all-in* :: $\text{real list} \Rightarrow \text{float interval list} \Rightarrow \text{bool}$
 $(\text{infix } (\text{all}'\text{-in}) \ 50)$
where $x \text{ all-in } I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_r I ! i))$

definition *all-subset* :: $'a::\text{order interval list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool}$
 $(\text{infix } (\text{all}'\text{-subset}) \ 50)$
where $I \text{ all-subset } J = (\text{length } I = \text{length } J \wedge (\forall i < \text{length } I. \text{set-of } (I!i) \subseteq \text{set-of } (J!i)))$

lemmas $[\text{simp}] = \text{all-in-def all-subset-def}$

lemma *all-subsetD*:
assumes $I \text{ all-subset } J$
assumes $x \text{ all-in } I$
shows $x \text{ all-in } J$

<proof>

lemma *round-interval-mono*: *set-of (round-interval prec X) ⊆ set-of (round-interval prec Y)*
if *set-of X ⊆ 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 ⊆ set-of Y ⟷ lower Y ≤ lower X ∧ upper X ≤ 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) = 2ⁿ*
<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 (λi. x ∈_r i) (subdivide-interval n I) if x ∈_r I for x::real
<proof>


```

fun interval-list-union :: 'a::lattice interval list  $\Rightarrow$  '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  $\neq$  []
assumes i < length S
shows set-of (S!i)  $\subseteq$  set-of (interval-list-union S)
  <proof>

```

lemma split-domain-correct:

```

fixes x :: real list
assumes x all-in I
assumes split-correct:  $\bigwedge x \ a \ I. x \in_r I \implies \text{list-ex } (\lambda i::\text{float interval}. x \in_r i) \ (\text{split } I)$ 
shows list-ex ( $\lambda s. x \text{ all-in } s$ ) (split-domain split I)
  <proof>

```

lift-definition(code-dt) inverse-float-interval::nat $\Rightarrow$ float interval $\Rightarrow$ float interval option **is**

```

 $\lambda \text{prec } (l, u). \text{ if } (0 < l \vee u < 0) \text{ then Some (float-divl prec 1 } u, \text{ float-divr prec 1 } l) \text{ else None}$ 
  <proof>

```

lemma inverse-float-interval-eq-Some-conv:

```

defines one  $\equiv$  (1::float)
shows
  inverse-float-interval p X = Some R  $\longleftrightarrow$ 
    (lower X > 0  $\vee$  upper X < 0)  $\wedge$ 
    lower R = float-divl p one (upper X)  $\wedge$ 
    upper R = float-divr p one (lower X)
  <proof>

```

lemma inverse-float-interval:

```

inverse ' set-of (real-interval X)  $\subseteq$  set-of (real-interval Y)
if inverse-float-interval p X = Some Y
  <proof>

```

lemma inverse-float-intervalI:

```

 $x \in_r X \implies \text{inverse } x \in \text{set-of}' (\text{inverse-float-interval } p \ X)$ 
  <proof>

```

lemma inverse-float-interval-eqI: inverse-float-interval p X = Some IVL $\implies x \in_r$

```

X  $\implies \text{inverse } x \in_r \text{ IVL}$ 
  <proof>

```

lemma *real-interval-abs-interval*[simp]:
 $\text{real-interval } (\text{abs-interval } x) = \text{abs-interval } (\text{real-interval } x)$
 ⟨proof⟩

lift-definition *floor-float-interval*::float interval $\Rightarrow$ float interval **is**
 $\lambda(l, u). (\text{floor-fl } l, \text{floor-fl } u)$
 ⟨proof⟩

lemma *lower-floor-float-interval*[simp]: $\text{lower } (\text{floor-float-interval } x) = \text{floor-fl } (\text{lower } x)$
 ⟨proof⟩

lemma *upper-floor-float-interval*[simp]: $\text{upper } (\text{floor-float-interval } x) = \text{floor-fl } (\text{upper } x)$
 ⟨proof⟩

lemma *floor-float-intervalI*: $\lfloor x \rfloor \in_r \text{floor-float-interval } X$ **if** $x \in_r X$
 ⟨proof⟩

end

48.4 constants for code generation

definition *lowerF*::float interval $\Rightarrow$ float **where** *lowerF* = *lower*

definition *upperF*::float interval $\Rightarrow$ float **where** *upperF* = *upper*

end

49 Immutable Arrays with Code Generation

theory *IArray*

imports *Main*

begin

49.1 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 $\Rightarrow$ 'a list **where**

list-of (*IArray* *xs*) = *xs*

qualified definition *of-fun* :: (*nat* $\Rightarrow$ '*a*) $\Rightarrow$ *nat* $\Rightarrow$ '*a* iarray **where**
 [simp]: *of-fun* *f* *n* = *IArray* (map *f* [0..*n*])

qualified definition *sub* :: '*a* iarray $\Rightarrow$ *nat* $\Rightarrow$ '*a* (infixl !! 100) **where**
 [simp]: *as* !! *n* = *IArray.list-of* *as* ! *n*

qualified definition *length* :: '*a* iarray $\Rightarrow$ *nat* **where**
 [simp]: *length* *as* = *List.length* (*IArray.list-of* *as*)

qualified definition *all* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a* iarray $\Rightarrow$ *bool* **where**
 [simp]: *all* *p* *as* $\longleftrightarrow$ ($\forall a \in \text{set } (\text{list-of } as). p\ a$)

qualified definition *exists* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a* iarray $\Rightarrow$ *bool* **where**
 [simp]: *exists* *p* *as* $\longleftrightarrow$ ($\exists a \in \text{set } (\text{list-of } as). p\ a$)

lemma *of-fun-nth*:
IArray.of-fun *f* *n* !! *i* = *f* *i* **if** *i* < *n*
 <proof>

end

49.2 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*)

$\langle \text{proof} \rangle$

lemma *list-of-code* [code]:

$IArray.\text{list-of } as = \text{map } (\lambda n. as !! n) [0 ..< IArray.\text{length } as]$

$\langle \text{proof} \rangle$

lemma [code]:

$HOL.\text{equal } as \ bs \longleftrightarrow HOL.\text{equal } (IArray.\text{list-of } as) (IArray.\text{list-of } bs)$

$\langle \text{proof} \rangle$

lemma [code]:

$IArray.\text{all } p = \text{Not} \circ IArray.\text{exists } (\text{Not} \circ p)$

$\langle \text{proof} \rangle$

context

includes *term-syntax*

begin

lemma [code]:

$\text{Code-Evaluation.term-of } (as :: 'a::\text{typerep } iarray) =$

$\text{Code-Evaluation.Const } (STR \text{ "IArray.iarray.IArray" }) (TYPEREP('a \text{ list} \Rightarrow 'a \text{ iarray})) <\cdot> (\text{Code-Evaluation.term-of } (IArray.\text{list-of } as))$

$\langle \text{proof} \rangle$

end

49.3 Auxiliary operations for code generation

context

begin

qualified primrec *tabulate* :: $\text{integer} \times (\text{integer} \Rightarrow 'a) \Rightarrow 'a \text{ iarray}$ **where**

$\text{tabulate } (n, f) = IArray (\text{map } (f \circ \text{integer-of-nat}) [0..<\text{nat-of-integer } n])$

lemma [code]:

$IArray.\text{of-fun } f \ n = IArray.\text{tabulate } (\text{integer-of-nat } n, f \circ \text{nat-of-integer})$

$\langle \text{proof} \rangle$ **primrec** *sub'* :: $'a \text{ iarray} \times \text{integer} \Rightarrow 'a$ **where**

$\text{sub'} } (as, n) = as !! \text{nat-of-integer } n$

lemma [code]:

$IArray.\text{sub'} } (IArray \ as, n) = as ! \text{nat-of-integer } n$

$\langle \text{proof} \rangle$

lemma [code]:

$as !! n = IArray.\text{sub'} } (as, \text{integer-of-nat } n)$

$\langle \text{proof} \rangle$ **definition** *length'* :: $'a \text{ iarray} \Rightarrow \text{integer}$ **where**

[simp]: $\text{length'} } as = \text{integer-of-nat } (List.\text{length } (IArray.\text{list-of } as))$

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*⟩

lemma [*code*]:

IArray.exists p as ⇔ *exists-upto p* (*length' as*) *as*
including *integer.lifting* ⟨*proof*⟩

end

49.4 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*

49.5 Code Generation for Haskell

We map 'a iarrays 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' IAr-
ray.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

50 Definition of Landau symbols

theory *Landau-Symbols*

imports

Complex-Main

begin

lemma *eventually-subst'*:

```

  eventually (λx. f x = g x) F ==> eventually (λx. P x (f x)) F = eventually (λx.
P x (g x)) F
  <proof>

```

50.1 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 $\Rightarrow$ ('a $\Rightarrow$ ('b :: real-normed-field)) $\Rightarrow$ ('a $\Rightarrow$ 'b) set
 $(\langle (1O[-]'(-)) \rangle)$
where *bigo* $F g = \{f. (\exists c > 0. \text{eventually } (\lambda x. \text{norm } (f x) \leq c * \text{norm } (g x)) F)\}$

definition *smallo* :: 'a filter $\Rightarrow$ ('a $\Rightarrow$ ('b :: real-normed-field)) $\Rightarrow$ ('a $\Rightarrow$ 'b) set
 $(\langle (1o[-]'(-)) \rangle)$
where *smallo* $F g = \{f. (\forall c > 0. \text{eventually } (\lambda x. \text{norm } (f x) \leq c * \text{norm } (g x)) F)\}$

definition *bigomega* :: 'a filter $\Rightarrow$ ('a $\Rightarrow$ ('b :: real-normed-field)) $\Rightarrow$ ('a $\Rightarrow$ 'b) set
 $(\langle (1\Omega[-]'(-)) \rangle)$
where *bigomega* $F g = \{f. (\exists c > 0. \text{eventually } (\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x)) F)\}$

definition *smallomega* :: 'a filter $\Rightarrow$ ('a $\Rightarrow$ ('b :: real-normed-field)) $\Rightarrow$ ('a $\Rightarrow$ 'b) set
 $(\langle (1\omega[-]'(-)) \rangle)$
where *smallomega* $F g = \{f. (\forall c > 0. \text{eventually } (\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x)) F)\}$

definition *bigtheta* :: 'a filter $\Rightarrow$ ('a $\Rightarrow$ ('b :: real-normed-field)) $\Rightarrow$ ('a $\Rightarrow$ 'b) set
 $(\langle (1\Theta[-]'(-)) \rangle)$
where *bigtheta* $F g = \text{bigo } F g \cap \text{bigomega } F g$

abbreviation *bigo-at-top* $(\langle (2O'(-)) \rangle)$ **where**
 $O(g) \equiv \text{bigo at-top } g$

abbreviation *smallo-at-top* $(\langle (2o'(-)) \rangle)$ **where**
 $o(g) \equiv \text{smallo at-top } g$

abbreviation *bigomega-at-top* $(\langle (2\Omega'(-)) \rangle)$ **where**
 $\Omega(g) \equiv \text{bigomega at-top } g$

abbreviation *smallomega-at-top* $(\langle (2\omega'(-)) \rangle)$ **where**
 $\omega(g) \equiv \text{smallomega at-top } g$

abbreviation *bigtheta-at-top* $(\langle (2\Theta'(-)) \rangle)$ **where**
 $\Theta(g) \equiv \text{bigtheta at-top } g$

The following is a set of properties that all Landau symbols satisfy.

named-theorems *landau-divide-simps*

locale *landau-symbol* =

fixes $L :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
and $L' :: 'c \text{ filter} \Rightarrow ('c \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('c \Rightarrow 'b) \text{ set}$
and $Lr :: 'a \text{ filter} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real}) \text{ set}$
assumes $\text{bot}' : L \text{ bot } f = \text{UNIV}$
assumes $\text{filter-mono}' : F1 \leq F2 \Longrightarrow L F2 f \subseteq L F1 f$
assumes in-filtermap-iff :
 $f' \in L (\text{filtermap } h' F') g' \longleftrightarrow (\lambda x. f' (h' x)) \in L' F' (\lambda x. g' (h' x))$
assumes filtercomap :
 $f' \in L F'' g' \Longrightarrow (\lambda x. f' (h' x)) \in L' (\text{filtercomap } h' F'') (\lambda x. g' (h' x))$
assumes sup : $f \in L F1 g \Longrightarrow f \in L F2 g \Longrightarrow f \in L (\text{sup } F1 F2) g$
assumes in-cong : $\text{eventually } (\lambda x. f x = g x) F \Longrightarrow f \in L F (h) \longleftrightarrow g \in L F (h)$
(h)
assumes cong : $\text{eventually } (\lambda x. f x = g x) F \Longrightarrow L F (f) = L F (g)$
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 $\text{cmult-in-iff [simp]}$: $c \neq 0 \Longrightarrow (\lambda x. c * f x) \in L F (g) \longleftrightarrow f \in L F (g)$
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 : $\text{eventually } (\lambda x. f x \neq 0) F \Longrightarrow \text{eventually } (\lambda x. g x \neq 0) F$
 $\Longrightarrow$
 $f \in L F (g) \Longrightarrow (\lambda x. \text{inverse } (g x)) \in L F (\lambda x. \text{inverse } (f x))$
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 : $f \in L F (g) \Longrightarrow \text{filterlim } h' F G \Longrightarrow (\lambda x. f (h' x)) \in L' G$
 $(\lambda x. g (h' x))$
assumes norm-iff [simp] : $(\lambda x. \text{norm } (f x)) \in Lr F (\lambda x. \text{norm } (g x)) \longleftrightarrow f \in L F (g)$
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] : $f \in L \text{ bot } g \langle \text{proof} \rangle$

lemma filter-mono : $F1 \leq F2 \Longrightarrow f \in L F2 g \Longrightarrow f \in L F1 g$
 $\langle \text{proof} \rangle$

lemma cong-ex :
 $\text{eventually } (\lambda x. f1 x = f2 x) F \Longrightarrow \text{eventually } (\lambda x. g1 x = g2 x) F \Longrightarrow$
 $f1 \in L F (g1) \longleftrightarrow f2 \in L F (g2)$
 $\langle \text{proof} \rangle$

lemma cong-ex-bigtheta :
 $f1 \in \Theta[F](f2) \Longrightarrow g1 \in \Theta[F](g2) \Longrightarrow f1 \in L F (g1) \longleftrightarrow f2 \in L F (g2)$
 $\langle \text{proof} \rangle$

lemma bigtheta-trans1 :
 $f \in L F (g) \Longrightarrow g \in \Theta[F](h) \Longrightarrow f \in L F (h)$
 $\langle \text{proof} \rangle$

lemma *bigheta-trans2*:

$$f \in \Theta[F](g) \implies g \in L F (h) \implies f \in L F (h)$$

<proof>

lemma *cmult'* [simp]: $c \neq 0 \implies L F (\lambda x. f x * c) = L F (f)$

<proof>

lemma *cmult-in-iff'* [simp]: $c \neq 0 \implies (\lambda x. f x * c) \in L F (g) \longleftrightarrow f \in L F (g)$

<proof>

lemma *cdiv* [simp]: $c \neq 0 \implies L F (\lambda x. f x / c) = L F (f)$

<proof>

lemma *cdiv-in-iff'* [simp]: $c \neq 0 \implies (\lambda x. f x / c) \in L F (g) \longleftrightarrow f \in L F (g)$

<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*: $c \neq 0 \implies L F (\lambda-. c) = L F (\lambda-. 1)$

<proof>

lemma *const'* [simp]: *NO-MATCH* $1 c \implies c \neq 0 \implies L F (\lambda-. c) = L F (\lambda-. 1)$

<proof>

lemma *const-in-iff'*: $c \neq 0 \implies (\lambda-. c) \in L F (f) \longleftrightarrow (\lambda-. 1) \in L F (f)$

<proof>

lemma *const-in-iff'* [simp]: *NO-MATCH* $1 c \implies c \neq 0 \implies (\lambda-. c) \in L F (f) \longleftrightarrow (\lambda-. 1) \in L F (f)$

<proof>

lemma *plus-subset2*: $g \in o[F](f) \implies L F (f) \subseteq L F (\lambda x. f x + g x)$

<proof>

lemma *mult-right* [simp]: $f \in L F (g) \implies (\lambda x. f x * h x) \in L F (\lambda x. g x * h x)$

<proof>

lemma *mult*: $f1 \in L F (g1) \implies f2 \in L F (g2) \implies (\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x)$

<proof>

lemma *inverse-cancel*:

assumes *eventually* $(\lambda x. f x \neq 0) F$
assumes *eventually* $(\lambda x. g x \neq 0) F$

shows $(\lambda x. \text{inverse } (f x)) \in L F (\lambda x. \text{inverse } (g x)) \longleftrightarrow g \in L F (f)$
 $\langle \text{proof} \rangle$

lemma *divide-right*:

assumes *eventually* $(\lambda x. h x \neq 0) F$
assumes $f \in L F (g)$
shows $(\lambda x. f x / h x) \in L F (\lambda x. g x / h x)$
 $\langle \text{proof} \rangle$

lemma *divide-right-iff*:

assumes *eventually* $(\lambda x. h x \neq 0) F$
shows $(\lambda x. f x / h x) \in L F (\lambda x. g x / h x) \longleftrightarrow f \in L F (g)$
 $\langle \text{proof} \rangle$

lemma *divide-left*:

assumes *eventually* $(\lambda x. f x \neq 0) F$
assumes *eventually* $(\lambda x. g x \neq 0) F$
assumes $g \in L F (f)$
shows $(\lambda x. h x / f x) \in L F (\lambda x. h x / g x)$
 $\langle \text{proof} \rangle$

lemma *divide-left-iff*:

assumes *eventually* $(\lambda x. f x \neq 0) F$
assumes *eventually* $(\lambda x. g x \neq 0) F$
assumes *eventually* $(\lambda x. h x \neq 0) F$
shows $(\lambda x. h x / f x) \in L F (\lambda x. h x / g x) \longleftrightarrow g \in L F (f)$
 $\langle \text{proof} \rangle$

lemma *divide*:

assumes *eventually* $(\lambda x. g1 x \neq 0) F$
assumes *eventually* $(\lambda x. g2 x \neq 0) F$
assumes $f1 \in L F (f2) \ g2 \in L F (g1)$
shows $(\lambda x. f1 x / g1 x) \in L F (\lambda x. f2 x / g2 x)$
 $\langle \text{proof} \rangle$

lemma *divide-eq1*:

assumes *eventually* $(\lambda x. h x \neq 0) F$
shows $f \in L F (\lambda x. g x / h x) \longleftrightarrow (\lambda x. f x * h x) \in L F (g)$
 $\langle \text{proof} \rangle$

lemma *divide-eq2*:

assumes *eventually* $(\lambda x. h x \neq 0) F$
shows $(\lambda x. f x / h x) \in L F (\lambda x. g x) \longleftrightarrow f \in L F (\lambda x. g x * h x)$
 $\langle \text{proof} \rangle$

lemma *inverse-eq1*:

assumes *eventually* $(\lambda x. g x \neq 0) F$
shows $f \in L F (\lambda x. \text{inverse } (g x)) \longleftrightarrow (\lambda x. f x * g x) \in L F (\lambda x. 1)$
 $\langle \text{proof} \rangle$

lemma *inverse-eq2*:

assumes *eventually* $(\lambda x. f\ x \neq 0)\ F$
shows $(\lambda x. \text{inverse}\ (f\ x)) \in L\ F\ (g) \longleftrightarrow (\lambda x. 1) \in L\ F\ (\lambda x. f\ x * g\ x)$
 $\langle \text{proof} \rangle$

lemma *inverse-flip*:

assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$
assumes $(\lambda x. \text{inverse}\ (g\ x)) \in L\ F\ (h)$
shows $(\lambda x. \text{inverse}\ (h\ x)) \in L\ F\ (g)$
 $\langle \text{proof} \rangle$

lemma *lift-trans*:

assumes $f \in L\ F\ (g)$
assumes $(\lambda x. t\ x\ (g\ x)) \in L\ F\ (h)$
assumes $\bigwedge f\ g. f \in L\ F\ (g) \implies (\lambda x. t\ x\ (f\ x)) \in L\ F\ (\lambda x. t\ x\ (g\ x))$
shows $(\lambda x. t\ x\ (f\ x)) \in L\ F\ (h)$
 $\langle \text{proof} \rangle$

lemma *lift-trans'*:

assumes $f \in L\ F\ (\lambda x. t\ x\ (g\ x))$
assumes $g \in L\ F\ (h)$
assumes $\bigwedge g\ h. g \in L\ F\ (h) \implies (\lambda x. t\ x\ (g\ x)) \in L\ F\ (\lambda x. t\ x\ (h\ x))$
shows $f \in L\ F\ (\lambda x. t\ x\ (h\ x))$
 $\langle \text{proof} \rangle$

lemma *lift-trans-bigtheta*:

assumes $f \in L\ F\ (g)$
assumes $(\lambda x. t\ x\ (g\ x)) \in \Theta[F](h)$
assumes $\bigwedge f\ g. f \in L\ F\ (g) \implies (\lambda x. t\ x\ (f\ x)) \in L\ F\ (\lambda x. t\ x\ (g\ x))$
shows $(\lambda x. t\ x\ (f\ x)) \in L\ F\ (h)$
 $\langle \text{proof} \rangle$

lemma *lift-trans-bigtheta'*:

assumes $f \in L\ F\ (\lambda x. t\ x\ (g\ x))$
assumes $g \in \Theta[F](h)$
assumes $\bigwedge g\ h. g \in \Theta[F](h) \implies (\lambda x. t\ x\ (g\ x)) \in \Theta[F](\lambda x. t\ x\ (h\ x))$
shows $f \in L\ F\ (\lambda x. t\ x\ (h\ x))$
 $\langle \text{proof} \rangle$

lemma (in *landau-symbol*) *mult-in-1*:

assumes $f \in L\ F\ (\lambda-. 1)\ g \in L\ F\ (\lambda-. 1)$
shows $(\lambda x. f\ x * g\ x) \in L\ F\ (\lambda-. 1)$
 $\langle \text{proof} \rangle$

lemma (in *landau-symbol*) *of-real-cancel*:

$(\lambda x. \text{of-real}\ (f\ x)) \in L\ F\ (\lambda x. \text{of-real}\ (g\ x)) \implies f \in Lr\ F\ g$
 $\langle \text{proof} \rangle$

lemma (in *landau-symbol*) *of-real-iff*:

$(\lambda x. \text{of-real } (f\ x)) \in L\ F\ (\lambda x. \text{of-real } (g\ x)) \longleftrightarrow f \in Lr\ F\ g$
 $\langle \text{proof} \rangle$

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 $L\ l :: 'a\ \text{filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b)\ \text{set}$
fixes $L'\ l' :: 'c\ \text{filter} \Rightarrow ('c \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('c \Rightarrow 'b)\ \text{set}$
fixes $Lr\ lr :: 'a\ \text{filter} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real})\ \text{set}$
and $R :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}$
assumes $L\text{-def}: L\ F\ g = \{f. \exists c>0. \text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g\ x)))\ F\}$
and $l\text{-def}: l\ F\ g = \{f. \forall c>0. \text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g\ x)))\ F\}$
and $L'\text{-def}: L'\ F'\ g' = \{f. \exists c>0. \text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g'\ x)))\ F'\}$
and $l'\text{-def}: l'\ F'\ g' = \{f. \forall c>0. \text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g'\ x)))\ F'\}$
and $Lr\text{-def}: Lr\ F''\ g'' = \{f. \exists c>0. \text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g''\ x)))\ F''\}$
and $lr\text{-def}: lr\ F''\ g'' = \{f. \forall c>0. \text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g''\ x)))\ F''\}$
and $R: R = (\leq) \vee R = (\geq)$

interpretation *landau-o*:

landau-pair bigo smallo bigo smallo bigo smallo $(\leq)$
 $\langle \text{proof} \rangle$

interpretation *landau-omega*:

landau-pair bigomega smallomega bigomega smallomega bigomega smallomega
 $(\geq)$
 $\langle \text{proof} \rangle$

context *landau-pair*

begin

lemmas $R\text{-E} = \text{disjE } [OF\ R, \text{case-names } le\ ge]$

lemma *bigI*:

$c > 0 \implies \text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g\ x)))\ F \implies f \in L\ F\ (g)$

$\langle proof \rangle$

lemma *bigE*:

assumes $f \in L F (g)$

obtains c **where** $c > 0$ *eventually* $(\lambda x. R (norm (f x)) (c * (norm (g x)))) F$

$\langle proof \rangle$

lemma *smallI*:

$(\bigwedge c. c > 0 \implies \text{eventually } (\lambda x. R (norm (f x)) (c * (norm (g x)))) F) \implies f \in l F (g)$

$\langle proof \rangle$

lemma *smallD*:

$f \in l F (g) \implies c > 0 \implies \text{eventually } (\lambda x. R (norm (f x)) (c * (norm (g x)))) F$

$\langle proof \rangle$

lemma *bigE-nonneg-real*:

assumes $f \in Lr F (g)$ *eventually* $(\lambda x. f x \geq 0) F$

obtains c **where** $c > 0$ *eventually* $(\lambda x. R (f x) (c * |g x|)) F$

$\langle proof \rangle$

lemma *smallD-nonneg-real*:

assumes $f \in lr F (g)$ *eventually* $(\lambda x. f x \geq 0) F$ $c > 0$

shows *eventually* $(\lambda x. R (f x) (c * |g x|)) F$

$\langle proof \rangle$

lemma *small-imp-big*: $f \in l F (g) \implies f \in L F (g)$

$\langle proof \rangle$

lemma *small-subset-big*: $l F (g) \subseteq L F (g)$

$\langle proof \rangle$

lemma *R-refl [simp]*: $R x x$ $\langle proof \rangle$

lemma *R-linear*: $\neg R x y \implies R y x$

$\langle proof \rangle$

lemma *R-trans [trans]*: $R a b \implies R b c \implies R a c$

$\langle proof \rangle$

lemma *R-mult-left-mono*: $R a b \implies c \geq 0 \implies R (c*a) (c*b)$

$\langle proof \rangle$

lemma *R-mult-right-mono*: $R a b \implies c \geq 0 \implies R (a*c) (b*c)$

$\langle proof \rangle$

lemma *big-trans*:

assumes $f \in L F (g)$ $g \in L F (h)$

shows $f \in L F (h)$

$\langle proof \rangle$

lemma *big-small-trans*:

assumes $f \in L\ F\ (g)\ g \in l\ F\ (h)$

shows $f \in l\ F\ (h)$

$\langle proof \rangle$

lemma *small-big-trans*:

assumes $f \in l\ F\ (g)\ g \in L\ F\ (h)$

shows $f \in l\ F\ (h)$

$\langle proof \rangle$

lemma *small-trans*:

$f \in l\ F\ (g) \implies g \in l\ F\ (h) \implies f \in l\ F\ (h)$

$\langle proof \rangle$

lemma *small-big-trans'*:

$f \in l\ F\ (g) \implies g \in L\ F\ (h) \implies f \in L\ F\ (h)$

$\langle proof \rangle$

lemma *big-small-trans'*:

$f \in L\ F\ (g) \implies g \in l\ F\ (h) \implies f \in L\ F\ (h)$

$\langle proof \rangle$

lemma *big-subsetI* [intro]: $f \in L\ F\ (g) \implies L\ F\ (f) \subseteq L\ F\ (g)$

$\langle proof \rangle$

lemma *small-subsetI* [intro]: $f \in L\ F\ (g) \implies l\ F\ (f) \subseteq l\ F\ (g)$

$\langle proof \rangle$

lemma *big-refl* [simp]: $f \in L\ F\ (f)$

$\langle proof \rangle$

lemma *small-refl-iff*: $f \in l\ F\ (f) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0)\ F$

$\langle proof \rangle$

lemma *big-small-asymmetric*: $f \in L\ F\ (g) \implies g \in l\ F\ (f) \implies \text{eventually } (\lambda x. f\ x = 0)\ F$

$\langle proof \rangle$

lemma *small-big-asymmetric*: $f \in l\ F\ (g) \implies g \in L\ F\ (f) \implies \text{eventually } (\lambda x. f\ x = 0)\ F$

$\langle proof \rangle$

lemma *small-asymmetric*: $f \in l\ F\ (g) \implies g \in l\ F\ (f) \implies \text{eventually } (\lambda x. f\ x = 0)\ F$

$\langle proof \rangle$

lemma *plus-aux*:
 assumes $f \in o[F](g)$
 shows $g \in L F (\lambda x. f x + g x)$
 $\langle proof \rangle$

end

lemma *bigomega-iff-bigo*: $g \in \Omega[F](f) \longleftrightarrow f \in O[F](g)$
 $\langle proof \rangle$

lemma *smallomega-iff-smalllo*: $g \in \omega[F](f) \longleftrightarrow f \in o[F](g)$
 $\langle proof \rangle$

context *landau-pair*
begin

lemma *big-mono*:
 eventually $(\lambda x. R (norm (f x)) (norm (g x))) F \implies f \in L F (g)$
 $\langle proof \rangle$

lemma *big-mult*:
 assumes $f1 \in L F (g1)$ $f2 \in L F (g2)$
 shows $(\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x)$
 $\langle proof \rangle$

lemma *small-big-mult*:
 assumes $f1 \in l F (g1)$ $f2 \in L F (g2)$
 shows $(\lambda x. f1 x * f2 x) \in l F (\lambda x. g1 x * g2 x)$
 $\langle proof \rangle$

lemma *big-small-mult*:
 $f1 \in L F (g1) \implies f2 \in l F (g2) \implies (\lambda x. f1 x * f2 x) \in l F (\lambda x. g1 x * g2 x)$
 $\langle proof \rangle$

lemma *small-mult*: $f1 \in l F (g1) \implies f2 \in l F (g2) \implies (\lambda x. f1 x * f2 x) \in l F (\lambda x. g1 x * g2 x)$
 $\langle proof \rangle$

lemmas *mult = big-mult small-big-mult big-small-mult small-mult*

sublocale *big*: *landau-symbol* $L L' Lr$
 $\langle proof \rangle$

sublocale *small*: *landau-symbol* $l l' lr$
 $\langle proof \rangle$

These rules allow chaining of Landau symbol propositions in Isar with

"also".

lemma *big-mult-1*: $f \in L F (g) \implies (\lambda-. 1) \in L F (h) \implies f \in L F (\lambda x. g x * h x)$
and *big-mult-1'*: $(\lambda-. 1) \in L F (g) \implies f \in L F (h) \implies f \in L F (\lambda x. g x * h x)$
and *small-mult-1*: $f \in l F (g) \implies (\lambda-. 1) \in L F (h) \implies f \in l F (\lambda x. g x * h x)$
and *small-mult-1'*: $(\lambda-. 1) \in L F (g) \implies f \in l F (h) \implies f \in l F (\lambda x. g x * h x)$
and *small-mult-1''*: $f \in L F (g) \implies (\lambda-. 1) \in l F (h) \implies f \in l F (\lambda x. g x * h x)$
and *small-mult-1'''*: $(\lambda-. 1) \in l F (g) \implies f \in L F (h) \implies f \in l F (\lambda x. g x * h x)$
 ⟨proof⟩

lemma *big-1-mult*: $f \in L F (g) \implies h \in L F (\lambda-. 1) \implies (\lambda x. f x * h x) \in L F (g)$
and *big-1-mult'*: $h \in L F (\lambda-. 1) \implies f \in L F (g) \implies (\lambda x. f x * h x) \in L F (g)$
and *small-1-mult*: $f \in l F (g) \implies h \in L F (\lambda-. 1) \implies (\lambda x. f x * h x) \in l F (g)$
and *small-1-mult'*: $h \in L F (\lambda-. 1) \implies f \in l F (g) \implies (\lambda x. f x * h x) \in l F (g)$
and *small-1-mult''*: $f \in L F (g) \implies h \in l F (\lambda-. 1) \implies (\lambda x. f x * h x) \in l F (g)$
and *small-1-mult'''*: $h \in l F (\lambda-. 1) \implies f \in L F (g) \implies (\lambda x. f x * h x) \in l F (g)$
 ⟨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 $\bigwedge x. x \in A \implies f x \in L F (g x)$
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 $\bigwedge x. x \in A \implies f x \in L F (\lambda-. 1)$
shows $(\lambda y. \prod x \in A. f x y) \in L F (\lambda-. 1)$
 ⟨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*: $g \in o[F](f) \implies L\ F\ (\lambda x. f\ x + g\ x) = L\ F\ (f)$

<proof>

lemma *diff-absorb1*: $f \in o[F](g) \implies L\ F\ (\lambda x. f\ x - g\ x) = L\ F\ (g)$

<proof>

lemma *diff-absorb2*: $g \in o[F](f) \implies L\ F\ (\lambda x. f\ x - g\ x) = L\ F\ (f)$

<proof>

lemmas *absorb* = *plus-absorb1 plus-absorb2 diff-absorb1 diff-absorb2*

end

lemma *bigthetaI* [*intro*]: $f \in O[F](g) \implies f \in \Omega[F](g) \implies f \in \Theta[F](g)$

<proof>

lemma *bigthetaD1* [*dest*]: $f \in \Theta[F](g) \implies f \in O[F](g)$

and *bigthetaD2* [*dest*]: $f \in \Theta[F](g) \implies f \in \Omega[F](g)$

<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 *bigO* [*intro*]:

assumes *eventually* $(\lambda x. (norm\ (f\ x)) \leq c * (norm\ (g\ x)))\ F$

shows $f \in O[F](g)$

<proof>

lemma *smallomegaD* [*dest*]:

assumes $f \in \omega[F](g)$

shows *eventually* $(\lambda x. (\text{norm } (f x)) \geq c * (\text{norm } (g x))) F$

<proof>

lemma *bighetaI'*:

assumes $c1 > 0$ $c2 > 0$

assumes *eventually* $(\lambda x. c1 * (\text{norm } (g x)) \leq (\text{norm } (f x)) \wedge (\text{norm } (f x)) \leq c2 * (\text{norm } (g x))) F$

shows $f \in \Theta[F](g)$

<proof>

lemma *bighetaI-cong*: *eventually* $(\lambda x. f x = g x) F \implies f \in \Theta[F](g)$

<proof>

lemma (*in landau-symbol*) *ev-eq-trans1*:

$f \in L F (\lambda x. g x (h x)) \implies \text{eventually } (\lambda x. h x = h' x) F \implies f \in L F (\lambda x. g x (h' x))$

<proof>

lemma (*in landau-symbol*) *ev-eq-trans2*:

eventually $(\lambda x. f x = f' x) F \implies (\lambda x. g x (f' x)) \in L F (h) \implies (\lambda x. g x (f x)) \in L F (h)$

<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*) *bigheta-trans1'*:

$f \in L F (g) \implies h \in \Theta[F](g) \implies f \in L F (h)$

<proof>

lemma (*in landau-symbol*) *bigheta-trans2'*:

$g \in \Theta[F](f) \implies g \in L F (h) \implies f \in L F (h)$

<proof>

lemma *bigo-bigomega-trans*: $f \in O[F](g) \implies h \in \Omega[F](g) \implies f \in O[F](h)$

and *bigo-smallomega-trans*: $f \in O[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$

and *smallo-bigomega-trans*: $f \in o[F](g) \implies h \in \Omega[F](g) \implies f \in o[F](h)$

and *smallo-smallomega-trans*: $f \in o[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$

and *bigomega-bigo-trans*: $f \in \Omega[F](g) \implies h \in O[F](g) \implies f \in \Omega[F](h)$

and *bigomega-smallo-trans*: $f \in \Omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$

and *smallomega-bigo-trans*: $f \in \omega[F](g) \implies h \in O[F](g) \implies f \in \omega[F](h)$

and *smallomega-smallo-trans*: $f \in \omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$

<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 $(\lambda x. \text{inverse } (f \ x)) \in \Theta[F](\lambda x. \text{inverse } (g \ x)) \longleftrightarrow f \in \Theta[F](g)$
 <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 $\text{eventually } (\lambda x. f \ x \neq 0) \ F \longleftrightarrow \text{eventually } (\lambda x. g \ x \neq 0) \ F$
 <proof>

50.2 Landau symbols and limits

lemma *bigoI-tendsto-norm*:
fixes $f \ g$
assumes $((\lambda x. \text{norm } (f \ x / g \ x)) \longrightarrow c) \ F$
assumes $\text{eventually } (\lambda x. g \ x \neq 0) \ F$
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 $f \in O[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigomegaI-tendsto-norm*:
assumes *c-not-0*: $(c::\text{real}) \neq 0$
assumes *lim*: $((\lambda x. \text{norm}\ (f\ x / g\ x)) \longrightarrow c)\ F$
shows $f \in \Omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigomegaI-tendsto*:
assumes *c-not-0*: $(c::\text{real}) \neq 0$
assumes *lim*: $((\lambda x. f\ x / g\ x) \longrightarrow c)\ F$
shows $f \in \Omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallomegaI-filterlim-at-top-norm*:
assumes *lim*: *filterlim* $(\lambda x. \text{norm}\ (f\ x / g\ x))\ \text{at-top}\ F$
shows $f \in \omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallomegaI-filterlim-at-infinity*:
assumes *lim*: *filterlim* $(\lambda x. f\ x / g\ x)\ \text{at-infinity}\ F$
shows $f \in \omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallomegaD-filterlim-at-top-norm*:
assumes $f \in \omega[F](g)$
assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
shows $\text{LIM } x\ F. \text{norm}\ (f\ x / g\ x) :> \text{at-top}$
 $\langle \text{proof} \rangle$

lemma *smallomegaD-filterlim-at-infinity*:
assumes $f \in \omega[F](g)$
assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
shows $\text{LIM } x\ F. f\ x / g\ x :> \text{at-infinity}$
 $\langle \text{proof} \rangle$

lemma *smallomega-1-conv-filterlim*: $f \in \omega[F](\lambda-. 1) \longleftrightarrow \text{filterlim } f\ \text{at-infinity}\ F$
 $\langle \text{proof} \rangle$

lemma *smalloI-tendsto*:
assumes *lim*: $((\lambda x. f\ x / g\ x) \longrightarrow 0)\ F$
assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
shows $f \in o[F](g)$
 $\langle \text{proof} \rangle$

lemma *smalloD-tendsto*:
assumes $f \in o[F](g)$

shows $((\lambda x. f\ x / g\ x) \longrightarrow 0)\ F$
 $\langle proof \rangle$

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 $f \in \Theta[F](g)$
 $\langle proof \rangle$

lemma *bigthetaI-tendsto*:

assumes *c-not-0*: $(c::real) \neq 0$
assumes *lim*: $((\lambda x. f\ x / g\ x) \longrightarrow c)\ F$
shows $f \in \Theta[F](g)$
 $\langle proof \rangle$

lemma *tendsto-add-smallo*:

assumes $(f1 \longrightarrow a)\ F$
assumes $f2 \in o[F](f1)$
shows $((\lambda x. f1\ x + f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

lemma *tendsto-diff-smallo*:

shows $(f1 \longrightarrow a)\ F \implies f2 \in o[F](f1) \implies ((\lambda x. f1\ x - f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

lemma *tendsto-add-smallo-iff*:

assumes $f2 \in o[F](f1)$
shows $(f1 \longrightarrow a)\ F \longleftrightarrow ((\lambda x. f1\ x + f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

lemma *tendsto-diff-smallo-iff*:

shows $f2 \in o[F](f1) \implies (f1 \longrightarrow a)\ F \longleftrightarrow ((\lambda x. f1\ x - f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

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** $(?f \longrightarrow -)\ -$)
 $\langle proof \rangle$

lemma *bigo-powr*:

fixes $f :: 'a \Rightarrow real$
assumes $f \in O[F](g)\ p \geq 0$
shows $(\lambda x. |f\ x| powr\ p) \in O[F](\lambda x. |g\ x| powr\ p)$
 $\langle proof \rangle$

lemma *smallo-powr*:

fixes $f :: 'a \Rightarrow \text{real}$
assumes $f \in o[F](g) \ p > 0$
shows $(\lambda x. |f\ x| \text{ powr } p) \in o[F](\lambda x. |g\ x| \text{ powr } p)$
 $\langle \text{proof} \rangle$

lemma *smallo-powr-nonneg*:

fixes $f :: 'a \Rightarrow \text{real}$
assumes $f \in o[F](g) \ p > 0 \text{ eventually } (\lambda x. f\ x \geq 0) \ F \text{ eventually } (\lambda x. g\ x \geq 0)$
 F
shows $(\lambda x. f\ x \text{ powr } p) \in o[F](\lambda x. g\ x \text{ powr } p)$
 $\langle \text{proof} \rangle$

lemma *bigheta-powr*:

fixes $f :: 'a \Rightarrow \text{real}$
shows $f \in \Theta[F](g) \implies (\lambda x. |f\ x| \text{ powr } p) \in \Theta[F](\lambda x. |g\ x| \text{ powr } p)$
 $\langle \text{proof} \rangle$

lemma *bigo-powr-nonneg*:

fixes $f :: 'a \Rightarrow \text{real}$
assumes $f \in O[F](g) \ p \geq 0 \text{ eventually } (\lambda x. f\ x \geq 0) \ F \text{ eventually } (\lambda x. g\ x \geq 0)$
 F
shows $(\lambda x. f\ x \text{ powr } p) \in O[F](\lambda x. g\ x \text{ powr } p)$
 $\langle \text{proof} \rangle$

lemma *zero-in-smallo* [simp]: $(\lambda-. 0) \in o[F](f)$
 $\langle \text{proof} \rangle$

lemma *zero-in-bigo* [simp]: $(\lambda-. 0) \in O[F](f)$
 $\langle \text{proof} \rangle$

lemma *in-bigomega-zero* [simp]: $f \in \Omega[F](\lambda x. 0)$
 $\langle \text{proof} \rangle$

lemma *in-smallomega-zero* [simp]: $f \in \omega[F](\lambda x. 0)$
 $\langle \text{proof} \rangle$

lemma *in-smallo-zero-iff* [simp]: $f \in o[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) \ F$
 $\langle \text{proof} \rangle$

lemma *in-bigo-zero-iff* [simp]: $f \in O[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) \ F$
 $\langle \text{proof} \rangle$

lemma *zero-in-smallomega-iff* [simp]: $(\lambda x. 0) \in \omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) \ F$
 $\langle \text{proof} \rangle$

lemma *zero-in-bigomega-iff* [simp]: $(\lambda x. 0) \in \Omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) \ F$

$\langle \text{proof} \rangle$

lemma *zero-in-bigtheta-iff* [simp]: $(\lambda-. 0) \in \Theta[F](f) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0)$
 F
 $\langle \text{proof} \rangle$

lemma *in-bigtheta-zero-iff* [simp]: $f \in \Theta[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0)$
 F
 $\langle \text{proof} \rangle$

lemma *cmult-in-bigo-iff* [simp]: $(\lambda x. c * f\ x) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$
 $O[F](g)$
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]: $(\lambda x. f\ x * c) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigo-const* [simp]: $(\lambda-. c) \in O[F](\lambda-. 1) \langle \text{proof} \rangle$

lemma *bigo-const-iff* [simp]: $(\lambda-. c1) \in O[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 = 0 \vee c2 \neq 0$
 $\langle \text{proof} \rangle$

lemma *bigomega-const-iff* [simp]: $(\lambda-. c1) \in \Omega[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 \neq 0 \vee c2 = 0$
 $\langle \text{proof} \rangle$

lemma *smallo-real-nat-transfer*:
 $(f :: \text{real} \Rightarrow \text{real}) \in o(g) \Longrightarrow (\lambda x::\text{nat}. f\ (\text{real } x)) \in o(\lambda x. g\ (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *bigo-real-nat-transfer*:
 $(f :: \text{real} \Rightarrow \text{real}) \in O(g) \Longrightarrow (\lambda x::\text{nat}. f\ (\text{real } x)) \in O(\lambda x. g\ (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *smallomega-real-nat-transfer*:
 $(f :: \text{real} \Rightarrow \text{real}) \in \omega(g) \Longrightarrow (\lambda x::\text{nat}. f\ (\text{real } x)) \in \omega(\lambda x. g\ (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *bigomega-real-nat-transfer*:
 $(f :: \text{real} \Rightarrow \text{real}) \in \Omega(g) \Longrightarrow (\lambda x::\text{nat}. f\ (\text{real } x)) \in \Omega(\lambda x. g\ (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *bigtheta-real-nat-transfer*:
 $(f :: \text{real} \Rightarrow \text{real}) \in \Theta(g) \Longrightarrow (\lambda x::\text{nat}. f\ (\text{real } x)) \in \Theta(\lambda x. g\ (\text{real } x))$
 $\langle \text{proof} \rangle$

lemmas *landau-real-nat-transfer* [intro] =

*bigo-real-nat-transfer smalllo-real-nat-transfer bigomega-real-nat-transfer
smallomega-real-nat-transfer bigheta-real-nat-transfer*

lemma *landau-symbol-if-at-top-eq* [simp]:

assumes *landau-symbol* $L\ L'\ Lr$

shows $L\ \text{at-top}\ (\lambda x::'a::\text{linordered-semidom. if } x = a \text{ then } f\ x \text{ else } g\ x) = L$
at-top (g)
 ⟨*proof*⟩

lemmas *landau-symbols-if-at-top-eq* [simp] = *landau-symbols*[*THEN landau-symbol-if-at-top-eq*]

lemma *sum-in-smalllo*:

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-smalllo*:

assumes $\bigwedge x. x \in A \implies f\ x \in o[F](g)$

shows $(\lambda x. \text{sum } (\lambda y. f\ y\ x)\ A) \in o[F](g)$

⟨*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 $\bigwedge x. x \in A \implies f\ x \in O[F](g)$

shows $(\lambda x. \text{sum } (\lambda y. f\ y\ x)\ A) \in O[F](g)$

⟨*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] = *bigheta-trans1 bigheta-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 $(\lambda x. f1\ x / f2\ x) \in L\ F\ (\lambda x. g1\ x / g2\ x) \longleftrightarrow f1 \in L\ F\ (g1)$
 $\langle proof \rangle$

lemma *divide-cancel-left*:

assumes $f1 \in \Theta[F](g1)$ **and eventually** $(\lambda x. g1\ x \neq 0)\ F$
shows $(\lambda x. f1\ x / f2\ x) \in L\ F\ (\lambda x. g1\ x / g2\ x) \longleftrightarrow$
 $(\lambda x. \text{inverse}\ (f2\ x)) \in L\ F\ (\lambda x. \text{inverse}\ (g2\ x))$
 $\langle proof \rangle$

end

lemma *powr-smallo-iff*:

assumes *filterlim g at-top F F $\neq$ bot*
shows $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in o[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p < q$
 $\langle proof \rangle$

lemma *powr-bigo-iff*:

assumes *filterlim g at-top F F $\neq$ bot*
shows $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in O[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p \leq q$
 $\langle proof \rangle$

lemma *powr-bigtheta-iff*:

assumes *filterlim g at-top F F $\neq$ bot*
shows $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in \Theta[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p = q$
 $\langle proof \rangle$

50.3 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 is that $\ln f(x) \in o(\ln g(x))$, which we show now.

lemma *ln-smallo-imp-flat*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$
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 $(\lambda x. f\ x\ \text{powr}\ p) \in o(\lambda x. g\ x\ \text{powr}\ q)$
 $\langle proof \rangle$

lemma *ln-smallo-imp-flat'*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

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 $(\lambda x. g x \text{ powr } q) \in o(\lambda x. f x \text{ powr } p)$
 <proof>

50.4 Asymptotic Equivalence

named-theorems *asympt-equiv-intros*

named-theorems *asympt-equiv-simps*

definition *asympt-equiv* :: $('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow 'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$

$(\langle \cdot \sim [\cdot] \rangle \rightarrow [51, 10, 51] 50)$

where $f \sim [F] g \longleftrightarrow ((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1) F$

abbreviation (*input*) *asympt-equiv-at-top* **where**

asympt-equiv-at-top $f g \equiv f \sim [\text{at-top}] g$

bundle *asympt-equiv-notation*

begin

notation *asympt-equiv-at-top* (**infix** $\langle \sim \rangle$ 50)

end

lemma *asympt-equivI*: $((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1)$
 $F \Longrightarrow f \sim [F] g$
 <proof>

lemma *asympt-equivD*: $f \sim [F] g \Longrightarrow ((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1) F$
 <proof>

lemma *asympt-equiv-filtermap-iff*:

$f \sim [\text{filtermap } h F] g \longleftrightarrow (\lambda x. f (h x)) \sim [F] (\lambda x. g (h x))$
 <proof>

lemma *asympt-equiv-refl* [*simp*, *asympt-equiv-intros*]: $f \sim [F] f$
 <proof>

lemma *asympt-equiv-symI*:

assumes $f \sim [F] g$

shows $g \sim [F] f$

<proof>

lemma *asympt-equiv-sym*: $f \sim [F] g \longleftrightarrow g \sim [F] f$
 <proof>

lemma *asympt-equivI'*:

assumes $((\lambda x. f\ x / g\ x) \longrightarrow 1)\ F$
shows $f \sim[F] g$
 $\langle proof \rangle$

lemma *asympt-equiv-cong*:
assumes *eventually* $(\lambda x. f1\ x = f2\ x)\ F$ *eventually* $(\lambda x. g1\ x = g2\ x)\ F$
shows $f1 \sim[F] g1 \longleftrightarrow f2 \sim[F] g2$
 $\langle proof \rangle$

lemma *asympt-equiv-eventually-zeros*:
fixes $f\ g :: 'a \Rightarrow 'b :: \text{real-normed-field}$
assumes $f \sim[F] g$
shows *eventually* $(\lambda x. f\ x = 0 \longleftrightarrow g\ x = 0)\ F$
 $\langle proof \rangle$

lemma *asympt-equiv-transfer*:
assumes $f1 \sim[F] g1$ *eventually* $(\lambda x. f1\ x = f2\ x)\ F$ *eventually* $(\lambda x. g1\ x = g2\ x)\ F$
shows $f2 \sim[F] g2$
 $\langle proof \rangle$

lemma *asympt-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 $(\lambda x. f\ x (h2\ x)) \sim[F] (\lambda x. g\ x (h2\ x))$
 $\langle proof \rangle$

lemma *asympt-equiv-trans* [trans]:
fixes $f\ g\ h$
assumes $f \sim[F] g$ $g \sim[F] h$
shows $f \sim[F] h$
 $\langle proof \rangle$

lemma *asympt-equiv-trans-lift1* [trans]:
assumes $a \sim[F] f$ $b \sim[F] c$ $\bigwedge c\ d. c \sim[F] d \implies f\ c \sim[F] f\ d$
shows $a \sim[F] f\ c$
 $\langle proof \rangle$

lemma *asympt-equiv-trans-lift2* [trans]:
assumes $f\ a \sim[F] b$ $a \sim[F] c$ $\bigwedge c\ d. c \sim[F] d \implies f\ c \sim[F] f\ d$
shows $f\ c \sim[F] b$
 $\langle proof \rangle$

lemma *asympt-equivD-const*:
assumes $f \sim[F] (\lambda \cdot. c)$
shows $(f \longrightarrow c)\ F$
 $\langle proof \rangle$

lemma *asympt-equiv-refl-ev*:

assumes *eventually* $(\lambda x. f\ x = g\ x)\ F$

shows $f \sim[F] g$

$\langle proof \rangle$

lemma *asympt-equiv-sandwich*:

fixes $f\ g\ h :: 'a \Rightarrow 'b :: \{\text{real-normed-field, order-topology, linordered-field}\}$

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 $g \sim[F] f\ g \sim[F] h$

$\langle proof \rangle$

lemma *asympt-equiv-imp-eventually-same-sign*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $f \sim[F] g$

shows *eventually* $(\lambda x. \text{sgn}\ (f\ x) = \text{sgn}\ (g\ x))\ F$

$\langle proof \rangle$

lemma

fixes $f\ g :: - \Rightarrow \text{real}$

assumes $f \sim[F] g$

shows *asympt-equiv-eventually-same-sign*: *eventually* $(\lambda x. \text{sgn}\ (f\ x) = \text{sgn}\ (g\ x))\ F$ (**is** ?th1)

and *asympt-equiv-eventually-neg-iff*: *eventually* $(\lambda x. f\ x < 0 \longleftrightarrow g\ x < 0)$ *F* (**is** ?th2)

and *asympt-equiv-eventually-pos-iff*: *eventually* $(\lambda x. f\ x > 0 \longleftrightarrow g\ x > 0)$ *F* (**is** ?th3)

$\langle proof \rangle$

lemma *asympt-equiv-tendsto-transfer*:

assumes $f \sim[F] g$ **and** $(f \longrightarrow c)\ F$

shows $(g \longrightarrow c)\ F$

$\langle proof \rangle$

lemma *tendsto-asympt-equiv-cong*:

assumes $f \sim[F] g$

shows $(f \longrightarrow c)\ F \longleftrightarrow (g \longrightarrow c)\ F$

$\langle proof \rangle$

lemma *smallo-imp-eventually-sgn*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $g \in o(f)$

shows *eventually* $(\lambda x. \text{sgn}\ (f\ x + g\ x) = \text{sgn}\ (f\ x))\ \text{at-top}$

$\langle proof \rangle$

context

begin

private lemma *asympt-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 *asympt-equiv-add-right* [*asympt-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 *asympt-equiv-add-left* [*asympt-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 *asympt-equiv-add-right'* [*asympt-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 *asympt-equiv-add-left'* [*asympt-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-asympt-equiv*:

assumes $(\lambda x. f x - g x) \in o[F](g)$

shows $f \sim[F] g$

<proof>

lemma *asympt-equiv-uminus* [*asympt-equiv-intros*]:

$f \sim[F] g \implies (\lambda x. -f x) \sim[F] (\lambda x. -g x)$

<proof>

lemma *asympt-equiv-uminus-iff* [*asympt-equiv-simps*]:

$(\lambda x. -f x) \sim[F] g \longleftrightarrow f \sim[F] (\lambda x. -g x)$

<proof>

lemma *asympt-equiv-mult* [*asympt-equiv-intros*]:

fixes $f1\ f2\ g1\ g2 :: 'a \Rightarrow 'b :: \text{real-normed-field}$

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 *asympt-equiv-power* [*asympt-equiv-intros*]:

$f \sim[F] g \implies (\lambda x. f x \hat{=} n) \sim[F] (\lambda x. g x \hat{=} n)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-inverse* [*asympt-equiv-intros*]:
assumes $f \sim[F] g$
shows $(\lambda x. \text{inverse } (f x)) \sim[F] (\lambda x. \text{inverse } (g x))$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-inverse-iff* [*asympt-equiv-simps*]:
 $(\lambda x. \text{inverse } (f x)) \sim[F] (\lambda x. \text{inverse } (g x)) \longleftrightarrow f \sim[F] g$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-divide* [*asympt-equiv-intros*]:
assumes $f1 \sim[F] g1$ $f2 \sim[F] g2$
shows $(\lambda x. f1 x / f2 x) \sim[F] (\lambda x. g1 x / g2 x)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-compose* [*asympt-equiv-intros*]:
assumes $f \sim[G] g$ *filterlim* h G F
shows $f \circ h \sim[F] g \circ h$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-compose'*:
assumes $f \sim[G] g$ *filterlim* h G F
shows $(\lambda x. f (h x)) \sim[F] (\lambda x. g (h x))$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-powr-real* [*asympt-equiv-intros*]:
fixes $f g :: 'a \Rightarrow \text{real}$
assumes $f \sim[F] g$ *eventually* $(\lambda x. f x \geq 0)$ F *eventually* $(\lambda x. g x \geq 0)$ F
shows $(\lambda x. f x \text{ powr } y) \sim[F] (\lambda x. g x \text{ powr } y)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-norm* [*asympt-equiv-intros*]:
fixes $f g :: 'a \Rightarrow 'b :: \text{real-normed-field}$
assumes $f \sim[F] g$
shows $(\lambda x. \text{norm } (f x)) \sim[F] (\lambda x. \text{norm } (g x))$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-abs-real* [*asympt-equiv-intros*]:
fixes $f g :: 'a \Rightarrow \text{real}$
assumes $f \sim[F] g$
shows $(\lambda x. |f x|) \sim[F] (\lambda x. |g x|)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-eventually-le*:
assumes $f \sim[F] g$ $c > 1$
shows *eventually* $(\lambda x. \text{norm } (f x) \leq c * \text{norm } (g x))$ F
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-eventually-ge*:

assumes $f \sim[F] g$ $c < 1$

shows *eventually* $(\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x)) F$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-bigo*:

assumes $f \sim[F] g$

shows $f \in O[F](g)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-bigomega*:

$f \sim[F] g \implies f \in \Omega[F](g)$

$\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-bigtheta*:

$f \sim[F] g \implies f \in \Theta[F](g)$

$\langle \text{proof} \rangle$

lemma *asympt-equiv-at-infinity-transfer*:

assumes $f \sim[F] g$ *filterlim* f *at-infinity* F

shows *filterlim* g *at-infinity* F
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-at-top-transfer*:

fixes $f g :: - \Rightarrow \text{real}$

assumes $f \sim[F] g$ *filterlim* f *at-top* F

shows *filterlim* g *at-top* F
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-at-bot-transfer*:

fixes $f g :: - \Rightarrow \text{real}$

assumes $f \sim[F] g$ *filterlim* f *at-bot* F

shows *filterlim* g *at-bot* F
 $\langle \text{proof} \rangle$

lemma *asympt-equivI'-const*:

assumes $((\lambda x. f x / g x) \longrightarrow c) F$ $c \neq 0$

shows $f \sim[F] (\lambda x. c * g x)$
 $\langle \text{proof} \rangle$

lemma *asympt-equivI'-inverse-const*:

assumes $((\lambda x. f x / g x) \longrightarrow \text{inverse } c) F$ $c \neq 0$

shows $(\lambda x. c * f x) \sim[F] g$
 $\langle \text{proof} \rangle$

lemma *filterlim-at-bot-imp-at-infinity*: *filterlim* f *at-bot* $F \implies$ *filterlim* f *at-infinity* F

for $f :: - \Rightarrow \text{real}$ $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-diff-smallo*:

assumes $f \sim[F] g$

shows $(\lambda x. f\ x - g\ x) \in o[F](g)$

<proof>

lemma *asympt-equiv-altdef*:

$f \sim[F] g \longleftrightarrow (\lambda x. f\ x - g\ x) \in o[F](g)$

<proof>

lemma *asympt-equiv-0-left-iff [simp]*: $(\lambda-. 0) \sim[F] f \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0)$
 F

and *asympt-equiv-0-right-iff [simp]*: $f \sim[F] (\lambda-. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0)$
 F

<proof>

lemma *asympt-equiv-sandwich-real*:

fixes $f\ g\ l\ u :: 'a \Rightarrow \text{real}$

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 *asympt-equiv-sandwich-real'*:

fixes $f\ g\ l\ u :: 'a \Rightarrow \text{real}$

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 *asympt-equiv-sandwich-real''*:

fixes $f\ g\ l\ u :: 'a \Rightarrow \text{real}$

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

theory *Lattice-Constructions*

imports *Main*

begin

50.5 Values extended by a bottom element

datatype $'a\ bot = \text{Value } 'a \mid Bot$

instantiation $bot :: (\text{preorder})\ \text{preorder}$

begin

definition *less-eq-bot* **where**

$x \leq y \longleftrightarrow (\text{case } x \text{ of Bot} \Rightarrow \text{True} \mid \text{Value } x \Rightarrow (\text{case } y \text{ of Bot} \Rightarrow \text{False} \mid \text{Value } y \Rightarrow x \leq y))$

definition *less-bot* **where**

$x < y \longleftrightarrow (\text{case } y \text{ of Bot} \Rightarrow \text{False} \mid \text{Value } y \Rightarrow (\text{case } x \text{ of Bot} \Rightarrow \text{True} \mid \text{Value } x \Rightarrow x < y))$

lemma *less-eq-bot-Bot* [simp]: $\text{Bot} \leq x$

<proof>

lemma *less-eq-bot-Bot-code* [code]: $\text{Bot} \leq x \longleftrightarrow \text{True}$

<proof>

lemma *less-eq-bot-Bot-is-Bot*: $x \leq \text{Bot} \implies x = \text{Bot}$

<proof>

lemma *less-eq-bot-Value-Bot* [simp, code]: $\text{Value } x \leq \text{Bot} \longleftrightarrow \text{False}$

<proof>

lemma *less-eq-bot-Value* [simp, code]: $\text{Value } x \leq \text{Value } y \longleftrightarrow x \leq y$

<proof>

lemma *less-bot-Bot* [simp, code]: $x < \text{Bot} \longleftrightarrow \text{False}$

<proof>

lemma *less-bot-Bot-is-Value*: $\text{Bot} < x \implies \exists z. x = \text{Value } z$

<proof>

lemma *less-bot-Bot-Value* [simp]: $\text{Bot} < \text{Value } x$

<proof>

lemma *less-bot-Bot-Value-code* [code]: $\text{Bot} < \text{Value } x \longleftrightarrow \text{True}$

<proof>

lemma *less-bot-Value* [simp, code]: $\text{Value } x < \text{Value } y \longleftrightarrow x < y$

<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

```

$\langle proof \rangle$

50.6 Values extended by a top element

datatype *'a top* = *Value 'a* | *Top*

instantiation *top* :: (*preorder*) *preorder*
begin

definition *less-eq-top* **where**

$x \leq y \longleftrightarrow (\text{case } y \text{ of } Top \Rightarrow True \mid \text{Value } y \Rightarrow (\text{case } x \text{ of } Top \Rightarrow False \mid \text{Value } x \Rightarrow x \leq y))$

definition *less-top* **where**

$x < y \longleftrightarrow (\text{case } x \text{ of } Top \Rightarrow False \mid \text{Value } x \Rightarrow (\text{case } y \text{ of } Top \Rightarrow True \mid \text{Value } y \Rightarrow x < y))$

lemma *less-eq-top-Top* [*simp*]: $x \leq Top$
 $\langle proof \rangle$

lemma *less-eq-top-Top-code* [*code*]: $x \leq Top \longleftrightarrow True$
 $\langle proof \rangle$

lemma *less-eq-top-is-Top*: $Top \leq x \Longrightarrow x = Top$
 $\langle proof \rangle$

lemma *less-eq-top-Top-Value* [*simp*, *code*]: $Top \leq \text{Value } x \longleftrightarrow False$
 $\langle proof \rangle$

lemma *less-eq-top-Value-Value* [*simp*, *code*]: $\text{Value } x \leq \text{Value } y \longleftrightarrow x \leq y$
 $\langle proof \rangle$

lemma *less-top-Top* [*simp*, *code*]: $Top < x \longleftrightarrow False$
 $\langle proof \rangle$

lemma *less-top-Top-is-Value*: $x < Top \Longrightarrow \exists z. x = \text{Value } z$
 $\langle proof \rangle$

lemma *less-top-Value-Top* [*simp*]: $\text{Value } x < Top$
 $\langle proof \rangle$

lemma *less-top-Value-Top-code* [*code*]: $\text{Value } x < Top \longleftrightarrow True$
 $\langle proof \rangle$

lemma *less-top-Value* [*simp*, *code*]: $\text{Value } x < \text{Value } y \longleftrightarrow x < y$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

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

$$\begin{array}{l} Top \Rightarrow Top \\ | \text{Value } v' \Rightarrow \text{Value } (\text{sup } v \ v') \end{array}$$
instance $\langle \text{proof} \rangle$ **end****instance** *top* :: (*lattice*) *bounded-lattice-top* $\langle \text{proof} \rangle$

50.7 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**

$$\begin{array}{l} x \leq y \equiv \\ \quad (\text{case } x \text{ of} \\ \quad \quad Bot \Rightarrow True \\ | \text{Value } v1 \Rightarrow \\ \quad \quad (\text{case } y \text{ of} \\ \quad \quad \quad Bot \Rightarrow False \\ \quad \quad \quad | \text{Value } v2 \Rightarrow v1 = v2 \\ \quad \quad \quad | Top \Rightarrow True) \\ | Top \Rightarrow y = Top) \end{array}$$
definition *less-flat-complete-lattice***where**

$$\begin{array}{l} x < y = \\ \quad (\text{case } x \text{ of} \\ \quad \quad Bot \Rightarrow y \neq Bot \\ | \text{Value } v1 \Rightarrow y = Top \\ | Top \Rightarrow False) \end{array}$$
lemma [*simp*]: *Bot* ≤ *y* $\langle \text{proof} \rangle$ **lemma** [*simp*]: *y* ≤ *Top* $\langle \text{proof} \rangle$ **lemma** *greater-than-two-values*:**assumes** *a* ≠ *b* *Value a* ≤ *z* *Value b* ≤ *z***shows** *z* = *Top* $\langle \text{proof} \rangle$

```

lemma lesser-than-two-values:
  assumes  $a \neq b$   $z \leq \text{Value } a$   $z \leq \text{Value } b$ 
  shows  $z = \text{Bot}$ 
   $\langle \text{proof} \rangle$ 

instance
   $\langle \text{proof} \rangle$ 

end

instantiation flat-complete-lattice :: (type) bot
begin
  definition bot = Bot
  instance  $\langle \text{proof} \rangle$ 
end

instantiation flat-complete-lattice :: (type) top
begin
  definition top = Top
  instance  $\langle \text{proof} \rangle$ 
end

instantiation flat-complete-lattice :: (type) lattice
begin

definition inf-flat-complete-lattice
where
  inf  $x$   $y$  =
    (case  $x$  of
      Bot  $\Rightarrow$  Bot
    | Value  $v1$   $\Rightarrow$ 
      (case  $y$  of
        Bot  $\Rightarrow$  Bot
      | Value  $v2$   $\Rightarrow$  if  $v1 = v2$  then  $x$  else Bot
      | Top  $\Rightarrow$   $x$ )
    | Top  $\Rightarrow$   $y$ )

definition sup-flat-complete-lattice
where
  sup  $x$   $y$  =
    (case  $x$  of
      Bot  $\Rightarrow$   $y$ 
    | Value  $v1$   $\Rightarrow$ 
      (case  $y$  of
        Bot  $\Rightarrow$   $x$ 
      | Value  $v2$   $\Rightarrow$  if  $v1 = v2$  then  $x$  else Top
      | Top  $\Rightarrow$  Top)
    | Top  $\Rightarrow$  Top)

```

instance $\langle proof \rangle$ **end****instantiation** *flat-complete-lattice* :: (type) complete-lattice**begin****definition** *Sup-flat-complete-lattice***where** $Sup\ A =$ $(if\ A = \{\} \vee A = \{Bot\}\ then\ Bot$ $else\ if\ \exists v. A - \{Bot\} = \{Value\ v\}\ then\ Value\ (THE\ v. A - \{Bot\} = \{Value\ v\})$ $else\ Top)$ **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** $\langle proof \rangle$ **end****end**

51 Infinite Streams

theory *Stream***imports** *Nat-Bijection***begin****codatatype** (*sset*: 'a) *stream* = $SCons\ (shd: 'a)\ (stl: 'a\ stream)\ (\text{infixr}\ \langle \#\#\rangle\ 65)$ **for***map*: *smap**rel*: *stream-all2***context****begin**

— for code generation only

qualified definition *smember* :: 'a $\Rightarrow$ 'a *stream* $\Rightarrow$ bool **where** $[code-abbrev]:\ smember\ x\ s \longleftrightarrow x \in sset\ s$

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* ∈ *sset* *s* and $\bigwedge s. P$ (*shd* *s*) *s* and $\bigwedge s y. \llbracket y \in \textit{sset} (\textit{stl } s); P y (\textit{stl } s) \rrbracket$
 $\implies P y s$
 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*⟩

51.1 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*⟩

51.2 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*]: $\llbracket a \in A; s \in \text{streams } A \rrbracket \Longrightarrow a \#\# s \in \text{streams } A$

end

lemma *in-streams*: $\text{stl } s \in \text{streams } S \Longrightarrow \text{shd } s \in S \Longrightarrow s \in \text{streams } S$
 <proof>

lemma *streamsE*: $s \in \text{streams } A \Longrightarrow (\text{shd } s \in A \Longrightarrow \text{stl } s \in \text{streams } A \Longrightarrow P) \Longrightarrow P$
 <proof>

lemma *Stream-image*: $x \#\# y \in ((\#\#) x') \text{ ' } Y \longleftrightarrow x = x' \wedge y \in Y$
 <proof>

lemma *shift-streams*: $\llbracket w \in \text{lists } A; s \in \text{streams } A \rrbracket \Longrightarrow w @- s \in \text{streams } A$
 <proof>

lemma *streams-Stream*: $x \#\# s \in \text{streams } A \longleftrightarrow x \in A \wedge s \in \text{streams } A$
 <proof>

lemma *streams-stl*: $s \in \text{streams } A \Longrightarrow \text{stl } s \in \text{streams } A$
 <proof>

lemma *streams-shd*: $s \in \text{streams } A \Longrightarrow \text{shd } s \in A$
 <proof>

lemma *sset-streams*:
assumes $\text{sset } s \subseteq A$
shows $s \in \text{streams } A$
 <proof>

lemma *streams-sset*:
assumes $s \in \text{streams } A$
shows $\text{sset } s \subseteq A$
 <proof>

lemma *streams-iff-sset*: $s \in \text{streams } A \longleftrightarrow \text{sset } s \subseteq A$
 <proof>

lemma *streams-mono*: $s \in \text{streams } A \Longrightarrow A \subseteq B \Longrightarrow s \in \text{streams } B$
 <proof>

lemma *streams-mono2*: $S \subseteq T \Longrightarrow \text{streams } S \subseteq \text{streams } T$
 <proof>

lemma *smap-streams*: $s \in \text{streams } A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow f x \in B) \Longrightarrow \text{smap } f s$

$\in \text{streams } B$
 $\langle \text{proof} \rangle$

lemma *streams-empty*: $\text{streams } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *streams-UNIV*[*simp*]: $\text{streams } \text{UNIV} = \text{UNIV}$
 $\langle \text{proof} \rangle$

51.3 nth, take, drop for streams

primrec *snth* :: $'a \text{ stream} \Rightarrow \text{nat} \Rightarrow 'a$ (**infixl** $\langle !! \rangle$ 100) **where**
 $s !! 0 = \text{shd } s$
 $| s !! \text{Suc } n = \text{stl } s !! n$

lemma *snth-Stream*: $(x \#\# s) !! \text{Suc } i = s !! i$
 $\langle \text{proof} \rangle$

lemma *snth-smap*[*simp*]: $\text{smap } f s !! n = f (s !! n)$
 $\langle \text{proof} \rangle$

lemma *shift-snth-less*[*simp*]: $p < \text{length } xs \Longrightarrow (xs @- s) !! p = xs ! p$
 $\langle \text{proof} \rangle$

lemma *shift-snth-ge*[*simp*]: $p \geq \text{length } xs \Longrightarrow (xs @- s) !! p = s !! (p - \text{length } xs)$
 $\langle \text{proof} \rangle$

lemma *shift-snth*: $(xs @- s) !! n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } s !! (n - \text{length } xs))$
 $\langle \text{proof} \rangle$

lemma *snth-sset*[*simp*]: $s !! n \in \text{sset } s$
 $\langle \text{proof} \rangle$

lemma *sset-range*: $\text{sset } s = \text{range } (\text{snth } s)$
 $\langle \text{proof} \rangle$

lemma *streams-iff-snth*: $s \in \text{streams } X \longleftrightarrow (\forall n. s !! n \in X)$
 $\langle \text{proof} \rangle$

lemma *snth-in*: $s \in \text{streams } X \Longrightarrow s !! n \in X$
 $\langle \text{proof} \rangle$

primrec *stake* :: $\text{nat} \Rightarrow 'a \text{ stream} \Rightarrow 'a \text{ list}$ **where**
 $\text{stake } 0 s = []$
 $| \text{stake } (\text{Suc } n) s = \text{shd } s \# \text{stake } n (\text{stl } s)$

lemma *length-stake*[*simp*]: $\text{length } (\text{stake } n s) = n$
 $\langle \text{proof} \rangle$

lemma *stake-smap[simp]*: $\text{stake } n \ (\text{smap } f \ s) = \text{map } f \ (\text{stake } n \ s)$
 $\langle \text{proof} \rangle$

lemma *take-stake*: $\text{take } n \ (\text{stake } m \ s) = \text{stake } (\min n \ m) \ s$
 $\langle \text{proof} \rangle$

primrec *sdrop* :: $\text{nat} \Rightarrow 'a \ \text{stream} \Rightarrow 'a \ \text{stream}$ **where**
 $\text{sdrop } 0 \ s = s$
 $|\ \text{sdrop } (\text{Suc } n) \ s = \text{sdrop } n \ (\text{stl } s)$

lemma *sdrop-simps[simp]*:
 $\text{shd } (\text{sdrop } n \ s) = s \ !! \ n \ \text{stl } (\text{sdrop } n \ s) = \text{sdrop } (\text{Suc } n) \ s$
 $\langle \text{proof} \rangle$

lemma *sdrop-smap[simp]*: $\text{sdrop } n \ (\text{smap } f \ s) = \text{smap } f \ (\text{sdrop } n \ s)$
 $\langle \text{proof} \rangle$

lemma *sdrop-stl*: $\text{sdrop } n \ (\text{stl } s) = \text{stl } (\text{sdrop } n \ s)$
 $\langle \text{proof} \rangle$

lemma *drop-stake*: $\text{drop } n \ (\text{stake } m \ s) = \text{stake } (m - n) \ (\text{sdrop } n \ s)$
 $\langle \text{proof} \rangle$

lemma *stake-sdrop*: $\text{stake } n \ s \ @- \ \text{sdrop } n \ s = s$
 $\langle \text{proof} \rangle$

lemma *id-stake-snth-sdrop*:
 $s = \text{stake } i \ s \ @- \ s \ !! \ i \ \#\# \ \text{sdrop } (\text{Suc } i) \ s$
 $\langle \text{proof} \rangle$

lemma *smap-alt*: $\text{smap } f \ s = s' \longleftrightarrow (\forall n. f \ (s \ !! \ n) = s' \ !! \ n) \ (\text{is } ?L = ?R)$
 $\langle \text{proof} \rangle$

lemma *stake-invert-Nil[iff]*: $\text{stake } n \ s = [] \longleftrightarrow n = 0$
 $\langle \text{proof} \rangle$

lemma *sdrop-shift*: $\text{sdrop } i \ (w \ @- \ s) = \text{drop } i \ w \ @- \ \text{sdrop } (i - \text{length } w) \ s$
 $\langle \text{proof} \rangle$

lemma *stake-shift*: $\text{stake } i \ (w \ @- \ s) = \text{take } i \ w \ @ \ \text{stake } (i - \text{length } w) \ s$
 $\langle \text{proof} \rangle$

lemma *stake-add[simp]*: $\text{stake } m \ s \ @ \ \text{stake } n \ (\text{sdrop } m \ s) = \text{stake } (m + n) \ s$
 $\langle \text{proof} \rangle$

lemma *sdrop-add[simp]*: $\text{sdrop } n \ (\text{sdrop } m \ s) = \text{sdrop } (m + n) \ s$
 $\langle \text{proof} \rangle$

lemma *sdrop-snth*: $\text{sdrop } n \ s \ !! \ m = s \ !! \ (n + m)$
 $\langle \text{proof} \rangle$

partial-function (*tailrec*) *sdrop-while* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \ \text{stream} \Rightarrow 'a \ \text{stream}$
where
 $\text{sdrop-while } P \ s = (\text{if } P \ (\text{shd } s) \text{ then } \text{sdrop-while } P \ (\text{stl } s) \text{ else } s)$

lemma *sdrop-while-SCons*[code]:
 $\text{sdrop-while } P \ (a \ \#\# \ s) = (\text{if } P \ a \text{ then } \text{sdrop-while } P \ s \text{ else } a \ \#\# \ s)$
 $\langle \text{proof} \rangle$

lemma *sdrop-while-sdrop-LEAST*:
assumes $\exists n. P \ (s \ !! \ n)$
shows $\text{sdrop-while } (\text{Not} \circ P) \ s = \text{sdrop } (\text{LEAST } n. P \ (s \ !! \ n)) \ s$
 $\langle \text{proof} \rangle$

primcorec *sfilter* **where**
 $\text{shd } (\text{sfilter } P \ s) = \text{shd } (\text{sdrop-while } (\text{Not} \circ P) \ s)$
 $|\ \text{stl } (\text{sfilter } P \ s) = \text{sfilter } P \ (\text{stl } (\text{sdrop-while } (\text{Not} \circ P) \ s))$

lemma *sfilter-Stream*: $\text{sfilter } P \ (x \ \#\# \ s) = (\text{if } P \ x \text{ then } x \ \#\# \ \text{sfilter } P \ s \text{ else } \text{sfilter } P \ s)$
 $\langle \text{proof} \rangle$

51.4 unary predicates lifted to streams

definition *stream-all* $P \ s = (\forall p. P \ (s \ !! \ p))$

lemma *stream-all-iff*[iff]: $\text{stream-all } P \ s \longleftrightarrow \text{Ball } (\text{sset } s) \ P$
 $\langle \text{proof} \rangle$

lemma *stream-all-shift*[simp]: $\text{stream-all } P \ (xs \ @- \ s) = (\text{list-all } P \ xs \wedge \text{stream-all } P \ s)$
 $\langle \text{proof} \rangle$

lemma *stream-all-Stream*: $\text{stream-all } P \ (x \ \#\# \ X) \longleftrightarrow P \ x \wedge \text{stream-all } P \ X$
 $\langle \text{proof} \rangle$

51.5 recurring stream out of a list

primcorec *cycle* :: $'a \ \text{list} \Rightarrow 'a \ \text{stream}$ **where**
 $\text{shd } (\text{cycle } xs) = \text{hd } xs$
 $|\ \text{stl } (\text{cycle } xs) = \text{cycle } (\text{tl } xs \ @ \ [\text{hd } xs])$

lemma *cycle-decomp*: $u \neq [] \Longrightarrow \text{cycle } u = u \ @- \ \text{cycle } u$
 $\langle \text{proof} \rangle$

lemma *cycle-Cons*[code]: $\text{cycle } (x \ \# \ xs) = x \ \#\# \ \text{cycle } (xs \ @ \ [x])$
 $\langle \text{proof} \rangle$

lemma *cycle-rotated*: $\llbracket v \neq []; \text{cycle } u = v @- s \rrbracket \implies \text{cycle } (tl\ u @ [hd\ u]) = tl\ v @- s$
 <proof>

lemma *stake-append*: $\text{stake } n\ (u @- s) = \text{take } (\min\ (\text{length } u)\ n)\ u @ \text{stake } (n - \text{length } u)\ s$
 <proof>

lemma *stake-cycle-le[simp]*:
 assumes $u \neq []$ $n < \text{length } u$
 shows $\text{stake } n\ (\text{cycle } u) = \text{take } n\ u$
 <proof>

lemma *stake-cycle-eq[simp]*: $u \neq [] \implies \text{stake } (\text{length } u)\ (\text{cycle } u) = u$
 <proof>

lemma *sdrop-cycle-eq[simp]*: $u \neq [] \implies \text{sdrop } (\text{length } u)\ (\text{cycle } u) = \text{cycle } u$
 <proof>

lemma *stake-cycle-eq-mod-0[simp]*: $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies \text{stake } n\ (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u)\ u)$
 <proof>

lemma *sdrop-cycle-eq-mod-0[simp]*: $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies \text{sdrop } n\ (\text{cycle } u) = \text{cycle } u$
 <proof>

lemma *stake-cycle*: $u \neq [] \implies \text{stake } n\ (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u)\ u) @ \text{take } (n \bmod \text{length } u)\ u$
 <proof>

lemma *sdrop-cycle*: $u \neq [] \implies \text{sdrop } n\ (\text{cycle } u) = \text{cycle } (\text{rotate } (n \bmod \text{length } u)\ u)$
 <proof>

lemma *sset-cycle[simp]*:
 assumes $xs \neq []$
 shows $\text{sset } (\text{cycle } xs) = \text{set } xs$
 <proof>

51.6 iterated application of a function

primcorec *siterate* where
 $\text{shd } (\text{siterate } f\ x) = x$
 $| \text{stl } (\text{siterate } f\ x) = \text{siterate } f\ (f\ x)$

lemma *stake-Suc*: $\text{stake } (\text{Suc } n)\ s = \text{stake } n\ s @ [s !! n]$
 <proof>

lemma *snth-siterate[simp]*: $\text{siterate } f \ x \ !! \ n = (\widetilde{f})^n \ x$
 $\langle \text{proof} \rangle$

lemma *sdrop-siterate[simp]*: $\text{sdrop } n \ (\text{siterate } f \ x) = \text{siterate } f \ ((\widetilde{f})^n \ x)$
 $\langle \text{proof} \rangle$

lemma *stake-siterate[simp]*: $\text{stake } n \ (\text{siterate } f \ x) = \text{map } (\lambda n. (\widetilde{f})^n \ x) \ [0 \ ..< \ n]$
 $\langle \text{proof} \rangle$

lemma *sset-siterate*: $\text{sset } (\text{siterate } f \ x) = \{(\widetilde{f})^n \ x \mid n. \text{True}\}$
 $\langle \text{proof} \rangle$

lemma *smap-siterate*: $\text{smap } f \ (\text{siterate } f \ x) = \text{siterate } f \ (f \ x)$
 $\langle \text{proof} \rangle$

51.7 stream repeating a single element

abbreviation *sconst* $\equiv \text{siterate } \text{id}$

lemma *shift-replicate-sconst[simp]*: $\text{replicate } n \ x \ @- \ \text{sconst } x = \text{sconst } x$
 $\langle \text{proof} \rangle$

lemma *sset-sconst[simp]*: $\text{sset } (\text{sconst } x) = \{x\}$
 $\langle \text{proof} \rangle$

lemma *sconst-alt*: $s = \text{sconst } x \longleftrightarrow \text{sset } s = \{x\}$
 $\langle \text{proof} \rangle$

lemma *sconst-cycle*: $\text{sconst } x = \text{cycle } [x]$
 $\langle \text{proof} \rangle$

lemma *smap-sconst*: $\text{smap } f \ (\text{sconst } x) = \text{sconst } (f \ x)$
 $\langle \text{proof} \rangle$

lemma *sconst-streams*: $x \in A \implies \text{sconst } x \in \text{streams } A$
 $\langle \text{proof} \rangle$

lemma *streams-empty-iff*: $\text{streams } S = \{\} \longleftrightarrow S = \{\}$
 $\langle \text{proof} \rangle$

51.8 stream of natural numbers

abbreviation *fromN* $\equiv \text{siterate } \text{Suc}$

abbreviation *nats* $\equiv \text{fromN } 0$

lemma *sset-fromN[simp]*: $\text{sset } (\text{fromN } n) = \{n \ ..\}$
 $\langle \text{proof} \rangle$

lemma *stream-smap-fromN*: $s = \text{smap } (\lambda j. \text{let } i = j - n \text{ in } s !! i) \text{ (fromN } n)$
 ⟨proof⟩

lemma *stream-smap-nats*: $s = \text{smap } (\text{snth } s) \text{ nats}$
 ⟨proof⟩

51.9 flatten a stream of lists

primcorec *flat* **where**

$\text{shd } (\text{flat } ws) = \text{hd } (\text{shd } ws)$
 $| \text{ stl } (\text{flat } ws) = \text{flat } (\text{if } \text{tl } (\text{shd } ws) = [] \text{ then } \text{stl } ws \text{ else } \text{tl } (\text{shd } ws) \text{ \#\# } \text{stl } ws)$

lemma *flat-Cons[simp, code]*: $\text{flat } ((x \# xs) \text{ \#\# } ws) = x \text{ \#\# } \text{flat } (\text{if } xs = [] \text{ then } ws \text{ else } xs \text{ \#\# } ws)$
 ⟨proof⟩

lemma *flat-Stream[simp]*: $xs \neq [] \implies \text{flat } (xs \text{ \#\# } ws) = xs @- \text{flat } ws$
 ⟨proof⟩

lemma *flat-unfold*: $\text{shd } ws \neq [] \implies \text{flat } ws = \text{shd } ws @- \text{flat } (\text{stl } ws)$
 ⟨proof⟩

lemma *flat-snth*: $\forall xs \in \text{sset } s. xs \neq [] \implies \text{flat } s !! n = (\text{if } n < \text{length } (\text{shd } s) \text{ then } \text{shd } s ! n \text{ else } \text{flat } (\text{stl } s) !! (n - \text{length } (\text{shd } s)))$
 ⟨proof⟩

lemma *sset-flat[simp]*: $\forall xs \in \text{sset } s. xs \neq [] \implies \text{sset } (\text{flat } s) = (\bigcup xs \in \text{sset } s. \text{set } xs) \text{ (is } ?P \implies ?L = ?R)$
 ⟨proof⟩

51.10 merge a stream of streams

definition *smerge* :: $'a \text{ stream} \Rightarrow 'a \text{ stream}$ **where**

$\text{smerge } ss = \text{flat } (\text{smap } (\lambda n. \text{map } (\lambda s. s !! n) (\text{stake } (\text{Suc } n) ss) @ \text{stake } n (ss !! n)) \text{ nats})$

lemma *stake-nth[simp]*: $m < n \implies \text{stake } n s ! m = s !! m$
 ⟨proof⟩

lemma *snth-sset-smerge*: $ss !! n !! m \in \text{sset } (\text{smerge } ss)$
 ⟨proof⟩

lemma *sset-smerge*: $\text{sset } (\text{smerge } ss) = \bigcup (\text{sset } ` (\text{sset } ss))$
 ⟨proof⟩

51.11 product of two streams

definition *sproduct* :: $'a \text{ stream} \Rightarrow 'b \text{ stream} \Rightarrow ('a \times 'b) \text{ stream}$ **where**

$\text{sproduct } s1 s2 = \text{smerge } (\text{smap } (\lambda x. \text{smap } (\text{Pair } x) s2) s1)$

lemma *sset-sproduct*: $sset (sproduct\ s1\ s2) = sset\ s1 \times sset\ s2$
 $\langle proof \rangle$

51.12 interleave two streams

primcorec *sinterleave* **where**

$shd\ (sinterleave\ s1\ s2) = shd\ s1$
 $|\ stl\ (sinterleave\ s1\ s2) = sinterleave\ s2\ (stl\ s1)$

lemma *sinterleave-code*[*code*]:
 $sinterleave\ (x\ \#\#\ s1)\ s2 = x\ \#\#\ sinterleave\ s2\ s1$
 $\langle proof \rangle$

lemma *sinterleave-snth*[*simp*]:
 $even\ n \implies sinterleave\ s1\ s2\ !!\ n = s1\ !!\ (n\ div\ 2)$
 $odd\ n \implies sinterleave\ s1\ s2\ !!\ n = s2\ !!\ (n\ div\ 2)$
 $\langle proof \rangle$

lemma *sset-sinterleave*: $sset\ (sinterleave\ s1\ s2) = sset\ s1 \cup sset\ s2$
 $\langle proof \rangle$

51.13 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*]: $szip\ (a\ \#\#\ s1)\ (b\ \#\#\ s2) = (a,\ b)\ \#\#\ (szip\ s1\ s2)$
 $\langle proof \rangle$

lemma *snth-szip*[*simp*]: $szip\ s1\ s2\ !!\ n = (s1\ !!\ n,\ s2\ !!\ n)$
 $\langle proof \rangle$

lemma *stake-szip*[*simp*]:
 $stake\ n\ (szip\ s1\ s2) = zip\ (stake\ n\ s1)\ (stake\ n\ s2)$
 $\langle proof \rangle$

lemma *sdrop-szip*[*simp*]: $sdrop\ n\ (szip\ s1\ s2) = szip\ (sdrop\ n\ s1)\ (sdrop\ n\ s2)$
 $\langle proof \rangle$

lemma *smap-szip-fst*:
 $smap\ (\lambda x.\ f\ (fst\ x))\ (szip\ s1\ s2) = smap\ f\ s1$
 $\langle proof \rangle$

lemma *smap-szip-snd*:
 $smap\ (\lambda x.\ g\ (snd\ x))\ (szip\ s1\ s2) = smap\ g\ s2$
 $\langle proof \rangle$

51.14 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]:

$smap2 f (a \#\# s1) (b \#\# s2) = f a b \#\# (smap2 f s1 s2)$
 $\langle proof \rangle$

lemma *smap2-szip*:

$smap2 f s1 s2 = smap (case-prod f) (szip s1 s2)$
 $\langle proof \rangle$

lemma *smap-smap2*[simp]:

$smap f (smap2 g s1 s2) = smap2 (\lambda x y. f (g x y)) s1 s2$
 $\langle proof \rangle$

lemma *smap2-alt*:

$(smap2 f s1 s2 = s) = (\forall n. f (s1 !! n) (s2 !! n) = s !! n)$
 $\langle proof \rangle$

lemma *snth-smap2*[simp]:

$smap2 f s1 s2 !! n = f (s1 !! n) (s2 !! n)$
 $\langle proof \rangle$

lemma *stake-smap2*[simp]:

$stake n (smap2 f s1 s2) = map (case-prod f) (zip (stake n s1) (stake n s2))$
 $\langle proof \rangle$

lemma *sdrop-smap2*[simp]:

$sdrop n (smap2 f s1 s2) = smap2 f (sdrop n s1) (sdrop n s2)$
 $\langle proof \rangle$

end

52 List prefixes, suffixes, and homeomorphic embedding

theory *Sublist*

imports *Main*

begin

52.1 Prefix order on lists

definition *prefix* :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ bool

where *prefix* *xs ys* $\longleftrightarrow (\exists zs. ys = xs @ zs)$

definition *strict-prefix* :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ bool

where $strict\text{-}prefix\ xs\ ys \longleftrightarrow prefix\ xs\ ys \wedge xs \neq ys$

global-interpretation $prefix\text{-}order$: *ordering prefix strict-prefix*
 $\langle proof \rangle$

interpretation $prefix\text{-}order$: *order prefix strict-prefix*
 $\langle proof \rangle$

global-interpretation $prefix\text{-}bot$: *ordering-top $\langle \lambda xs\ ys.\ prefix\ ys\ xs \rangle \langle \lambda xs\ ys.\ strict\text{-}prefix\ ys\ xs \rangle \langle [] \rangle$*
 $\langle proof \rangle$

interpretation $prefix\text{-}bot$: *order-bot Nil prefix strict-prefix*
 $\langle proof \rangle$

lemma $prefixI$ [*intro?*]: $ys = xs @ zs \implies prefix\ xs\ ys$
 $\langle proof \rangle$

lemma $prefixE$ [*elim?*]:
assumes $prefix\ xs\ ys$
obtains zs **where** $ys = xs @ zs$
 $\langle proof \rangle$

lemma $strict\text{-}prefixI'$ [*intro?*]: $ys = xs @ z \# zs \implies strict\text{-}prefix\ xs\ ys$
 $\langle proof \rangle$

lemma $strict\text{-}prefixE'$ [*elim?*]:
assumes $strict\text{-}prefix\ xs\ ys$
obtains $z\ zs$ **where** $ys = xs @ z \# zs$
 $\langle proof \rangle$

lemma $strict\text{-}prefixI$ [*intro?*]: $prefix\ xs\ ys \implies xs \neq ys \implies strict\text{-}prefix\ xs\ ys$
 $\langle proof \rangle$

lemma $strict\text{-}prefixE$ [*elim?*]:
fixes $xs\ ys :: 'a\ list$
assumes $strict\text{-}prefix\ xs\ ys$
obtains $prefix\ xs\ ys$ **and** $xs \neq ys$
 $\langle proof \rangle$

52.2 Basic properties of prefixes

theorem $Nil\text{-}prefix$ [*simp*]: $prefix\ []\ xs$
 $\langle proof \rangle$

theorem $prefix\text{-}Nil$ [*simp*]: $(prefix\ xs\ []) = (xs = [])$
 $\langle proof \rangle$

lemma *prefix-snoc* [simp]: $\text{prefix } xs \ (ys \ @ \ [y]) \longleftrightarrow xs = ys \ @ \ [y] \vee \text{prefix } xs \ ys$
 ⟨proof⟩

lemma *Cons-prefix-Cons* [simp]: $\text{prefix } (x \# xs) \ (y \# ys) = (x = y \wedge \text{prefix } xs \ ys)$
 ⟨proof⟩

lemma *prefix-code* [code]:
 $\text{prefix } [] \ xs \longleftrightarrow \text{True}$
 $\text{prefix } (x \# xs) \ [] \longleftrightarrow \text{False}$
 $\text{prefix } (x \# xs) \ (y \# ys) \longleftrightarrow x = y \wedge \text{prefix } xs \ ys$
 ⟨proof⟩

lemma *same-prefix-prefix* [simp]: $\text{prefix } (xs \ @ \ ys) \ (xs \ @ \ zs) = \text{prefix } ys \ zs$
 ⟨proof⟩

lemma *same-prefix-nil* [simp]: $\text{prefix } (xs \ @ \ ys) \ xs = (ys = [])$
 ⟨proof⟩

lemma *prefix-prefix* [simp]: $\text{prefix } xs \ ys \Longrightarrow \text{prefix } xs \ (ys \ @ \ zs)$
 ⟨proof⟩

lemma *append-prefixD*: $\text{prefix } (xs \ @ \ ys) \ zs \Longrightarrow \text{prefix } xs \ zs$
 ⟨proof⟩

theorem *prefix-Cons*: $\text{prefix } xs \ (y \# ys) = (xs = [] \vee (\exists zs. xs = y \# zs \wedge \text{prefix } zs \ ys))$
 ⟨proof⟩

theorem *prefix-append*:
 $\text{prefix } xs \ (ys \ @ \ zs) = (\text{prefix } xs \ ys \vee (\exists us. xs = ys \ @ \ us \wedge \text{prefix } us \ zs))$
 ⟨proof⟩

lemma *append-one-prefix*:
 $\text{prefix } xs \ ys \Longrightarrow \text{length } xs < \text{length } ys \Longrightarrow \text{prefix } (xs \ @ \ [ys \ ! \ \text{length } xs]) \ ys$
 ⟨proof⟩

theorem *prefix-length-le*: $\text{prefix } xs \ ys \Longrightarrow \text{length } xs \leq \text{length } ys$
 ⟨proof⟩

lemma *prefix-same-cases*:
 $\text{prefix } (xs_1 :: 'a \ \text{list}) \ ys \Longrightarrow \text{prefix } xs_2 \ ys \Longrightarrow \text{prefix } xs_1 \ xs_2 \vee \text{prefix } xs_2 \ xs_1$
 ⟨proof⟩

lemma *prefix-length-prefix*:
 $\text{prefix } ps \ xs \Longrightarrow \text{prefix } qs \ xs \Longrightarrow \text{length } ps \leq \text{length } qs \Longrightarrow \text{prefix } ps \ qs$
 ⟨proof⟩

lemma *set-mono-prefix*: $\text{prefix } xs \ ys \Longrightarrow \text{set } xs \subseteq \text{set } ys$

$\langle \text{proof} \rangle$

lemma *take-is-prefix*: $\text{prefix } (\text{take } n \text{ } xs) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-is-prefix*: $\text{prefix } (\text{takeWhile } P \text{ } xs) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *prefixeq-butlast*: $\text{prefix } (\text{butlast } xs) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *prefix-map-rightE*:
 assumes $\text{prefix } xs \text{ } (\text{map } f \text{ } ys)$
 shows $\exists xs'. \text{prefix } xs' \text{ } ys \wedge xs = \text{map } f \text{ } xs'$
 $\langle \text{proof} \rangle$

lemma *map-mono-prefix*: $\text{prefix } xs \text{ } ys \implies \text{prefix } (\text{map } f \text{ } xs) \text{ } (\text{map } f \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *filter-mono-prefix*: $\text{prefix } xs \text{ } ys \implies \text{prefix } (\text{filter } P \text{ } xs) \text{ } (\text{filter } P \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *sorted-antimono-prefix*: $\text{prefix } xs \text{ } ys \implies \text{sorted } ys \implies \text{sorted } xs$
 $\langle \text{proof} \rangle$

lemma *prefix-length-less*: $\text{strict-prefix } xs \text{ } ys \implies \text{length } xs < \text{length } ys$
 $\langle \text{proof} \rangle$

lemma *prefix-snocD*: $\text{prefix } (xs@[x]) \text{ } ys \implies \text{strict-prefix } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *strict-prefix-simps* [simp, code]:
 $\text{strict-prefix } xs \text{ } [] \longleftrightarrow \text{False}$
 $\text{strict-prefix } [] \text{ } (x \# xs) \longleftrightarrow \text{True}$
 $\text{strict-prefix } (x \# xs) \text{ } (y \# ys) \longleftrightarrow x = y \wedge \text{strict-prefix } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *take-strict-prefix*: $\text{strict-prefix } xs \text{ } ys \implies \text{strict-prefix } (\text{take } n \text{ } xs) \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *prefix-takeWhile*:
 assumes $\text{prefix } xs \text{ } ys$
 shows $\text{prefix } (\text{takeWhile } P \text{ } xs) \text{ } (\text{takeWhile } P \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *prefix-dropWhile*:
 assumes $\text{prefix } xs \text{ } ys$
 shows $\text{prefix } (\text{dropWhile } P \text{ } xs) \text{ } (\text{dropWhile } P \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *prefix-remdups-adj*:

assumes *prefix xs ys*

shows *prefix (remdups-adj xs) (remdups-adj ys)*

<proof>

lemma *not-prefix-cases*:

assumes *pf_x: ¬ 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>*

52.3 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*:

$prefixes (xs @ ys) = prefixes xs @ map (\ys'. xs @ ys') (tl (prefixes ys))$
 $\langle proof \rangle$

lemma *prefixes-eq-snoc*:

$prefixes ys = xs @ [x] \longleftrightarrow$
 $(ys = [] \wedge xs = [] \vee (\exists z zs. ys = zs @ [z] \wedge xs = prefixes zs)) \wedge x = ys$
 $\langle proof \rangle$

lemma *prefixes-tailrec* [code]:

$prefixes xs = rev (snd (foldl (\lambda(acc1, acc2) x. (x \# acc1, rev (x \# acc1) \# acc2))$
 $([], []) xs))$
 $\langle proof \rangle$

lemma *set-prefixes-eq*: $set (prefixes xs) = \{ys. prefix ys xs\}$

$\langle proof \rangle$

lemma *card-set-prefixes* [simp]: $card (set (prefixes xs)) = Suc (length xs)$

$\langle proof \rangle$

lemma *set-prefixes-append*:

$set (prefixes (xs @ ys)) = set (prefixes xs) \cup \{xs @ ys' \mid ys'. ys' \in set (prefixes$
 $ys)\}$
 $\langle proof \rangle$

52.4 Longest Common Prefix

definition *Longest-common-prefix* :: 'a list set $\Rightarrow$ 'a list **where**

Longest-common-prefix $L = (ARG-MAX \text{ length } ps. \forall xs \in L. prefix ps xs)$

lemma *Longest-common-prefix-ex*: $L \neq \{\} \implies$

$\exists ps. (\forall xs \in L. prefix ps xs) \wedge (\forall qs. (\forall xs \in L. prefix qs xs) \longrightarrow size qs \leq size$
 $ps)$

(**is** - $\implies \exists ps. ?P L ps$)

$\langle proof \rangle$

lemma *Longest-common-prefix-unique*:

$\langle \exists! ps. (\forall xs \in L. prefix ps xs) \wedge (\forall qs. (\forall xs \in L. prefix qs xs) \longrightarrow length qs \leq$
 $length ps) \rangle$

if $\langle L \neq \{\} \rangle$

$\langle proof \rangle$

lemma *Longest-common-prefix-eq*:

$\llbracket L \neq \{\}; \forall xs \in L. prefix ps xs;$
 $\forall qs. (\forall xs \in L. prefix qs xs) \longrightarrow size qs \leq size ps \rrbracket$

$\implies \text{Longest-common-prefix } L = ps$

$\langle proof \rangle$

lemma *Longest-common-prefix-prefix*:

$xs \in L \implies prefix (\text{Longest-common-prefix } L) xs$

$\langle \text{proof} \rangle$

lemma *Longest-common-prefix-longest:*

$L \neq \{\} \implies \forall xs \in L. \text{prefix } ps \ xs \implies \text{length } ps \leq \text{length}(\text{Longest-common-prefix } L)$

$\langle \text{proof} \rangle$

lemma *Longest-common-prefix-max-prefix:*

$L \neq \{\} \implies \forall xs \in L. \text{prefix } ps \ xs \implies \text{prefix } ps \ (\text{Longest-common-prefix } L)$

$\langle \text{proof} \rangle$

lemma *Longest-common-prefix-Nil:* $[] \in L \implies \text{Longest-common-prefix } L = []$

$\langle \text{proof} \rangle$

lemma *Longest-common-prefix-image-Cons:* $L \neq \{\} \implies$

$\text{Longest-common-prefix } ((\#) \ x \ 'L) = x \ \# \ \text{Longest-common-prefix } L$

$\langle \text{proof} \rangle$

lemma *Longest-common-prefix-eq-Cons:* **assumes** $L \neq \{\} \ [] \notin L \ \forall xs \in L. \text{hd } xs = x$

shows $\text{Longest-common-prefix } L = x \ \# \ \text{Longest-common-prefix } \{ys. x \ \# \ ys \in L\}$

$\langle \text{proof} \rangle$

lemma *Longest-common-prefix-eq-Nil:*

$\llbracket x \ \# \ ys \in L; y \ \# \ zs \in L; x \neq y \rrbracket \implies \text{Longest-common-prefix } L = []$

$\langle \text{proof} \rangle$

fun *longest-common-prefix* :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ 'a list **where**

$\text{longest-common-prefix } (x \ \# \ xs) \ (y \ \# \ ys) =$

$(\text{if } x=y \text{ then } x \ \# \ \text{longest-common-prefix } xs \ ys \text{ else } []) \mid$

$\text{longest-common-prefix } - - = []$

lemma *longest-common-prefix-prefix1:*

$\text{prefix } (\text{longest-common-prefix } xs \ ys) \ xs$

$\langle \text{proof} \rangle$

lemma *longest-common-prefix-prefix2:*

$\text{prefix } (\text{longest-common-prefix } xs \ ys) \ ys$

$\langle \text{proof} \rangle$

lemma *longest-common-prefix-max-prefix:*

$\llbracket \text{prefix } ps \ xs; \text{prefix } ps \ ys \rrbracket$

$\implies \text{prefix } ps \ (\text{longest-common-prefix } xs \ ys)$

$\langle \text{proof} \rangle$

52.5 Parallel lists

definition *parallel* :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ bool (**infixl** $\parallel$ 50)

where $(xs \parallel ys) = (\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs)$

lemma *parallelI* [intro]: $\neg \text{prefix } xs \ ys \implies \neg \text{prefix } ys \ xs \implies xs \parallel ys$
 ⟨proof⟩

lemma *parallelE* [elim]:
 assumes $xs \parallel ys$
 obtains $\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs$
 ⟨proof⟩

theorem *prefix-cases*:
 obtains $\text{prefix } xs \ ys \mid \text{strict-prefix } ys \ xs \mid xs \parallel ys$
 ⟨proof⟩

lemma *parallel-cancel*: $a \# xs \parallel a \# ys \implies xs \parallel ys$
 ⟨proof⟩

theorem *parallel-decomp*:
 $xs \parallel ys \implies \exists as \ b \ bs \ c \ cs. \ b \neq c \wedge xs = as \ @ \ b \ \# \ bs \wedge ys = as \ @ \ c \ \# \ cs$
 ⟨proof⟩

lemma *parallel-append*: $a \parallel b \implies a \ @ \ c \parallel b \ @ \ d$
 ⟨proof⟩

lemma *parallel-appendI*: $xs \parallel ys \implies x = xs \ @ \ xs' \implies y = ys \ @ \ ys' \implies x \parallel y$
 ⟨proof⟩

lemma *parallel-commute*: $a \parallel b \longleftrightarrow b \parallel a$
 ⟨proof⟩

52.6 Suffix order on lists

definition *suffix* :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ bool
 where $\text{suffix } xs \ ys = (\exists zs. \ ys = zs \ @ \ xs)$

definition *strict-suffix* :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ bool
 where $\text{strict-suffix } xs \ ys \longleftrightarrow \text{suffix } xs \ ys \wedge xs \neq ys$

global-interpretation *suffix-order*: ordering *suffix* *strict-suffix*
 ⟨proof⟩

interpretation *suffix-order*: order *suffix* *strict-suffix*
 ⟨proof⟩

global-interpretation *suffix-bot*: ordering-top $\langle \lambda xs \ ys. \text{suffix } ys \ xs \rangle \langle \lambda xs \ ys. \text{strict-suffix } ys \ xs \rangle \langle [] \rangle$
 ⟨proof⟩

interpretation *suffix-bot*: order-bot Nil *suffix* *strict-suffix*
 ⟨proof⟩

lemma *suffixI* [*intro?*]: $ys = zs @ xs \implies \text{suffix } xs \text{ } ys$
 ⟨*proof*⟩

lemma *suffixE* [*elim?*]:
assumes *suffix xs ys*
obtains *zs where* $ys = zs @ xs$
 ⟨*proof*⟩

lemma *suffix-tl* [*simp*]: $\text{suffix } (tl \text{ } xs) \text{ } xs$
 ⟨*proof*⟩

lemma *strict-suffix-tl* [*simp*]: $xs \neq [] \implies \text{strict-suffix } (tl \text{ } xs) \text{ } xs$
 ⟨*proof*⟩

lemma *Nil-suffix* [*simp*]: $\text{suffix } [] \text{ } xs$
 ⟨*proof*⟩

lemma *suffix-Nil* [*simp*]: $(\text{suffix } xs \text{ } []) = (xs = [])$
 ⟨*proof*⟩

lemma *suffix-ConsI*: $\text{suffix } xs \text{ } ys \implies \text{suffix } xs \text{ } (y \# ys)$
 ⟨*proof*⟩

lemma *suffix-ConsD*: $\text{suffix } (x \# xs) \text{ } ys \implies \text{suffix } xs \text{ } ys$
 ⟨*proof*⟩

lemma *suffix-appendI*: $\text{suffix } xs \text{ } ys \implies \text{suffix } xs \text{ } (zs @ ys)$
 ⟨*proof*⟩

lemma *suffix-appendD*: $\text{suffix } (zs @ xs) \text{ } ys \implies \text{suffix } xs \text{ } ys$
 ⟨*proof*⟩

lemma *strict-suffix-set-subset*: $\text{strict-suffix } xs \text{ } ys \implies \text{set } xs \subseteq \text{set } ys$
 ⟨*proof*⟩

lemma *set-mono-suffix*: $\text{suffix } xs \text{ } ys \implies \text{set } xs \subseteq \text{set } ys$
 ⟨*proof*⟩

lemma *sorted-antimono-suffix*: $\text{suffix } xs \text{ } ys \implies \text{sorted } ys \implies \text{sorted } xs$
 ⟨*proof*⟩

lemma *suffix-ConsD2*: $\text{suffix } (x \# xs) \text{ } (y \# ys) \implies \text{suffix } xs \text{ } ys$
 ⟨*proof*⟩

lemma *suffix-to-prefix* [*code*]: $\text{suffix } xs \text{ } ys \longleftrightarrow \text{prefix } (rev \text{ } xs) \text{ } (rev \text{ } ys)$
 ⟨*proof*⟩

lemma *strict-suffix-to-prefix* [*code*]: $\text{strict-suffix } xs \text{ } ys \longleftrightarrow \text{strict-prefix } (rev \text{ } xs) \text{ } (rev \text{ } ys)$

$ys)$
 $\langle proof \rangle$

lemma *distinct-suffix*: $distinct\ ys \implies suffix\ xs\ ys \implies distinct\ xs$
 $\langle proof \rangle$

lemma *map-mono-suffix*: $suffix\ xs\ ys \implies suffix\ (map\ f\ xs)\ (map\ f\ ys)$
 $\langle proof \rangle$

lemma *filter-mono-suffix*: $suffix\ xs\ ys \implies suffix\ (filter\ P\ xs)\ (filter\ P\ ys)$
 $\langle proof \rangle$

lemma *suffix-drop*: $suffix\ (drop\ n\ as)\ as$
 $\langle proof \rangle$

lemma *suffix-dropWhile*: $suffix\ (dropWhile\ P\ xs)\ xs$
 $\langle proof \rangle$

lemma *suffix-take*: $suffix\ xs\ ys \implies ys = take\ (length\ ys - length\ xs)\ ys @ xs$
 $\langle proof \rangle$

lemma *strict-suffix-reflclp-conv*: $strict\_suffix^{==} = suffix$
 $\langle proof \rangle$

lemma *suffix-lists*: $suffix\ xs\ ys \implies ys \in lists\ A \implies xs \in lists\ A$
 $\langle proof \rangle$

lemma *suffix-snoc [simp]*: $suffix\ xs\ (ys @ [y]) \longleftrightarrow xs = [] \vee (\exists zs. xs = zs @ [y] \wedge suffix\ zs\ ys)$
 $\langle proof \rangle$

lemma *snoc-suffix-snoc [simp]*: $suffix\ (xs @ [x])\ (ys @ [y]) = (x = y \wedge suffix\ xs\ ys)$
 $\langle proof \rangle$

lemma *same-suffix-suffix [simp]*: $suffix\ (ys @ xs)\ (zs @ xs) = suffix\ ys\ zs$
 $\langle proof \rangle$

lemma *same-suffix-nil [simp]*: $suffix\ (ys @ xs)\ xs = (ys = [])$
 $\langle proof \rangle$

theorem *suffix-Cons*: $suffix\ xs\ (y \# ys) \longleftrightarrow xs = y \# ys \vee suffix\ xs\ ys$
 $\langle proof \rangle$

theorem *suffix-append*:
 $suffix\ xs\ (ys @ zs) \longleftrightarrow suffix\ xs\ zs \vee (\exists xs'. xs = xs' @ zs \wedge suffix\ xs'\ ys)$
 $\langle proof \rangle$

theorem *suffix-length-le*: $suffix\ xs\ ys \implies length\ xs \leq length\ ys$

$\langle proof \rangle$

lemma *suffix-same-cases*:

$suffix\ (xs_1::'a\ list)\ ys \implies suffix\ xs_2\ ys \implies suffix\ xs_1\ xs_2 \vee suffix\ xs_2\ xs_1$
 $\langle proof \rangle$

lemma *suffix-length-suffix*:

$suffix\ ps\ xs \implies suffix\ qs\ xs \implies length\ ps \leq length\ qs \implies suffix\ ps\ qs$
 $\langle proof \rangle$

lemma *suffix-length-less*: $strict\_suffix\ xs\ ys \implies length\ xs < length\ ys$

$\langle proof \rangle$

lemma *suffix-ConsD'*: $suffix\ (x\#\!xs)\ ys \implies strict\_suffix\ xs\ ys$

$\langle proof \rangle$

lemma *drop-strict-suffix*: $strict\_suffix\ xs\ ys \implies strict\_suffix\ (drop\ n\ xs)\ ys$

$\langle proof \rangle$

lemma *suffix-map-rightE*:

assumes $suffix\ xs\ (map\ f\ ys)$

shows $\exists xs'. suffix\ xs'\ ys \wedge xs = map\ f\ xs'$

$\langle proof \rangle$

lemma *suffix-remdups-adj*: $suffix\ xs\ ys \implies suffix\ (remdups\_adj\ xs)\ (remdups\_adj\ ys)$

$\langle proof \rangle$

lemma *not-suffix-cases*:

assumes $pf\ x: \neg suffix\ ps\ ls$

obtains

$(c1)\ ps \neq []\ \text{and}\ ls = []$

$| (c2)\ a\ as\ x\ xs\ \text{where}\ ps = as@[a]\ \text{and}\ ls = xs@[x]\ \text{and}\ x = a\ \text{and}\ \neg suffix\ as\ xs$

$| (c3)\ a\ as\ x\ xs\ \text{where}\ ps = as@[a]\ \text{and}\ ls = xs@[x]\ \text{and}\ x \neq a$

$\langle proof \rangle$

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 $r1: \bigwedge x\ xs\ y\ ys. x \neq y \implies P\ (xs@[x])\ (ys@[y])$

and $r2: \bigwedge x\ xs\ y\ ys. \llbracket x = y; \neg suffix\ xs\ ys; P\ xs\ ys \rrbracket \implies P\ (xs@[x])\ (ys@[y])$

shows $P\ ps\ ls\ \langle proof \rangle$

lemma *parallelD1*: $x \parallel y \implies \neg prefix\ x\ y$

$\langle proof \rangle$

lemma *parallelD2*: $x \parallel y \implies \neg prefix\ y\ x$

$\langle \text{proof} \rangle$

lemma *parallel-Nil1* [simp]: $\neg x \parallel []$
 $\langle \text{proof} \rangle$

lemma *parallel-Nil2* [simp]: $\neg [] \parallel x$
 $\langle \text{proof} \rangle$

lemma *Cons-parallelI1*: $a \neq b \implies a \# as \parallel b \# bs$
 $\langle \text{proof} \rangle$

lemma *Cons-parallelI2*: $\llbracket a = b; as \parallel bs \rrbracket \implies a \# as \parallel b \# bs$
 $\langle \text{proof} \rangle$

lemma *not-equal-is-parallel*:
 assumes *neg*: $xs \neq ys$
 and *len*: $\text{length } xs = \text{length } ys$
 shows $xs \parallel ys$
 $\langle \text{proof} \rangle$

52.7 Suffixes

primrec *suffixes* where
 $\text{suffixes } [] = [[]]$
 $| \text{suffixes } (x \# xs) = \text{suffixes } xs @ [x \# xs]$

lemma *in-set-suffixes* [simp]: $xs \in \text{set } (\text{suffixes } ys) \longleftrightarrow \text{suffix } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *distinct-suffixes* [intro]: $\text{distinct } (\text{suffixes } xs)$
 $\langle \text{proof} \rangle$

lemma *length-suffixes* [simp]: $\text{length } (\text{suffixes } xs) = \text{Suc } (\text{length } xs)$
 $\langle \text{proof} \rangle$

lemma *suffixes-snoc* [simp]: $\text{suffixes } (xs @ [x]) = [] \# \text{map } (\lambda ys. ys @ [x]) (\text{suffixes } xs)$
 $\langle \text{proof} \rangle$

lemma *suffixes-not-Nil* [simp]: $\text{suffixes } xs \neq []$
 $\langle \text{proof} \rangle$

lemma *hd-suffixes* [simp]: $\text{hd } (\text{suffixes } xs) = []$
 $\langle \text{proof} \rangle$

lemma *last-suffixes* [simp]: $\text{last } (\text{suffixes } xs) = xs$
 $\langle \text{proof} \rangle$

lemma *suffixes-append*:

$\text{suffixes } (xs @ ys) = \text{suffixes } ys @ \text{map } (\lambda xs'. xs' @ ys) (tl (\text{suffixes } xs))$
 $\langle \text{proof} \rangle$

lemma *suffixes-eq-snoc*:

$\text{suffixes } ys = xs @ [x] \longleftrightarrow$
 $(ys = [] \wedge xs = [] \vee (\exists z zs. ys = z \# zs \wedge xs = \text{suffixes } zs)) \wedge x = ys$
 $\langle \text{proof} \rangle$

lemma *suffixes-tailrec* [code]:

$\text{suffixes } xs = \text{rev } (\text{snd } (\text{foldl } (\lambda(acc1, acc2) x. (x \# acc1, (x \# acc1) \# acc2)) ([], []))$
 $(\text{rev } xs))$
 $\langle \text{proof} \rangle$

lemma *set-suffixes-eq*: $\text{set } (\text{suffixes } xs) = \{ys. \text{suffix } ys \ xs\}$
 $\langle \text{proof} \rangle$

lemma *card-set-suffixes* [simp]: $\text{card } (\text{set } (\text{suffixes } xs)) = \text{Suc } (\text{length } xs)$
 $\langle \text{proof} \rangle$

lemma *set-suffixes-append*:

$\text{set } (\text{suffixes } (xs @ ys)) = \text{set } (\text{suffixes } ys) \cup \{xs' @ ys \mid xs'. xs' \in \text{set } (\text{suffixes } xs)\}$
 $\langle \text{proof} \rangle$

lemma *suffixes-conv-prefixes*: $\text{suffixes } xs = \text{map rev } (\text{prefixes } (\text{rev } xs))$
 $\langle \text{proof} \rangle$

lemma *prefixes-conv-suffixes*: $\text{prefixes } xs = \text{map rev } (\text{suffixes } (\text{rev } xs))$
 $\langle \text{proof} \rangle$

lemma *prefixes-rev*: $\text{prefixes } (\text{rev } xs) = \text{map rev } (\text{suffixes } xs)$
 $\langle \text{proof} \rangle$

lemma *suffixes-rev*: $\text{suffixes } (\text{rev } xs) = \text{map rev } (\text{prefixes } xs)$
 $\langle \text{proof} \rangle$

52.8 Homeomorphic embedding on lists

inductive *list-emb* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$
for $P :: ('a \Rightarrow 'a \Rightarrow \text{bool})$

where

list-emb-Nil [intro, simp]: $\text{list-emb } P [] ys$
 $| \text{list-emb-Cons}$ [intro] : $\text{list-emb } P xs ys \Longrightarrow \text{list-emb } P xs (y \# ys)$
 $| \text{list-emb-Cons2}$ [intro]: $P x y \Longrightarrow \text{list-emb } P xs ys \Longrightarrow \text{list-emb } P (x \# xs) (y \# ys)$

lemma *list-emb-mono*:

assumes $\bigwedge x y. P x y \longrightarrow Q x y$
shows $\text{list-emb } P xs ys \longrightarrow \text{list-emb } Q xs ys$

$\langle proof \rangle$

lemma *list-emb-Nil2* [simp]:
 assumes *list-emb* *P* *xs* [] **shows** *xs* = []
 $\langle proof \rangle$

lemma *list-emb-refl*:
 assumes $\bigwedge x. x \in \text{set } xs \implies P \ x \ x$
 shows *list-emb* *P* *xs* *xs*
 $\langle proof \rangle$

lemma *list-emb-Cons-Nil* [simp]: *list-emb* *P* (*x* # *xs*) [] = *False*
 $\langle proof \rangle$

lemma *list-emb-append2* [intro]: *list-emb* *P* *xs* *ys* $\implies$ *list-emb* *P* *xs* (*zs* @ *ys*)
 $\langle proof \rangle$

lemma *list-emb-prefix* [intro]:
 assumes *list-emb* *P* *xs* *ys* **shows** *list-emb* *P* *xs* (*ys* @ *zs*)
 $\langle proof \rangle$

lemma *list-emb-ConsD*:
 assumes *list-emb* *P* (*x* # *xs*) *ys*
 shows $\exists us \ v \ vs. \ ys = us \ @ \ v \ \# \ vs \wedge P \ x \ v \wedge \text{list-emb } P \ xs \ vs$
 $\langle proof \rangle$

lemma *list-emb-appendD*:
 assumes *list-emb* *P* (*xs* @ *ys*) *zs*
 shows $\exists us \ vs. \ zs = us \ @ \ vs \wedge \text{list-emb } P \ xs \ us \wedge \text{list-emb } P \ ys \ vs$
 $\langle proof \rangle$

lemma *list-emb-strict-suffix*:
 assumes *list-emb* *P* *xs* *ys* **and** *strict-suffix* *ys* *zs*
 shows *list-emb* *P* *xs* *zs*
 $\langle proof \rangle$

lemma *list-emb-suffix*:
 assumes *list-emb* *P* *xs* *ys* **and** *suffix* *ys* *zs*
 shows *list-emb* *P* *xs* *zs*
 $\langle proof \rangle$

lemma *list-emb-length*: *list-emb* *P* *xs* *ys* $\implies \text{length } xs \leq \text{length } ys$
 $\langle proof \rangle$

lemma *list-emb-trans*:
 assumes $\bigwedge x \ y \ z. \llbracket x \in \text{set } xs; y \in \text{set } ys; z \in \text{set } zs; P \ x \ y; P \ y \ z \rrbracket \implies P \ x \ z$
 shows $\llbracket \text{list-emb } P \ xs \ ys; \text{list-emb } P \ ys \ zs \rrbracket \implies \text{list-emb } P \ xs \ zs$
 $\langle proof \rangle$

lemma *list-emb-set*:

assumes *list-emb* P xs ys **and** $x \in \text{set } xs$

obtains y **where** $y \in \text{set } ys$ **and** $P\ x\ y$

<proof>

lemma *list-emb-Cons-iff1* [*simp*]:

assumes $P\ x\ y$

shows $\text{list-emb } P\ (x\#xs)\ (y\#ys) \longleftrightarrow \text{list-emb } P\ xs\ ys$

<proof>

lemma *list-emb-Cons-iff2* [*simp*]:

assumes $\neg P\ x\ y$

shows $\text{list-emb } P\ (x\#xs)\ (y\#ys) \longleftrightarrow \text{list-emb } P\ (x\#xs)\ ys$

<proof>

lemma *list-emb-code* [*code*]:

$\text{list-emb } P\ []\ ys \longleftrightarrow \text{True}$

$\text{list-emb } P\ (x\#xs)\ [] \longleftrightarrow \text{False}$

$\text{list-emb } P\ (x\#xs)\ (y\#ys) \longleftrightarrow (\text{if } P\ x\ y \text{ then } \text{list-emb } P\ xs\ ys \text{ else } \text{list-emb } P\ (x\#xs)\ ys)$

<proof>

52.9 Subsequences (special case of homeomorphic embedding)

abbreviation *subseq* :: $'a\ \text{list} \Rightarrow 'a\ \text{list} \Rightarrow \text{bool}$

where $\text{subseq } xs\ ys \equiv \text{list-emb } (=)\ xs\ ys$

definition *strict-subseq* **where** $\text{strict-subseq } xs\ ys \longleftrightarrow xs \neq ys \wedge \text{subseq } xs\ ys$

lemma *subseq-Cons2*: $\text{subseq } xs\ ys \implies \text{subseq } (x\#xs)\ (x\#ys)$ *<proof>*

lemma *subseq-same-length*:

assumes $\text{subseq } xs\ ys$ **and** $\text{length } xs = \text{length } ys$ **shows** $xs = ys$

<proof>

lemma *not-subseq-length* [*simp*]: $\text{length } ys < \text{length } xs \implies \neg \text{subseq } xs\ ys$

<proof>

lemma *subseq-Cons'*: $\text{subseq } (x\#xs)\ ys \implies \text{subseq } xs\ ys$

<proof>

lemma *subseq-Cons2'*:

assumes $\text{subseq } (x\#xs)\ (x\#ys)$ **shows** $\text{subseq } xs\ ys$

<proof>

lemma *subseq-Cons2-neg*:

assumes $\text{subseq } (x\#xs)\ (y\#ys)$

shows $x \neq y \implies \text{subseq } (x\#xs)\ ys$

$\langle \text{proof} \rangle$

lemma *subseq-Cons2-iff* [simp]:

$\text{subseq } (x\#xs) (y\#ys) = (\text{if } x = y \text{ then } \text{subseq } xs \ ys \text{ else } \text{subseq } (x\#xs) \ ys)$

$\langle \text{proof} \rangle$

lemma *subseq-append'*: $\text{subseq } (zs @ xs) (zs @ ys) \longleftrightarrow \text{subseq } xs \ ys$

$\langle \text{proof} \rangle$

global-interpretation *subseq-order*: *ordering subseq strict-subseq*

$\langle \text{proof} \rangle$

interpretation *subseq-order*: *order subseq strict-subseq*

$\langle \text{proof} \rangle$

lemma *in-set-subseqs* [simp]: $xs \in \text{set } (\text{subseqs } ys) \longleftrightarrow \text{subseq } xs \ ys$

$\langle \text{proof} \rangle$

lemma *set-subseqs-eq*: $\text{set } (\text{subseqs } ys) = \{xs. \text{subseq } xs \ ys\}$

$\langle \text{proof} \rangle$

lemma *subseq-append-le-same-iff*: $\text{subseq } (xs @ ys) \ ys \longleftrightarrow xs = []$

$\langle \text{proof} \rangle$

lemma *subseq-singleton-left*: $\text{subseq } [x] \ ys \longleftrightarrow x \in \text{set } ys$

$\langle \text{proof} \rangle$

lemma *list-emb-append-mono*:

$\llbracket \text{list-emb } P \ xs \ xs'; \text{list-emb } P \ ys \ ys' \rrbracket \Longrightarrow \text{list-emb } P \ (xs@ys) \ (xs'@ys')$

$\langle \text{proof} \rangle$

lemma *prefix-imp-subseq* [intro]: $\text{prefix } xs \ ys \Longrightarrow \text{subseq } xs \ ys$

$\langle \text{proof} \rangle$

lemma *suffix-imp-subseq* [intro]: $\text{suffix } xs \ ys \Longrightarrow \text{subseq } xs \ ys$

$\langle \text{proof} \rangle$

52.10 Appending elements

lemma *subseq-append* [simp]:

$\text{subseq } (xs @ zs) (ys @ zs) \longleftrightarrow \text{subseq } xs \ ys \ (\text{is } ?l = ?r)$

$\langle \text{proof} \rangle$

lemma *subseq-append-iff*:

$\text{subseq } xs \ (ys @ zs) \longleftrightarrow (\exists xs1 \ xs2. xs = xs1 @ xs2 \wedge \text{subseq } xs1 \ ys \wedge \text{subseq } xs2 \ zs)$

$(\text{is } ?lhs = ?rhs)$

$\langle \text{proof} \rangle$

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 $\implies$ subseq xs (zs @ ys)*
<proof>

lemma *subseq-rev-drop-many*: *subseq xs ys $\implies$ subseq xs (ys @ zs)*
<proof>

52.11 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 $\longleftrightarrow (\exists N. xs = nths ys N) \text{ (is } ?L = ?R)$
<proof>

52.12 Contiguous sublists

52.12.1 sublist

definition *sublist* :: '*a list* $\Rightarrow$ '*a list* $\Rightarrow$ bool **where**
sublist xs ys = ($\exists ps ss. ys = ps @ xs @ ss$)

definition *strict-sublist* :: '*a list* $\Rightarrow$ '*a list* $\Rightarrow$ bool **where**
strict-sublist xs ys $\longleftrightarrow$ sublist xs ys $\wedge$ xs $\neq$ ys

interpretation *sublist-order*: *order sublist strict-sublist*
<proof>

lemma *sublist-Nil-left* [*simp, intro*]: *sublist [] ys*
<proof>

lemma *sublist-Cons-Nil* [*simp*]: $\neg$ *sublist (x#xs) []*
<proof>

lemma *sublist-Nil-right* [*simp*]: *sublist xs [] $\longleftrightarrow$ 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* $\longleftrightarrow$ $(\exists ys'. \text{prefix } ys' \text{ } ys \wedge \text{suffix } xs \text{ } ys')$
 ⟨*proof*⟩

lemma *sublist-altdef'*: *sublist xs ys* $\longleftrightarrow$ $(\exists ys'. \text{suffix } ys' \text{ } ys \wedge \text{prefix } xs \text{ } ys')$
 ⟨*proof*⟩

lemma *sublist-Cons-right*: *sublist xs (y # ys)* $\longleftrightarrow$ *prefix xs (y # ys)* $\vee$ *sublist xs ys*
 ⟨*proof*⟩

lemma *sublist-code* [*code*]:
sublist [] ys $\longleftrightarrow$ *True*
sublist (x # xs) [] $\longleftrightarrow$ *False*
sublist (x # xs) (y # ys) $\longleftrightarrow$ *prefix (x # xs) (y # ys)* $\vee$ *sublist (x # xs) ys*
 ⟨*proof*⟩

lemma *sublist-append*:
sublist xs (ys @ zs) $\longleftrightarrow$
sublist xs ys $\vee$ *sublist xs zs* $\vee$ $(\exists xs1 \text{ } xs2. \text{xs} = \text{xs1} @ \text{xs2} \wedge \text{suffix } \text{xs1} \text{ } \text{ys} \wedge \text{prefix } \text{xs2} \text{ } \text{zs})$
 ⟨*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* $\implies$ *length xs* $\leq$ *length ys*
 ⟨*proof*⟩

lemma *set-mono-sublist*: *sublist xs ys* $\implies$ *set xs* $\subseteq$ *set ys*
 ⟨*proof*⟩

lemma *prefix-imp-sublist* [*simp*, *intro*]: *prefix xs ys* $\implies$ *sublist xs ys*
 ⟨*proof*⟩

lemma *suffix-imp-sublist* [*simp*, *intro*]: *suffix xs ys* $\implies$ *sublist xs ys*
 ⟨*proof*⟩

lemma *sublist-take* [*simp*, *intro*]: *sublist (take n xs) xs*

$\langle \text{proof} \rangle$

lemma *sublist-takeWhile* [*simp*, *intro*]: *sublist (takeWhile P xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-drop* [*simp*, *intro*]: *sublist (drop n xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-dropWhile* [*simp*, *intro*]: *sublist (dropWhile P xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-tl* [*simp*, *intro*]: *sublist (tl xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-butlast* [*simp*, *intro*]: *sublist (butlast xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-rev* [*simp*]: *sublist (rev xs) (rev ys) = sublist xs ys*
 $\langle \text{proof} \rangle$

lemma *sublist-rev-left*: *sublist (rev xs) ys = sublist xs (rev ys)*
 $\langle \text{proof} \rangle$

lemma *sublist-rev-right*: *sublist xs (rev ys) = sublist (rev xs) ys*
 $\langle \text{proof} \rangle$

lemma *snoc-sublist-snoc*:
 $\text{sublist } (xs @ [x]) (ys @ [y]) \longleftrightarrow$
 $(x = y \wedge \text{suffix } xs \text{ } ys \vee \text{sublist } (xs @ [x]) \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *sublist-snoc*:
 $\text{sublist } xs (ys @ [y]) \longleftrightarrow \text{suffix } xs (ys @ [y]) \vee \text{sublist } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *sublist-imp-subseq* [*intro*]: *sublist xs ys $\implies$ subseq xs ys*
 $\langle \text{proof} \rangle$

lemma *sublist-map-rightE*:
assumes *sublist xs (map f ys)*
shows $\exists xs'. \text{sublist } xs' \text{ } ys \wedge xs = \text{map } f \text{ } xs'$
 $\langle \text{proof} \rangle$

lemma *sublist-remdups-adj*:
assumes *sublist xs ys*
shows *sublist (remdups-adj xs) (remdups-adj ys)*
 $\langle \text{proof} \rangle$

52.12.2 *sublists*

primrec *sublists* :: 'a list $\Rightarrow$ 'a list list **where**

sublists [] = []
 | *sublists* (x # xs) = *sublists* xs @ map ((#) x) (*prefixes* xs)

lemma *in-set-sublists* [simp]: $xs \in \text{set } (\text{sublists } ys) \longleftrightarrow \text{sublist } xs \text{ } ys$
 <proof>

lemma *set-sublists-eq*: $\text{set } (\text{sublists } xs) = \{ys. \text{sublist } ys \text{ } xs\}$
 <proof>

lemma *length-sublists* [simp]: $\text{length } (\text{sublists } xs) = \text{Suc } (\text{length } xs * \text{Suc } (\text{length } xs) \text{ div } 2)$
 <proof>

52.13 Parametricity

context includes *lifting-syntax*
begin

private lemma *prefix-primrec*:

prefix = *rec-list* ($\lambda xs. \text{True}$) ($\lambda x \text{ } xs \text{ } xsa \text{ } ys.$
 case *ys* of [] $\Rightarrow$ *False* | *y* # *ys* $\Rightarrow$ $x = y \wedge xsa \text{ } ys$)

<proof> **lemma** *sublist-primrec*:

sublist = ($\lambda xs \text{ } ys. \text{rec-list } (\lambda xs. xs = [])$) ($\lambda y \text{ } ys \text{ } ysa \text{ } xs. \text{prefix } xs \text{ } (y \# ys) \vee ysa \text{ } xs$) *ys* *xs*)

<proof> **lemma** *list-emb-primrec*:

list-emb = ($\lambda uu \text{ } uua \text{ } uuaa. \text{rec-list } (\lambda P \text{ } xs. \text{List.null } xs)$) ($\lambda y \text{ } ys \text{ } ysa \text{ } P \text{ } xs. \text{case } xs$
 of [] $\Rightarrow$ *True*
 | *x* # *xs* $\Rightarrow$ *if* *P* *x* *y* *then* *ysa* *P* *xs* *else* *ysa* *P* (*x* # *xs*)) *uuaa* *uu* *uua*)
 <proof>

lemma *prefix-transfer* [transfer-rule]:

assumes [transfer-rule]: *bi-unique* *A*

shows ($\text{list-all2 } A \implies \text{list-all2 } A \implies (=)$) *prefix* *prefix*

<proof>

lemma *suffix-transfer* [transfer-rule]:

assumes [transfer-rule]: *bi-unique* *A*

shows ($\text{list-all2 } A \implies \text{list-all2 } A \implies (=)$) *suffix* *suffix*

<proof>

lemma *sublist-transfer* [transfer-rule]:

assumes [transfer-rule]: *bi-unique* *A*

shows ($\text{list-all2 } A \implies \text{list-all2 } A \implies (=)$) *sublist* *sublist*

<proof>

lemma *parallel-transfer* [transfer-rule]:

assumes [transfer-rule]: *bi-unique* *A*

shows $(list-all2\ A ==> list-all2\ A ==> (=))\ parallel\ parallel$
 $\langle proof \rangle$

lemma *list-emb-transfer* [*transfer-rule*]:
 $((A ==> A ==> (=)) ==> list-all2\ A ==> list-all2\ A ==> (=))$
list-emb list-emb
 $\langle proof \rangle$

lemma *strict-prefix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(list-all2\ A ==> list-all2\ A ==> (=))\ strict-prefix\ strict-prefix$
 $\langle proof \rangle$

lemma *strict-suffix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(list-all2\ A ==> list-all2\ A ==> (=))\ strict-suffix\ strict-suffix$
 $\langle proof \rangle$

lemma *strict-subseq-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(list-all2\ A ==> list-all2\ A ==> (=))\ strict-subseq\ strict-subseq$
 $\langle proof \rangle$

lemma *strict-sublist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(list-all2\ A ==> list-all2\ A ==> (=))\ strict-sublist\ strict-sublist$
 $\langle proof \rangle$

lemma *prefixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(list-all2\ A ==> list-all2\ (list-all2\ A))\ prefixes\ prefixes$
 $\langle proof \rangle$

lemma *suffixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(list-all2\ A ==> list-all2\ (list-all2\ A))\ suffixes\ suffixes$
 $\langle proof \rangle$

lemma *sublists-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(list-all2\ A ==> list-all2\ (list-all2\ A))\ sublists\ sublists$
 $\langle proof \rangle$

end

end

53 Linear Temporal Logic on Streams

theory *Linear-Temporal-Logic-on-Streams*
imports *Stream Sublist Extended-Nat Infinite-Set*
begin

54 Preliminaries

lemma *shift-prefix*:
assumes $xl @- xs = yl @- ys$ **and** $\text{length } xl \leq \text{length } yl$
shows $\text{prefix } xl \ yl$
 $\langle \text{proof} \rangle$

lemma *shift-prefix-cases*:
assumes $xl @- xs = yl @- ys$
shows $\text{prefix } xl \ yl \vee \text{prefix } yl \ xl$
 $\langle \text{proof} \rangle$

55 Linear temporal logic

Propositional connectives:

abbreviation (*input*) *IMPL* (**infix** *impl* 60)
where $\varphi \text{ impl } \psi \equiv \lambda xs. \varphi \ xs \longrightarrow \psi \ xs$

abbreviation (*input*) *OR* (**infix** *or* 60)
where $\varphi \text{ or } \psi \equiv \lambda xs. \varphi \ xs \vee \psi \ xs$

abbreviation (*input*) *AND* (**infix** *aand* 60)
where $\varphi \text{ aand } \psi \equiv \lambda xs. \varphi \ xs \wedge \psi \ xs$

abbreviation (*input*) *not* **where** $\text{not } \varphi \equiv \lambda xs. \neg \varphi \ xs$

abbreviation (*input*) *true* $\equiv \lambda xs. \text{True}$

abbreviation (*input*) *false* $\equiv \lambda xs. \text{False}$

lemma *impl-not-or*: $\varphi \text{ impl } \psi = (\text{not } \varphi) \text{ or } \psi$
 $\langle \text{proof} \rangle$

lemma *not-or*: $\text{not } (\varphi \text{ or } \psi) = (\text{not } \varphi) \text{ aand } (\text{not } \psi)$
 $\langle \text{proof} \rangle$

lemma *not-aand*: $\text{not } (\varphi \text{ aand } \psi) = (\text{not } \varphi) \text{ or } (\text{not } \psi)$
 $\langle \text{proof} \rangle$

lemma *non-not[simp]*: $\text{not } (\text{not } \varphi) = \varphi$ $\langle \text{proof} \rangle$

Temporal (LTL) connectives:

fun *holds* **where** *holds* $P\ xs \longleftrightarrow P\ (shd\ xs)$
fun *next* **where** *next* $\varphi\ xs = \varphi\ (stl\ xs)$

definition $HLD\ s = holds\ (\lambda x. x \in s)$

abbreviation $HLD\text{-}next$ (**infixr** $\cdot\ 65$) **where**
 $s \cdot P \equiv HLD\ s\ aand\ next\ P$

context
notes $[[inductive\text{-}internals]]$
begin

inductive *ev* **for** φ **where**
base: $\varphi\ xs \Longrightarrow ev\ \varphi\ xs$
 $|$
step: $ev\ \varphi\ (stl\ xs) \Longrightarrow ev\ \varphi\ xs$

coinductive *alw* **for** φ **where**
alw: $[[\varphi\ xs; alw\ \varphi\ (stl\ xs)]] \Longrightarrow alw\ \varphi\ xs$

— weak until:

coinductive *UNTIL* (**infix** *until* 60) **for** $\varphi\ \psi$ **where**
base: $\psi\ xs \Longrightarrow (\varphi\ until\ \psi)\ xs$
 $|$
step: $[[\varphi\ xs; (\varphi\ until\ \psi)\ (stl\ xs)]] \Longrightarrow (\varphi\ until\ \psi)\ xs$

end

lemma *holds-mono*:
assumes *holds*: *holds* $P\ xs$ **and** $0: \bigwedge x. P\ x \Longrightarrow Q\ x$
shows *holds* $Q\ xs$
 $\langle proof \rangle$

lemma *holds-aand*:
 $(holds\ P\ aand\ holds\ Q)\ steps \longleftrightarrow holds\ (\lambda step. P\ step \wedge Q\ step)\ steps\ \langle proof \rangle$

lemma *HLD-iff*: $HLD\ s\ \omega \longleftrightarrow shd\ \omega \in s$
 $\langle proof \rangle$

lemma *HLD-Stream[simp]*: $HLD\ X\ (x\ \#\#\ \omega) \longleftrightarrow x \in X$
 $\langle proof \rangle$

lemma *next-mono*:
assumes *next*: *next* $\varphi\ xs$ **and** $0: \bigwedge xs. \varphi\ xs \Longrightarrow \psi\ xs$
shows *next* $\psi\ xs$
 $\langle proof \rangle$

declare *ev.intros* $[intro]$
declare *alw.cases* $[elim]$

lemma *ev-induct-strong*[consumes 1, case-names base step]:

$ev\ \varphi\ x \implies (\bigwedge xs. \varphi\ xs \implies P\ xs) \implies (\bigwedge xs. ev\ \varphi\ (stl\ xs) \implies \neg\ \varphi\ xs \implies P\ (stl\ xs)) \implies P\ xs \implies P\ x$
 ⟨proof⟩

lemma *alw-coinduct*[consumes 1, case-names alw stl]:

$X\ x \implies (\bigwedge x. X\ x \implies \varphi\ x) \implies (\bigwedge x. X\ x \implies \neg\ alw\ \varphi\ (stl\ x) \implies X\ (stl\ x)) \implies alw\ \varphi\ x$
 ⟨proof⟩

lemma *ev-mono*:

assumes *ev*: $ev\ \varphi\ xs$ **and** *0*: $\bigwedge xs. \varphi\ xs \implies \psi\ xs$

shows $ev\ \psi\ xs$

⟨proof⟩

lemma *alw-mono*:

assumes *alw*: $alw\ \varphi\ xs$ **and** *0*: $\bigwedge xs. \varphi\ xs \implies \psi\ xs$

shows $alw\ \psi\ xs$

⟨proof⟩

lemma *until-monoL*:

assumes *until*: $(\varphi\ 1\ until\ \psi)\ xs$ **and** *0*: $\bigwedge xs. \varphi\ 1\ xs \implies \varphi\ 2\ xs$

shows $(\varphi\ 2\ until\ \psi)\ xs$

⟨proof⟩

lemma *until-monoR*:

assumes *until*: $(\varphi\ until\ \psi\ 1)\ xs$ **and** *0*: $\bigwedge xs. \psi\ 1\ xs \implies \psi\ 2\ xs$

shows $(\varphi\ until\ \psi\ 2)\ xs$

⟨proof⟩

lemma *until-mono*:

assumes *until*: $(\varphi\ 1\ until\ \psi\ 1)\ xs$ **and**

0: $\bigwedge xs. \varphi\ 1\ xs \implies \varphi\ 2\ xs \bigwedge xs. \psi\ 1\ xs \implies \psi\ 2\ xs$

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]*: $alw (alw \varphi) = alw \varphi$
 $\langle proof \rangle$

lemma *ev-shift*:
assumes $ev \varphi xs$
shows $ev \varphi (xl @- xs)$
 $\langle proof \rangle$

lemma *ev-imp-shift*:
assumes $ev \varphi xs$ **shows** $\exists xl xs2. xs = xl @- xs2 \wedge \varphi xs2$
 $\langle proof \rangle$

lemma *alw-ev-shift*: $alw \varphi xs1 \implies ev (alw \varphi) (xl @- xs1)$
 $\langle proof \rangle$

lemma *alw-shift*:
assumes $alw \varphi (xl @- xs)$
shows $alw \varphi xs$
 $\langle proof \rangle$

lemma *ev-ex-nxt*:
assumes $ev \varphi xs$
shows $\exists n. (nxt \rightsquigarrow n) \varphi xs$
 $\langle proof \rangle$

lemma *alw-sdrop*:
assumes $alw \varphi xs$ **shows** $alw \varphi (sdrop n xs)$
 $\langle proof \rangle$

lemma *nxt-sdrop*: $(nxt \rightsquigarrow n) \varphi xs \longleftrightarrow \varphi (sdrop n xs)$
 $\langle proof \rangle$

definition $wait \varphi xs \equiv LEAST n. (nxt \rightsquigarrow n) \varphi xs$

lemma *nxt-wait*:
assumes $ev \varphi xs$ **shows** $(nxt \rightsquigarrow (wait \varphi xs)) \varphi xs$
 $\langle proof \rangle$

lemma *nxt-wait-least*:
assumes $ev: ev \varphi xs$ **and** $nxt: (nxt \rightsquigarrow n) \varphi xs$ **shows** $wait \varphi xs \leq n$
 $\langle proof \rangle$

lemma *sdrop-wait*:
assumes $ev \varphi xs$ **shows** $\varphi (sdrop (wait \varphi xs) xs)$
 $\langle proof \rangle$

lemma *sdrop-wait-least*:
assumes $ev: ev \varphi xs$ **and** $nxt: \varphi (sdrop n xs)$ **shows** $wait \varphi xs \leq n$
 $\langle proof \rangle$

lemma *next-ev*: $(\text{next} \leadsto n) \varphi \text{ xs} \implies \text{ev } \varphi \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *not-ev*: $\text{not } (\text{ev } \varphi) = \text{alw } (\text{not } \varphi)$
 $\langle \text{proof} \rangle$

lemma *not-alw*: $\text{not } (\text{alw } \varphi) = \text{ev } (\text{not } \varphi)$
 $\langle \text{proof} \rangle$

lemma *not-ev-not[simp]*: $\text{not } (\text{ev } (\text{not } \varphi)) = \text{alw } \varphi$
 $\langle \text{proof} \rangle$

lemma *not-alw-not[simp]*: $\text{not } (\text{alw } (\text{not } \varphi)) = \text{ev } \varphi$
 $\langle \text{proof} \rangle$

lemma *alw-ev-sdrop*:
assumes $\text{alw } (\text{ev } \varphi) \text{ (sdrop } m \text{ xs)}$
shows $\text{alw } (\text{ev } \varphi) \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *ev-alw-imp-alw-ev*:
assumes $\text{ev } (\text{alw } \varphi) \text{ xs}$ **shows** $\text{alw } (\text{ev } \varphi) \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *alw-aand*: $\text{alw } (\varphi \text{ aand } \psi) = \text{alw } \varphi \text{ aand } \text{alw } \psi$
 $\langle \text{proof} \rangle$

lemma *ev-or*: $\text{ev } (\varphi \text{ or } \psi) = \text{ev } \varphi \text{ or } \text{ev } \psi$
 $\langle \text{proof} \rangle$

lemma *ev-alw-aand*:
assumes $\varphi: \text{ev } (\text{alw } \varphi) \text{ xs}$ **and** $\psi: \text{ev } (\text{alw } \psi) \text{ xs}$
shows $\text{ev } (\text{alw } (\varphi \text{ aand } \psi)) \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *ev-alw-alw-impl*:
assumes $\text{ev } (\text{alw } \varphi) \text{ xs}$ **and** $\text{alw } (\text{alw } \varphi \text{ impl } \text{ev } \psi) \text{ xs}$
shows $\text{ev } \psi \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *ev-alw-stl[simp]*: $\text{ev } (\text{alw } \varphi) \text{ (stl } x) \longleftrightarrow \text{ev } (\text{alw } \varphi) \text{ x}$
 $\langle \text{proof} \rangle$

lemma *alw-alw-impl-ev*:
 $\text{alw } (\text{alw } \varphi \text{ impl } \text{ev } \psi) = (\text{ev } (\text{alw } \varphi) \text{ impl } \text{alw } (\text{ev } \psi)) \text{ (is ?A = ?B)}$
 $\langle \text{proof} \rangle$

lemma *ev-alw-impl*:

assumes $ev\ \varphi\ xs$ **and** $alw\ (\varphi\ impl\ \psi)\ xs$ **shows** $ev\ \psi\ xs$
 $\langle proof \rangle$

lemma *ev-alw-impl-ev*:
assumes $ev\ \varphi\ xs$ **and** $alw\ (\varphi\ impl\ ev\ \psi)\ xs$ **shows** $ev\ \psi\ xs$
 $\langle proof \rangle$

lemma *alw-mp*:
assumes $alw\ \varphi\ xs$ **and** $alw\ (\varphi\ impl\ \psi)\ xs$
shows $alw\ \psi\ xs$
 $\langle proof \rangle$

lemma *all-imp-alw*:
assumes $\bigwedge xs. \varphi\ xs$ **shows** $alw\ \varphi\ xs$
 $\langle proof \rangle$

lemma *alw-impl-ev-alw*:
assumes $alw\ (\varphi\ impl\ ev\ \psi)\ xs$
shows $alw\ (ev\ \varphi\ impl\ ev\ \psi)\ xs$
 $\langle proof \rangle$

lemma *ev-holds-sset*:
 $ev\ (holds\ P)\ xs \longleftrightarrow (\exists x \in sset\ xs. P\ x)$ **(is** $?L \longleftrightarrow ?R)$
 $\langle proof \rangle$

LTL as a program logic:

lemma *alw-invar*:
assumes $\varphi\ xs$ **and** $alw\ (\varphi\ impl\ next\ \varphi)\ xs$
shows $alw\ \varphi\ xs$
 $\langle proof \rangle$

lemma *variance*:
assumes $1: \varphi\ xs$ **and** $2: alw\ (\varphi\ impl\ (\psi\ or\ next\ \varphi))\ xs$
shows $(alw\ \varphi\ or\ ev\ \psi)\ xs$
 $\langle proof \rangle$

lemma *ev-alw-imp-next*:
assumes $e: ev\ \varphi\ xs$ **and** $a: alw\ (\varphi\ impl\ (next\ \varphi))\ xs$
shows $ev\ (alw\ \varphi)\ xs$
 $\langle proof \rangle$

inductive *ev-at* :: $('a\ stream \Rightarrow bool) \Rightarrow nat \Rightarrow 'a\ stream \Rightarrow bool$ **for** $P :: 'a\ stream \Rightarrow bool$ **where**
 $base: P\ \omega \Longrightarrow ev-at\ P\ 0\ \omega$
 $| step: \neg P\ \omega \Longrightarrow ev-at\ P\ n\ (stl\ \omega) \Longrightarrow ev-at\ P\ (Suc\ n)\ \omega$

inductive-simps *ev-at-0[simp]*: $ev-at\ P\ 0\ \omega$
inductive-simps *ev-at-Suc[simp]*: $ev-at\ P\ (Suc\ n)\ \omega$

lemma *ev-at-imp-snth*: $ev\text{-}at\ P\ n\ \omega \implies P\ (sdrop\ n\ \omega)$
 $\langle proof \rangle$

lemma *ev-at-HLD-imp-snth*: $ev\text{-}at\ (HLD\ X)\ n\ \omega \implies \omega !!\ n \in X$
 $\langle proof \rangle$

lemma *ev-at-HLD-single-imp-snth*: $ev\text{-}at\ (HLD\ \{x\})\ n\ \omega \implies \omega !!\ n = x$
 $\langle proof \rangle$

lemma *ev-at-unique*: $ev\text{-}at\ P\ n\ \omega \implies ev\text{-}at\ P\ m\ \omega \implies n = m$
 $\langle proof \rangle$

lemma *ev-iff-ev-at*: $ev\ P\ \omega \longleftrightarrow (\exists n. ev\text{-}at\ P\ n\ \omega)$
 $\langle proof \rangle$

lemma *ev-at-shift*: $ev\text{-}at\ (HLD\ X)\ i\ (stake\ (Suc\ i)\ \omega @-\ \omega' :: 's\ stream) \longleftrightarrow ev\text{-}at\ (HLD\ X)\ i\ \omega$
 $\langle proof \rangle$

lemma *ev-iff-ev-at-unique*: $ev\ P\ \omega \longleftrightarrow (\exists! n. ev\text{-}at\ P\ n\ \omega)$
 $\langle proof \rangle$

lemma *alw-HLD-iff-streams*: $alw\ (HLD\ X)\ \omega \longleftrightarrow \omega \in streams\ X$
 $\langle proof \rangle$

lemma *not-HLD*: $not\ (HLD\ X) = HLD\ (-\ X)$
 $\langle proof \rangle$

lemma *not-alw-iff*: $\neg (alw\ P\ \omega) \longleftrightarrow ev\ (not\ P)\ \omega$
 $\langle proof \rangle$

lemma *not-ev-iff*: $\neg (ev\ P\ \omega) \longleftrightarrow alw\ (not\ P)\ \omega$
 $\langle proof \rangle$

lemma *ev-Stream*: $ev\ P\ (x\ \#\#\ s) \longleftrightarrow P\ (x\ \#\#\ s) \vee ev\ P\ s$
 $\langle proof \rangle$

lemma *alw-ev-imp-ev-alw*:
assumes $alw\ (ev\ P)\ \omega$ **shows** $ev\ (P\ a\ and\ alw\ (ev\ P))\ \omega$
 $\langle proof \rangle$

lemma *ev-False*: $ev\ (\lambda x. False)\ \omega \longleftrightarrow False$
 $\langle proof \rangle$

lemma *alw-False*: $alw\ (\lambda x. False)\ \omega \longleftrightarrow False$
 $\langle proof \rangle$

lemma *ev-iff-sdrop*: $ev\ P\ \omega \longleftrightarrow (\exists m. P\ (sdrop\ m\ \omega))$

$\langle proof \rangle$

lemma *alw-iff-sdrop*: $alw\ P\ \omega \longleftrightarrow (\forall m. P\ (sdrop\ m\ \omega))$
 $\langle proof \rangle$

lemma *infinite-iff-alw-ev*: $infinite\ \{m. P\ (sdrop\ m\ \omega)\} \longleftrightarrow alw\ (ev\ P)\ \omega$
 $\langle proof \rangle$

lemma *alw-inv*:
assumes *stl*: $\bigwedge s. f\ (stl\ s) = stl\ (f\ s)$
shows $alw\ P\ (f\ s) \longleftrightarrow alw\ (\lambda x. P\ (f\ x))\ s$
 $\langle proof \rangle$

lemma *ev-inv*:
assumes *stl*: $\bigwedge s. f\ (stl\ s) = stl\ (f\ s)$
shows $ev\ P\ (f\ s) \longleftrightarrow ev\ (\lambda x. P\ (f\ x))\ s$
 $\langle proof \rangle$

lemma *alw-smap*: $alw\ P\ (smap\ f\ s) \longleftrightarrow alw\ (\lambda x. P\ (smap\ f\ x))\ s$
 $\langle proof \rangle$

lemma *ev-smap*: $ev\ P\ (smap\ f\ s) \longleftrightarrow ev\ (\lambda x. P\ (smap\ f\ x))\ s$
 $\langle proof \rangle$

lemma *alw-cong*:
assumes *P*: $alw\ P\ \omega$ **and** *eq*: $\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$
shows $alw\ Q1\ \omega \longleftrightarrow alw\ Q2\ \omega$
 $\langle proof \rangle$

lemma *ev-cong*:
assumes *P*: $alw\ P\ \omega$ **and** *eq*: $\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$
shows $ev\ Q1\ \omega \longleftrightarrow ev\ Q2\ \omega$
 $\langle proof \rangle$

lemma *alwD*: $alw\ P\ x \implies P\ x$
 $\langle proof \rangle$

lemma *alw-alwD*: $alw\ P\ \omega \implies alw\ (alw\ P)\ \omega$
 $\langle proof \rangle$

lemma *alw-ev-stl*: $alw\ (ev\ P)\ (stl\ \omega) \longleftrightarrow alw\ (ev\ P)\ \omega$
 $\langle proof \rangle$

lemma *holds-Stream*: $holds\ P\ (x\ \#\#\ s) \longleftrightarrow P\ x$
 $\langle proof \rangle$

lemma *holds-eq1[simp]*: $holds\ ((=)\ x) = HLD\ \{x\}$
 $\langle proof \rangle$

lemma *holds-eq2[simp]*: *holds* $(\lambda y. y = x) = \text{HLD } \{x\}$
 ⟨*proof*⟩

lemma *not-holds-eq[simp]*: *holds* $(- (=) x) = \text{not } (\text{HLD } \{x\})$
 ⟨*proof*⟩

Strong until

context

notes $[[\text{inductive-internals}]]$

begin

inductive *suntil* (**infix** *suntil* 60) **for** $\varphi \ \psi$ **where**

base: $\psi \ \omega \implies (\varphi \ \text{suntil} \ \psi) \ \omega$

| *step*: $\varphi \ \omega \implies (\varphi \ \text{suntil} \ \psi) \ (\text{stl} \ \omega) \implies (\varphi \ \text{suntil} \ \psi) \ \omega$

inductive-simps *suntil-Stream*: $(\varphi \ \text{suntil} \ \psi) \ (x \ \#\# \ s)$

end

lemma *suntil-induct-strong*[*consumes 1, case-names base step*]:

$(\varphi \ \text{suntil} \ \psi) \ x \implies$

$(\bigwedge \omega. \psi \ \omega \implies P \ \omega) \implies$

$(\bigwedge \omega. \varphi \ \omega \implies \neg \psi \ \omega \implies (\varphi \ \text{suntil} \ \psi) \ (\text{stl} \ \omega) \implies P \ (\text{stl} \ \omega) \implies P \ \omega) \implies P \ x$

⟨*proof*⟩

lemma *ev-suntil*: $(\varphi \ \text{suntil} \ \psi) \ \omega \implies \text{ev} \ \psi \ \omega$

⟨*proof*⟩

lemma *suntil-inv*:

assumes *stl*: $\bigwedge s. f \ (\text{stl} \ s) = \text{stl} \ (f \ s)$

shows $(P \ \text{suntil} \ Q) \ (f \ s) \longleftrightarrow ((\lambda x. P \ (f \ x)) \ \text{suntil} \ (\lambda x. Q \ (f \ x))) \ s$

⟨*proof*⟩

lemma *suntil-smap*: $(P \ \text{suntil} \ Q) \ (\text{smap} \ f \ s) \longleftrightarrow ((\lambda x. P \ (\text{smap} \ f \ x)) \ \text{suntil} \ (\lambda x. Q \ (\text{smap} \ f \ x))) \ s$

⟨*proof*⟩

lemma *hld-smap*: $\text{HLD} \ x \ (\text{smap} \ f \ s) = \text{holds} \ (\lambda y. f \ y \in x) \ s$

⟨*proof*⟩

lemma *suntil-mono*:

assumes *eq*: $\bigwedge \omega. P \ \omega \implies Q1 \ \omega \implies Q2 \ \omega \ \bigwedge \omega. P \ \omega \implies R1 \ \omega \implies R2 \ \omega$

assumes *: $(Q1 \ \text{suntil} \ R1) \ \omega \ \text{alw} \ P \ \omega$ **shows** $(Q2 \ \text{suntil} \ R2) \ \omega$

⟨*proof*⟩

lemma *suntil-cong*:

$\text{alw} \ P \ \omega \implies (\bigwedge \omega. P \ \omega \implies Q1 \ \omega \longleftrightarrow Q2 \ \omega) \implies (\bigwedge \omega. P \ \omega \implies R1 \ \omega \longleftrightarrow R2$

$\omega) \implies$

$(Q1 \ \text{suntil} \ R1) \ \omega \longleftrightarrow (Q2 \ \text{suntil} \ R2) \ \omega$

$\langle \text{proof} \rangle$

lemma *ev-suntil-iff*: $ev (P \text{ until } Q) \omega \longleftrightarrow ev Q \omega$
 $\langle \text{proof} \rangle$

lemma *true-suntil*: $((\lambda -. \text{True}) \text{ until } P) = ev P$
 $\langle \text{proof} \rangle$

lemma *suntil-lfp*: $(\varphi \text{ until } \psi) = lfp (\lambda P s. \psi s \vee (\varphi s \wedge P (stl s)))$
 $\langle \text{proof} \rangle$

lemma *sfilter-P[simp]*: $P (shd s) \implies sfilter P s = shd s \#\# sfilter P (stl s)$
 $\langle \text{proof} \rangle$

lemma *sfilter-not-P[simp]*: $\neg P (shd s) \implies sfilter P s = sfilter P (stl s)$
 $\langle \text{proof} \rangle$

lemma *sfilter-eq*:
assumes $ev (\text{holds } P) s$
shows $sfilter P s = x \#\# s' \longleftrightarrow$
 $P x \wedge (\text{not } (\text{holds } P) \text{ until } (HLD \{x\} \text{ aand } nxt (\lambda s. sfilter P s = s')))$ s
 $\langle \text{proof} \rangle$

lemma *sfilter-streams*:
 $alw (ev (\text{holds } P)) \omega \implies \omega \in \text{streams } A \implies sfilter P \omega \in \text{streams } \{x \in A. P x\}$
 $\langle \text{proof} \rangle$

lemma *alw-sfilter*:
assumes $alw (ev (\text{holds } P)) s$
shows $alw Q (sfilter P s) \longleftrightarrow alw (\lambda x. Q (sfilter P x)) s$
 $\langle \text{proof} \rangle$

lemma *ev-sfilter*:
assumes $alw (ev (\text{holds } P)) s$
shows $ev Q (sfilter P s) \longleftrightarrow ev (\lambda x. Q (sfilter P x)) s$
 $\langle \text{proof} \rangle$

lemma *holds-sfilter*:
assumes $ev (\text{holds } Q) s$ **shows** $\text{holds } P (sfilter Q s) \longleftrightarrow (\text{not } (\text{holds } Q) \text{ until } (\text{holds } (Q \text{ aand } P))) s$
 $\langle \text{proof} \rangle$

lemma *suntil-aand-nxt*:
 $(\varphi \text{ until } (\varphi \text{ aand } nxt \psi)) \omega \longleftrightarrow (\varphi \text{ aand } nxt (\varphi \text{ until } \psi)) \omega$
 $\langle \text{proof} \rangle$

lemma *alw-sconst*: $alw P (sconst x) \longleftrightarrow P (sconst x)$
 $\langle \text{proof} \rangle$

lemma *ev-sconst*: $ev\ P\ (sconst\ x) \longleftrightarrow P\ (sconst\ x)$
 $\langle proof \rangle$

lemma *suntil-sconst*: $(\varphi\ suntil\ \psi)\ (sconst\ x) \longleftrightarrow \psi\ (sconst\ x)$
 $\langle proof \rangle$

lemma *hld-smap'*: $HLD\ x\ (smap\ f\ s) = HLD\ (f\ -'x)\ s$
 $\langle proof \rangle$

lemma *pigeonhole-stream*:
 assumes *alw* $(HLD\ s)\ \omega$
 assumes *finite* *s*
 shows $\exists x \in s. alw\ (ev\ (HLD\ \{x\}))\ \omega$
 $\langle proof \rangle$

lemma *ev-eq-suntil*: $ev\ P\ \omega \longleftrightarrow (not\ P\ suntil\ P)\ \omega$
 $\langle proof \rangle$

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

lemma *suntil-implies-until*: $(\varphi\ suntil\ \psi)\ \omega \implies (\varphi\ until\ \psi)\ \omega$
 $\langle proof \rangle$

lemma *alw-implies-until*: $alw\ \varphi\ \omega \implies (\varphi\ until\ \psi)\ \omega$
 $\langle proof \rangle$

lemma *until-ev-suntil*: $(\varphi\ until\ \psi)\ \omega \implies ev\ \psi\ \omega \implies (\varphi\ suntil\ \psi)\ \omega$
 $\langle proof \rangle$

lemma *suntil-as-until*: $(\varphi\ suntil\ \psi)\ \omega = ((\varphi\ until\ \psi)\ \omega \wedge ev\ \psi\ \omega)$
 $\langle proof \rangle$

lemma *until-not-released-now*: $(\varphi\ until\ \psi)\ \omega \implies \neg\ \psi\ \omega \implies \varphi\ \omega$
 $\langle proof \rangle$

lemma *until-must-release-ev*: $(\varphi\ until\ \psi)\ \omega \implies ev\ (not\ \varphi)\ \omega \implies ev\ \psi\ \omega$
 $\langle proof \rangle$

lemma *until-as-suntil*: $(\varphi\ until\ \psi)\ \omega = ((\varphi\ suntil\ \psi)\ or\ (alw\ \varphi))\ \omega$
 $\langle proof \rangle$

lemma *alw-holds*: $alw\ (holds\ P)\ (h\ \#\ t) = (P\ h \wedge alw\ (holds\ P)\ t)$
 $\langle proof \rangle$

lemma *alw-holds2*: $alw\ (holds\ P)\ ss = (P\ (shd\ ss) \wedge alw\ (holds\ P)\ (stl\ ss))$
 $\langle proof \rangle$

lemma *alw-eq-sconst*: $(\text{alw } (\text{HLD } \{h\}) \ t) = (t = \text{sconst } h)$
 $\langle \text{proof} \rangle$

lemma *sdrop-if-suntil*: $(p \ \text{suntil} \ q) \ \omega \implies \exists j. \ q \ (\text{sdrop } j \ \omega) \wedge (\forall k < j. \ p \ (\text{sdrop } k \ \omega))$
 $\langle \text{proof} \rangle$

lemma *not-suntil*: $(\neg (p \ \text{suntil} \ q) \ \omega) = (\neg (p \ \text{until} \ q) \ \omega \vee \text{alw } (\text{not } q) \ \omega)$
 $\langle \text{proof} \rangle$

lemma *sdrop-until*: $q \ (\text{sdrop } j \ \omega) \implies \forall k < j. \ p \ (\text{sdrop } k \ \omega) \implies (p \ \text{until} \ q) \ \omega$
 $\langle \text{proof} \rangle$

lemma *sdrop-suntil*: $q \ (\text{sdrop } j \ \omega) \implies (\forall k < j. \ p \ (\text{sdrop } k \ \omega)) \implies (p \ \text{suntil} \ q) \ \omega$
 $\langle \text{proof} \rangle$

lemma *suntil-iff-sdrop*: $(p \ \text{suntil} \ q) \ \omega = (\exists j. \ q \ (\text{sdrop } j \ \omega) \wedge (\forall k < j. \ p \ (\text{sdrop } k \ \omega)))$
 $\langle \text{proof} \rangle$

end

57 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* :: $('a::\text{times}) \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list}$ (**infix** $*_s$ 70)
where $x *_s xs \equiv \text{map } ((*) \ x) \ xs$

lemma *scaleI[simp]*: $(1::'a::\text{monoid-mult}) *_s xs = xs$
 $\langle \text{proof} \rangle$

57.1 + and −

fun *zipwith0* :: $('a::\text{zero} \Rightarrow 'b::\text{zero} \Rightarrow 'c) \Rightarrow 'a \ \text{list} \Rightarrow 'b \ \text{list} \Rightarrow 'c \ \text{list}$
where
 $\text{zipwith0 } f \ [] \ [] = [] \mid$
 $\text{zipwith0 } f \ (x\#xs) \ (y\#ys) = f \ x \ y \ \# \ \text{zipwith0 } f \ xs \ ys \mid$
 $\text{zipwith0 } f \ (x\#xs) \ [] = f \ x \ 0 \ \# \ \text{zipwith0 } f \ xs \ [] \mid$
 $\text{zipwith0 } f \ [] \ (y\#ys) = f \ 0 \ y \ \# \ \text{zipwith0 } f \ [] \ ys$

instantiation *list* :: $(\{ \text{zero}, \text{plus} \}) \ \text{plus}$
begin

definition

list-add-def: $(+) = \text{zipwith0 } (+)$

instance $\langle \text{proof} \rangle$

end

instantiation *list* :: $(\{\text{zero}, \text{uminus}\}) \text{ uminus}$
begin

definition

list-uminus-def: $\text{uminus} = \text{map uminus}$

instance $\langle \text{proof} \rangle$

end

instantiation *list* :: $(\{\text{zero}, \text{minus}\}) \text{ minus}$
begin

definition

list-diff-def: $(-) = \text{zipwith0 } (-)$

instance $\langle \text{proof} \rangle$

end

lemma *zipwith0-Nil[simp]*: $\text{zipwith0 } f \ [] \ ys = \text{map } (f \ 0) \ ys$
 $\langle \text{proof} \rangle$

lemma *list-add-Nil[simp]*: $[] + xs = (xs :: 'a :: \text{monoid-add list})$
 $\langle \text{proof} \rangle$

lemma *list-add-Nil2[simp]*: $xs + [] = (xs :: 'a :: \text{monoid-add list})$
 $\langle \text{proof} \rangle$

lemma *list-add-Cons[simp]*: $(x \# xs) + (y \# ys) = (x + y) \# (xs + ys)$
 $\langle \text{proof} \rangle$

lemma *list-diff-Nil[simp]*: $[] - xs = -(xs :: 'a :: \text{group-add list})$
 $\langle \text{proof} \rangle$

lemma *list-diff-Nil2[simp]*: $xs - [] = (xs :: 'a :: \text{group-add list})$
 $\langle \text{proof} \rangle$

lemma *list-diff-Cons-Cons[simp]*: $(x \# xs) - (y \# ys) = (x - y) \# (xs - ys)$
 $\langle \text{proof} \rangle$

lemma *list-uminus-Cons*[simp]: $-(x\#xs) = (-x)\#(-xs)$
 $\langle proof \rangle$

lemma *self-list-diff*:
 $xs - xs = replicate\ (length(xs::'a::group-add\ list))\ 0$
 $\langle proof \rangle$

lemma *list-add-assoc*: **fixes** $xs :: 'a::monoid-add\ list$
shows $(xs+ys)+zs = xs+(ys+zs)$
 $\langle proof \rangle$

57.2 Inner product

definition *iprod* :: $'a::ring\ list \Rightarrow 'a\ list \Rightarrow 'a\ (\langle -, - \rangle)$ **where**
 $\langle xs, ys \rangle = (\sum (x, y) \leftarrow zip\ xs\ ys.\ x*y)$

lemma *iprod-Nil*[simp]: $\langle [], ys \rangle = 0$
 $\langle proof \rangle$

lemma *iprod-Nil2*[simp]: $\langle xs, [] \rangle = 0$
 $\langle proof \rangle$

lemma *iprod-Cons*[simp]: $\langle x\#xs, y\#ys \rangle = x*y + \langle xs, ys \rangle$
 $\langle proof \rangle$

lemma *iprod0-if-coeffs0*: $\forall c \in set\ cs.\ c = 0 \implies \langle cs, xs \rangle = 0$
 $\langle proof \rangle$

lemma *iprod-uminus*[simp]: $\langle -xs, ys \rangle = -\langle xs, ys \rangle$
 $\langle proof \rangle$

lemma *iprod-left-add-distrib*: $\langle xs + ys, zs \rangle = \langle xs, zs \rangle + \langle ys, zs \rangle$
 $\langle proof \rangle$

lemma *iprod-left-diff-distrib*: $\langle xs - ys, zs \rangle = \langle xs, zs \rangle - \langle ys, zs \rangle$
 $\langle proof \rangle$

lemma *iprod-assoc*: $\langle x *_s xs, ys \rangle = x * \langle xs, ys \rangle$
 $\langle proof \rangle$

end

58 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 $\Rightarrow$ 'a::ord $\Rightarrow$ bool (infixl $\langle * \leq \rangle$ 70)
where $S * \leq x = (\forall y \in S. y \leq x)$

definition *setge* :: 'a::ord $\Rightarrow$ 'a set $\Rightarrow$ bool (infixl $\langle \leq * \rangle$ 70)
where $x \leq * S = (\forall y \in S. x \leq y)$

58.1 Rules for the Relations $* \leq$ and $\leq *$

lemma *setleI*: $\forall y \in S. y \leq x \Longrightarrow S * \leq x$
 $\langle proof \rangle$

lemma *setleD*: $S * \leq x \Longrightarrow y \in S \Longrightarrow y \leq x$
 $\langle proof \rangle$

lemma *setgeI*: $\forall y \in S. x \leq y \Longrightarrow x \leq * S$
 $\langle proof \rangle$

lemma *setgeD*: $x \leq * S \Longrightarrow y \in S \Longrightarrow x \leq y$
 $\langle proof \rangle$

definition *leastP* :: ('a $\Rightarrow$ bool) $\Rightarrow$ 'a::ord $\Rightarrow$ bool
where $leastP P x = (P x \wedge x \leq * Collect P)$

definition *isUb* :: 'a set $\Rightarrow$ 'a set $\Rightarrow$ 'a::ord $\Rightarrow$ bool
where $isUb R S x = (S * \leq x \wedge x \in R)$

definition *isLub* :: 'a set $\Rightarrow$ 'a set $\Rightarrow$ 'a::ord $\Rightarrow$ bool
where $isLub R S x = leastP (isUb R S) x$

definition *ubs* :: 'a set $\Rightarrow$ 'a::ord set $\Rightarrow$ 'a set
where $ubs R S = Collect (isUb R S)$

58.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: $leastP P x \Longrightarrow P x$
 $\langle proof \rangle$

lemma *leastPD2*: $leastP P x \Longrightarrow x \leq * Collect P$
 $\langle proof \rangle$

lemma *leastPD3*: $leastP P x \Longrightarrow y \in Collect P \Longrightarrow x \leq y$
 $\langle proof \rangle$

lemma *isLubD1*: $isLub R S x \Longrightarrow S * \leq x$
 $\langle proof \rangle$

lemma *isLubD1a*: $isLub R S x \Longrightarrow x \in R$

$\langle proof \rangle$

lemma *isLub-isUb*: $isLub\ R\ S\ x \implies isUb\ R\ S\ x$
 $\langle proof \rangle$

lemma *isLubD2*: $isLub\ R\ S\ x \implies y \in S \implies y \leq x$
 $\langle proof \rangle$

lemma *isLubD3*: $isLub\ R\ S\ x \implies leastP\ (isUb\ R\ S)\ x$
 $\langle proof \rangle$

lemma *isLubI1*: $leastP(isUb\ R\ S)\ x \implies isLub\ R\ S\ x$
 $\langle proof \rangle$

lemma *isLubI2*: $isUb\ R\ S\ x \implies x \leq_* Collect\ (isUb\ R\ S) \implies isLub\ R\ S\ x$
 $\langle proof \rangle$

lemma *isUbD*: $isUb\ R\ S\ x \implies y \in S \implies y \leq x$
 $\langle proof \rangle$

lemma *isUbD2*: $isUb\ R\ S\ x \implies S \leq_* x$
 $\langle proof \rangle$

lemma *isUbD2a*: $isUb\ R\ S\ x \implies x \in R$
 $\langle proof \rangle$

lemma *isUbI*: $S \leq_* x \implies x \in R \implies isUb\ R\ S\ x$
 $\langle proof \rangle$

lemma *isLub-le-isUb*: $isLub\ R\ S\ x \implies isUb\ R\ S\ y \implies x \leq y$
 $\langle proof \rangle$

lemma *isLub-ubs*: $isLub\ R\ S\ x \implies x \leq_* ub\ R\ S$
 $\langle proof \rangle$

lemma *isLub-unique*: $[| isLub\ R\ S\ x; isLub\ R\ S\ y |] \implies x = (y::'a::linorder)$
 $\langle proof \rangle$

lemma *isUb-UNIV-I*: $(\bigwedge y. y \in S \implies y \leq u) \implies isUb\ UNIV\ S\ u$
 $\langle proof \rangle$

definition *greatestP* :: $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$
where *greatestP* $P\ x = (P\ x \wedge Collect\ P\ \leq_*\ x)$

definition *isLb* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where *isLb* $R\ S\ x = (x \leq_* S \wedge x \in R)$

definition *isGlb* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$

where $isGlb\ R\ S\ x = greatestP\ (isLb\ R\ S)\ x$

definition $lbs :: 'a\ set \Rightarrow 'a::ord\ set \Rightarrow 'a\ set$
where $lbs\ R\ S = Collect\ (isLb\ R\ S)$

58.3 Rules about the Operators $greatestP$, $isLb$ and $isGlb$

lemma $greatestPD1: greatestP\ P\ x \Longrightarrow P\ x$
 $\langle proof \rangle$

lemma $greatestPD2: greatestP\ P\ x \Longrightarrow Collect\ P\ *<= x$
 $\langle proof \rangle$

lemma $greatestPD3: greatestP\ P\ x \Longrightarrow y \in Collect\ P \Longrightarrow x \geq y$
 $\langle proof \rangle$

lemma $isGlbD1: isGlb\ R\ S\ x \Longrightarrow x <=* S$
 $\langle proof \rangle$

lemma $isGlbD1a: isGlb\ R\ S\ x \Longrightarrow x \in R$
 $\langle proof \rangle$

lemma $isGlb-isLb: isGlb\ R\ S\ x \Longrightarrow isLb\ R\ S\ x$
 $\langle proof \rangle$

lemma $isGlbD2: isGlb\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \geq x$
 $\langle proof \rangle$

lemma $isGlbD3: isGlb\ R\ S\ x \Longrightarrow greatestP\ (isLb\ R\ S)\ x$
 $\langle proof \rangle$

lemma $isGlbI1: greatestP\ (isLb\ R\ S)\ x \Longrightarrow isGlb\ R\ S\ x$
 $\langle proof \rangle$

lemma $isGlbI2: isLb\ R\ S\ x \Longrightarrow Collect\ (isLb\ R\ S)\ *<= x \Longrightarrow isGlb\ R\ S\ x$
 $\langle proof \rangle$

lemma $isLbD: isLb\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \geq x$
 $\langle proof \rangle$

lemma $isLbD2: isLb\ R\ S\ x \Longrightarrow x <=* S$
 $\langle proof \rangle$

lemma $isLbD2a: isLb\ R\ S\ x \Longrightarrow x \in R$
 $\langle proof \rangle$

lemma $isLbI: x <=* S \Longrightarrow x \in R \Longrightarrow isLb\ R\ S\ x$
 $\langle proof \rangle$

lemma *isGlb-le-isLb*: $\text{isGlb } R \ S \ x \Longrightarrow \text{isLb } R \ S \ y \Longrightarrow x \geq y$
 ⟨proof⟩

lemma *isGlb-ubs*: $\text{isGlb } R \ S \ x \Longrightarrow \text{lbs } R \ S \ * \leq x$
 ⟨proof⟩

lemma *isGlb-unique*: $[\text{isGlb } R \ S \ x; \text{isGlb } R \ S \ y] \Longrightarrow x = (y::'a::\text{linorder})$
 ⟨proof⟩

lemma *bdd-above-setle*: $\text{bdd-above } A \longleftrightarrow (\exists a. A \leq a)$
 ⟨proof⟩

lemma *bdd-below-setge*: $\text{bdd-below } A \longleftrightarrow (\exists a. a \leq A)$
 ⟨proof⟩

lemma *isLub-cSup*:
 $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \Longrightarrow (\exists b. S \leq b) \Longrightarrow \text{isLub}$
 $\text{UNIV } S \ (\text{Sup } S)$
 ⟨proof⟩

lemma *isGlb-cInf*:
 $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \Longrightarrow (\exists b. b \leq S) \Longrightarrow \text{isGlb}$
 $\text{UNIV } S \ (\text{Inf } S)$
 ⟨proof⟩

lemma *cSup-le*: $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \Longrightarrow S \leq b \Longrightarrow$
 $\text{Sup } S \leq b$
 ⟨proof⟩

lemma *cInf-ge*: $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \Longrightarrow b \leq S \Longrightarrow$
 $\text{Inf } S \geq b$
 ⟨proof⟩

lemma *cSup-bounds*:
fixes $S :: 'a :: \text{conditionally-complete-lattice set}$
shows $S \neq \{\} \Longrightarrow a \leq S \Longrightarrow S \leq b \Longrightarrow a \leq \text{Sup } S \wedge \text{Sup } S \leq b$
 ⟨proof⟩

lemma *cSup-unique*: $(S::'a :: \{\text{conditionally-complete-linorder, no-bot}\} \text{ set}) \leq b \Longrightarrow$
 $(\forall b' < b. \exists x \in S. b' < x) \Longrightarrow \text{Sup } S = b$
 ⟨proof⟩

lemma *cInf-unique*: $b \leq (S::'a :: \{\text{conditionally-complete-linorder, no-top}\} \text{ set}) \Longrightarrow$
 $(\forall b' > b. \exists x \in S. b' > x) \Longrightarrow \text{Inf } S = b$
 ⟨proof⟩

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: $\exists X. X \in S \Longrightarrow \exists Y. \text{isUb } (\text{UNIV}::\text{real set}) \ S \ Y \Longrightarrow \exists t.$

isLub (*UNIV* :: *real set*) *S t*
 ⟨*proof*⟩

lemma *Bseq-isUb*: $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \Longrightarrow \exists U. \text{isUb } (UNIV :: \text{real set}) \{x. \exists n. X\ n = x\} U$
 ⟨*proof*⟩

lemma *Bseq-isLub*: $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \Longrightarrow \exists U. \text{isLub } (UNIV :: \text{real set}) \{x. \exists n. X\ n = x\} U$
 ⟨*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*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists b. b \leq^* s \Longrightarrow \text{Inf } s \leq \text{Inf } (t :: \text{real set})$
 ⟨*proof*⟩

lemma *real-ge-sup-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists b. s \leq^* b \Longrightarrow \text{Sup } s \geq \text{Sup } (t :: \text{real set})$
 ⟨*proof*⟩

end

59 An abstract view on maps for code generation.

theory *Mapping*
imports *Main AList*
begin

59.1 Parametricity transfer rules

lemma *map-of-foldr*: $\text{map-of } xs = \text{foldr } (\lambda(k, v) m. m(k \mapsto v))\ xs\ \text{Map.empty}$
 ⟨*proof*⟩

context **includes** *lifting-syntax*
begin

lemma *empty-parametric*: $(A \Longrightarrow B) \Longrightarrow \text{Map.empty } A \Longrightarrow \text{Map.empty } B$
 ⟨*proof*⟩

lemma *lookup-parametric*: $((A \Longrightarrow B) \Longrightarrow A \Longrightarrow B) (\lambda m\ k. m\ k) (\lambda m\ k. m\ k)$

$\langle \text{proof} \rangle$

lemma *update-parametric*:

assumes $[transfer\text{-}rule]: bi\text{-}unique\ A$

shows $(A ==> B ==> (A ==> rel\text{-}option\ B) ==> A ==> rel\text{-}option\ B)$

$(\lambda k\ v\ m. m(k \mapsto v)) (\lambda k\ v\ m. m(k \mapsto v))$

$\langle \text{proof} \rangle$

lemma *delete-parametric*:

assumes $[transfer\text{-}rule]: bi\text{-}unique\ A$

shows $(A ==> (A ==> rel\text{-}option\ B) ==> A ==> rel\text{-}option\ B)$

$(\lambda k\ m. m(k := None)) (\lambda k\ m. m(k := None))$

$\langle \text{proof} \rangle$

lemma *is-none-parametric* $[transfer\text{-}rule]:$

$(rel\text{-}option\ A ==> HOL.eq)\ Option.is\ none\ Option.is\ none$

$\langle \text{proof} \rangle$

lemma *dom-parametric*:

assumes $[transfer\text{-}rule]: bi\text{-}total\ A$

shows $((A ==> rel\text{-}option\ B) ==> rel\text{-}set\ A)\ dom\ dom$

$\langle \text{proof} \rangle$

lemma *graph-parametric*:

assumes $bi\text{-}total\ A$

shows $((A ==> rel\text{-}option\ B) ==> rel\text{-}set\ (rel\text{-}prod\ A\ B))\ Map.graph\ Map.graph$

$\langle \text{proof} \rangle$

lemma *map-of-parametric* $[transfer\text{-}rule]:$

assumes $[transfer\text{-}rule]: bi\text{-}unique\ R1$

shows $(list\text{-}all2\ (rel\text{-}prod\ R1\ R2) ==> R1 ==> rel\text{-}option\ R2)\ map\text{-}of$

$\langle \text{proof} \rangle$

lemma *map-entry-parametric* $[transfer\text{-}rule]:$

assumes $[transfer\text{-}rule]: bi\text{-}unique\ A$

shows $(A ==> (B ==> B) ==> (A ==> rel\text{-}option\ B) ==> A ==> rel\text{-}option\ B)$

$(\lambda k\ f\ m. (case\ m\ k\ of\ None \Rightarrow m$
 $\quad | Some\ v \Rightarrow m\ (k \mapsto (f\ v)))) (\lambda k\ f\ m. (case\ m\ k\ of\ None \Rightarrow m$
 $\quad | Some\ v \Rightarrow m\ (k \mapsto (f\ v))))$

$\langle \text{proof} \rangle$

lemma *tabulate-parametric*:

assumes $[transfer\text{-}rule]: bi\text{-}unique\ A$

shows $(list\text{-}all2\ A ==> (A ==> B) ==> A ==> rel\text{-}option\ B)$

$(\lambda ks\ f. (map\text{-}of\ (map\ (\lambda k. (k, f\ k))\ ks))) (\lambda ks\ f. (map\text{-}of\ (map\ (\lambda k. (k, f\ k))\ ks)))$

⟨proof⟩

lemma *bulkload-parametric*:

(*list-all2* $A \implies HOL.eq \implies rel-option\ A$)
 ($\lambda xs\ k. \text{if } k < \text{length } xs \text{ then } Some\ (xs\ !\ k) \text{ else } None$)
 ($\lambda xs\ k. \text{if } k < \text{length } xs \text{ then } Some\ (xs\ !\ k) \text{ else } None$)
 ⟨proof⟩

lemma *map-parametric*:

(($A \implies B \implies (C \implies D) \implies (B \implies rel-option\ C) \implies A \implies rel-option\ D$)
 ($\lambda f\ g\ m. (map-option\ g \circ m \circ f)$) ($\lambda f\ g\ m. (map-option\ g \circ m \circ f)$)
 ⟨proof⟩

lemma *combine-with-key-parametric*:

(($A \implies B \implies B \implies B$) $\implies (A \implies rel-option\ B) \implies (A \implies rel-option\ B) \implies$
 ($A \implies rel-option\ B$) ($\lambda f\ m1\ m2\ x. combine-options\ (f\ x)\ (m1\ x)\ (m2\ x)$)
 ($\lambda f\ m1\ m2\ x. combine-options\ (f\ x)\ (m1\ x)\ (m2\ x)$)
 ⟨proof⟩

lemma *combine-parametric*:

(($B \implies B \implies B$) $\implies (A \implies rel-option\ B) \implies (A \implies rel-option\ B) \implies$
 ($A \implies rel-option\ B$) ($\lambda f\ m1\ m2\ x. combine-options\ f\ (m1\ x)\ (m2\ x)$)
 ($\lambda f\ m1\ m2\ x. combine-options\ f\ (m1\ x)\ (m2\ x)$)
 ⟨proof⟩

end

59.2 Type definition and primitive operations

typedef ($'a, 'b$) *mapping* = *UNIV* :: ($'a \rightarrow 'b$) *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* $\Rightarrow 'a \Rightarrow 'b\ option$

is $\lambda m\ k. m\ k$ **parametric** *lookup-parametric* ⟨proof⟩

definition *lookup-default* $d\ m\ k = (\text{case } Mapping.lookup\ m\ k \text{ of } None \Rightarrow d \mid Some\ v \Rightarrow v)$

lift-definition *update* :: ($'a \Rightarrow 'b \Rightarrow ('a, 'b)\ mapping \Rightarrow ('a, 'b)\ mapping$

is $\lambda k\ v\ m. m(k \mapsto v)$ **parametric** *update-parametric* ⟨proof⟩

lift-definition *delete* :: ('a $\Rightarrow$ ('a, 'b) mapping $\Rightarrow$ ('a, 'b) mapping
 is $\lambda k\ m.$ *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** *bulkload-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-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]]

59.3 Functorial structure

functor *map*: *map*
 <proof>

59.4 Derived operations

definition *ordered-keys* :: ('a::linorder, 'b) mapping $\Rightarrow$ 'a list

where *ordered-keys* $m = (\text{if finite } (\text{keys } m) \text{ then sorted-list-of-set } (\text{keys } m) \text{ else } [])$

definition *ordered-entries* $:: ('a::\text{linorder}, 'b) \text{ mapping} \Rightarrow ('a \times 'b) \text{ list}$
where *ordered-entries* $m = (\text{if finite } (\text{entries } m) \text{ then sorted-key-list-of-set fst } (\text{entries } m) \text{ else } [])$

definition *fold* $:: ('a::\text{linorder} \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow 'c \Rightarrow 'c$
where *fold* $f \ m \ a = \text{List.fold } (\text{case-prod } f) (\text{ordered-entries } m) \ a$

definition *is-empty* $:: ('a, 'b) \text{ mapping} \Rightarrow \text{bool}$
where *is-empty* $m \longleftrightarrow \text{keys } m = \{\}$

definition *size* $:: ('a, 'b) \text{ mapping} \Rightarrow \text{nat}$
where *size* $m = (\text{if finite } (\text{keys } m) \text{ then card } (\text{keys } m) \text{ else } 0)$

definition *replace* $:: 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow ('a, 'b) \text{ mapping}$
where *replace* $k \ v \ m = (\text{if } k \in \text{keys } m \text{ then update } k \ v \ m \text{ else } m)$

definition *default* $:: 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow ('a, 'b) \text{ mapping}$
where *default* $k \ v \ m = (\text{if } k \in \text{keys } m \text{ then } m \text{ else update } k \ v \ m)$

Manual derivation of transfer rule is non-trivial

lift-definition *map-entry* $:: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow ('a, 'b) \text{ mapping}$
is

$\lambda k \ f \ m.$
 $(\text{case } m \ k \ \text{of}$
 $\quad \text{None} \Rightarrow m$
 $\quad | \text{Some } v \Rightarrow m \ (k \mapsto (f \ v)))$ **parametric** *map-entry-parametric* $\langle \text{proof} \rangle$

lemma *map-entry-code* [code]:

map-entry $k \ f \ m =$
 $(\text{case lookup } m \ k \ \text{of}$
 $\quad \text{None} \Rightarrow m$
 $\quad | \text{Some } v \Rightarrow \text{update } k \ (f \ v) \ m)$
 $\langle \text{proof} \rangle$

definition *map-default* $:: 'a \Rightarrow 'b \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow ('a, 'b) \text{ mapping}$
where *map-default* $k \ v \ f \ m = \text{map-entry } k \ f \ (\text{default } k \ v \ m)$

definition *of-alist* $:: ('k \times 'v) \text{ list} \Rightarrow ('k, 'v) \text{ mapping}$
where *of-alist* $xs = \text{foldr } (\lambda(k, v) \ m. \text{update } k \ v \ m) \ xs \ \text{empty}$

instantiation *mapping* $:: (\text{type}, \text{type}) \text{ equal}$
begin

definition *HOL.equal* $m1 \ m2 \longleftrightarrow (\forall k. \text{lookup } m1 \ k = \text{lookup } m2 \ k)$

```

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

```

59.5 Properties

```

lemma mapping-eqI: ( $\bigwedge x. \text{lookup } m \ x = \text{lookup } m' \ x$ )  $\implies m = m'$ 
  ⟨proof⟩

lemma mapping-eqI':
  assumes  $\bigwedge x. x \in \text{Mapping.keys } m \implies \text{Mapping.lookup-default } d \ m \ x = \text{Mapping.lookup-default } d \ m' \ x$ 
  and  $\text{Mapping.keys } m = \text{Mapping.keys } m'$ 
  shows  $m = m'$ 
  ⟨proof⟩

lemma lookup-update[simp]:  $\text{lookup } (\text{update } k \ v \ m) \ k = \text{Some } v$ 
  ⟨proof⟩

lemma lookup-update-neq[simp]:  $k \neq k' \implies \text{lookup } (\text{update } k \ v \ m) \ k' = \text{lookup } m \ k'$ 
  ⟨proof⟩

lemma lookup-update':  $\text{lookup } (\text{update } k \ v \ m) \ k' = (\text{if } k = k' \text{ then } \text{Some } v \text{ else } \text{lookup } m \ k')$ 
  ⟨proof⟩

lemma lookup-empty[simp]:  $\text{lookup } \text{empty } k = \text{None}$ 
  ⟨proof⟩

lemma lookup-delete[simp]:  $\text{lookup } (\text{delete } k \ m) \ k = \text{None}$ 

```

<proof>

lemma *lookup-delete-neq[simp]:* $k \neq k' \implies \text{lookup } (\text{delete } k \ m) \ k' = \text{lookup } m \ k'$
<proof>

lemma *lookup-filter:*
 $\text{lookup } (\text{filter } P \ m) \ k =$
 $(\text{case } \text{lookup } m \ k \ \text{of}$
 $\quad \text{None} \Rightarrow \text{None}$
 $\quad | \text{Some } v \Rightarrow \text{if } P \ k \ v \ \text{then } \text{Some } v \ \text{else } \text{None})$
<proof>

lemma *lookup-map-values:* $\text{lookup } (\text{map-values } f \ m) \ k = \text{map-option } (f \ k) \ (\text{lookup } m \ k)$
<proof>

lemma *lookup-default-empty:* $\text{lookup-default } d \ \text{empty} \ k = d$
<proof>

lemma *lookup-default-update:* $\text{lookup-default } d \ (\text{update } k \ v \ m) \ k = v$
<proof>

lemma *lookup-default-update-neq:*
 $k \neq k' \implies \text{lookup-default } d \ (\text{update } k \ v \ m) \ k' = \text{lookup-default } d \ m \ k'$
<proof>

lemma *lookup-default-update':*
 $\text{lookup-default } d \ (\text{update } k \ v \ m) \ k' = (\text{if } k = k' \ \text{then } v \ \text{else } \text{lookup-default } d \ m \ k')$
<proof>

lemma *lookup-default-filter:*
 $\text{lookup-default } d \ (\text{filter } P \ m) \ k =$
 $(\text{if } P \ k \ (\text{lookup-default } d \ m \ k) \ \text{then } \text{lookup-default } d \ m \ k \ \text{else } d)$
<proof>

lemma *lookup-default-map-values:*
 $\text{lookup-default } (f \ k \ d) \ (\text{map-values } f \ m) \ k = f \ k \ (\text{lookup-default } d \ m \ k)$
<proof>

lemma *lookup-combine-with-key:*
 $\text{Mapping.lookup } (\text{combine-with-key } f \ m1 \ m2) \ x =$
 $\text{combine-options } (f \ x) \ (\text{Mapping.lookup } m1 \ x) \ (\text{Mapping.lookup } m2 \ x)$
<proof>

lemma *combine-altdef:* $\text{combine } f \ m1 \ m2 = \text{combine-with-key } (\lambda \cdot. f) \ m1 \ m2$
<proof>

lemma *lookup-combine:*
 $\text{Mapping.lookup } (\text{combine } f \ m1 \ m2) \ x =$

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*: $x \neq y \implies \text{lookup } (\text{map-entry } x\ f\ m)\ y = \text{lookup } m\ y$
 ⟨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*: $x \neq y \implies \text{lookup } (\text{default } x\ d\ m)\ y = \text{lookup } m\ y$
 ⟨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*: $x \neq y \implies \text{lookup } (\text{map-default } x\ d\ f\ m)\ y = \text{lookup } m\ y$
 ⟨proof⟩

lemma *lookup-map-default'*:

lookup (*map-default* $x\ d\ f\ m$) y =

(if $x = y$ then $\text{Some } (f \text{ (lookup-default } d \text{ } m \text{ } x))$ else $\text{lookup } m \text{ } y$)
 ⟨proof⟩

lemma *lookup-tabulate*:

assumes *distinct xs*

shows $\text{Mapping.lookup } (\text{Mapping.tabulate } xs \text{ } f) \text{ } x = (\text{if } x \in \text{set } xs \text{ then } \text{Some } (f \text{ } x) \text{ else } \text{None})$

⟨proof⟩

lemma *lookup-of-alist*: $\text{lookup } (\text{of-alist } xs) \text{ } k = \text{map-of } xs \text{ } k$

⟨proof⟩

lemma *keys-is-none-rep* [code-unfold]: $k \in \text{keys } m \longleftrightarrow \neg (\text{Option.is-none } (\text{lookup } m \text{ } k))$

⟨proof⟩

lemma *update-update*:

$\text{update } k \text{ } v \text{ } (\text{update } k \text{ } w \text{ } m) = \text{update } k \text{ } v \text{ } m$

$k \neq l \implies \text{update } k \text{ } v \text{ } (\text{update } l \text{ } w \text{ } m) = \text{update } l \text{ } w \text{ } (\text{update } k \text{ } v \text{ } m)$

⟨proof⟩

lemma *update-delete* [simp]: $\text{update } k \text{ } v \text{ } (\text{delete } k \text{ } m) = \text{update } k \text{ } v \text{ } m$

⟨proof⟩

lemma *delete-update*:

$\text{delete } k \text{ } (\text{update } k \text{ } v \text{ } m) = \text{delete } k \text{ } m$

$k \neq l \implies \text{delete } k \text{ } (\text{update } l \text{ } v \text{ } m) = \text{update } l \text{ } v \text{ } (\text{delete } k \text{ } m)$

⟨proof⟩

lemma *delete-empty* [simp]: $\text{delete } k \text{ } \text{empty} = \text{empty}$

⟨proof⟩

lemma *Mapping-delete-if-notin-keys*[simp]:

$k \notin \text{keys } m \implies \text{delete } k \text{ } m = m$

⟨proof⟩

lemma *replace-update*:

$k \notin \text{keys } m \implies \text{replace } k \text{ } v \text{ } m = m$

$k \in \text{keys } m \implies \text{replace } k \text{ } v \text{ } m = \text{update } k \text{ } v \text{ } m$

⟨proof⟩

lemma *map-values-update*: $\text{map-values } f \text{ } (\text{update } k \text{ } v \text{ } m) = \text{update } k \text{ } (f \text{ } v) \text{ } (\text{map-values } f \text{ } m)$

⟨proof⟩

lemma *size-mono*: $\text{finite } (\text{keys } m') \implies \text{keys } m \subseteq \text{keys } m' \implies \text{size } m \leq \text{size } m'$

⟨proof⟩

lemma *size-empty* [simp]: $\text{size } \text{empty} = 0$

$\langle \text{proof} \rangle$

lemma *size-update*:

$\text{finite } (\text{keys } m) \implies \text{size } (\text{update } k \ v \ m) =$
 $(\text{if } k \in \text{keys } m \text{ then } \text{size } m \text{ else } \text{Suc } (\text{size } m))$
 $\langle \text{proof} \rangle$

lemma *size-delete*: $\text{size } (\text{delete } k \ m) = (\text{if } k \in \text{keys } m \text{ then } \text{size } m - 1 \text{ else } \text{size } m)$
 $\langle \text{proof} \rangle$

lemma *size-tabulate* [simp]: $\text{size } (\text{tabulate } ks \ f) = \text{length } (\text{remdups } ks)$
 $\langle \text{proof} \rangle$

lemma *keys-filter*: $\text{keys } (\text{filter } P \ m) \subseteq \text{keys } m$
 $\langle \text{proof} \rangle$

lemma *size-filter*: $\text{finite } (\text{keys } m) \implies \text{size } (\text{filter } P \ m) \leq \text{size } m$
 $\langle \text{proof} \rangle$

lemma *bulkload-tabulate*: $\text{bulkload } xs = \text{tabulate } [0..<\text{length } xs] \ (\text{nth } xs)$
 $\langle \text{proof} \rangle$

lemma *is-empty-empty* [simp]: $\text{is-empty } \text{empty}$
 $\langle \text{proof} \rangle$

lemma *is-empty-update* [simp]: $\neg \text{is-empty } (\text{update } k \ v \ m)$
 $\langle \text{proof} \rangle$

lemma *is-empty-delete*: $\text{is-empty } (\text{delete } k \ m) \longleftrightarrow \text{is-empty } m \vee \text{keys } m = \{k\}$
 $\langle \text{proof} \rangle$

lemma *is-empty-replace* [simp]: $\text{is-empty } (\text{replace } k \ v \ m) \longleftrightarrow \text{is-empty } m$
 $\langle \text{proof} \rangle$

lemma *is-empty-default* [simp]: $\neg \text{is-empty } (\text{default } k \ v \ m)$
 $\langle \text{proof} \rangle$

lemma *is-empty-map-entry* [simp]: $\text{is-empty } (\text{map-entry } k \ f \ m) \longleftrightarrow \text{is-empty } m$
 $\langle \text{proof} \rangle$

lemma *is-empty-map-values* [simp]: $\text{is-empty } (\text{map-values } f \ m) \longleftrightarrow \text{is-empty } m$
 $\langle \text{proof} \rangle$

lemma *is-empty-map-default* [simp]: $\neg \text{is-empty } (\text{map-default } k \ v \ f \ m)$
 $\langle \text{proof} \rangle$

lemma *keys-dom-lookup*: $\text{keys } m = \text{dom } (\text{Mapping.lookup } m)$
 $\langle \text{proof} \rangle$

lemma *keys-empty* [simp]: *keys empty* = {}
 ⟨proof⟩

lemma *in-keysD*: $k \in \text{keys } m \implies \exists v. \text{lookup } m \ k = \text{Some } v$
 ⟨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-entry k f m)* = *keys m*
 ⟨proof⟩

lemma *keys-map-default* [simp]: *keys (map-default k v f m)* = *insert k (keys m)*
 ⟨proof⟩

lemma *keys-map-values* [simp]: *keys (map-values f m)* = *keys m*
 ⟨proof⟩

lemma *keys-combine-with-key* [simp]:
Mapping.keys (combine-with-key f m1 m2) = *Mapping.keys m1* ∪ *Mapping.keys m2*
 ⟨proof⟩

lemma *keys-combine* [simp]: *Mapping.keys (combine f m1 m2)* = *Mapping.keys m1* ∪ *Mapping.keys m2*
 ⟨proof⟩

lemma *keys-tabulate* [simp]: *keys (tabulate ks f)* = *set ks*
 ⟨proof⟩

lemma *keys-of-alist* [simp]: *keys (of-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]:

$\text{finite } (\text{Mapping.keys } m) \implies \text{set } (\text{Mapping.ordered-keys } m) = \text{Mapping.keys } m$
 ⟨proof⟩

lemma *distinct-ordered-keys* [simp]: $\text{distinct } (\text{ordered-keys } m)$

⟨proof⟩

lemma *ordered-keys-infinite* [simp]: $\neg \text{finite } (\text{keys } m) \implies \text{ordered-keys } m = []$

⟨proof⟩

lemma *ordered-keys-empty* [simp]: $\text{ordered-keys empty} = []$

⟨proof⟩

lemma *sorted-ordered-keys* [simp]: $\text{sorted } (\text{ordered-keys } m)$

⟨proof⟩

lemma *ordered-keys-update* [simp]:

$k \in \text{keys } m \implies \text{ordered-keys } (\text{update } k \ v \ m) = \text{ordered-keys } m$

$\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies$

$\text{ordered-keys } (\text{update } k \ v \ m) = \text{insort } k \ (\text{ordered-keys } m)$

⟨proof⟩

lemma *ordered-keys-delete* [simp]: $\text{ordered-keys } (\text{delete } k \ m) = \text{remove1 } k \ (\text{ordered-keys } m)$

⟨proof⟩

lemma *ordered-keys-replace* [simp]: $\text{ordered-keys } (\text{replace } k \ v \ m) = \text{ordered-keys } m$

⟨proof⟩

lemma *ordered-keys-default* [simp]:

$k \in \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{ordered-keys } m$

$\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{insort } k \ (\text{ordered-keys } m)$

⟨proof⟩

lemma *ordered-keys-map-entry* [simp]: $\text{ordered-keys } (\text{map-entry } k \ f \ m) = \text{ordered-keys } m$

⟨proof⟩

lemma *ordered-keys-map-default* [simp]:

$k \in \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{ordered-keys } m$

$\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{insort } k \ (\text{ordered-keys } m)$

⟨proof⟩

lemma *ordered-keys-tabulate* [simp]: $\text{ordered-keys } (\text{tabulate } ks \ f) = \text{sort } (\text{remdups } ks)$

⟨proof⟩

lemma *ordered-keys-bulkload* [simp]: *ordered-keys* (*bulkload* *ks*) = [0..*length* *ks*]
 ⟨*proof*⟩

lemma *tabulate-fold*: *tabulate* *xs* *f* = *List.fold* ($\lambda k\ m.\ \text{update } k\ (f\ k)\ m$) *xs* *empty*
 ⟨*proof*⟩

lemma *All-mapping-mono*:
 $(\bigwedge k\ v.\ k \in \text{keys } m \implies P\ k\ v \implies Q\ k\ v) \implies \text{All-mapping } m\ P \implies \text{All-mapping } m\ Q$
 ⟨*proof*⟩

lemma *All-mapping-empty* [simp]: *All-mapping* *Mapping.empty* *P*
 ⟨*proof*⟩

lemma *All-mapping-update-iff*:
 $\text{All-mapping } (\text{Mapping.update } k\ v\ m)\ P \longleftrightarrow P\ k\ v \wedge \text{All-mapping } m\ (\lambda k'\ v'.\ k = k' \vee P\ k'\ v')$
 ⟨*proof*⟩

lemma *All-mapping-update*:
 $P\ k\ v \implies \text{All-mapping } m\ (\lambda k'\ v'.\ k = k' \vee P\ k'\ v') \implies \text{All-mapping } (\text{Mapping.update } k\ v\ m)\ P$
 ⟨*proof*⟩

lemma *All-mapping-filter-iff*: *All-mapping* (*filter* *P* *m*) *Q* $\longleftrightarrow \text{All-mapping } m\ (\lambda k\ v.\ P\ k\ v \longrightarrow Q\ k\ v)$
 ⟨*proof*⟩

lemma *All-mapping-filter*: *All-mapping* *m* *Q* $\implies \text{All-mapping } (\text{filter } P\ m)\ Q$
 ⟨*proof*⟩

lemma *All-mapping-map-values*: *All-mapping* (*map-values* *f* *m*) *P* $\longleftrightarrow \text{All-mapping } m\ (\lambda k\ v.\ P\ k\ (f\ k\ v))$
 ⟨*proof*⟩

lemma *All-mapping-tabulate*: $(\forall x \in \text{set } xs.\ P\ x\ (f\ x)) \implies \text{All-mapping } (\text{Mapping.tabulate } xs\ f)\ P$
 ⟨*proof*⟩

lemma *All-mapping-alist*:
 $(\bigwedge k\ v.\ (k,\ v) \in \text{set } xs \implies P\ k\ v) \implies \text{All-mapping } (\text{Mapping.of-alist } xs)\ P$
 ⟨*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*:
 $\text{combine.F } g \text{ (set } xs) = \text{foldr } (\lambda x. \text{ combine } f \text{ (} g \text{ } x)) \text{ (remdups } xs) \text{ Mapping.empty}$
<proof>

lemma *keys-fold-combine*: $\text{finite } A \implies \text{Mapping.keys (combine.F } g \text{ } A) = (\bigcup_{x \in A. \text{ Mapping.keys (} g \text{ } x)})$
<proof>

end

59.5.1 *entries, ordered-entries, and fold*

context *linorder*
begin

sublocale *folding-Map-graph: folding-insort-key ($\leq$) ($<$) Map.graph m fst for m*
<proof>

end

lemma *sorted-fst-list-of-set-insort-Map-graph[simp]*:
assumes $\text{finite (dom } m) \text{ fst } x \notin \text{dom } m$
shows $\text{sorted-key-list-of-set fst (insert } x \text{ (Map.graph } m))$
 $= \text{insort-key fst } x \text{ (sorted-key-list-of-set fst (Map.graph } m))$
<proof>

lemma *sorted-fst-list-of-set-insort-insert-Map-graph[simp]*:
assumes $\text{finite (dom } m) \text{ fst } x \notin \text{dom } m$
shows $\text{sorted-key-list-of-set fst (insert } x \text{ (Map.graph } m))$
 $= \text{insort-insert-key fst } x \text{ (sorted-key-list-of-set fst (Map.graph } m))$
<proof>

lemma *linorder-finite-Map-induct[consumes 1, case-names empty update]*:
fixes $m :: 'a::\text{linorder} \rightarrow 'b$
assumes $\text{finite (dom } m)$
assumes $P \text{ Map.empty}$
assumes $\bigwedge k \ v \ m. \llbracket \text{finite (dom } m); k \notin \text{dom } m; (\bigwedge k'. k' \in \text{dom } m \implies k' \leq k);$
 $P \ m \rrbracket$
 $\implies P \ (m(k \mapsto v))$
shows $P \ m$
<proof>

lemma *delete-insort-fst*[simp]: $AList.delete\ k\ (insort\text{-}key\ fst\ (k,\ v)\ xs) = AList.delete\ k\ xs$
 ⟨proof⟩

lemma *insort-fst-delete*: $\llbracket fst\ x \neq k2;\ sorted\ (List.map\ fst\ xs) \rrbracket$
 $\implies insort\text{-}key\ fst\ x\ (AList.delete\ k2\ xs) = AList.delete\ k2\ (insort\text{-}key\ fst\ x\ xs)$
 ⟨proof⟩

lemma *sorted-fst-list-of-set-Map-graph-fun-upd-None*[simp]:
 $sorted\text{-}key\text{-}list\text{-}of\text{-}set\ fst\ (Map.graph\ (m(k := None)))$
 $= AList.delete\ k\ (sorted\text{-}key\text{-}list\text{-}of\text{-}set\ fst\ (Map.graph\ m))$
 ⟨proof⟩

lemma *entries-empty*[simp]: $entries\ empty = \{\}$
 ⟨proof⟩

lemma *entries-lookup*: $entries\ m = Map.graph\ (lookup\ m)$
 ⟨proof⟩

lemma *in-entriesI*: $lookup\ m\ k = Some\ v \implies (k,\ v) \in entries\ m$
 ⟨proof⟩

lemma *in-entriesD*: $(k,\ v) \in entries\ m \implies lookup\ m\ k = Some\ v$
 ⟨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*:
 $entries\ (delete\ k\ m) = \{e \in entries\ m.\ fst\ e \neq k\}$
 ⟨proof⟩

lemma *entries-of-alist*[simp]:
 $distinct\ (List.map\ fst\ xs) \implies entries\ (of\text{-}alist\ xs) = set\ xs$
 ⟨proof⟩

lemma *entries-keysD*:
 $x \in entries\ m \implies fst\ x \in keys\ m$
 ⟨proof⟩

lemma *set-ordered-entries*[simp]:

finite (keys *m*) $\implies$ *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]:

$\neg$ *finite* (Mapping.keys *m*) $\implies$ 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* $\notin$ *f* ‘*set* *xs* $\forall x \in$ *set* *xs*. *f x* $\leq$ *f y*

shows *insort-key* *f y xs* = *xs* @ [*y*]

⟨proof⟩

lemma *fold-update*:

assumes *finite* (keys *m*)

assumes *k* $\notin$ keys *m* $\wedge$ *k'*. *k'* $\in$ keys *m* $\implies$ *k'* $\leq$ *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  $\bigwedge k\ v\ m.$ 
   $\llbracket \text{finite } (\text{keys } m); k \notin \text{keys } m; (\bigwedge k'. k' \in \text{keys } m \implies k' \leq k); P\ m \rrbracket$ 
   $\implies P\ (\text{update } k\ v\ m)$ 
shows  $P\ m$ 
 $\langle \text{proof} \rangle$ 

```

59.6 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

```

60 Monad notation for arbitrary types

```

theory Monad-Syntax
  imports Adhoc-Overloading
begin

```

We provide a convenient *do*-notation for monadic expressions well-known from Haskell. *const>Let* is printed specially in *do*-expressions.

```

consts
   $\text{bind} :: 'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd$  (infixl  $\gg=$  54)

```

```

notation (ASCII)
   $\text{bind}$  (infixl  $\gg=$  54)

```

```

abbreviation (do-notation)
   $\text{bind-do} :: 'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd$ 
  where  $\text{bind-do} \equiv \text{bind}$ 

```

```

notation (output)
   $\text{bind-do}$  (infixl  $\gg=$  54)

```

```

notation (ASCII output)
   $\text{bind-do}$  (infixl  $\gg=$  54)

```

```

nonterminal do-binds and do-bind
syntax

```

```

 $\text{-do-block} :: \text{do-binds} \Rightarrow 'a$  ( $\text{do } \{ // (2\ -) // \}$  [12] 62)
 $\text{-do-bind} :: [\text{pttrn}, 'a] \Rightarrow \text{do-bind } ((2\ \leftarrow / -) 13)$ 
 $\text{-do-let} :: [\text{pttrn}, 'a] \Rightarrow \text{do-bind } ((2\ \text{let } - = / -) [1000, 13] 13)$ 
 $\text{-do-then} :: 'a \Rightarrow \text{do-bind } (- [14] 13)$ 

```



```

-do-final :: 'a ⇒ do-binds (-)
-do-cons :: [do-bind, do-binds] ⇒ do-binds (-;/- [13, 12] 12)
-thenM :: ['a, 'b] ⇒ 'c (infixl >> 54)

```

syntax (*ASCII*)

```

-do-bind :: [pttrn, 'a] ⇒ do-bind ((2- <-/- -) 13)
-thenM :: ['a, 'b] ⇒ 'c (infixl >> 54)

```

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 (λp. 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

61 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]:

$$P\ x \implies \text{strip-while}\ P\ (xs\ @\ [x]) = \text{strip-while}\ P\ xs$$

<proof>

lemma *strip-while-Cons* [simp]:

$$\neg P\ x \implies \text{strip-while}\ P\ (x\ \# xs) = x\ \# \text{strip-while}\ P\ xs$$

<proof>

lemma *strip-while-eq-Nil* [simp]:

$$\text{strip-while}\ P\ xs = [] \iff (\forall x \in \text{set}\ xs. P\ x)$$

<proof>

lemma *strip-while-eq-Cons-rec*:

$$\text{strip-while}\ P\ (x\ \# xs) = x\ \# \text{strip-while}\ P\ xs \iff \neg (P\ x \wedge (\forall x \in \text{set}\ xs. P\ x))$$

<proof>

lemma *split-strip-while-append*:

fixes $xs :: 'a\ \text{list}$

obtains $ys\ zs :: 'a\ \text{list}$

where $\text{strip-while}\ P\ xs = ys$ **and** $\forall x \in \text{set}\ zs. P\ x$ **and** $xs = ys\ @\ zs$

<proof>

lemma *strip-while-snoc* [simp]:

$$\text{strip-while}\ P\ (xs\ @\ [x]) = (\text{if}\ P\ x\ \text{then}\ \text{strip-while}\ P\ xs\ \text{else}\ xs\ @\ [x])$$

<proof>

lemma *strip-while-map*:

$$\text{strip-while}\ P\ (\text{map}\ f\ xs) = \text{map}\ f\ (\text{strip-while}\ (P \circ f)\ xs)$$

<proof>

lemma *strip-while-dropWhile-commute*:

$$\text{strip-while}\ P\ (\text{dropWhile}\ Q\ xs) = \text{dropWhile}\ Q\ (\text{strip-while}\ P\ xs)$$

<proof>

lemma *dropWhile-strip-while-commute*:

$$\text{dropWhile}\ P\ (\text{strip-while}\ Q\ xs) = \text{strip-while}\ Q\ (\text{dropWhile}\ P\ xs)$$

<proof>

definition *no-leading* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a\ \text{list} \Rightarrow \text{bool}$

where

$$\text{no-leading}\ P\ xs \iff (xs \neq [] \longrightarrow \neg P\ (\text{hd}\ xs))$$

lemma *no-leading-Nil* [simp, intro!]:

$$\text{no-leading}\ P\ []$$

<proof>

lemma *no-leading-Cons* [simp, intro!]:

$$\text{no-leading}\ P\ (x\ \# xs) \iff \neg P\ x$$

$\langle \text{proof} \rangle$

lemma *no-leading-append* [simp]:

$\text{no-leading } P \ (xs @ ys) \longleftrightarrow \text{no-leading } P \ xs \wedge (xs = [] \longrightarrow \text{no-leading } P \ ys)$

$\langle \text{proof} \rangle$

lemma *no-leading-dropWhile* [simp]:

$\text{no-leading } P \ (\text{dropWhile } P \ xs)$

$\langle \text{proof} \rangle$

lemma *dropWhile-eq-obtain-leading*:

assumes $\text{dropWhile } P \ xs = ys$

obtains zs **where** $xs = zs @ ys$ **and** $\bigwedge z. z \in \text{set } zs \implies P \ z$ **and** $\text{no-leading } P \ ys$

$\langle \text{proof} \rangle$

lemma *dropWhile-idem-iff*:

$\text{dropWhile } P \ xs = xs \longleftrightarrow \text{no-leading } P \ xs$

$\langle \text{proof} \rangle$

abbreviation *no-trailing* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$

where

$\text{no-trailing } P \ xs \equiv \text{no-leading } P \ (\text{rev } xs)$

lemma *no-trailing-unfold*:

$\text{no-trailing } P \ xs \longleftrightarrow (xs \neq [] \longrightarrow \neg P \ (\text{last } xs))$

$\langle \text{proof} \rangle$

lemma *no-trailing-Nil* [simp, intro!]:

$\text{no-trailing } P \ []$

$\langle \text{proof} \rangle$

lemma *no-trailing-Cons* [simp]:

$\text{no-trailing } P \ (x \# xs) \longleftrightarrow \text{no-trailing } P \ xs \wedge (xs = [] \longrightarrow \neg P \ x)$

$\langle \text{proof} \rangle$

lemma *no-trailing-append*:

$\text{no-trailing } P \ (xs @ ys) \longleftrightarrow \text{no-trailing } P \ ys \wedge (ys = [] \longrightarrow \text{no-trailing } P \ xs)$

$\langle \text{proof} \rangle$

lemma *no-trailing-append-Cons* [simp]:

$\text{no-trailing } P \ (xs @ y \# ys) \longleftrightarrow \text{no-trailing } P \ (y \# ys)$

$\langle \text{proof} \rangle$

lemma *no-trailing-strip-while* [simp]:

$\text{no-trailing } P \ (\text{strip-while } P \ xs)$

$\langle \text{proof} \rangle$

lemma *strip-while-idem* [simp]:
 $\text{no-trailing } P \text{ } xs \implies \text{strip-while } P \text{ } xs = xs$
 ⟨proof⟩

lemma *strip-while-eq-obtain-trailing*:
assumes $\text{strip-while } P \text{ } xs = ys$
obtains zs **where** $xs = ys @ zs$ **and** $\bigwedge z. z \in \text{set } zs \implies P \text{ } z$ **and** $\text{no-trailing } P \text{ } ys$
 ⟨proof⟩

lemma *strip-while-idem-iff*:
 $\text{strip-while } P \text{ } xs = xs \longleftrightarrow \text{no-trailing } P \text{ } xs$
 ⟨proof⟩

lemma *no-trailing-map*:
 $\text{no-trailing } P \text{ } (\text{map } f \text{ } xs) \longleftrightarrow \text{no-trailing } (P \circ f) \text{ } xs$
 ⟨proof⟩

lemma *no-trailing-drop* [simp]:
 $\text{no-trailing } P \text{ } (\text{drop } n \text{ } xs) \text{ if } \text{no-trailing } P \text{ } xs$
 ⟨proof⟩

lemma *no-trailing-upt* [simp]:
 $\text{no-trailing } P \text{ } [n..<m] \longleftrightarrow (n < m \longrightarrow \neg P \text{ } (m - 1))$
 ⟨proof⟩

definition *nth-default* :: $'a \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a$
where
 $\text{nth-default } dftt \text{ } xs \text{ } n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } dftt)$

lemma *nth-default-nth*:
 $n < \text{length } xs \implies \text{nth-default } dftt \text{ } xs \text{ } n = xs ! n$
 ⟨proof⟩

lemma *nth-default-beyond*:
 $\text{length } xs \leq n \implies \text{nth-default } dftt \text{ } xs \text{ } n = dftt$
 ⟨proof⟩

lemma *nth-default-Nil* [simp]:
 $\text{nth-default } dftt \text{ } [] \text{ } n = dftt$
 ⟨proof⟩

lemma *nth-default-Cons*:
 $\text{nth-default } dftt \text{ } (x \# xs) \text{ } n = (\text{case } n \text{ of } 0 \Rightarrow x \mid \text{Suc } n' \Rightarrow \text{nth-default } dftt \text{ } xs \text{ } n')$
 ⟨proof⟩

lemma *nth-default-Cons-0* [simp]:
 $\text{nth-default } dftt \text{ } (x \# xs) \text{ } 0 = x$

$\langle \text{proof} \rangle$

lemma *nth-default-Cons-Suc* [simp]:

$\text{nth-default dflt } (x \# xs) \text{ (Suc } n) = \text{nth-default dflt } xs \text{ } n$

$\langle \text{proof} \rangle$

lemma *nth-default-replicate-dflt* [simp]:

$\text{nth-default dflt } (\text{replicate } n \text{ dflt}) \text{ } m = \text{dflt}$

$\langle \text{proof} \rangle$

lemma *nth-default-append*:

$\text{nth-default dflt } (xs @ ys) \text{ } n =$

$(\text{if } n < \text{length } xs \text{ then } \text{nth } xs \text{ } n \text{ else } \text{nth-default dflt } ys \text{ } (n - \text{length } xs))$

$\langle \text{proof} \rangle$

lemma *nth-default-append-trailing* [simp]:

$\text{nth-default dflt } (xs @ \text{replicate } n \text{ dflt}) = \text{nth-default dflt } xs$

$\langle \text{proof} \rangle$

lemma *nth-default-snoc-default* [simp]:

$\text{nth-default dflt } (xs @ [\text{dflt}]) = \text{nth-default dflt } xs$

$\langle \text{proof} \rangle$

lemma *nth-default-eq-dflt-iff*:

$\text{nth-default dflt } xs \text{ } k = \text{dflt} \longleftrightarrow (k < \text{length } xs \longrightarrow xs ! k = \text{dflt})$

$\langle \text{proof} \rangle$

lemma *nth-default-take-eq*:

$\text{nth-default dflt } (\text{take } m \text{ } xs) \text{ } n =$

$(\text{if } n < m \text{ then } \text{nth-default dflt } xs \text{ } n \text{ else } \text{dflt})$

$\langle \text{proof} \rangle$

lemma *in-enumerate-iff-nth-default-eq*:

$x \neq \text{dflt} \implies (n, x) \in \text{set } (\text{enumerate } 0 \text{ } xs) \longleftrightarrow \text{nth-default dflt } xs \text{ } n = x$

$\langle \text{proof} \rangle$

lemma *last-conv-nth-default*:

assumes $xs \neq []$

shows $\text{last } xs = \text{nth-default dflt } xs \text{ } (\text{length } xs - 1)$

$\langle \text{proof} \rangle$

lemma *nth-default-map-eq*:

$f \text{ dflt}' = \text{dflt} \implies \text{nth-default dflt } (\text{map } f \text{ } xs) \text{ } n = f \text{ } (\text{nth-default dflt}' \text{ } xs \text{ } n)$

$\langle \text{proof} \rangle$

lemma *finite-nth-default-neq-default* [simp]:

$\text{finite } \{k. \text{nth-default dflt } xs \text{ } k \neq \text{dflt}\}$

$\langle \text{proof} \rangle$

lemma *sorted-list-of-set-nth-default*:

sorted-list-of-set $\{k. \text{nth-default dflt } xs \ k \neq \text{dflt}\} = \text{map fst } (\text{filter } (\lambda(-, x). x \neq \text{dflt}) \ (\text{enumerate } 0 \ xs))$
 ⟨proof⟩

lemma *map-nth-default*:

map $(\text{nth-default } x \ xs) \ [0..<\text{length } xs] = xs$
 ⟨proof⟩

lemma *range-nth-default [simp]*:

range $(\text{nth-default dflt } xs) = \text{insert dflt } (\text{set } xs)$
 ⟨proof⟩

lemma *nth-strip-while*:

assumes $n < \text{length } (\text{strip-while } P \ xs)$
shows $\text{strip-while } P \ xs \ ! \ n = xs \ ! \ n$
 ⟨proof⟩

lemma *length-strip-while-le*:

length $(\text{strip-while } P \ xs) \leq \text{length } xs$
 ⟨proof⟩

lemma *nth-default-strip-while-dflt [simp]*:

nth-default dflt $(\text{strip-while } ((=) \ \text{dflt}) \ xs) = \text{nth-default dflt } xs$
 ⟨proof⟩

lemma *nth-default-eq-iff*:

nth-default dflt $xs = \text{nth-default dflt } ys$
 $\longleftrightarrow \text{strip-while } (\text{HOL.eq dflt}) \ xs = \text{strip-while } (\text{HOL.eq dflt}) \ ys \ (\text{is } ?P \longleftrightarrow ?Q)$
 ⟨proof⟩

lemma *nth-default-map2*:

$\langle \text{nth-default } d \ (\text{map2 } f \ xs \ ys) \ n = f \ (\text{nth-default } d1 \ xs \ n) \ (\text{nth-default } d2 \ ys \ n) \rangle$
if $\langle \text{length } xs = \text{length } ys \rangle$ **and** $\langle f \ d1 \ d2 = d \rangle$ **for** $bs \ cs$
 ⟨proof⟩

end

theory *Cancellation*

imports *Main*

begin

named-theorems *cancelation-simproc-pre* $\langle$ These theorems are here to normalise the term. Special

handling of constructors should be here. Remark that only the simproc $\text{@}\{\text{term NO-MATCH}\}$ is also included. $\rangle$

named-theorems *cancelation-simproc-post* $\langle$ These theorems are here to normalise the term, after the cancelation simproc. Normalisation of $\langle$ iterate-add $\rangle$ back to the normale representation should be put here. $\rangle$

named-theorems *cancelation-simproc-eq-elim* $\langle$ These theorems are here to help deriving contradiction (e.g., $\langle$ Suc - = 0 $\rangle$). $\rangle$

definition *iterate-add* :: $\langle$ nat $\Rightarrow$ 'a::cancel-comm-monoid-add $\Rightarrow$ 'a $\rangle$ **where**
 $\langle$ iterate-add n a = (((+) a) $\widehat{\sim}$ n) 0 $\rangle$

lemma *iterate-add-simps[simp]*:
 $\langle$ iterate-add 0 a = 0 $\rangle$
 $\langle$ iterate-add (Suc n) a = a + iterate-add n a $\rangle$
 $\langle$ proof $\rangle$

lemma *iterate-add-empty[simp]*: $\langle$ iterate-add n 0 = 0 $\rangle$
 $\langle$ proof $\rangle$

lemma *iterate-add-distrib[simp]*: $\langle$ iterate-add (m+n) a = iterate-add m a + iterate-add n a $\rangle$
 $\langle$ proof $\rangle$

lemma *iterate-add-Numeral1*: $\langle$ iterate-add n Numeral1 = of-nat n $\rangle$
 $\langle$ proof $\rangle$

lemma *iterate-add-1*: $\langle$ iterate-add n 1 = of-nat n $\rangle$
 $\langle$ proof $\rangle$

lemma *iterate-add-eq-add-iff1*:
 $\langle$ i $\leq$ j $\implies$ (iterate-add j u + m = iterate-add i u + n) = (iterate-add (j - i) u + m = n) $\rangle$
 $\langle$ proof $\rangle$

lemma *iterate-add-eq-add-iff2*:
 $\langle$ i $\leq$ j $\implies$ (iterate-add i u + m = iterate-add j u + n) = (m = iterate-add (j - i) u + n) $\rangle$
 $\langle$ proof $\rangle$

lemma *iterate-add-less-iff1*:
 $j \leq (i::nat) \implies (\text{iterate-add } i \text{ (u::'a :: \{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le\})} + m < \text{iterate-add } j \text{ u + n}) = (\text{iterate-add } (i-j) \text{ u + m} < n)$
 $\langle$ proof $\rangle$

lemma *iterate-add-less-iff2*:
 $i \leq (j::nat) \implies (\text{iterate-add } i \text{ (u::'a :: \{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le\})}$

$+ m < \text{iterate-add } j \ u + n = (m < \text{iterate-add } (j - i) \ u + n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-less-eq-iff1*:

$j \leq (i::\text{nat}) \implies (\text{iterate-add } i \ (u:: 'a :: \{\text{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le}\}))$
 $+ m \leq \text{iterate-add } j \ u + n = (\text{iterate-add } (i-j) \ u + m \leq n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-less-eq-iff2*:

$i \leq (j::\text{nat}) \implies (\text{iterate-add } i \ (u:: 'a :: \{\text{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le}\}))$
 $+ m \leq \text{iterate-add } j \ u + n = (m \leq \text{iterate-add } (j - i) \ u + n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-add-eq1*:

$j \leq (i::\text{nat}) \implies ((\text{iterate-add } i \ u + m) - (\text{iterate-add } j \ u + n)) = ((\text{iterate-add } (i-j) \ u + m) - n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-diff-add-eq2*:

$i \leq (j::\text{nat}) \implies ((\text{iterate-add } i \ u + m) - (\text{iterate-add } j \ u + n)) = (m - (\text{iterate-add } (j-i) \ u + n))$
 $\langle \text{proof} \rangle$

61.1 Simproc Set-Up

$\langle \text{ML} \rangle$

end

62 (Finite) Multisets

theory *Multiset*

imports *Cancellation*

begin

62.1 The type of multisets

typedef $'a \text{ multiset} = \langle \{f :: 'a \Rightarrow \text{nat. finite } \{x. f \ x > 0\}\} \rangle$
morphisms *count Abs-multiset*
 $\langle \text{proof} \rangle$

setup-lifting *type-definition-multiset*

lemma *count-Abs-multiset*:

$\langle \text{count } (\text{Abs-multiset } f) = f \rangle \text{ if } \langle \text{finite } \{x. f \ x > 0\} \rangle$
 $\langle \text{proof} \rangle$

lemma *multiset-eq-iff*: $M = N \longleftrightarrow (\forall a. \text{count } M \ a = \text{count } N \ a)$

$\langle \text{proof} \rangle$

lemma *multiset-eqI*: $(\bigwedge x. \text{count } A \ x = \text{count } B \ x) \implies A = B$
 $\langle \text{proof} \rangle$

Preservation of the representing set *multiset*.

lemma *diff-preserves-multiset*:
 $\langle \text{finite } \{x. 0 < M \ x - N \ x\} \rangle \text{ if } \langle \text{finite } \{x. 0 < M \ x\} \rangle \text{ for } M \ N :: \langle 'a \Rightarrow \text{nat} \rangle$
 $\langle \text{proof} \rangle$

lemma *filter-preserves-multiset*:
 $\langle \text{finite } \{x. 0 < (\text{if } P \ x \text{ then } M \ x \text{ else } 0)\} \rangle \text{ if } \langle \text{finite } \{x. 0 < M \ x\} \rangle \text{ for } M \ N ::$
 $\langle 'a \Rightarrow \text{nat} \rangle$
 $\langle \text{proof} \rangle$

lemmas *in-multiset* = *diff-preserves-multiset* *filter-preserves-multiset*

62.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (type) *cancel-comm-monoid-add*
begin

lift-definition *zero-multiset* :: $\langle 'a \text{ multiset} \rangle$
is $\langle \lambda a. 0 \rangle$
 $\langle \text{proof} \rangle$

abbreviation *empty-mset* :: $\langle 'a \text{ multiset} \rangle (\langle \{\#\} \rangle)$
where $\langle \text{empty-mset} \equiv 0 \rangle$

lift-definition *plus-multiset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$
is $\langle \lambda M \ N \ a. M \ a + N \ a \rangle$
 $\langle \text{proof} \rangle$

lift-definition *minus-multiset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$
is $\langle \lambda M \ N \ a. M \ a - N \ a \rangle$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

context
begin

qualified definition *is-empty* :: $\langle 'a \text{ multiset} \Rightarrow \text{bool} \rangle$ **where**
 $[\text{code-abbrev}]: \text{is-empty } A \longleftrightarrow A = \{\#\}$

end

lemma *add-mset-in-multiset*:

$\langle \text{finite } \{x. 0 < (\text{if } x = a \text{ then } \text{Suc } (M x) \text{ else } M x)\} \rangle$
if $\langle \text{finite } \{x. 0 < M x\} \rangle$
 $\langle \text{proof} \rangle$

lift-definition *add-mset* :: $'a \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$ **is**

$\lambda a M b. \text{if } b = a \text{ then } \text{Suc } (M b) \text{ else } M b$
 $\langle \text{proof} \rangle$

syntax

-multiset :: $\text{args} \Rightarrow 'a \text{ multiset} \quad (\{\#(-)\# \})$

translations

$\{\#x, xs\# \} == \text{CONST } \text{add-mset } x \ \{\#xs\# \}$
 $\{\#x\# \} == \text{CONST } \text{add-mset } x \ \{\# \}$

lemma *count-empty* [simp]: $\text{count } \{\# \} a = 0$

$\langle \text{proof} \rangle$

lemma *count-add-mset* [simp]:

$\text{count } (\text{add-mset } b A) a = (\text{if } b = a \text{ then } \text{Suc } (\text{count } A a) \text{ else } \text{count } A a)$
 $\langle \text{proof} \rangle$

lemma *count-single*: $\text{count } \{\#b\# \} a = (\text{if } b = a \text{ then } 1 \text{ else } 0)$

$\langle \text{proof} \rangle$

lemma

add-mset-not-empty [simp]: $\langle \text{add-mset } a A \neq \{\# \} \rangle$ **and**
empty-not-add-mset [simp]: $\{\# \} \neq \text{add-mset } a A$
 $\langle \text{proof} \rangle$

lemma *add-mset-add-mset-same-iff* [simp]:

$\text{add-mset } a A = \text{add-mset } a B \longleftrightarrow A = B$
 $\langle \text{proof} \rangle$

lemma *add-mset-commute*:

$\text{add-mset } x (\text{add-mset } y M) = \text{add-mset } y (\text{add-mset } x M)$
 $\langle \text{proof} \rangle$

62.3 Basic operations

62.3.1 Conversion to set and membership

definition *set-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ set} \rangle$

where $\langle \text{set-mset } M = \{x. \text{count } M x > 0\} \rangle$

abbreviation *member-mset* :: $\langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$

where $\langle \text{member-mset } a M \equiv a \in \text{set-mset } M \rangle$

notation

member-mset $\langle \iota'(\in \#') \rangle$ **and**
member-mset $\langle \iota(-/\in \# -) \rangle [50, 51] 50)$

notation (ASCII)

member-mset $\langle \iota'(:\#') \rangle$ **and**
member-mset $\langle \iota(-/:\# -) \rangle [50, 51] 50)$

abbreviation *not-member-mset* $:: \langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$
where $\langle \text{not-member-mset } a \ M \equiv a \notin \text{set-mset } M \rangle$

notation

not-member-mset $\langle \iota'(\notin \#') \rangle$ **and**
not-member-mset $\langle \iota(-/\notin \# -) \rangle [50, 51] 50)$

notation (ASCII)

not-member-mset $\langle \iota'(\sim \#') \rangle$ **and**
not-member-mset $\langle \iota(-/\sim \# -) \rangle [50, 51] 50)$

context

begin

qualified abbreviation *Ball* $:: 'a \text{ multiset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where $\text{Ball } M \equiv \text{Set.Ball } (\text{set-mset } M)$

qualified abbreviation *Bex* $:: 'a \text{ multiset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where $\text{Bex } M \equiv \text{Set.Bex } (\text{set-mset } M)$

end

syntax

-MBall $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \forall -\in \# -) [0, 0, 10] 10)$
-MBex $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \exists -\in \# -) [0, 0, 10] 10)$

syntax (ASCII)

-MBall $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \forall -:\# -) [0, 0, 10] 10)$
-MBex $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \exists -:\# -) [0, 0, 10] 10)$

translations

$\forall x \in \# A. P \Rightarrow \text{CONST Multiset.Ball } A (\lambda x. P)$
 $\exists x \in \# A. P \Rightarrow \text{CONST Multiset.Bex } A (\lambda x. P)$

$\langle ML \rangle$

lemma *count-eq-zero-iff*:

$\text{count } M \ x = 0 \longleftrightarrow x \notin \# M$
 $\langle \text{proof} \rangle$

lemma *not-in-iff*:

$x \notin \# M \longleftrightarrow \text{count } M \ x = 0$

$\langle \text{proof} \rangle$

lemma *count-greater-zero-iff* [simp]:
 $\text{count } M \ x > 0 \longleftrightarrow x \in \# \ M$
 $\langle \text{proof} \rangle$

lemma *count-inI*:
assumes $\text{count } M \ x = 0 \implies \text{False}$
shows $x \in \# \ M$
 $\langle \text{proof} \rangle$

lemma *in-countE*:
assumes $x \in \# \ M$
obtains n **where** $\text{count } M \ x = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *count-greater-eq-Suc-zero-iff* [simp]:
 $\text{count } M \ x \geq \text{Suc } 0 \longleftrightarrow x \in \# \ M$
 $\langle \text{proof} \rangle$

lemma *count-greater-eq-one-iff* [simp]:
 $\text{count } M \ x \geq 1 \longleftrightarrow x \in \# \ M$
 $\langle \text{proof} \rangle$

lemma *set-mset-empty* [simp]:
 $\text{set-mset } \{\#\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *set-mset-single*:
 $\text{set-mset } \{\#b\# \} = \{b\}$
 $\langle \text{proof} \rangle$

lemma *set-mset-eq-empty-iff* [simp]:
 $\text{set-mset } M = \{\} \longleftrightarrow M = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *finite-set-mset* [iff]:
 $\text{finite } (\text{set-mset } M)$
 $\langle \text{proof} \rangle$

lemma *set-mset-add-mset-insert* [simp]: $\langle \text{set-mset } (\text{add-mset } a \ A) = \text{insert } a \ (\text{set-mset } A) \rangle$
 $\langle \text{proof} \rangle$

lemma *multiset-nonemptyE* [elim]:
assumes $A \neq \{\#\}$
obtains x **where** $x \in \# \ A$
 $\langle \text{proof} \rangle$

62.3.2 Union

lemma *count-union* [simp]:
 $\text{count } (M + N) \ a = \text{count } M \ a + \text{count } N \ a$
 ⟨proof⟩

lemma *set-mset-union* [simp]:
 $\text{set-mset } (M + N) = \text{set-mset } M \cup \text{set-mset } N$
 ⟨proof⟩

lemma *union-mset-add-mset-left* [simp]:
 $\text{add-mset } a \ A + B = \text{add-mset } a \ (A + B)$
 ⟨proof⟩

lemma *union-mset-add-mset-right* [simp]:
 $A + \text{add-mset } a \ B = \text{add-mset } a \ (A + B)$
 ⟨proof⟩

lemma *add-mset-add-single*: $\langle \text{add-mset } a \ A = A + \{\#a\# \} \rangle$
 ⟨proof⟩

62.3.3 Difference

instance *multiset* :: (type) *comm-monoid-diff*
 ⟨proof⟩

lemma *count-diff* [simp]:
 $\text{count } (M - N) \ a = \text{count } M \ a - \text{count } N \ a$
 ⟨proof⟩

lemma *add-mset-diff-bothsides*:
 $\langle \text{add-mset } a \ M - \text{add-mset } a \ A = M - A \rangle$
 ⟨proof⟩

lemma *in-diff-count*:
 $a \in\# \ M - N \longleftrightarrow \text{count } N \ a < \text{count } M \ a$
 ⟨proof⟩

lemma *count-in-diffI*:
 assumes $\bigwedge n. \text{count } N \ x = n + \text{count } M \ x \implies \text{False}$
 shows $x \in\# \ M - N$
 ⟨proof⟩

lemma *in-diff-countE*:
 assumes $x \in\# \ M - N$
 obtains n where $\text{count } M \ x = \text{Suc } n + \text{count } N \ x$
 ⟨proof⟩

lemma *in-diffD*:
 assumes $a \in\# \ M - N$

shows $a \in \# M$
 $\langle proof \rangle$

lemma *set-mset-diff*:
 $set\text{-}mset (M - N) = \{a. count\ N\ a < count\ M\ a\}$
 $\langle proof \rangle$

lemma *diff-empty [simp]*: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
 $\langle proof \rangle$

lemma *diff-cancel*: $A - A = \{\#\}$
 $\langle proof \rangle$

lemma *diff-union-cancelR*: $M + N - N = (M :: 'a\ multiset)$
 $\langle proof \rangle$

lemma *diff-union-cancelL*: $N + M - N = (M :: 'a\ multiset)$
 $\langle proof \rangle$

lemma *diff-right-commute*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - N - Q = M - Q - N$
 $\langle proof \rangle$

lemma *diff-add*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - (N + Q) = M - N - Q$
 $\langle proof \rangle$

lemma *insert-DiffM [simp]*: $x \in \# M \implies add\text{-}mset\ x\ (M - \{\#x\}) = M$
 $\langle proof \rangle$

lemma *insert-DiffM2*: $x \in \# M \implies (M - \{\#x\}) + \{\#x\} = M$
 $\langle proof \rangle$

lemma *diff-union-swap*: $a \neq b \implies add\text{-}mset\ b\ (M - \{\#a\}) = add\text{-}mset\ b\ M - \{\#a\}$
 $\langle proof \rangle$

lemma *diff-add-mset-swap [simp]*: $b \notin \# A \implies add\text{-}mset\ b\ M - A = add\text{-}mset\ b\ (M - A)$
 $\langle proof \rangle$

lemma *diff-union-swap2 [simp]*: $y \in \# M \implies add\text{-}mset\ x\ M - \{\#y\} = add\text{-}mset\ x\ (M - \{\#y\})$
 $\langle proof \rangle$

lemma *diff-diff-add-mset [simp]*: $(M :: 'a\ multiset) - N - P = M - (N + P)$
 $\langle proof \rangle$

lemma *diff-union-single-conv*:

$a \in\# J \implies I + J - \{\#a\# \} = I + (J - \{\#a\# \})$
 $\langle \text{proof} \rangle$

lemma *mset-add [elim?]*:

assumes $a \in\# A$
obtains B **where** $A = \text{add-mset } a \ B$
 $\langle \text{proof} \rangle$

lemma *union-iff*:

$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
 $\langle \text{proof} \rangle$

62.3.4 Min and Max

abbreviation *Min-mset* :: $'a::\text{linorder multiset} \Rightarrow 'a$ **where**
 $\text{Min-mset } m \equiv \text{Min } (\text{set-mset } m)$

abbreviation *Max-mset* :: $'a::\text{linorder multiset} \Rightarrow 'a$ **where**
 $\text{Max-mset } m \equiv \text{Max } (\text{set-mset } m)$

62.3.5 Equality of multisets

lemma *single-eq-single [simp]*: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *union-eq-empty [iff]*: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *empty-eq-union [iff]*: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *multi-self-add-other-not-self [simp]*: $M = \text{add-mset } x \ M \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *add-mset-remove-trivial [simp]*: $\langle \text{add-mset } x \ M - \{\#x\#\} = M \rangle$
 $\langle \text{proof} \rangle$

lemma *diff-single-trivial*: $\neg x \in\# M \implies M - \{\#x\#\} = M$
 $\langle \text{proof} \rangle$

lemma *diff-single-eq-union*: $x \in\# M \implies M - \{\#x\#\} = N \longleftrightarrow M = \text{add-mset } x \ N$
 $\langle \text{proof} \rangle$

lemma *union-single-eq-diff*: $\text{add-mset } x \ M = N \implies M = N - \{\#x\#\}$
 $\langle \text{proof} \rangle$

lemma *union-single-eq-member*: $\text{add-mset } x \ M = N \implies x \in\# N$

$\langle \text{proof} \rangle$

lemma *add-mset-remove-trivial-If*:

$\text{add-mset } a \ (N - \{\#a\# \}) = (\text{if } a \in\# \ N \text{ then } N \text{ else } \text{add-mset } a \ N)$

$\langle \text{proof} \rangle$

lemma *add-mset-remove-trivial-eq*: $\langle N = \text{add-mset } a \ (N - \{\#a\# \}) \longleftrightarrow a \in\# \ N \rangle$

$\langle \text{proof} \rangle$

lemma *union-is-single*:

$M + N = \{\#a\# \} \longleftrightarrow M = \{\#a\# \} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\# \}$

(**is** ?lhs = ?rhs)

$\langle \text{proof} \rangle$

lemma *single-is-union*: $\{\#a\# \} = M + N \longleftrightarrow \{\#a\# \} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\# \} = N$

$\langle \text{proof} \rangle$

lemma *add-eq-conv-diff*:

$\text{add-mset } a \ M = \text{add-mset } b \ N \longleftrightarrow M = N \wedge a = b \vee M = \text{add-mset } b \ (N - \{\#a\# \}) \wedge N = \text{add-mset } a \ (M - \{\#b\# \})$

(**is** ?lhs $\longleftrightarrow$?rhs)

$\langle \text{proof} \rangle$

lemma *add-mset-eq-single [iff]*: $\text{add-mset } b \ M = \{\#a\# \} \longleftrightarrow b = a \wedge M = \{\#\}$

$\langle \text{proof} \rangle$

lemma *single-eq-add-mset [iff]*: $\{\#a\# \} = \text{add-mset } b \ M \longleftrightarrow b = a \wedge M = \{\#\}$

$\langle \text{proof} \rangle$

lemma *insert-noteq-member*:

assumes *BC*: $\text{add-mset } b \ B = \text{add-mset } c \ C$

and *bnotc*: $b \neq c$

shows $c \in\# \ B$

$\langle \text{proof} \rangle$

lemma *add-eq-conv-ex*:

$(\text{add-mset } a \ M = \text{add-mset } b \ N) =$

$(M = N \wedge a = b \vee (\exists K. M = \text{add-mset } b \ K \wedge N = \text{add-mset } a \ K))$

$\langle \text{proof} \rangle$

lemma *multi-member-split*: $x \in\# \ M \implies \exists A. M = \text{add-mset } x \ A$

$\langle \text{proof} \rangle$

lemma *multiset-add-sub-el-shuffle*:

assumes $c \in\# \ B$

and $b \neq c$

shows $\text{add-mset } b \ (B - \{\#c\# \}) = \text{add-mset } b \ B - \{\#c\# \}$
 $\langle \text{proof} \rangle$

lemma $\text{add-mset-eq-singleton-iff}[\text{iff}]$:
 $\text{add-mset } x \ M = \{\#y\#\} \longleftrightarrow M = \{\#\} \wedge x = y$
 $\langle \text{proof} \rangle$

62.3.6 Pointwise ordering induced by count

definition $\text{subseteq-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\subseteq\#$ 50)
where $A \subseteq\# B \longleftrightarrow (\forall a. \text{count } A \ a \leq \text{count } B \ a)$

definition $\text{subset-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\subset\#$ 50)
where $A \subset\# B \longleftrightarrow A \subseteq\# B \wedge A \neq B$

abbreviation (*input*) $\text{supseteq-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\supseteq\#$ 50)
where $\text{supseteq-mset } A \ B \equiv B \subseteq\# A$

abbreviation (*input*) $\text{supset-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\supset\#$ 50)
where $\text{supset-mset } A \ B \equiv B \subset\# A$

notation (*input*)
 subseq-mset (**infix** $\leq\#$ 50) **and**
 supseq-mset (**infix** $\geq\#$ 50)

notation (*ASCII*)
 subseq-mset (**infix** $\leq\#$ 50) **and**
 subset-mset (**infix** $<\#$ 50) **and**
 supseq-mset (**infix** $\geq\#$ 50) **and**
 supset-mset (**infix** $>\#$ 50)

global-interpretation subset-mset : *ordering* $\langle(\subseteq\#)\rangle \langle(\subset\#)\rangle$
 $\langle \text{proof} \rangle$

interpretation subset-mset : *ordered-ab-semigroup-add-imp-le* $\langle(+)\rangle \langle(-)\rangle \langle(\subseteq\#)\rangle$
 $\langle(\subset\#)\rangle$
 $\langle \text{proof} \rangle$

interpretation subset-mset : *ordered-ab-semigroup-monoid-add-imp-le* $(+) \ 0 \ (-)$
 $(\subseteq\#) \ (\subset\#)$
 $\langle \text{proof} \rangle$

lemma mset-subset-eqI :
 $(\bigwedge a. \text{count } A \ a \leq \text{count } B \ a) \implies A \subseteq\# B$
 $\langle \text{proof} \rangle$

lemma $\text{mset-subset-eq-count}$:

$A \subseteq\# B \implies \text{count } A \ a \leq \text{count } B \ a$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-exists-conv*: $(A::'a \text{ multiset}) \subseteq\# B \longleftrightarrow (\exists C. B = A + C)$
 $\langle \text{proof} \rangle$

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff* $(+)$ 0 $(\subseteq\#)$ $(\subset\#)$
 $(-)$
 $\langle \text{proof} \rangle$

declare *subset-mset.add-diff-assoc*[simp] *subset-mset.add-diff-assoc2*[simp]

lemma *mset-subset-eq-mono-add-right-cancel*: $(A::'a \text{ multiset}) + C \subseteq\# B + C$
 $\longleftrightarrow A \subseteq\# B$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-mono-add-left-cancel*: $C + (A::'a \text{ multiset}) \subseteq\# C + B \longleftrightarrow$
 $A \subseteq\# B$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-mono-add*: $(A::'a \text{ multiset}) \subseteq\# B \implies C \subseteq\# D \implies A +$
 $C \subseteq\# B + D$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-add-left*: $(A::'a \text{ multiset}) \subseteq\# A + B$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-add-right*: $B \subseteq\# (A::'a \text{ multiset}) + B$
 $\langle \text{proof} \rangle$

lemma *single-subset-iff* [simp]:
 $\{\#a\# \} \subseteq\# M \longleftrightarrow a \in\# M$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-single*: $a \in\# B \implies \{\#a\# \} \subseteq\# B$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-add-mset-cancel*: $\langle \text{add-mset } a \ A \subseteq\# \text{ add-mset } a \ B \longleftrightarrow A$
 $\subseteq\# B \rangle$
 $\langle \text{proof} \rangle$

lemma *multiset-diff-union-assoc*:
fixes $A \ B \ C \ D :: 'a \text{ multiset}$
shows $C \subseteq\# B \implies A + B - C = A + (B - C)$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-multiset-union-diff-commute*:
fixes $A \ B \ C \ D :: 'a \text{ multiset}$
shows $B \subseteq\# A \implies A - B + C = A + C - B$

$\langle \text{proof} \rangle$

lemma *diff-subset-eq-self[simp]*:
 $(M::'a \text{ multiset}) - N \subseteq\# M$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eqD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
 $\langle \text{proof} \rangle$

lemma *mset-subsetD*:
 $A \subset\# B \implies x \in\# A \implies x \in\# B$
 $\langle \text{proof} \rangle$

lemma *set-mset-mono*:
 $A \subseteq\# B \implies \text{set-mset } A \subseteq \text{set-mset } B$
 $\langle \text{proof} \rangle$

lemma *mset-subset-eq-insertD*:
 $\text{add-mset } x \ A \subseteq\# B \implies x \in\# B \wedge A \subset\# B$
 $\langle \text{proof} \rangle$

lemma *mset-subset-insertD*:
 $\text{add-mset } x \ A \subset\# B \implies x \in\# B \wedge A \subset\# B$
 $\langle \text{proof} \rangle$

lemma *mset-subset-of-empty[simp]*: $A \subset\# \{\#\} \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *empty-subset-add-mset[simp]*: $\{\#\} \subset\# \text{add-mset } x \ M$
 $\langle \text{proof} \rangle$

lemma *empty-le*: $\{\#\} \subseteq\# A$
 $\langle \text{proof} \rangle$

lemma *insert-subset-eq-iff*:
 $\text{add-mset } a \ A \subseteq\# B \longleftrightarrow a \in\# B \wedge A \subseteq\# B - \{\#a\# \}$
 $\langle \text{proof} \rangle$

lemma *insert-union-subset-iff*:
 $\text{add-mset } a \ A \subset\# B \longleftrightarrow a \in\# B \wedge A \subset\# B - \{\#a\# \}$
 $\langle \text{proof} \rangle$

lemma *subset-eq-diff-conv*:
 $A - C \subseteq\# B \longleftrightarrow A \subseteq\# B + C$
 $\langle \text{proof} \rangle$

lemma *multi-psub-of-add-self [simp]*: $A \subset\# \text{add-mset } x \ A$

$\langle \text{proof} \rangle$

lemma *multi-psub-self*: $A \subset\# A = \text{False}$
 $\langle \text{proof} \rangle$

lemma *mset-subset-add-mset* [simp]: $\text{add-mset } x \ N \subset\# \text{add-mset } x \ M \longleftrightarrow N \subset\# M$
 $\langle \text{proof} \rangle$

lemma *mset-subset-diff-self*: $c \in\# B \implies B - \{\#c\} \subset\# B$
 $\langle \text{proof} \rangle$

lemma *Diff-eq-empty-iff-mset*: $A - B = \{\#\} \longleftrightarrow A \subseteq\# B$
 $\langle \text{proof} \rangle$

lemma *add-mset-subseteq-single-iff*[iff]: $\text{add-mset } a \ M \subseteq\# \{\#b\} \longleftrightarrow M = \{\#\} \wedge a = b$
 $\langle \text{proof} \rangle$

62.3.7 Intersection and bounded union

definition *inter-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ (**infixl** $\langle \cap\# \rangle$ 70)
where $\langle A \cap\# B = A - (A - B) \rangle$

lemma *count-inter-mset* [simp]:
 $\langle \text{count } (A \cap\# B) \ x = \min (\text{count } A \ x) (\text{count } B \ x) \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset*: *semilattice-inf* $\langle (\cap\#) \rangle \langle (\subseteq\#) \rangle \langle (\subset\#) \rangle$
 $\langle \text{proof} \rangle$

definition *union-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ (**infixl** $\langle \cup\# \rangle$ 70)
where $\langle A \cup\# B = A + (B - A) \rangle$

lemma *count-union-mset* [simp]:
 $\langle \text{count } (A \cup\# B) \ x = \max (\text{count } A \ x) (\text{count } B \ x) \rangle$
 $\langle \text{proof} \rangle$

global-interpretation *subset-mset*: *semilattice-neutr-order* $\langle (\cup\#) \rangle \langle \{\#\} \rangle \langle (\supseteq\#) \rangle \langle (\supset\#) \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset*: *semilattice-sup* $\langle (\cup\#) \rangle \langle (\subseteq\#) \rangle \langle (\subset\#) \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset: bounded-lattice-bot* $(\cap\#)$ $(\subseteq\#)$ $(\subset\#)$
 $(\cup\#)$ $\{\#\}$
 $\langle proof \rangle$

62.3.8 Additional intersection facts

lemma *set-mset-inter [simp]*:
 $set\text{-}mset\ (A \cap\# B) = set\text{-}mset\ A \cap set\text{-}mset\ B$
 $\langle proof \rangle$

lemma *diff-intersect-left-idem [simp]*:
 $M - M \cap\# N = M - N$
 $\langle proof \rangle$

lemma *diff-intersect-right-idem [simp]*:
 $M - N \cap\# M = M - N$
 $\langle proof \rangle$

lemma *multiset-inter-single[simp]*: $a \neq b \implies \{\#a\# \} \cap\# \{\#b\# \} = \{\#\}$
 $\langle proof \rangle$

lemma *multiset-union-diff-commute*:
assumes $B \cap\# C = \{\#\}$
shows $A + B - C = A - C + B$
 $\langle proof \rangle$

lemma *disjunct-not-in*:
 $A \cap\# B = \{\#\} \longleftrightarrow (\forall a. a \notin\# A \vee a \notin\# B)$ (**is** $?P \longleftrightarrow ?Q$)
 $\langle proof \rangle$

lemma *inter-mset-empty-distrib-right*: $A \cap\# (B + C) = \{\#\} \longleftrightarrow A \cap\# B = \{\#\} \wedge A \cap\# C = \{\#\}$
 $\langle proof \rangle$

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$
 $\langle proof \rangle$

lemma *add-mset-inter-add-mset [simp]*:
 $add\text{-}mset\ a\ A \cap\# add\text{-}mset\ a\ B = add\text{-}mset\ a\ (A \cap\# B)$
 $\langle proof \rangle$

lemma *add-mset-disjoint [simp]*:
 $add\text{-}mset\ a\ A \cap\# B = \{\#\} \longleftrightarrow a \notin\# B \wedge A \cap\# B = \{\#\}$
 $\{\#\} = add\text{-}mset\ a\ A \cap\# B \longleftrightarrow a \notin\# B \wedge \{\#\} = A \cap\# B$
 $\langle proof \rangle$

lemma *disjoint-add-mset [simp]*:
 $B \cap\# add\text{-}mset\ a\ A = \{\#\} \longleftrightarrow a \notin\# B \wedge B \cap\# A = \{\#\}$

$$\{\#\} = A \cap\# \text{ add-mset } b \ B \longleftrightarrow b \notin\# A \wedge \{\#\} = A \cap\# B$$

<proof>

lemma *inter-add-left1*: $\neg x \in\# N \implies (\text{add-mset } x \ M) \cap\# N = M \cap\# N$
<proof>

lemma *inter-add-left2*: $x \in\# N \implies (\text{add-mset } x \ M) \cap\# N = \text{add-mset } x \ (M \cap\# (N - \{\#x\#}))$
<proof>

lemma *inter-add-right1*: $\neg x \in\# N \implies N \cap\# (\text{add-mset } x \ M) = N \cap\# M$
<proof>

lemma *inter-add-right2*: $x \in\# N \implies N \cap\# (\text{add-mset } x \ M) = \text{add-mset } x \ ((N - \{\#x\#}) \cap\# M)$
<proof>

lemma *disjunct-set-mset-diff*:
assumes $M \cap\# N = \{\#\}$
shows $\text{set-mset } (M - N) = \text{set-mset } M$
<proof>

lemma *at-most-one-mset-mset-diff*:
assumes $a \notin\# M - \{\#a\#$
shows $\text{set-mset } (M - \{\#a\#}) = \text{set-mset } M - \{a\}$
<proof>

lemma *more-than-one-mset-mset-diff*:
assumes $a \in\# M - \{\#a\#$
shows $\text{set-mset } (M - \{\#a\#}) = \text{set-mset } M$
<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>

62.3.9 Additional bounded union facts

lemma *set-mset-sup* [simp]:

$$\langle \text{set-mset } (A \cup\# B) = \text{set-mset } A \cup \text{set-mset } B \rangle$$

<proof>

lemma *sup-union-left1* [simp]: $\neg x \in\# N \implies (\text{add-mset } x M) \cup\# N = \text{add-mset } x (M \cup\# N)$

<proof>

lemma *sup-union-left2*: $x \in\# N \implies (\text{add-mset } x M) \cup\# N = \text{add-mset } x (M \cup\# (N - \{\#x\}))$

<proof>

lemma *sup-union-right1* [simp]: $\neg x \in\# N \implies N \cup\# (\text{add-mset } x M) = \text{add-mset } x (N \cup\# M)$

<proof>

lemma *sup-union-right2*: $x \in\# N \implies N \cup\# (\text{add-mset } x M) = \text{add-mset } x ((N - \{\#x\}) \cup\# M)$

<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*:

$$\langle \text{add-mset } a A \cup\# \text{add-mset } a B = \text{add-mset } a (A \cup\# B) \rangle$$

<proof>

62.4 Replicate and repeat operations

definition *replicate-mset* :: $\text{nat} \Rightarrow 'a \Rightarrow 'a \text{ multiset}$ **where**

$$\text{replicate-mset } n x = (\text{add-mset } x \overset{\sim}{\sim} n) \{\#\}$$

lemma *replicate-mset-0*[simp]: $\text{replicate-mset } 0 x = \{\#\}$

<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* :: $\langle \text{nat} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$
is $\langle \lambda n M a. n * M a \rangle$ ⟨proof⟩

lemma *count-repeat-mset* [simp]: *count (repeat-mset i A) a = i * count A a*
 ⟨proof⟩

lemma *repeat-mset-0* [simp]:
 $\langle \text{repeat-mset } 0 M = \{\#\} \rangle$
 ⟨proof⟩

lemma *repeat-mset-Suc* [simp]:
 $\langle \text{repeat-mset (Suc n) } M = M + \text{repeat-mset } n M \rangle$
 ⟨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'*: $\langle \text{repeat-mset } (i - j) u = \text{repeat-mset } i u - \text{repeat-mset } j u \rangle$
 ⟨proof⟩

lemma *left-add-mult-distrib-mset*:
 $\text{repeat-mset } i u + (\text{repeat-mset } j u + k) = \text{repeat-mset } (i+j) u + k$
 ⟨proof⟩

lemma *repeat-mset-distrib*:
 $\text{repeat-mset } (m + n) A = \text{repeat-mset } m A + \text{repeat-mset } n A$
 ⟨proof⟩

lemma *repeat-mset-distrib2*[simp]:
 $\text{repeat-mset } n (A + B) = \text{repeat-mset } n A + \text{repeat-mset } n B$
 ⟨proof⟩

lemma *repeat-mset-replicate-mset*[simp]:
 $\text{repeat-mset } n \{\#a\# \} = \text{replicate-mset } n a$
 ⟨proof⟩

lemma *repeat-mset-distrib-add-mset*[simp]:
 $\text{repeat-mset } n (\text{add-mset } a A) = \text{replicate-mset } n a + \text{repeat-mset } n A$
 ⟨proof⟩

lemma *repeat-mset-empty[simp]*: $\text{repeat-mset } n \ \{\#\} = \{\#\}$
 ⟨proof⟩

62.4.1 Simprocs

lemma *repeat-mset-iterate-add*: $\langle \text{repeat-mset } n \ M = \text{iterate-add } n \ M \rangle$
 ⟨proof⟩

lemma *mset-subseteq-add-iff1*:
 $j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subseteq\# n)$
 ⟨proof⟩

lemma *mset-subseteq-add-iff2*:
 $i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{repeat-mset } j \ u + n) = (m \subseteq\# \text{repeat-mset } (j-i) \ u + n)$
 ⟨proof⟩

lemma *mset-subset-add-iff1*:
 $j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subset\# n)$
 ⟨proof⟩

lemma *mset-subset-add-iff2*:
 $i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{repeat-mset } j \ u + n) = (m \subset\# \text{repeat-mset } (j-i) \ u + n)$
 ⟨proof⟩

⟨ML⟩

lemma *add-mset-replicate-mset-safe[cancelation-simproc-pre]*: $\langle \text{NO-MATCH } \{\#\} \ M \implies \text{add-mset } a \ M = \{\#a\# \} + M \rangle$
 ⟨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⟩

62.4.2 Conditionally complete lattice

instantiation *multiset* :: (type) Inf
begin

lift-definition *Inf-multiset* :: 'a multiset set $\Rightarrow$ 'a multiset **is**
 $\lambda A \ i. \text{ if } A = \{\} \text{ then } 0 \text{ else } \text{Inf } ((\lambda f. f \ i) \text{ ' } A)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *Inf-multiset-empty*: $\text{Inf } \{\} = \{\# \}$
 $\langle \text{proof} \rangle$

lemma *count-Inf-multiset-nonempty*: $A \neq \{\} \implies \text{count } (\text{Inf } A) \ x = \text{Inf } ((\lambda X. \text{count } X \ x) \text{ ' } A)$
 $\langle \text{proof} \rangle$

instantiation *multiset* :: (type) Sup
begin

definition *Sup-multiset* :: 'a multiset set $\Rightarrow$ 'a multiset **where**
 $\text{Sup-multiset } A = (\text{if } A \neq \{\} \wedge \text{subset-mset.bdd-above } A \text{ then}$
 $\text{Abs-multiset } (\lambda i. \text{Sup } ((\lambda X. \text{count } X \ i) \text{ ' } A)) \text{ else } \{\# \})$

lemma *Sup-multiset-empty*: $\text{Sup } \{\} = \{\# \}$
 $\langle \text{proof} \rangle$

lemma *Sup-multiset-unbounded*: $\neg \text{subset-mset.bdd-above } A \implies \text{Sup } A = \{\# \}$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *bdd-above-multiset-imp-bdd-above-count*:
assumes *subset-mset.bdd-above* ($A :: \text{'a multiset set}$)
shows *bdd-above* $((\lambda X. \text{count } X \ x) \text{ ' } A)$
 $\langle \text{proof} \rangle$

lemma *bdd-above-multiset-imp-finite-support*:
assumes $A \neq \{\}$ *subset-mset.bdd-above* ($A :: \text{'a multiset set}$)
shows *finite* $(\bigcup X \in A. \{x. \text{count } X \ x > 0\})$
 $\langle \text{proof} \rangle$

lemma *Sup-multiset-in-multiset*:
 $\langle \text{finite } \{i. 0 < (\text{SUP } M \in A. \text{count } M \ i)\} \rangle$

if $\langle A \neq \{\} \rangle \langle \text{subset-mset.bdd-above } A \rangle$
 $\langle \text{proof} \rangle$

lemma *count-Sup-multiset-nonempty*:
 $\langle \text{count } (\text{Sup } A) \ x = (\text{SUP } X \in A. \text{count } X \ x) \rangle$
if $\langle A \neq \{\} \rangle \langle \text{subset-mset.bdd-above } A \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset: conditionally-complete-lattice* $\text{Inf Sup } (\cap\#) (\subseteq\#) (\subset\#)$
 $(\cup\#)$
 $\langle \text{proof} \rangle$

lemma *set-mset-Inf*:
assumes $A \neq \{\}$
shows $\text{set-mset } (\text{Inf } A) = (\bigcap X \in A. \text{set-mset } X)$
 $\langle \text{proof} \rangle$

lemma *in-Inf-multiset-iff*:
assumes $A \neq \{\}$
shows $x \in\# \text{Inf } A \longleftrightarrow (\forall X \in A. x \in\# X)$
 $\langle \text{proof} \rangle$

lemma *in-Inf-multisetD*: $x \in\# \text{Inf } A \implies X \in A \implies x \in\# X$
 $\langle \text{proof} \rangle$

lemma *set-mset-Sup*:
assumes $\text{subset-mset.bdd-above } A$
shows $\text{set-mset } (\text{Sup } A) = (\bigcup X \in A. \text{set-mset } X)$
 $\langle \text{proof} \rangle$

lemma *in-Sup-multiset-iff*:
assumes $\text{subset-mset.bdd-above } A$
shows $x \in\# \text{Sup } A \longleftrightarrow (\exists X \in A. x \in\# X)$
 $\langle \text{proof} \rangle$

lemma *in-Sup-multisetD*:
assumes $x \in\# \text{Sup } A$
shows $\exists X \in A. x \in\# X$
 $\langle \text{proof} \rangle$

interpretation *subset-mset: distrib-lattice* $(\cap\#) (\subseteq\#) (\subset\#) (\cup\#)$
 $\langle \text{proof} \rangle$

62.4.3 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$
is $\lambda P M. \lambda x. \text{if } P \ x \text{ then } M \ x \text{ else } 0$
 $\langle \text{proof} \rangle$

syntax (*ASCII*)

$-MCollect :: pttrn \Rightarrow 'a\ multiset \Rightarrow bool \Rightarrow 'a\ multiset \quad ((1\{\#- : \# \ - / \ -\# \}))$

syntax

$-MCollect :: pttrn \Rightarrow 'a\ multiset \Rightarrow bool \Rightarrow 'a\ multiset \quad ((1\{\#- \in \# \ - / \ -\# \}))$

translations

$\{\#x \in \# \ M. \ P\# \} == CONST\ filter\text{-}mset\ (\lambda x. \ P)\ M$

lemma *count-filter-mset* [simp]:

$count\ (filter\text{-}mset\ P\ M)\ a = (if\ P\ a\ then\ count\ M\ a\ else\ 0)$

<proof>

lemma *set-mset-filter* [simp]:

$set\text{-}mset\ (filter\text{-}mset\ P\ M) = \{a \in set\text{-}mset\ M. \ P\ a\}$

<proof>

lemma *filter-empty-mset* [simp]: $filter\text{-}mset\ P\ \{\#\} = \{\#\}$

<proof>

lemma *filter-single-mset*: $filter\text{-}mset\ P\ \{\#x\# \} = (if\ P\ x\ then\ \{\#x\# \} \ else\ \{\#\})$

<proof>

lemma *filter-union-mset* [simp]: $filter\text{-}mset\ P\ (M + N) = filter\text{-}mset\ P\ M + filter\text{-}mset\ P\ N$

<proof>

lemma *filter-diff-mset* [simp]: $filter\text{-}mset\ P\ (M - N) = filter\text{-}mset\ P\ M - filter\text{-}mset\ P\ N$

<proof>

lemma *filter-inter-mset* [simp]: $filter\text{-}mset\ P\ (M \cap \# \ N) = filter\text{-}mset\ P\ M \cap \# \ filter\text{-}mset\ P\ N$

<proof>

lemma *filter-sup-mset*[simp]: $filter\text{-}mset\ P\ (A \cup \# \ B) = filter\text{-}mset\ P\ A \cup \# \ filter\text{-}mset\ P\ B$

<proof>

lemma *filter-mset-add-mset* [simp]:

$filter\text{-}mset\ P\ (add\text{-}mset\ x\ A) =$
 $(if\ P\ x\ then\ add\text{-}mset\ x\ (filter\text{-}mset\ P\ A) \ else\ filter\text{-}mset\ P\ A)$

<proof>

lemma *multiset-filter-subset*[simp]: $filter\text{-}mset\ f\ M \subseteq \# \ M$

<proof>

lemma *multiset-filter-mono*:

assumes $A \subseteq \# \ B$

shows $filter\text{-}mset\ f\ A \subseteq \# \ filter\text{-}mset\ f\ B$

⟨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-filter-mset*: *filter-mset* $P\ (\text{filter-mset}\ Q\ M) = \{\#x \in\# M. Q\ x \wedge P\ x\}$

⟨proof⟩

lemma

filter-mset-True[simp]: $\{\#y \in\# M. \text{True}\#\} = M$ **and**

filter-mset-False[simp]: $\{\#y \in\# M. \text{False}\#\} = \{\#\}$

⟨proof⟩

62.4.4 Size

definition *wcount* **where** *wcount* $f\ M = (\lambda x. \text{count}\ M\ x * \text{Suc}\ (f\ x))$

lemma *wcount-union*: *wcount* $f\ (M + N)\ a = \text{wcount}\ f\ M\ a + \text{wcount}\ f\ N\ a$
 ⟨proof⟩

lemma *wcount-add-mset*:

wcount $f\ (\text{add-mset}\ x\ M)\ a = (\text{if}\ x = a\ \text{then}\ \text{Suc}\ (f\ a)\ \text{else}\ 0) + \text{wcount}\ f\ M\ a$

⟨proof⟩

definition *size-multiset* :: $('a \Rightarrow \text{nat}) \Rightarrow 'a\ \text{multiset} \Rightarrow \text{nat}$ **where**

size-multiset $f\ M = \text{sum}\ (\text{wcount}\ f\ M)\ (\text{set-mset}\ M)$

lemmas *size-multiset-eq* = *size-multiset-def*[*unfolded wcount-def*]

instantiation *multiset* :: $(\text{type})\ \text{size}$

begin

definition *size-multiset* **where**

size-multiset-overloaded-def: *size-multiset* = *Multiset.size-multiset* $(\lambda\_. 0)$

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 $\{\#b\# \} = \text{Suc } (f\ b)$
 ⟨proof⟩

lemma *size-single*: *size* $\{\#b\# \} = 1$
 ⟨proof⟩

lemma *sum-wcount-Int*:
 $\text{finite } A \implies \text{sum } (\text{wcount } f\ N) (A \cap \text{set-mset } N) = \text{sum } (\text{wcount } f\ N) A$
 ⟨proof⟩

lemma *size-multiset-union* [simp]:
 $\text{size-multiset } f\ (M + N :: 'a \text{ multiset}) = \text{size-multiset } f\ M + \text{size-multiset } f\ N$
 ⟨proof⟩

lemma *size-multiset-add-mset* [simp]:
 $\text{size-multiset } f\ (\text{add-mset } a\ M) = \text{Suc } (f\ a) + \text{size-multiset } f\ M$
 ⟨proof⟩

lemma *size-add-mset* [simp]: $\text{size } (\text{add-mset } a\ A) = \text{Suc } (\text{size } A)$
 ⟨proof⟩

lemma *size-union* [simp]: $\text{size } (M + N :: 'a \text{ multiset}) = \text{size } M + \text{size } N$
 ⟨proof⟩

lemma *size-multiset-eq-0-iff-empty* [iff]:
 $\text{size-multiset } f\ M = 0 \longleftrightarrow M = \{\#\}$
 ⟨proof⟩

lemma *size-eq-0-iff-empty* [iff]: $(\text{size } M = 0) = (M = \{\#\})$
 ⟨proof⟩

lemma *nonempty-has-size*: $(S \neq \{\#\}) = (0 < \text{size } S)$
 ⟨proof⟩

lemma *size-eq-Suc-imp-elem*: $\text{size } M = \text{Suc } n \implies \exists a. a \in\# M$
 ⟨proof⟩

lemma *size-eq-Suc-imp-eq-union*:
 assumes $\text{size } M = \text{Suc } n$
 shows $\exists a\ N. M = \text{add-mset } a\ N$
 ⟨proof⟩

lemma *size-mset-mono*:
 fixes $A\ B :: 'a \text{ multiset}$
 assumes $A \subseteq\# B$
 shows $\text{size } A \leq \text{size } B$
 ⟨proof⟩

lemma *size-filter-mset-lesseq[simp]*: $\text{size } (\text{filter-mset } f \ M) \leq \text{size } M$
 $\langle \text{proof} \rangle$

lemma *size-Diff-submset*:
 $M \subseteq\# M' \implies \text{size } (M' - M) = \text{size } M' - \text{size}(M::'a \text{ multiset})$
 $\langle \text{proof} \rangle$

62.5 Induction and case splits

theorem *multiset-induct* [*case-names empty add, induct type: multiset*]:
assumes *empty*: $P \ \{\#\}$
assumes *add*: $\bigwedge x \ M. P \ M \implies P \ (\text{add-mset } x \ M)$
shows $P \ M$
 $\langle \text{proof} \rangle$

lemma *multiset-induct-min* [*case-names empty add*]:
fixes $M :: 'a::\text{linorder multiset}$
assumes
 empty: $P \ \{\#\}$ **and**
 add: $\bigwedge x \ M. P \ M \implies (\forall y \in\# M. y \geq x) \implies P \ (\text{add-mset } x \ M)$
shows $P \ M$
 $\langle \text{proof} \rangle$

lemma *multiset-induct-max* [*case-names empty add*]:
fixes $M :: 'a::\text{linorder multiset}$
assumes
 empty: $P \ \{\#\}$ **and**
 add: $\bigwedge x \ M. P \ M \implies (\forall y \in\# M. y \leq x) \implies P \ (\text{add-mset } x \ M)$
shows $P \ M$
 $\langle \text{proof} \rangle$

lemma *multi-nonempty-split*: $M \neq \{\#\} \implies \exists A \ a. M = \text{add-mset } a \ A$
 $\langle \text{proof} \rangle$

lemma *multiset-cases* [*cases type*]:
obtains (*empty*) $M = \{\#\}$
 | (*add*) $x \ N$ **where** $M = \text{add-mset } x \ N$
 $\langle \text{proof} \rangle$

lemma *multi-drop-mem-not-eq*: $c \in\# B \implies B - \{\#c\# \} \neq B$
 $\langle \text{proof} \rangle$

lemma *union-filter-mset-complement[simp]*:
 $\forall x. P \ x = (\neg Q \ x) \implies \text{filter-mset } P \ M + \text{filter-mset } Q \ M = M$
 $\langle \text{proof} \rangle$

lemma *multiset-partition*: $M = \{\#x \in\# M. P \ x\# \} + \{\#x \in\# M. \neg P \ x\# \}$
 $\langle \text{proof} \rangle$

lemma *mset-subset-size*: $A \subset\# B \implies \text{size } A < \text{size } B$
 $\langle \text{proof} \rangle$

lemma *size-1-singleton-mset*: $\text{size } M = 1 \implies \exists a. M = \{\#a\#\}$
 $\langle \text{proof} \rangle$

62.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-subset-mset-rel*: $\text{wf } \{(M, N :: 'a \text{ multiset}). M \subset\# N\}$
 $\langle \text{proof} \rangle$

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B. \forall (A :: 'a \text{ multiset}). A \subset\# B \longrightarrow P A \implies P B$
shows $P B$
 $\langle \text{proof} \rangle$

lemma *multi-subset-induct* [*consumes 2, case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P \{\#\}$
and *insert*: $\bigwedge a F. a \in\# A \implies P F \implies P (\text{add-mset } a F)$
shows $P F$
 $\langle \text{proof} \rangle$

62.6 The fold combinator

definition *fold-mset* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ multiset} \Rightarrow 'b$
where
 $\text{fold-mset } f s M = \text{Finite-Set.fold } (\lambda x. f x \text{ } \sim\!\!\sim \text{ count } M x) s (\text{set-mset } M)$

lemma *fold-mset-empty* [*simp*]: $\text{fold-mset } f s \{\#\} = s$
 $\langle \text{proof} \rangle$

context *comp-fun-commute*
begin

lemma *fold-mset-add-mset* [*simp*]: $\text{fold-mset } f s (\text{add-mset } x M) = f x (\text{fold-mset } f s M)$
 $\langle \text{proof} \rangle$

corollary *fold-mset-single*: $\text{fold-mset } f s \{\#x\#\} = f x s$
 $\langle \text{proof} \rangle$

lemma *fold-mset-fun-left-comm*: $f x (\text{fold-mset } f s M) = \text{fold-mset } f (f x s) M$
 $\langle \text{proof} \rangle$

lemma *fold-mset-union* [*simp*]: $\text{fold-mset } f s (M + N) = \text{fold-mset } f (\text{fold-mset } f s M) N$
 $\langle \text{proof} \rangle$

lemma *fold-mset-fusion*:

assumes *comp-fun-commute* g

and $*$: $\bigwedge x y. h (g x y) = f x (h y)$

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.

62.7 Image

definition *image-mset* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ multiset} \Rightarrow 'b \text{ multiset}$ **where**

image-mset $f = fold-mset (add-mset \circ f) \{\#\}$

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* $(add-mset \circ f)$

<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 *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* $f M = \{\#\} \longleftrightarrow M = \{\#\}$

<proof>

lemma *image-mset-If*:

image-mset $(\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else } g \ x) \ A =$
image-mset $f \ (\text{filter-mset } P \ A) + \text{image-mset } g \ (\text{filter-mset } (\lambda x. \neg P \ x) \ A)$
 $\langle \text{proof} \rangle$

lemma *image-mset-Diff*:

assumes $B \subseteq\# A$
shows *image-mset* $f \ (A - B) = \text{image-mset } f \ A - \text{image-mset } f \ B$
 $\langle \text{proof} \rangle$

lemma *count-image-mset*:

$\langle \text{count } (\text{image-mset } f \ A) \ x = (\sum y \in f \text{ --} \{x\} \cap \text{set-mset } A. \text{count } A \ y) \rangle$
 $\langle \text{proof} \rangle$

lemma *count-image-mset'*:

$\langle \text{count } (\text{image-mset } f \ X) \ y = (\sum x \mid x \in\# X \wedge y = f \ x. \text{count } X \ x) \rangle$
 $\langle \text{proof} \rangle$

lemma *image-mset-subseteq-mono*: $A \subseteq\# B \implies \text{image-mset } f \ A \subseteq\# \text{image-mset } f \ B$

$\langle \text{proof} \rangle$

lemma *image-mset-subset-mono*: $M \subset\# N \implies \text{image-mset } f \ M \subset\# \text{image-mset } f \ N$

$\langle \text{proof} \rangle$

syntax (ASCII)

-comprehension-mset :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset } ((\{\#-/ . - : \# - \# \}))$

syntax

-comprehension-mset :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset } ((\{\#-/ . - \in \# - \# \}))$

translations

$\{\#e. x \in\# M \# \} \Rightarrow \text{CONST } \text{image-mset } (\lambda x. e) \ M$

syntax (ASCII)

-comprehension-mset' :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow \text{bool} \Rightarrow 'a \text{ multiset } ((\{\#-/ \mid - : \# - / - \# \}))$

syntax

-comprehension-mset' :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow \text{bool} \Rightarrow 'a \text{ multiset } ((\{\#-/ \mid - \in \# - / - \# \}))$

translations

$\{\#e \mid x \in\# M. P \# \} \mapsto \{\#e. x \in\# \{\# x \in\# M. P \# \} \# \}$

This allows to write not just filters like $\{\#x \in\# M. x < c \# \}$ but also images like $\{\#x + x. x \in\# M \# \}$ and $\{\#x + x \mid x \in\# M. x < c \# \}$, where the latter is currently displayed as $\{\#x + x. x \in\# \{\#x \in\# M. x < c \# \} \# \}$.

lemma *in-image-mset*: $y \in\# \{\#f \ x. x \in\# M \# \} \longleftrightarrow y \in f \text{ --} \text{set-mset } M$

$\langle \text{proof} \rangle$

functor *image-mset*: *image-mset*
 $\langle \text{proof} \rangle$

declare
image-mset.id [simp]
image-mset.identity [simp]

lemma *image-mset-id*[simp]: *image-mset id x = x*
 $\langle \text{proof} \rangle$

lemma *image-mset-cong*: $(\bigwedge x. x \in\# M \implies f x = g x) \implies \{\#f x. x \in\# M\# \} = \{\#g x. x \in\# M\# \}$
 $\langle \text{proof} \rangle$

lemma *image-mset-cong-pair*:
 $(\forall x y. (x, y) \in\# M \longrightarrow f x y = g x y) \implies \{\#f x y. (x, y) \in\# M\# \} = \{\#g x y. (x, y) \in\# M\# \}$
 $\langle \text{proof} \rangle$

lemma *image-mset-const-eq*:
 $\{\#c. a \in\# M\# \} = \text{replicate-mset (size M) c}$
 $\langle \text{proof} \rangle$

62.8 Further conversions

primrec *mset* :: 'a list $\Rightarrow$ 'a multiset **where**
mset [] = {#} |
mset (a # x) = *add-mset* a (*mset* x)

lemma *in-multiset-in-set*:
 $x \in\# \text{mset } xs \longleftrightarrow x \in \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *count-mset*:
 $\text{count (mset xs) } x = \text{length (filter } (\lambda y. x = y) \text{ xs)}$
 $\langle \text{proof} \rangle$

lemma *mset-zero-iff*[simp]: $(\text{mset } x = \{\#\}) = (x = [])$
 $\langle \text{proof} \rangle$

lemma *mset-zero-iff-right*[simp]: $(\{\#\} = \text{mset } x) = (x = [])$
 $\langle \text{proof} \rangle$

lemma *count-mset-gt-0*: $x \in \text{set } xs \implies \text{count (mset xs) } x > 0$
 $\langle \text{proof} \rangle$

lemma *count-mset-0-iff* [simp]: $\text{count (mset xs) } x = 0 \longleftrightarrow x \notin \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *mset-single-iff*[*iff*]: $mset\ xs = \{\#x\# \} \longleftrightarrow xs = [x]$
 ⟨*proof*⟩

lemma *mset-single-iff-right*[*iff*]: $\{\#x\# \} = mset\ xs \longleftrightarrow xs = [x]$
 ⟨*proof*⟩

lemma *set-mset-mset*[*simp*]: $set\text{-}mset\ (mset\ xs) = set\ xs$
 ⟨*proof*⟩

lemma *set-mset-comp-mset* [*simp*]: $set\text{-}mset \circ mset = set$
 ⟨*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*]: $mset\ (filter\ P\ xs) = \{\#x \in \# mset\ xs. P\ x\ \#\}$
 ⟨*proof*⟩

lemma *mset-rev* [*simp*]:
 $mset\ (rev\ xs) = mset\ xs$
 ⟨*proof*⟩

lemma *surj-mset*: *surj* *mset*
 ⟨*proof*⟩

lemma *distinct-count-atmost-1*:
 $distinct\ x = (\forall a. count\ (mset\ x)\ a = (if\ a \in set\ x\ then\ 1\ else\ 0))$
 ⟨*proof*⟩

lemma *mset-eq-setD*:
assumes $mset\ xs = mset\ ys$
shows $set\ xs = set\ ys$
 ⟨*proof*⟩

lemma *set-eq-iff-mset-eq-distinct*:
 $\langle distinct\ x \implies distinct\ y \implies set\ x = set\ y \longleftrightarrow mset\ x = mset\ y \rangle$
 ⟨*proof*⟩

lemma *set-eq-iff-mset-remdups-eq*:
 $\langle set\ x = set\ y \longleftrightarrow mset\ (remdups\ x) = mset\ (remdups\ y) \rangle$
 ⟨*proof*⟩

lemma *mset-eq-imp-distinct-iff*:
 $\langle distinct\ xs \longleftrightarrow distinct\ ys \rangle$ **if** $\langle mset\ xs = mset\ ys \rangle$
 ⟨*proof*⟩

lemma *nth-mem-mset*: $i < \text{length } ls \implies (ls ! i) \in \# \text{ mset } ls$
 ⟨proof⟩

lemma *mset-remove1[simp]*: $\text{mset } (\text{remove1 } a \text{ } xs) = \text{mset } xs - \{\#a\# \}$
 ⟨proof⟩

lemma *mset-eq-length*:
 assumes $\text{mset } xs = \text{mset } ys$
 shows $\text{length } xs = \text{length } ys$
 ⟨proof⟩

lemma *mset-eq-length-filter*:
 assumes $\text{mset } xs = \text{mset } ys$
 shows $\text{length } (\text{filter } (\lambda x. z = x) \text{ } xs) = \text{length } (\text{filter } (\lambda y. z = y) \text{ } ys)$
 ⟨proof⟩

lemma *fold-multiset-equiv*:
 ⟨ $\text{List.fold } f \text{ } xs = \text{List.fold } f \text{ } ys$ ⟩
 if f : $\langle \bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies f x \circ f y = f y \circ f x \rangle$
 and $\langle \text{mset } xs = \text{mset } ys \rangle$
 ⟨proof⟩

lemma *fold-permuted-eq*:
 ⟨ $\text{List.fold } (\odot) \text{ } xs \text{ } z = \text{List.fold } (\odot) \text{ } ys \text{ } z$ ⟩
 if $\langle \text{mset } xs = \text{mset } ys \rangle$
 and $\langle P \text{ } z \rangle$ and P : $\langle \bigwedge x z. x \in \text{set } xs \implies P z \implies P (x \odot z) \rangle$
 and f : $\langle \bigwedge x y z. x \in \text{set } xs \implies y \in \text{set } xs \implies P z \implies x \odot (y \odot z) = y \odot (x \odot z) \rangle$
 for f (infixl $\langle \odot \rangle$ 70)
 ⟨proof⟩

lemma *mset-shuffles*: $zs \in \text{shuffles } xs \text{ } ys \implies \text{mset } zs = \text{mset } xs + \text{mset } ys$
 ⟨proof⟩

lemma *mset-insort [simp]*: $\text{mset } (\text{insort } x \text{ } xs) = \text{add-mset } x \text{ } (\text{mset } xs)$
 ⟨proof⟩

lemma *mset-map[simp]*: $\text{mset } (\text{map } f \text{ } xs) = \text{image-mset } f \text{ } (\text{mset } xs)$
 ⟨proof⟩

global-interpretation *mset-set*: folding $\text{add-mset } \{\# \}$
 defines $\text{mset-set} = \text{folding-on.F } \text{add-mset } \{\# \}$
 ⟨proof⟩

lemma *sum-multiset-singleton [simp]*: $\text{sum } (\lambda n. \{\#n\# \}) \text{ } A = \text{mset-set } A$
 ⟨proof⟩

lemma *count-mset-set [simp]*:
 finite $A \implies x \in A \implies \text{count } (\text{mset-set } A) \text{ } x = 1$ (is PROP ?P)

$\neg \text{finite } A \implies \text{count } (\text{mset-set } A) \ x = 0 \text{ (is PROP ?Q)}$
 $x \notin A \implies \text{count } (\text{mset-set } A) \ x = 0 \text{ (is PROP ?R)}$
 <proof>

lemma *elem-mset-set*[simp, intro]: $\text{finite } A \implies x \in\# \text{ mset-set } A \longleftrightarrow x \in A$
 <proof>

lemma *mset-set-Union*:

$\text{finite } A \implies \text{finite } B \implies A \cap B = \{\} \implies \text{mset-set } (A \cup B) = \text{mset-set } A +$
 $\text{mset-set } B$
 <proof>

lemma *filter-mset-mset-set* [simp]:

$\text{finite } A \implies \text{filter-mset } P \ (\text{mset-set } A) = \text{mset-set } \{x \in A. P \ x\}$
 <proof>

lemma *mset-set-Diff*:

assumes $\text{finite } A \ B \subseteq A$
shows $\text{mset-set } (A - B) = \text{mset-set } A - \text{mset-set } B$
 <proof>

lemma *mset-set-set*: $\text{distinct } xs \implies \text{mset-set } (\text{set } xs) = \text{mset } xs$
 <proof>

lemma *count-mset-set'*: $\text{count } (\text{mset-set } A) \ x = (\text{if } \text{finite } A \wedge x \in A \text{ then } 1 \text{ else } 0)$
 <proof>

lemma *subset-imp-msubset-mset-set*:

assumes $A \subseteq B \ \text{finite } B$
shows $\text{mset-set } A \subseteq\# \text{ mset-set } B$
 <proof>

lemma *mset-set-set-mset-msubset*: $\text{mset-set } (\text{set-mset } A) \subseteq\# A$
 <proof>

lemma *mset-set-upto-eq-mset-upto*:

$\langle \text{mset-set } \{..<n\} = \text{mset } [0..<n] \rangle$
 <proof>

context *linorder*

begin

definition *sorted-list-of-multiset* :: $'a \text{ multiset} \Rightarrow 'a \text{ list}$

where

$\text{sorted-list-of-multiset } M = \text{fold-mset insert } [] \ M$

lemma *sorted-list-of-multiset-empty* [simp]:

$\text{sorted-list-of-multiset } \{\#\} = []$

$\langle proof \rangle$

lemma *sorted-list-of-multiset-singleton* [simp]:

sorted-list-of-multiset $\{\#x\# \} = [x]$

$\langle proof \rangle$

lemma *sorted-list-of-multiset-insert* [simp]:

sorted-list-of-multiset (add-mset x M) = *List.insort* x (*sorted-list-of-multiset* M)

$\langle proof \rangle$

end

lemma *mset-sorted-list-of-multiset*[simp]: *mset* (*sorted-list-of-multiset* M) = M

$\langle proof \rangle$

lemma *sorted-list-of-multiset-mset*[simp]: *sorted-list-of-multiset* (*mset* xs) = *sort* xs

$\langle proof \rangle$

lemma *finite-set-mset-mset-set*[simp]: *finite* $A \implies$ *set-mset* (*mset-set* A) = A

$\langle proof \rangle$

lemma *mset-set-empty-iff*: *mset-set* $A = \{\#\} \longleftrightarrow A = \{\} \vee$ *infinite* A

$\langle proof \rangle$

lemma *infinite-set-mset-mset-set*: $\neg$ *finite* $A \implies$ *set-mset* (*mset-set* A) = $\{\}$

$\langle proof \rangle$

lemma *set-sorted-list-of-multiset* [simp]:

set (*sorted-list-of-multiset* M) = *set-mset* M

$\langle proof \rangle$

lemma *sorted-list-of-mset-set* [simp]:

sorted-list-of-multiset (*mset-set* A) = *sorted-list-of-set* A

$\langle proof \rangle$

lemma *mset-upt* [simp]: *mset* $[m..<n] =$ *mset-set* $\{m..<n\}$

$\langle proof \rangle$

lemma *image-mset-map-of*:

distinct (*map fst* xs) $\implies \{\#the (map-of xs i). i \in \# mset (map fst xs)\# \} =$ *mset* (*map snd* xs)

$\langle proof \rangle$

lemma *msubset-mset-set-iff*[simp]:

assumes *finite* A *finite* B

shows *mset-set* $A \subseteq \# mset-set B \longleftrightarrow A \subseteq B$

$\langle proof \rangle$

lemma *mset-set-eq-iff*[simp]:
 assumes *finite A finite B*
 shows *mset-set A = mset-set B $\longleftrightarrow$ 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>

62.9 More properties of the replicate and repeat operations

lemma *in-replicate-mset*[simp]: *x $\in$ # replicate-mset n y $\longleftrightarrow$ n > 0 $\wedge$ 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 *count-le-replicate-mset-subset-eq*: *n $\leq$ count M x $\longleftrightarrow$ replicate-mset n x $\subseteq$ # M*
<proof>

lemma *filter-eq-replicate-mset*: *{#y $\in$ # D. y = x#} = replicate-mset (count D x) x*
<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 = {#} $\longleftrightarrow$ n = 0*
<proof>

lemma *replicate-mset-eq-iff*:
replicate-mset m a = replicate-mset n b $\longleftrightarrow$ m = 0 $\wedge$ n = 0 $\vee$ m = n $\wedge$ a = b
<proof>

lemma *repeat-mset-cancel1*: *repeat-mset a A = repeat-mset a B $\longleftrightarrow$ A = B $\vee$ a = 0*
<proof>

lemma *repeat-mset-cancel2*: *repeat-mset a A = repeat-mset b A $\longleftrightarrow$ a = b $\vee$ A = {#}*
<proof>

lemma *repeat-mset-eq-empty-iff*: $\text{repeat-mset } n \ A = \{\#\} \longleftrightarrow n = 0 \vee A = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *image-replicate-mset* [simp]:
 $\text{image-mset } f \ (\text{replicate-mset } n \ a) = \text{replicate-mset } n \ (f \ a)$
 $\langle \text{proof} \rangle$

lemma *replicate-mset-msubseteq-iff*:
 $\text{replicate-mset } m \ a \subseteq\# \text{ replicate-mset } n \ b \longleftrightarrow m = 0 \vee a = b \wedge m \leq n$
 $\langle \text{proof} \rangle$

lemma *msubseteq-replicate-msetE*:
 assumes $A \subseteq\# \text{ replicate-mset } n \ a$
 obtains m where $m \leq n$ and $A = \text{replicate-mset } m \ a$
 $\langle \text{proof} \rangle$

62.10 Big operators

locale *comm-monoid-mset* = *comm-monoid*
begin

interpretation *comp-fun-commute* f
 $\langle \text{proof} \rangle$

interpretation *comp?*: *comp-fun-commute* $f \circ g$
 $\langle \text{proof} \rangle$

context
begin

definition $F :: 'a \text{ multiset} \Rightarrow 'a$
 where *eq-fold*: $F \ M = \text{fold-mset } f \ \mathbf{1} \ M$

lemma *empty* [simp]: $F \ \{\#\} = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *singleton* [simp]: $F \ \{\#x\# \} = x$
 $\langle \text{proof} \rangle$

lemma *union* [simp]: $F \ (M + N) = F \ M * F \ N$
 $\langle \text{proof} \rangle$

lemma *add-mset* [simp]: $F \ (\text{add-mset } x \ N) = x * F \ N$
 $\langle \text{proof} \rangle$

lemma *insert* [simp]:
 shows $F \ (\text{image-mset } g \ (\text{add-mset } x \ A)) = g \ x * F \ (\text{image-mset } g \ A)$
 $\langle \text{proof} \rangle$

lemma *remove*:

assumes $x \in\# A$

shows $F A = x * F (A - \{\#x\# \})$

$\langle proof \rangle$

lemma *neutral*:

$\forall x \in\# A. x = \mathbf{1} \implies F A = \mathbf{1}$

$\langle proof \rangle$

lemma *neutral-const [simp]*:

$F (image-mset (\lambda\_. \mathbf{1}) A) = \mathbf{1}$

$\langle proof \rangle$ **lemma** *F-image-mset-product*:

$F \{\#g\ x\ j * F \{\#g\ i\ j. i \in\# A\#\}. j \in\# B\#\} =$

$F (image-mset (g\ x)\ B) * F \{\#F \{\#g\ i\ j. i \in\# A\#\}. j \in\# B\#\}$

$\langle proof \rangle$

lemma *swap*:

$F (image-mset (\lambda i. F (image-mset (g\ i)\ B))\ A) =$

$F (image-mset (\lambda j. F (image-mset (\lambda i. g\ i\ j)\ A))\ B)$

$\langle proof \rangle$

lemma *distrib*: $F (image-mset (\lambda x. g\ x * h\ x)\ A) = F (image-mset\ g\ A) * F (image-mset\ h\ A)$

$\langle proof \rangle$

lemma *union-disjoint*:

$A \cap\# B = \{\#\} \implies F (image-mset\ g\ (A \cup\# B)) = F (image-mset\ g\ A) * F$

$(image-mset\ g\ B)$

$\langle proof \rangle$

end

end

lemma *comp-fun-commute-plus-mset[simp]*: $comp-fun-commute ((+) :: 'a\ multiset \Rightarrow - \Rightarrow -)$

$\langle proof \rangle$

declare *comp-fun-commute.fold-mset-add-mset*[OF *comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff[iff]*: $x \in\# fold-mset\ (+)\ M\ NN \longleftrightarrow x \in\# M \vee (\exists N. N \in\# NN \wedge x \in\# N)$

$\langle proof \rangle$

context *comm-monoid-add*

begin

sublocale *sum-mset*: *comm-monoid-mset plus 0*

defines $sum-mset = sum-mset.F\ \langle proof \rangle$

lemma *sum-unfold-sum-mset*:

$sum\ f\ A = sum\text{-}mset\ (image\text{-}mset\ f\ (mset\text{-}set\ A))$
 $\langle proof \rangle$

end

notation *sum-mset* $(\sum \#)$

syntax (*ASCII*)

-sum-mset-image :: *pttrn* $\Rightarrow$ 'b *set* $\Rightarrow$ 'a $\Rightarrow$ 'a::*comm-monoid-add* ((3*SUM* -:#-.
 -) [0, 51, 10] 10)

syntax

-sum-mset-image :: *pttrn* $\Rightarrow$ 'b *set* $\Rightarrow$ 'a $\Rightarrow$ 'a::*comm-monoid-add* ((3 $\sum$ - $\in$ #-. -)
 [0, 51, 10] 10)

translations

$\sum i \in \# A. b \equiv CONST\ sum\text{-}mset\ (CONST\ image\text{-}mset\ (\lambda i. b)\ A)$

context *comm-monoid-add*

begin

lemma *sum-mset-sum-list*:

$sum\text{-}mset\ (mset\ xs) = sum\text{-}list\ xs$
 $\langle proof \rangle$

end

context *canonically-ordered-monoid-add*

begin

lemma *sum-mset-0-iff* [*simp*]:

$sum\text{-}mset\ M = 0 \iff (\forall x \in set\text{-}mset\ M. x = 0)$
 $\langle proof \rangle$

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 $\bigwedge i. i \in \# K \implies f\ i \leq g\ i$
 $\langle proof \rangle$

end

context *cancel-comm-monoid-add*

begin

lemma *sum-mset-diff*:

sum-mset ($M - N$) = *sum-mset* M - *sum-mset* N if $N \subseteq\# M$ for $M N :: 'a$
multiset
 ⟨*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 $f :: 'a::\{comm-monoid-add,times\} \Rightarrow 'b::semiring-0$
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* ($\lambda x. \text{if } x = y \text{ then } c \text{ else } 0$) A) = $c * \text{of-nat } (\text{count } A y)$
 ⟨*proof*⟩

lemma *sum-mset-delta'*:

sum-mset (*image-mset* ($\lambda x. \text{if } y = x \text{ then } c \text{ else } 0$) A) = $c * \text{of-nat } (\text{count } A y)$
 ⟨*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*]:
 $\text{size } (\text{mset-set } A) = \text{card } A$
 $\langle \text{proof} \rangle$

lemma *sum-mset-constant* [*simp*]:
fixes $y :: 'b::\text{semiring-1}$
shows $\langle (\sum x \in \#A. y) = \text{of-nat } (\text{size } A) * y \rangle$
 $\langle \text{proof} \rangle$

lemma *set-mset-Union-mset* [*simp*]: $\text{set-mset } (\sum \# MM) = (\bigcup M \in \text{set-mset } MM. \text{set-mset } M)$
 $\langle \text{proof} \rangle$

lemma *in-Union-mset-iff* [*iff*]: $x \in \# \sum \# MM \longleftrightarrow (\exists M. M \in \# MM \wedge x \in \# M)$
 $\langle \text{proof} \rangle$

lemma *count-sum*:
 $\text{count } (\text{sum } f A) x = \text{sum } (\lambda a. \text{count } (f a) x) A$
 $\langle \text{proof} \rangle$

lemma *sum-eq-empty-iff*:
assumes *finite A*
shows $\text{sum } f A = \{\#\} \longleftrightarrow (\forall a \in A. f a = \{\#\})$
 $\langle \text{proof} \rangle$

lemma *Union-mset-empty-conv* [*simp*]: $\sum \# M = \{\#\} \longleftrightarrow (\forall i \in \# M. i = \{\#\})$
 $\langle \text{proof} \rangle$

lemma *Union-image-single-mset* [*simp*]: $\sum \# (\text{image-mset } (\lambda x. \{\#x\# \}) m) = m$
 $\langle \text{proof} \rangle$

context *comm-monoid-mult*
begin

sublocale *prod-mset*: *comm-monoid-mset times 1*
defines $\text{prod-mset} = \text{prod-mset}.F$ $\langle \text{proof} \rangle$

lemma *prod-mset-empty*:
 $\text{prod-mset } \{\#\} = 1$
 $\langle \text{proof} \rangle$

lemma *prod-mset-singleton*:
 $\text{prod-mset } \{\#x\# \} = x$
 $\langle \text{proof} \rangle$

lemma *prod-mset-Un*:
 $\text{prod-mset } (A + B) = \text{prod-mset } A * \text{prod-mset } B$

$\langle \text{proof} \rangle$

lemma *prod-mset-prod-list*:

$\text{prod-mset} (\text{mset } xs) = \text{prod-list } xs$

$\langle \text{proof} \rangle$

lemma *prod-mset-replicate-mset* [simp]:

$\text{prod-mset} (\text{replicate-mset } n \ a) = a \wedge n$

$\langle \text{proof} \rangle$

lemma *prod-unfold-prod-mset*:

$\text{prod } f \ A = \text{prod-mset} (\text{image-mset } f \ (\text{mset-set } A))$

$\langle \text{proof} \rangle$

lemma *prod-mset-multiplicity*:

$\text{prod-mset } M = \text{prod} (\lambda x. x \wedge \text{count } M \ x) \ (\text{set-mset } M)$

$\langle \text{proof} \rangle$

lemma *prod-mset-delta*: $\text{prod-mset} (\text{image-mset} (\lambda x. \text{if } x = y \text{ then } c \text{ else } 1) \ A) =$

$c \wedge \text{count } A \ y$

$\langle \text{proof} \rangle$

lemma *prod-mset-delta'*: $\text{prod-mset} (\text{image-mset} (\lambda x. \text{if } y = x \text{ then } c \text{ else } 1) \ A) =$

$c \wedge \text{count } A \ y$

$\langle \text{proof} \rangle$

lemma *prod-mset-subset-imp-dvd*:

assumes $A \subseteq\# B$

shows $\text{prod-mset } A \ \text{dvd} \ \text{prod-mset } B$

$\langle \text{proof} \rangle$

lemma *dvd-prod-mset*:

assumes $x \in\# A$

shows $x \ \text{dvd} \ \text{prod-mset } A$

$\langle \text{proof} \rangle$

end

notation *prod-mset* $(\prod\#)$

syntax (ASCII)

-prod-mset-image :: *pttrn* $\Rightarrow$ 'b *set* $\Rightarrow$ 'a $\Rightarrow$ 'a::comm-monoid-mult ((3PROD
-:#-. -) [0, 51, 10] 10)

syntax

-prod-mset-image :: *pttrn* $\Rightarrow$ 'b *set* $\Rightarrow$ 'a $\Rightarrow$ 'a::comm-monoid-mult ((3 $\prod\#$ -:#-.
-) [0, 51, 10] 10)

translations

$\prod\# i \in\# A. b \Rightarrow \text{CONST } \text{prod-mset} (\text{CONST } \text{image-mset} (\lambda i. b) \ A)$

lemma *prod-mset-constant* [simp]: $(\prod - \in \# A. c) = c \wedge \text{size } A$
 <proof>

lemma (in *semidom*) *prod-mset-zero-iff* [iff]:
 $\text{prod-mset } A = 0 \longleftrightarrow 0 \in \# A$
 <proof>

lemma (in *semidom-divide*) *prod-mset-diff*:
 assumes $B \subseteq \# A$ and $0 \notin \# B$
 shows $\text{prod-mset } (A - B) = \text{prod-mset } A \text{ div prod-mset } B$
 <proof>

lemma (in *semidom-divide*) *prod-mset-minus*:
 assumes $a \in \# A$ and $a \neq 0$
 shows $\text{prod-mset } (A - \{\# a \# \}) = \text{prod-mset } A \text{ div } a$
 <proof>

lemma (in *normalization-semidom*) *normalize-prod-mset-normalize*:
 $\text{normalize } (\text{prod-mset } (\text{image-mset normalize } A)) = \text{normalize } (\text{prod-mset } A)$
 <proof>

lemma (in *algebraic-semidom*) *is-unit-prod-mset-iff*:
 $\text{is-unit } (\text{prod-mset } A) \longleftrightarrow (\forall x \in \# A. \text{is-unit } x)$
 <proof>

lemma (in *normalization-semidom-multiplicative*) *normalize-prod-mset*:
 $\text{normalize } (\text{prod-mset } A) = \text{prod-mset } (\text{image-mset normalize } A)$
 <proof>

lemma (in *normalization-semidom-multiplicative*) *normalized-prod-msetI*:
 assumes $\bigwedge a. a \in \# A \implies \text{normalize } a = a$
 shows $\text{normalize } (\text{prod-mset } A) = \text{prod-mset } A$
 <proof>

62.11 Multiset as order-ignorant lists

context *linorder*
begin

lemma *mset-insort* [simp]:
 $\text{mset } (\text{insort-key } k \ x \ xs) = \text{add-mset } x \ (\text{mset } xs)$
 <proof>

lemma *mset-sort* [simp]:
 $\text{mset } (\text{sort-key } k \ xs) = \text{mset } xs$
 <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 $\bigwedge k. k \in set\ ys \implies filter\ (\lambda x. f\ k = f\ x)\ ys = filter\ (\lambda x. f\ k = f\ x)\ xs$
and $sorted\ (map\ f\ ys)$
shows $sort\text{-}key\ f\ xs = ys$
 $\langle proof \rangle$

lemma *properties-for-sort*:
assumes *multiset*: $mset\ ys = mset\ xs$
and $sorted\ ys$
shows $sort\ xs = ys$
 $\langle proof \rangle$

lemma *sort-key-inj-key-eq*:
assumes *mset-equal*: $mset\ xs = mset\ ys$
and $inj\text{-}on\ f\ (set\ xs)$
and $sorted\ (map\ f\ ys)$
shows $sort\text{-}key\ f\ xs = ys$
 $\langle proof \rangle$

lemma *sort-key-eq-sort-key*:
assumes $mset\ xs = mset\ ys$
and $inj\text{-}on\ f\ (set\ xs)$
shows $sort\text{-}key\ f\ xs = sort\text{-}key\ f\ ys$
 $\langle proof \rangle$

lemma *sort-key-by-quicksort*:
 $sort\text{-}key\ f\ xs = sort\text{-}key\ f\ [x \leftarrow xs. f\ x < f\ (xs\ !\ (length\ xs\ div\ 2))]$
 $\quad @\ [x \leftarrow xs. f\ x = f\ (xs\ !\ (length\ xs\ div\ 2))]$
 $\quad @\ sort\text{-}key\ f\ [x \leftarrow xs. f\ x > f\ (xs\ !\ (length\ xs\ div\ 2))]\ (is\ sort\text{-}key\ f\ ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *sort-by-quicksort*:
 $sort\ xs = sort\ [x \leftarrow xs. x < xs\ !\ (length\ xs\ div\ 2)]$
 $\quad @\ [x \leftarrow xs. x = xs\ !\ (length\ xs\ div\ 2)]$
 $\quad @\ sort\ [x \leftarrow xs. x > xs\ !\ (length\ xs\ div\ 2)]\ (is\ sort\ ?lhs = ?rhs)$
 $\langle proof \rangle$

A stable parameterized quicksort

definition *part* :: $('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'b\ list \times 'b\ list \times 'b\ list$ **where**
 $part\ f\ pivot\ xs = ([x \leftarrow xs. f\ x < pivot], [x \leftarrow xs. f\ x = pivot], [x \leftarrow xs. pivot < f\ x])$

lemma *part-code* [code]:
 $part\ f\ pivot\ [] = ([], [], [])$
 $part\ f\ pivot\ (x \# xs) = (let\ (lts, eqs, gts) = part\ f\ pivot\ xs; x' = f\ x\ in$
 $\quad if\ x' < pivot\ then\ (x \# lts, eqs, gts)$
 $\quad else\ if\ x' > pivot\ then\ (lts, eqs, x \# gts)$
 $\quad else\ (lts, x \# eqs, gts))$
 $\langle proof \rangle$

lemma *sort-key-by-quicksort-code* [code]:
 $\text{sort-key } f \text{ } xs =$
 $(\text{case } xs \text{ of}$
 $\quad [] \Rightarrow []$
 $\quad | [x] \Rightarrow xs$
 $\quad | [x, y] \Rightarrow (\text{if } f \ x \leq f \ y \text{ then } xs \text{ else } [y, x])$
 $\quad | - \Rightarrow$
 $\quad \quad \text{let } (lts, eqs, gts) = \text{part } f \ (f \ (xs \ ! \ (\text{length } xs \ \text{div } 2))) \ xs$
 $\quad \quad \text{in } \text{sort-key } f \ lts \ @ \ eqs \ @ \ \text{sort-key } f \ gts)$
 $\langle \text{proof} \rangle$
end

hide-const (open) *part*

lemma *mset-remdups-subset-eq*: $mset \ (\text{remdups } xs) \subseteq\# \ mset \ xs$
 $\langle \text{proof} \rangle$

lemma *mset-update*:
 $i < \text{length } ls \implies mset \ (ls[i := v]) = \text{add-mset } v \ (mset \ ls - \{\#ls \ ! \ i\# \})$
 $\langle \text{proof} \rangle$

lemma *mset-swap*:
 $i < \text{length } ls \implies j < \text{length } ls \implies$
 $mset \ (ls[j := ls \ ! \ i, i := ls \ ! \ j]) = mset \ ls$
 $\langle \text{proof} \rangle$

lemma *mset-eq-finite*:
 $\langle \text{finite } \{ys. mset \ ys = mset \ xs\} \rangle$
 $\langle \text{proof} \rangle$

62.12 The multiset order

62.12.1 Well-foundedness

definition *mult1* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ multiset} \times 'a \text{ multiset}) \text{ set}$ **where**
 $\text{mult1 } r = \{(N, M). \exists a \ M0 \ K. M = \text{add-mset } a \ M0 \wedge N = M0 + K \wedge$
 $(\forall b. b \in\# K \longrightarrow (b, a) \in r)\}$

definition *mult* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ multiset} \times 'a \text{ multiset}) \text{ set}$ **where**
 $\text{mult } r = (\text{mult1 } r)^+$

lemma *mult1I*:
assumes $M = \text{add-mset } a \ M0$ **and** $N = M0 + K$ **and** $\bigwedge b. b \in\# K \implies (b, a) \in r$
shows $(N, M) \in \text{mult1 } r$
 $\langle \text{proof} \rangle$

lemma *mult1E*:

assumes $(N, M) \in \text{mult1 } r$
obtains $a \ M0 \ K$ **where** $M = \text{add-mset } a \ M0 \ N = M0 + K \wedge b. b \in \# K \implies$
 $(b, a) \in r$
 $\langle \text{proof} \rangle$

lemma *mono-mult1*:
assumes $r \subseteq r'$ **shows** $\text{mult1 } r \subseteq \text{mult1 } r'$
 $\langle \text{proof} \rangle$

lemma *mono-mult*:
assumes $r \subseteq r'$ **shows** $\text{mult } r \subseteq \text{mult } r'$
 $\langle \text{proof} \rangle$

lemma *not-less-empty [iff]*: $(M, \{\#\}) \notin \text{mult1 } r$
 $\langle \text{proof} \rangle$

lemma *less-add*:
assumes $\text{mult1}: (N, \text{add-mset } a \ M0) \in \text{mult1 } r$
shows
 $(\exists M. (M, M0) \in \text{mult1 } r \wedge N = \text{add-mset } a \ M) \vee$
 $(\exists K. (\forall b. b \in \# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
 $\langle \text{proof} \rangle$

lemma *all-accessible*:
assumes $\text{wf } r$
shows $\forall M. M \in \text{Wellfounded.acc } (\text{mult1 } r)$
 $\langle \text{proof} \rangle$

theorem *wf-mult1*: $\text{wf } r \implies \text{wf } (\text{mult1 } r)$
 $\langle \text{proof} \rangle$

theorem *wf-mult*: $\text{wf } r \implies \text{wf } (\text{mult } r)$
 $\langle \text{proof} \rangle$

62.12.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
assumes
 $\text{trans: trans } r$ **and**
 $MN: (M, N) \in \text{mult } r$
shows $\exists I \ J \ K. N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in \text{set-mset } K. \exists j$
 $\in \text{set-mset } J. (k, j) \in r)$
 $\langle \text{proof} \rangle$

lemma *one-step-implies-mult*:
assumes
 $J \neq \{\#\}$ **and**
 $\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J. (k, j) \in r$

shows $(I + K, I + J) \in \text{mult } r$
 $\langle \text{proof} \rangle$

lemma *subset-implies-mult*:
assumes *sub*: $A \subset\# B$
shows $(A, B) \in \text{mult } r$
 $\langle \text{proof} \rangle$

62.13 The multiset extension is cancellative for multiset union

lemma *mult-cancel*:
assumes *trans s* **and** *irrefl s*
shows $(X + Z, Y + Z) \in \text{mult } s \longleftrightarrow (X, Y) \in \text{mult } s$ (**is** ?L $\longleftrightarrow$?R)
 $\langle \text{proof} \rangle$

lemmas *mult-cancel-add-mset* =
 $\text{mult-cancel}[\text{of } - \text{ } \{ \# - \# \}, \text{unfolded union-mset-add-mset-right add.comm-neutral}]$

lemma *mult-cancel-max*:
assumes *trans s* **and** *irrefl s*
shows $(X, Y) \in \text{mult } s \longleftrightarrow (X - X \cap\# Y, Y - X \cap\# Y) \in \text{mult } s$ (**is** ?L $\longleftrightarrow$?R)
 $\langle \text{proof} \rangle$

62.14 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* and *multeqp*.

definition *multp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ **where**
 $\text{multp } P \ N \ M =$
 $(\text{let } Z = M \cap\# N; X = M - Z \text{ in}$
 $X \neq \{\#\} \wedge (\text{let } Y = N - Z \text{ in } (\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P \ y \ x)))$

definition *multeqp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ **where**
 $\text{multeqp } P \ N \ M =$
 $(\text{let } Z = M \cap\# N; X = M - Z; Y = N - Z \text{ in}$
 $(\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P \ y \ x))$

lemma *multp-iff*:
assumes *irrefl R* **and** *trans R* **and** [*simp*]: $\bigwedge x \ y. P \ x \ y \longleftrightarrow (x, y) \in R$
shows $\text{multp } P \ N \ M \longleftrightarrow (N, M) \in \text{mult } R$ (**is** ?L $\longleftrightarrow$?R)
 $\langle \text{proof} \rangle$

lemma *multeqp-iff*:
assumes *irrefl R* **and** *trans R* **and** $\bigwedge x \ y. P \ x \ y \longleftrightarrow (x, y) \in R$
shows $\text{multeqp } P \ N \ M \longleftrightarrow (N, M) \in (\text{mult } R)^=$
 $\langle \text{proof} \rangle$

62.14.1 Partial-order properties

lemma (in *preorder*) *mult1-lessE*:
 assumes $(N, M) \in \text{mult1 } \{(a, b). a < b\}$
 obtains $a \ M0 \ K$ **where** $M = \text{add-mset } a \ M0 \ N = M0 + K$
 $a \notin \# K \wedge b. b \in \# K \implies b < a$
 $\langle \text{proof} \rangle$

instantiation *multiset* :: (*preorder*) *order*
begin

definition *less-multiset* :: '*a multiset* $\Rightarrow$ '*a multiset* $\Rightarrow$ *bool*
where $M' < M \longleftrightarrow (M', M) \in \text{mult } \{(x', x). x' < x\}$

definition *less-eq-multiset* :: '*a multiset* $\Rightarrow$ '*a multiset* $\Rightarrow$ *bool*
where *less-eq-multiset* $M' \ M \longleftrightarrow M' < M \vee M' = M$

instance
 $\langle \text{proof} \rangle$

end

lemma *mset-le-irrefl* [*elim!*]:
fixes $M :: 'a::\text{preorder multiset}$
shows $M < M \implies R$
 $\langle \text{proof} \rangle$

62.14.2 Monotonicity of multiset union

lemma *mult1-union*: $(B, D) \in \text{mult1 } r \implies (C + B, C + D) \in \text{mult1 } r$
 $\langle \text{proof} \rangle$

lemma *union-le-mono2*: $B < D \implies C + B < C + (D::'a::\text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *union-le-mono1*: $B < D \implies B + C < D + (C::'a::\text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *union-less-mono*:
fixes $A \ B \ C \ D :: 'a::\text{preorder multiset}$
shows $A < C \implies B < D \implies A + B < C + D$
 $\langle \text{proof} \rangle$

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-add*
begin
instance
 $\langle \text{proof} \rangle$
end

62.14.3 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in \# XS \implies x \in \# \{\# y \# \} + XS$
and *multi-member-this*: $x \in \# \{\# x \# \} + XS$
and *multi-member-last*: $x \in \# \{\# x \# \}$
 $\langle \text{proof} \rangle$

definition *ms-strict* = *mult pair-less*

definition *ms-weak* = *ms-strict* $\cup$ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
 $\langle \text{proof} \rangle$

lemma *smsI*:
 $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z + B) \in \text{ms-strict}$
 $\langle \text{proof} \rangle$

lemma *wmsI*:
 $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee A = \{\#\} \wedge B = \{\#\}$
 $\implies (Z + A, Z + B) \in \text{ms-weak}$
 $\langle \text{proof} \rangle$

inductive *pw-leq*

where

pw-leq-empty: $\text{pw-leq } \{\#\} \{\#\}$
 $| \text{pw-leq-step}$: $\llbracket (x, y) \in \text{pair-leq}; \text{pw-leq } X Y \rrbracket \implies \text{pw-leq } (\{\#x\# \} + X) (\{\#y\# \} + Y)$

lemma *pw-leq-lstep*:
 $(x, y) \in \text{pair-leq} \implies \text{pw-leq } \{\#x\# \} \{\#y\# \}$
 $\langle \text{proof} \rangle$

lemma *pw-leq-split*:
assumes *pw-leq* $X Y$
shows $\exists A B Z. X = A + Z \wedge Y = B + Z \wedge ((\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$
 $\langle \text{proof} \rangle$

lemma
assumes *pwleq*: *pw-leq* $Z Z'$
shows *ms-strictI*: $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B) \in \text{ms-strict}$
and *ms-weakI1*: $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B) \in \text{ms-weak}$
and *ms-weakI2*: $(Z + \{\#\}, Z' + \{\#\}) \in \text{ms-weak}$
 $\langle \text{proof} \rangle$

lemma *empty-neutral*: $\{\#\} + x = x$ $x + \{\#\} = x$
and *nonempty-plus*: $\{\# x \# \} + rs \neq \{\#\}$
and *nonempty-single*: $\{\# x \# \} \neq \{\#\}$

$\langle proof \rangle$

$\langle ML \rangle$

62.15 Legacy theorem bindings

lemmas *multi-count-eq = multiset-eq-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a \text{ multiset})$
 $\langle proof \rangle$

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a \text{ multiset}))$
 $\langle proof \rangle$

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a \text{ multiset}))$
 $\langle proof \rangle$

lemmas *union-ac = union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a \text{ multiset})$
 $\langle proof \rangle$

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a \text{ multiset})$
 $\langle proof \rangle$

lemma *multi-union-self-other-eq*: $(A::'a \text{ multiset}) + X = A + Y \Longrightarrow X = Y$
 $\langle proof \rangle$

lemma *mset-subset-trans*: $(M::'a \text{ multiset}) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
 $\langle proof \rangle$

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
 $\langle proof \rangle$

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
 $\langle proof \rangle$

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
 $\langle proof \rangle$

lemmas *multiset-inter-ac =*
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg M < (M::'a::\text{preorder multiset})$
 $\langle proof \rangle$

lemma *mset-le-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::\text{preorder multiset})$

$\langle \text{proof} \rangle$

lemma *mset-le-not-sym*: $M < N \implies \neg N < (M::'a::\text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *mset-le-asym*: $M < N \implies (\neg P \implies N < (M::'a::\text{preorder multiset})) \implies P$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

62.16 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: $\{\#\} = \text{mset } []$
 $\langle \text{proof} \rangle$

lemma [*code*]: $\text{add-mset } x (\text{mset } xs) = \text{mset } (x \# xs)$
 $\langle \text{proof} \rangle$

lemma [*code*]: $\text{Multiset.is-empty } (\text{mset } xs) \longleftrightarrow \text{List.null } xs$
 $\langle \text{proof} \rangle$

lemma *union-code* [*code*]: $\text{mset } xs + \text{mset } ys = \text{mset } (xs @ ys)$
 $\langle \text{proof} \rangle$

lemma [*code*]: $\text{image-mset } f (\text{mset } xs) = \text{mset } (\text{map } f xs)$
 $\langle \text{proof} \rangle$

lemma [*code*]: $\text{filter-mset } f (\text{mset } xs) = \text{mset } (\text{filter } f xs)$
 $\langle \text{proof} \rangle$

lemma [*code*]: $\text{mset } xs - \text{mset } ys = \text{mset } (\text{fold } \text{remove1 } ys xs)$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\text{mset } xs \cap \# \text{mset } ys =$
 $\text{mset } (\text{snd } (\text{fold } (\lambda x (ys, zs).$
 $\quad \text{if } x \in \text{set } ys \text{ then } (\text{remove1 } x ys, x \# zs) \text{ else } (ys, zs)) xs (ys, []))$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\text{mset } xs \cup \# \text{mset } ys =$
 $\text{mset } (\text{case-prod } \text{append } (\text{fold } (\lambda x (ys, zs). (\text{remove1 } x ys, x \# zs)) xs (ys, [])))$
 $\langle \text{proof} \rangle$

declare *in-multiset-in-set* [*code-unfold*]

lemma [code]: $\text{count } (\text{mset } xs) \ x = \text{fold } (\lambda y. \text{if } x = y \text{ then } \text{Suc} \text{ else } \text{id}) \ xs \ 0$
 ⟨proof⟩

declare *set-mset-mset* [code]

declare *sorted-list-of-multiset-mset* [code]

lemma [code]: — not very efficient, but representation-ignorant!
 $\text{mset-set } A = \text{mset } (\text{sorted-list-of-set } A)$
 ⟨proof⟩

declare *size-mset* [code]

fun *subset-eq-mset-impl* :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ bool option **where**
 subset-eq-mset-impl [] *ys* = Some (*ys* $\neq$ [])
 | *subset-eq-mset-impl* (Cons *x xs*) *ys* = (case *List.extract* ((=) *x*) *ys* of
 None $\Rightarrow$ None
 | Some (*ys1*, -, *ys2*) $\Rightarrow$ *subset-eq-mset-impl xs (ys1 @ ys2)*)

lemma *subset-eq-mset-impl*: (*subset-eq-mset-impl xs ys* = None $\longleftrightarrow \neg \text{mset } xs \subseteq \# \text{mset } ys$) $\wedge$
 (*subset-eq-mset-impl xs ys* = Some True $\longleftrightarrow \text{mset } xs \subset \# \text{mset } ys$) $\wedge$
 (*subset-eq-mset-impl xs ys* = Some False $\longrightarrow \text{mset } xs = \text{mset } ys$)
 ⟨proof⟩

lemma [code]: $\text{mset } xs \subseteq \# \text{mset } ys \longleftrightarrow \text{subset-eq-mset-impl } xs \ ys \neq \text{None}$
 ⟨proof⟩

lemma [code]: $\text{mset } xs \subset \# \text{mset } ys \longleftrightarrow \text{subset-eq-mset-impl } xs \ ys = \text{Some True}$
 ⟨proof⟩

instantiation *multiset* :: (equal) equal
begin

definition

[code del]: $\text{HOL.equal } A \ (B :: 'a \text{ multiset}) \longleftrightarrow A = B$

lemma [code]: $\text{HOL.equal } (\text{mset } xs) \ (\text{mset } ys) \longleftrightarrow \text{subset-eq-mset-impl } xs \ ys = \text{Some False}$
 ⟨proof⟩

instance
 ⟨proof⟩

end

declare *sum-mset-sum-list* [code]

lemma [code]: $\text{prod-mset } (\text{mset } xs) = \text{fold times } xs \ 1$
 ⟨proof⟩

Exercise for the casual reader: add implementations for $(\leq)$ and $(<)$ (multiset order).

Quickcheck generators

context

includes *term-syntax*

begin

definition

msetify :: 'a::typerep list $\times$ (unit $\Rightarrow$ Code-Evaluation.term)

$\Rightarrow$ 'a multiset $\times$ (unit $\Rightarrow$ 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* $\circ\rightarrow$ ($\lambda xs.$ Pair (*msetify* *xs*))

instance $\langle proof \rangle$

end

end

instantiation *multiset* :: (full-exhaustive) full-exhaustive

begin

definition *full-exhaustive-multiset* :: ('a multiset $\times$ (unit $\Rightarrow$ term) $\Rightarrow$ (bool $\times$ term list) option) $\Rightarrow$ natural $\Rightarrow$ (bool $\times$ term list) option

where

full-exhaustive-multiset *f* *i* = *Quickcheck-Exhaustive.full-exhaustive* ($\lambda xs.$ *f* (*msetify* *xs*)) *i*

instance $\langle proof \rangle$

end

hide-const (open) *msetify*

62.17 BNF setup

definition *rel-mset* **where**

$rel\text{-}mset\ R\ X\ Y \longleftrightarrow (\exists xs\ ys. mset\ xs = X \wedge mset\ ys = Y \wedge list\text{-}all2\ R\ xs\ ys)$

lemma *mset-zip-take-Cons-drop-twice*:

assumes $length\ xs = length\ ys\ j \leq length\ xs$

shows $mset\ (zip\ (take\ j\ xs\ @\ x\ \# \ drop\ j\ xs)\ (take\ j\ ys\ @\ y\ \# \ drop\ j\ ys)) =$
 $add\text{-}mset\ (x,y)\ (mset\ (zip\ xs\ ys))$

<proof>

lemma *ex-mset-zip-left*:

assumes $length\ xs = length\ ys\ mset\ xs' = mset\ xs$

shows $\exists ys'. length\ ys' = length\ xs' \wedge mset\ (zip\ xs'\ ys') = mset\ (zip\ xs\ ys)$

<proof>

lemma *list-all2-reorder-left-invariance*:

assumes $rel: list\text{-}all2\ R\ xs\ ys$ **and** $ms\text{-}x: mset\ xs' = mset\ xs$

shows $\exists ys'. list\text{-}all2\ R\ xs'\ ys' \wedge mset\ ys' = mset\ ys$

<proof>

lemma *ex-mset*: $\exists xs. mset\ xs = X$

<proof>

inductive *pred-mset* :: $('a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow bool$

where

$pred\text{-}mset\ P\ \{\#\}$

| $\llbracket P\ a; pred\text{-}mset\ P\ M \rrbracket \Longrightarrow pred\text{-}mset\ P\ (add\text{-}mset\ a\ M)$

lemma *pred-mset-iff*: — TODO: alias for *Multiset.Ball*

$\langle pred\text{-}mset\ P\ M \longleftrightarrow Multiset.Ball\ M\ P \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)

<proof>

bnf *'a multiset*

map: *image-mset*

sets: *set-mset*

bd: *natLeq*

wits: $\{\#\}$

rel: *rel-mset*

pred: *pred-mset*

<proof>

inductive *rel-mset'* :: $\langle ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'b\ multiset \Rightarrow bool \rangle$

where

Zero[*intro*]: $rel\text{-}mset'\ R\ \{\#\}\ \{\#\}$

| *Plus*[*intro*]: $\llbracket R\ a\ b; rel\text{-}mset'\ R\ M\ N \rrbracket \Longrightarrow rel\text{-}mset'\ R\ (add\text{-}mset\ a\ M)\ (add\text{-}mset\ b\ N)$

lemma *rel-mset-Zero*: $rel\text{-}mset\ R\ \{\#\}\ \{\#\}$

<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* $R\ (add-mset\ a\ M)\ (add-mset\ b\ N)$
 $\langle proof \rangle$

lemma *rel-mset'-imp-rel-mset*: *rel-mset'* $R\ M\ N \implies rel-mset\ R\ M\ N$
 $\langle proof \rangle$

lemma *rel-mset-size*: *rel-mset* $R\ M\ N \implies size\ M = size\ N$
 $\langle proof \rangle$

lemma *rel-mset-Zero-iff* [*simp*]:
shows *rel-mset* $rel\ \{\#\}\ Y \longleftrightarrow Y = \{\#\}$ **and** *rel-mset* $rel\ X\ \{\#\} \longleftrightarrow X = \{\#\}$
 $\langle proof \rangle$

lemma *multiset-induct2*[*case-names empty addL addR*]:
assumes *empty*: $P\ \{\#\}\ \{\#\}$
and *addL*: $\bigwedge a\ M\ N. P\ M\ N \implies P\ (add-mset\ a\ M)\ N$
and *addR*: $\bigwedge a\ M\ N. P\ M\ N \implies P\ M\ (add-mset\ a\ N)$
shows $P\ M\ N$
 $\langle proof \rangle$

lemma *multiset-induct2-size*[*consumes 1, case-names empty add*]:
assumes *c*: $size\ M = size\ N$
and *empty*: $P\ \{\#\}\ \{\#\}$
and *add*: $\bigwedge a\ b\ M\ N\ a\ b. P\ M\ N \implies P\ (add-mset\ a\ M)\ (add-mset\ b\ N)$
shows $P\ M\ N$
 $\langle proof \rangle$

lemma *msed-map-invL*:
assumes *image-mset* $f\ (add-mset\ a\ M) = N$
shows $\exists N1. N = add-mset\ (f\ a)\ N1 \wedge image-mset\ f\ M = N1$
 $\langle proof \rangle$

lemma *msed-map-invR*:
assumes *image-mset* $f\ M = add-mset\ b\ N$
shows $\exists M1\ a. M = add-mset\ a\ M1 \wedge f\ a = b \wedge image-mset\ f\ M1 = N$
 $\langle proof \rangle$

lemma *msed-rel-invL*:
assumes *rel-mset* $R\ (add-mset\ a\ M)\ N$
shows $\exists N1\ b. N = add-mset\ b\ N1 \wedge R\ a\ b \wedge rel-mset\ R\ M\ N1$
 $\langle proof \rangle$

lemma *msed-rel-invR*:

assumes *rel-mset* $R\ M$ (*add-mset* $b\ N$)
shows $\exists M1\ a.\ M = \text{add-mset}\ a\ M1 \wedge R\ a\ b \wedge \text{rel-mset}\ R\ M1\ N$
 $\langle \text{proof} \rangle$

lemma *rel-mset-imp-rel-mset'*:
assumes *rel-mset* $R\ M\ N$
shows *rel-mset'* $R\ M\ N$
 $\langle \text{proof} \rangle$

lemma *rel-mset-rel-mset'*: *rel-mset* $R\ M\ N = \text{rel-mset}'\ R\ M\ N$
 $\langle \text{proof} \rangle$

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*]]

62.18 Size setup

lemma *size-multiset-o-map*: *size-multiset* $g \circ \text{image-mset}\ f = \text{size-multiset}\ (g \circ f)$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

62.19 Lemmas about Size

lemma *size-mset-SucE*: *size* $A = \text{Suc}\ n \implies (\bigwedge a\ B.\ A = \{\#a\# \} + B \implies \text{size}\ B = n \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *size-Suc-Diff1*: $x \in \#\ M \implies \text{Suc}\ (\text{size}\ (M - \{\#x\# \})) = \text{size}\ M$
 $\langle \text{proof} \rangle$

lemma *size-Diff-singleton*: $x \in \#\ M \implies \text{size}\ (M - \{\#x\# \}) = \text{size}\ M - 1$
 $\langle \text{proof} \rangle$

lemma *size-Diff-singleton-if*: *size* $(A - \{\#x\# \}) = (\text{if } x \in \#\ A \text{ then } \text{size}\ A - 1 \text{ else } \text{size}\ A)$
 $\langle \text{proof} \rangle$

lemma *size-Un-Int*: *size* $A + \text{size}\ B = \text{size}\ (A \cup \# \ B) + \text{size}\ (A \cap \# \ B)$
 $\langle \text{proof} \rangle$

lemma *size-Un-disjoint*: $A \cap \# \ B = \{\#\} \implies \text{size}\ (A \cup \# \ B) = \text{size}\ A + \text{size}\ B$
 $\langle \text{proof} \rangle$

lemma *size-Diff-subset-Int*: *size* $(M - M') = \text{size}\ M - \text{size}\ (M \cap \# \ M')$
 $\langle \text{proof} \rangle$

lemma *diff-size-le-size-Diff*: *size* $(M :: \text{- multiset}) - \text{size}\ M' \leq \text{size}\ (M - M')$
 $\langle \text{proof} \rangle$

lemma *size-Diff1-less*: $x \in \# M \implies \text{size } (M - \{\#x\}) < \text{size } M$
 ⟨proof⟩

lemma *size-Diff2-less*: $x \in \# M \implies y \in \# M \implies \text{size } (M - \{\#x\} - \{\#y\}) < \text{size } M$
 ⟨proof⟩

lemma *size-Diff1-le*: $\text{size } (M - \{\#x\}) \leq \text{size } M$
 ⟨proof⟩

lemma *size-psubset*: $M \subseteq \# M' \implies \text{size } M < \text{size } M' \implies M \subset \# M'$
 ⟨proof⟩

hide-const (**open**) *wcount*

end

63 More Theorems about the Multiset Order

theory *Multiset-Order*
imports *Multiset*
begin

63.1 Alternative Characterizations

context *preorder*
begin

lemma *order-mult*: *class.order*
 $(\lambda M N. (M, N) \in \text{mult } \{(x, y). x < y\} \vee M = N)$
 $(\lambda M N. (M, N) \in \text{mult } \{(x, y). x < y\})$
 (**is** *class.order* ?le ?less)
 ⟨proof⟩

The Dershowitz–Manna ordering:

definition *less-multiset_{DM}* **where**
 $\text{less-multiset}_{DM} M N \longleftrightarrow$
 $(\exists X Y. X \neq \{\#\} \wedge X \subseteq \# N \wedge M = (N - X) + Y \wedge (\forall k. k \in \# Y \longrightarrow (\exists a. a \in \# X \wedge k < a)))$

The Huet–Oppen ordering:

definition *less-multiset_{HO}* **where**
 $\text{less-multiset}_{HO} M N \longleftrightarrow M \neq N \wedge (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x. y < x \wedge \text{count } M x < \text{count } N x))$

lemma *mult-imp-less-multiset_{HO}*:
 $(M, N) \in \text{mult } \{(x, y). x < y\} \implies \text{less-multiset}_{HO} M N$
 ⟨proof⟩

lemma *less-multiset_{DM}-imp-mult*:

less-multiset_{DM} $M\ N \implies (M, N) \in \text{mult } \{(x, y). x < y\}$
 $\langle \text{proof} \rangle$

lemma *less-multiset_{HO}-imp-less-multiset_{DM}*: *less-multiset_{HO}* $M\ N \implies \text{less-multiset}_{DM}\ M\ N$
 $\langle \text{proof} \rangle$

lemma *mult-less-multiset_{DM}*: $(M, N) \in \text{mult } \{(x, y). x < y\} \longleftrightarrow \text{less-multiset}_{DM}\ M\ N$
 $\langle \text{proof} \rangle$

lemma *mult-less-multiset_{HO}*: $(M, N) \in \text{mult } \{(x, y). x < y\} \longleftrightarrow \text{less-multiset}_{HO}\ M\ N$
 $\langle \text{proof} \rangle$

lemmas $\text{mult}_{DM} = \text{mult-less-multiset}_{DM}[\text{unfolded less-multiset}_{DM}\text{-def}]$

lemmas $\text{mult}_{HO} = \text{mult-less-multiset}_{HO}[\text{unfolded less-multiset}_{HO}\text{-def}]$

end

lemma *less-multiset-less-multiset_{HO}*: $M < N \longleftrightarrow \text{less-multiset}_{HO}\ M\ N$
 $\langle \text{proof} \rangle$

lemmas $\text{less-multiset}_{DM} = \text{mult}_{DM}[\text{folded less-multiset-def}]$

lemmas $\text{less-multiset}_{HO} = \text{mult}_{HO}[\text{folded less-multiset-def}]$

lemma *subset-eq-imp-le-multiset*:

shows $M \subseteq\# N \implies M \leq N$

$\langle \text{proof} \rangle$

lemma *le-multiset-right-total*: $M < \text{add-mset } x\ M$
 $\langle \text{proof} \rangle$

lemma *less-eq-multiset-empty-left[simp]*:

shows $\{\#\} \leq M$

$\langle \text{proof} \rangle$

lemma *ex-gt-imp-less-multiset*: $(\exists y. y \in\# N \wedge (\forall x. x \in\# M \longrightarrow x < y)) \implies M < N$

$\langle \text{proof} \rangle$

lemma *less-eq-multiset-empty-right[simp]*: $M \neq \{\#\} \implies \neg M \leq \{\#\}$

$\langle \text{proof} \rangle$

lemma *le-multiset-empty-left[simp]*: $M \neq \{\#\} \implies \{\#\} < M$

$\langle \text{proof} \rangle$

lemma *le-multiset-empty-right[simp]*: $\neg M < \{\#\}$
 $\langle \text{proof} \rangle$

lemma *union-le-diff-plus*: $P \subseteq \# M \implies N < P \implies M - P + N < M$
 $\langle \text{proof} \rangle$

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-monoid-add-imp-le*
begin

lemma *less-eq-multiset_{HO}*:
 $M \leq N \iff (\forall y. \text{count } N \ y < \text{count } M \ y \longrightarrow (\exists x. y < x \wedge \text{count } M \ x < \text{count } N \ x))$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

lemma
fixes $M \ N :: 'a \text{ multiset}$
shows
less-eq-multiset-plus-left: $N \leq (M + N)$ **and**
less-eq-multiset-plus-right: $M \leq (M + N)$
 $\langle \text{proof} \rangle$

lemma
fixes $M \ N :: 'a \text{ multiset}$
shows
le-multiset-plus-left-nonempty: $M \neq \{\#\} \implies N < M + N$ **and**
le-multiset-plus-right-nonempty: $N \neq \{\#\} \implies M < M + N$
 $\langle \text{proof} \rangle$

end

lemma *all-lt-Max-imp-lt-mset*: $N \neq \{\#\} \implies (\forall a \in \# M. a < \text{Max } (\text{set-mset } N)) \implies M < N$
 $\langle \text{proof} \rangle$

lemma *lt-imp-ex-count-lt*: $M < N \implies \exists y. \text{count } M \ y < \text{count } N \ y$
 $\langle \text{proof} \rangle$

lemma *subset-imp-less-mset*: $A \subset \# B \implies A < B$
 $\langle \text{proof} \rangle$

lemma *image-mset-strict-mono*:
assumes
mono-f: $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f \ x < f \ y$ **and**

less: $M < N$
shows $\text{image-mset } f \ M < \text{image-mset } f \ N$
 $\langle \text{proof} \rangle$

lemma *image-mset-mono*:
assumes
 mono-f: $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f \ x < f \ y$ **and**
 less: $M \leq N$
shows $\text{image-mset } f \ M \leq \text{image-mset } f \ N$
 $\langle \text{proof} \rangle$

lemma *mset-lt-single-right-iff[simp]*: $M < \{\#y\# \} \longleftrightarrow (\forall x \in \# \ M. x < y)$ **for** $y :: 'a::\text{linorder}$
 $\langle \text{proof} \rangle$

lemma *mset-le-single-right-iff[simp]*:
 $M \leq \{\#y\# \} \longleftrightarrow M = \{\#y\# \} \vee (\forall x \in \# \ M. x < y)$ **for** $y :: 'a::\text{linorder}$
 $\langle \text{proof} \rangle$

63.2 Simprocs

lemma *mset-le-add-iff1*:
 $j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \leq \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \leq n)$
 $\langle \text{proof} \rangle$

lemma *mset-le-add-iff2*:
 $i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \leq \text{repeat-mset } j \ u + n) = (m \leq \text{repeat-mset } (j-i) \ u + n)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

63.3 Additional facts and instantiations

lemma *ex-gt-count-imp-le-multiset*:
 $(\forall y :: 'a :: \text{order}. y \in \# \ M + N \longrightarrow y \leq x) \implies \text{count } M \ x < \text{count } N \ x \implies M < N$
 $\langle \text{proof} \rangle$

lemma *mset-lt-single-iff[iff]*: $\{\#x\# \} < \{\#y\# \} \longleftrightarrow x < y$
 $\langle \text{proof} \rangle$

lemma *mset-le-single-iff[iff]*: $\{\#x\# \} \leq \{\#y\# \} \longleftrightarrow x \leq y$ **for** $x \ y :: 'a::\text{order}$
 $\langle \text{proof} \rangle$

instance *multiset* :: $(\text{linorder}) \ \text{linordered-cancel-ab-semigroup-add}$
 $\langle \text{proof} \rangle$

lemma *less-eq-multiset-total*:


```

fixes  $M\ N :: 'a :: \text{linorder multiset}$ 
shows  $\neg M \leq N \implies N \leq M$ 
 $\langle \text{proof} \rangle$ 

instantiation  $\text{multiset} :: (\text{wellorder}) \text{ wellorder}$ 
begin

lemma  $\text{wf-less-multiset}: \text{wf } \{(M :: 'a \text{ multiset}, N). M < N\}$ 
 $\langle \text{proof} \rangle$ 

instance  $\langle \text{proof} \rangle$ 

end

instantiation  $\text{multiset} :: (\text{preorder}) \text{ order-bot}$ 
begin

definition  $\text{bot-multiset} :: 'a \text{ multiset}$  where  $\text{bot-multiset} = \{\#\}$ 

instance  $\langle \text{proof} \rangle$ 

end

instance  $\text{multiset} :: (\text{preorder}) \text{ no-top}$ 
 $\langle \text{proof} \rangle$ 

instance  $\text{multiset} :: (\text{preorder}) \text{ ordered-cancel-comm-monoid-add}$ 
 $\langle \text{proof} \rangle$ 

instantiation  $\text{multiset} :: (\text{linorder}) \text{ distrib-lattice}$ 
begin

definition  $\text{inf-multiset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$  where
 $\text{inf-multiset } A\ B = (\text{if } A < B \text{ then } A \text{ else } B)$ 

definition  $\text{sup-multiset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$  where
 $\text{sup-multiset } A\ B = (\text{if } B > A \text{ then } B \text{ else } A)$ 

instance
 $\langle \text{proof} \rangle$ 

end

end

```

64 Non-negative, non-positive integers and reals

```

theory Nonpos-Ints
imports Complex-Main

```

begin

64.1 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** $\mathbb{Z}_{\leq 0} = \{of-int\ n \mid n. n \leq 0\}$

lemma *zero-in-nonpos-Ints* [*simp,intro*]: $0 \in \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *neg-one-in-nonpos-Ints* [*simp,intro*]: $-1 \in \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *neg-numeral-in-nonpos-Ints* [*simp,intro*]: $-numeral\ n \in \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *one-notin-nonpos-Ints* [*simp*]: $(1 :: 'a :: ring-char-0) \notin \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *numeral-notin-nonpos-Ints* [*simp*]: $(numeral\ n :: 'a :: ring-char-0) \notin \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *minus-of-nat-in-nonpos-Ints* [*simp, intro*]: $- of-nat\ n \in \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *of-nat-in-nonpos-Ints-iff*: $(of-nat\ n :: 'a :: \{ring-1, ring-char-0\}) \in \mathbb{Z}_{\leq 0} \iff n = 0$
 $\langle proof \rangle$

lemma *nonpos-Ints-of-int*: $n \leq 0 \implies of-int\ n \in \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *nonpos-IntsI*:
 $x \in \mathbb{Z} \implies x \leq 0 \implies (x :: 'a :: linordered-idom) \in \mathbb{Z}_{\leq 0}$
 $\langle proof \rangle$

lemma *nonpos-Ints-subset-Ints*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{Z}$
 $\langle proof \rangle$

lemma *nonpos-Ints-nonpos* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \implies x \leq (0 :: 'a :: linordered-idom)$
 $\langle proof \rangle$

lemma *nonpos-Ints-Int* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \implies x \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *nonpos-Ints-cases*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = of-int\ n \wedge n \leq 0$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-cases'*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = -\text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *of-real-in-nonpos-Ints-iff*: $(\text{of-real } x :: 'a :: \text{real-algebra-1}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *nonpos-Ints-altdef*: $\mathbb{Z}_{\leq 0} = \{n \in \mathbb{Z}. (n :: 'a :: \text{linordered-idom}) \leq 0\}$
 $\langle \text{proof} \rangle$

lemma *uminus-in-Nats-iff*: $-x \in \mathbb{N} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *uminus-in-nonpos-Ints-iff*: $-x \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{N}$
 $\langle \text{proof} \rangle$

lemma *nonpos-Ints-mult*: $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{Z}_{\leq 0} \implies x * y \in \mathbb{N}$
 $\langle \text{proof} \rangle$

lemma *Nats-mult-nonpos-Ints*: $x \in \mathbb{N} \implies y \in \mathbb{Z}_{\leq 0} \implies x * y \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *nonpos-Ints-mult-Nats*:
 $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{N} \implies x * y \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *nonpos-Ints-add*:
 $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{Z}_{\leq 0} \implies x + y \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *nonpos-Ints-diff-Nats*:
 $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{N} \implies x - y \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *Nats-diff-nonpos-Ints*:
 $x \in \mathbb{N} \implies y \in \mathbb{Z}_{\leq 0} \implies x - y \in \mathbb{N}$
 $\langle \text{proof} \rangle$

lemma *plus-of-nat-eq-0-imp*: $z + \text{of-nat } n = 0 \implies z \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

64.2 Non-negative reals

definition *nonneg-Reals* :: $'a :: \text{real-algebra-1}$ *set* $(\mathbb{R}_{\geq 0})$
where $\mathbb{R}_{\geq 0} = \{\text{of-real } r \mid r. r \geq 0\}$

lemma *nonneg-Reals-of-real-iff* [simp]: *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]: $x \in \mathbb{R}_{\geq 0} \implies x \in \mathbb{R}$
 ⟨proof⟩

lemma *nonneg-Reals-of-nat-I* [simp]: *of-nat* $n \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-cases*:
 assumes $x \in \mathbb{R}_{\geq 0}$
 obtains r where $x = \text{of-real } r$ $r \geq 0$
 ⟨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]: *numeral* $w \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-minus-numeral-I* [simp]: $- \text{numeral } w \notin \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-add-I* [simp]: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a + b \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-mult-I* [simp]: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a * b \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-inverse-I* [simp]:
 fixes $a :: 'a::\text{real-div-algebra}$
 shows $a \in \mathbb{R}_{\geq 0} \implies \text{inverse } a \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-divide-I* [simp]:
 fixes $a :: 'a::\text{real-div-algebra}$
 shows $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a / b \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-pow-I* [simp]: $a \in \mathbb{R}_{\geq 0} \implies a^n \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *complex-nonneg-Reals-iff*: $z \in \mathbb{R}_{\geq 0} \longleftrightarrow \text{Re } z \geq 0 \wedge \text{Im } z = 0$
 ⟨proof⟩

lemma *ii-not-nonneg-Reals* [iff]: $i \notin \mathbb{R}_{\geq 0}$
 ⟨proof⟩

64.3 Non-positive reals

definition *nonpos-Reals* :: ‘a::real-algebra-1 set ($\mathbb{R}_{\leq 0}$)
 where $\mathbb{R}_{\leq 0} = \{\text{of-real } r \mid r. r \leq 0\}$

lemma *nonpos-Reals-of-real-iff* [simp]: $\text{of-real } r \in \mathbb{R}_{\leq 0} \longleftrightarrow r \leq 0$
 ⟨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]: $\text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow n=0$
 ⟨proof⟩

lemma *nonpos-Reals-Real* [dest]: $x \in \mathbb{R}_{\leq 0} \implies x \in \mathbb{R}$
 ⟨proof⟩

lemma *nonpos-Reals-cases*:
 assumes $x \in \mathbb{R}_{\leq 0}$
 obtains r where $x = \text{of-real } r$ $r \leq 0$
 ⟨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]: $\text{numeral } w \notin \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-add-I* [simp]: $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a + b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-mult-I1*: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-mult-I2*: $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-mult-of-nat-iff*:
 fixes $a :: 'a :: \text{real-div-algebra}$ **shows** $a * \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n = 0$
 ⟨proof⟩

lemma *nonpos-Reals-inverse-I*:
 fixes $a :: 'a :: \text{real-div-algebra}$
shows $a \in \mathbb{R}_{\leq 0} \implies \text{inverse } a \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-divide-I1*:
 fixes $a :: 'a :: \text{real-div-algebra}$
shows $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-divide-I2*:
 fixes $a :: 'a :: \text{real-div-algebra}$
shows $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-divide-of-nat-iff*:
 fixes $a :: 'a :: \text{real-div-algebra}$ **shows** $a / \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n = 0$
 ⟨proof⟩

lemma *nonpos-Reals-inverse-iff* [simp]:
 fixes $a :: 'a :: \text{real-div-algebra}$
shows $\text{inverse } a \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-pow-I*: $\llbracket a \in \mathbb{R}_{\leq 0}; \text{odd } n \rrbracket \implies a^{\wedge n} \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *complex-nonpos-Reals-iff*: $z \in \mathbb{R}_{\leq 0} \longleftrightarrow \text{Re } z \leq 0 \wedge \text{Im } z = 0$
 ⟨proof⟩

lemma *ii-not-nonpos-Reals* [iff]: $i \notin \mathbb{R}_{\leq 0}$
 ⟨proof⟩

end

65 Numeral Syntax for Types

```
theory Numeral-Type
imports Cardinality
begin
```

65.1 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:  $\neg$  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>
```

65.2 num1

```
instance num1 :: finite
<proof>

instantiation num1 :: CARD-1
begin

instance
<proof>

end

lemma num1-eq-iff: (x::num1) = (y::num1)  $\longleftrightarrow$  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]:  $(\forall i::\text{num1}. P\ i) \longleftrightarrow P\ 1$ 
  ⟨proof⟩

lemma ex-1 [simp]:  $(\exists x::\text{num1}. P\ x) \longleftrightarrow P\ 1$ 
  ⟨proof⟩

instantiation num1 :: linorder begin
definition a < b  $\longleftrightarrow \text{Rep-num1}\ a < \text{Rep-num1}\ b$ 
definition a ≤ b  $\longleftrightarrow \text{Rep-num1}\ a \leq \text{Rep-num1}\ b$ 
instance
  ⟨proof⟩
end

instance num1 :: wellorder
  ⟨proof⟩

instance bit0 :: (finite) card2
  ⟨proof⟩

instance bit1 :: (finite) card2
  ⟨proof⟩

```

65.3 Locales for modular arithmetic subtypes

```

locale mod-type =
  fixes n :: int
  and Rep :: 'a::{zero,one,plus,times,uminus,minus}  $\Rightarrow$  int
  and Abs :: int  $\Rightarrow$  'a::{zero,one,plus,times,uminus,minus}
  assumes type: type-definition Rep Abs {0..<n}
  and size1:  $1 < n$ 
  and zero-def:  $0 = \text{Abs}\ 0$ 
  and one-def:  $1 = \text{Abs}\ 1$ 
  and add-def:  $x + y = \text{Abs}\ ((\text{Rep}\ x + \text{Rep}\ y) \bmod n)$ 
  and mult-def:  $x * y = \text{Abs}\ ((\text{Rep}\ x * \text{Rep}\ y) \bmod n)$ 

```


and *diff-def*: $x - y = \text{Abs } ((\text{Rep } x - \text{Rep } y) \bmod n)$
and *minus-def*: $-x = \text{Abs } ((-\text{Rep } x) \bmod n)$
begin

lemma *size0*: $0 < n$
 $\langle \text{proof} \rangle$

lemmas *definitions* =
zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: $\text{Rep } x < n$
 $\langle \text{proof} \rangle$

lemma *Rep-le-n*: $\text{Rep } x \leq n$
 $\langle \text{proof} \rangle$

lemma *Rep-inject-sym*: $x = y \longleftrightarrow \text{Rep } x = \text{Rep } y$
 $\langle \text{proof} \rangle$

lemma *Rep-inverse*: $\text{Abs } (\text{Rep } x) = x$
 $\langle \text{proof} \rangle$

lemma *Abs-inverse*: $m \in \{0..<n\} \implies \text{Rep } (\text{Abs } m) = m$
 $\langle \text{proof} \rangle$

lemma *Rep-Abs-mod*: $\text{Rep } (\text{Abs } (m \bmod n)) = m \bmod n$
 $\langle \text{proof} \rangle$

lemma *Rep-Abs-0*: $\text{Rep } (\text{Abs } 0) = 0$
 $\langle \text{proof} \rangle$

lemma *Rep-0*: $\text{Rep } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *Rep-Abs-1*: $\text{Rep } (\text{Abs } 1) = 1$
 $\langle \text{proof} \rangle$

lemma *Rep-1*: $\text{Rep } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *Rep-mod*: $\text{Rep } x \bmod n = \text{Rep } x$
 $\langle \text{proof} \rangle$

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)*
 $\langle \text{proof} \rangle$

end

locale *mod-ring* = *mod-type* *n Rep Abs*
for *n* :: *int*
and *Rep* :: '*a*::{*comm-ring-1*} $\Rightarrow$ *int*
and *Abs* :: *int* $\Rightarrow$ '*a*::{*comm-ring-1*}
begin

lemma *of-nat-eq*: *of-nat* *k* = *Abs* (*int* *k mod n*)
 $\langle$ *proof* $\rangle$

lemma *of-int-eq*: *of-int* *z* = *Abs* (*z mod n*)
 $\langle$ *proof* $\rangle$

lemma *Rep-numeral*:
Rep (*numeral* *w*) = *numeral* *w mod n*
 $\langle$ *proof* $\rangle$

lemma *iszero-numeral*:
iszero (*numeral* *w*::'*a*) $\longleftrightarrow$ *numeral* *w mod n* = 0
 $\langle$ *proof* $\rangle$

lemma *cases*:
assumes 1: $\bigwedge z. \llbracket (x::'a) = \text{of-int } z; 0 \leq z; z < n \rrbracket \Longrightarrow P$
shows *P*
 $\langle$ *proof* $\rangle$

lemma *induct*:
 $(\bigwedge z. \llbracket 0 \leq z; z < n \rrbracket \Longrightarrow P (\text{of-int } z)) \Longrightarrow P (x::'a)$
 $\langle$ *proof* $\rangle$

lemma *UNIV-eq*: (*UNIV* :: '*a* set) = *Abs* ' {0..*n*}
 $\langle$ *proof* $\rangle$

lemma *CARD-eq*: *CARD*('a) = *nat* *n*
 $\langle$ *proof* $\rangle$

lemma *CHAR-eq* [*simp*]: *CHAR*('a) = *CARD*('a)
 $\langle$ *proof* $\rangle$

end

65.4 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 $Abs-bit0' :: int \Rightarrow 'a \text{ bit0}$ **where**
 $Abs-bit0' x = Abs-bit0 (x \bmod int \text{ CARD}('a \text{ bit0}))$

definition $Abs-bit1' :: int \Rightarrow 'a \text{ bit1}$ **where**
 $Abs-bit1' x = Abs-bit1 (x \bmod int \text{ CARD}('a \text{ bit1}))$

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 $\langle proof \rangle$

end

interpretation $bit0$:

$mod\text{-}type \text{ int } CARD('a::finite \text{ bit0})$

$Rep\text{-}bit0 :: 'a::finite \text{ bit0} \Rightarrow int$

$Abs\text{-}bit0 :: int \Rightarrow 'a::finite \text{ bit0}$

$\langle proof \rangle$

interpretation $bit1$:

$mod\text{-}type \text{ int } CARD('a::finite \text{ bit1})$

$Rep\text{-}bit1 :: 'a::finite \text{ bit1} \Rightarrow int$

$Abs\text{-}bit1 :: int \Rightarrow 'a::finite \text{ bit1}$

$\langle proof \rangle$

instance $bit0 :: (finite) \text{ comm-ring-1}$

$\langle proof \rangle$

instance $bit1 :: (finite) \text{ comm-ring-1}$

$\langle proof \rangle$

interpretation $bit0$:

$mod\text{-}ring \text{ int } CARD('a::finite \text{ bit0})$

$Rep\text{-}bit0 :: 'a::finite \text{ bit0} \Rightarrow int$

$Abs\text{-}bit0 :: int \Rightarrow 'a::finite \text{ bit0}$

$\langle proof \rangle$

interpretation *bit1*:

mod-ring int CARD('a::finite bit1)
Rep-bit1 :: 'a::finite bit1 $\Rightarrow$ int
Abs-bit1 :: int $\Rightarrow$ '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*

65.5 Order instances

instantiation *bit0* and *bit1 :: (finite) linorder* **begin**

definition *a < b $\longleftrightarrow$ Rep-bit0 a < Rep-bit0 b*

definition *a $\leq$ b $\longleftrightarrow$ Rep-bit0 a $\leq$ Rep-bit0 b*

definition *a < b $\longleftrightarrow$ Rep-bit1 a < Rep-bit1 b*

definition *a $\leq$ b $\longleftrightarrow$ Rep-bit1 a $\leq$ Rep-bit1 b*

instance

<proof>

end

lemma (**in** *preorder*) *trancpl-less: (<)⁺⁺ = (<)*

<proof>

instance *bit0* and *bit1 :: (finite) wellorder*

<proof>

65.6 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 $\Rightarrow$ num0 $\Rightarrow$ 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* $\Rightarrow$ '*a bit1*) {*0..<1 + 2 * int CARD('a :: finite)*}
 ⟨proof⟩

instantiation *bit0* and *bit1* :: (finite) equal begin

definition *equal-class.equal* *x y* $\longleftrightarrow$ *Rep-bit0* *x* = *Rep-bit0* *y*

definition *equal-class.equal* *x y* $\longleftrightarrow$ *Rep-bit1* *x* = *Rep-bit1* *y*

instance
 ⟨proof⟩

end

instantiation *bit0* :: (finite) enum begin

definition (*enum-class.enum* :: '*a bit0* list) = *map* (*Abs-bit0'* $\circ$ *int*) (*upt* 0 (*CARD('a bit0)*))

definition *enum-class.enum-all* *P* = ($\forall$ *b* :: '*a bit0* $\in$ set *enum-class.enum*. *P b*)

definition *enum-class.enum-ex* *P* = ($\exists$ *b* :: '*a bit0* $\in$ set *enum-class.enum*. *P b*)

instance ⟨proof⟩

end

instantiation *bit1* :: (finite) enum begin

definition (*enum-class.enum* :: '*a bit1* list) = *map* (*Abs-bit1'* $\circ$ *int*) (*upt* 0 (*CARD('a bit1)*))

definition *enum-class.enum-all* *P* = ($\forall$ *b* :: '*a bit1* $\in$ set *enum-class.enum*. *P b*)

definition *enum-class.enum-ex* *P* = ($\exists$ *b* :: '*a bit1* $\in$ 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

```

65.7 Syntax

```

syntax
  -NumeralType :: num-token => type (-)
  -NumeralType0 :: type (0)
  -NumeralType1 :: type (1)

```

```

translations
  (type) 1 == (type) num1
  (type) 0 == (type) num0

```

<ML>

65.8 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

```

66 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.

66.1 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

$'a \text{ word} = \text{nat} \Rightarrow 'a$

We can prefix a finite word to an ω -word, and a way to obtain an ω -word from a finite, non-empty word is by ω -iteration.

definition

$\text{conc} :: ['a \text{ list}, 'a \text{ word}] \Rightarrow 'a \text{ word}$ (**infixr** $\langle \frown \rangle$ 65)
where $w \frown x == \lambda n. \text{if } n < \text{length } w \text{ then } w!n \text{ else } x (n - \text{length } w)$

definition

$\text{iter} :: 'a \text{ list} \Rightarrow 'a \text{ word}$ ($\langle (-)^\omega \rangle$ [1000])
where $\text{iter } w == \text{if } w = [] \text{ then undefined else } (\lambda n. w!(n \bmod (\text{length } w)))$

lemma conc-empty[simp] : $[] \frown w = w$
 $\langle \text{proof} \rangle$

lemma conc-fst[simp] : $n < \text{length } w \implies (w \frown x) n = w!n$
 $\langle \text{proof} \rangle$

lemma conc-snd[simp] : $\neg(n < \text{length } w) \implies (w \frown x) n = x (n - \text{length } w)$
 $\langle \text{proof} \rangle$

lemma iter-nth [simp] : $0 < \text{length } w \implies w^\omega n = w!(n \bmod (\text{length } w))$
 $\langle \text{proof} \rangle$

lemma conc-conc[simp] : $u \frown v \frown w = (u @ v) \frown w$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemma range-conc[simp] : $\text{range } (w_1 \frown w_2) = \text{set } w_1 \cup \text{range } w_2$
 $\langle \text{proof} \rangle$

lemma iter-unroll : $0 < \text{length } w \implies w^\omega = w \frown w^\omega$
 $\langle \text{proof} \rangle$

66.2 Subsequence, Prefix, and Suffix

definition $\text{suffix} :: [\text{nat}, 'a \text{ word}] \Rightarrow 'a \text{ word}$
where $\text{suffix } k \ x \equiv \lambda n. x (k+n)$

definition $\text{subsequence} :: 'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ list}$ ($\langle [- \rightarrow -] \rangle$ 900)
where $\text{subsequence } w \ i \ j \equiv \text{map } w \ [i..<j]$

abbreviation $\text{prefix} :: \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ list}$
where $\text{prefix } n \ w \equiv \text{subsequence } w \ 0 \ n$

lemma suffix-nth [simp] : $(\text{suffix } k \ x) n = x (k+n)$
 $\langle \text{proof} \rangle$

lemma *suffix-0* [*simp*]: $\text{suffix } 0 \ x = x$
 $\langle \text{proof} \rangle$

lemma *suffix-suffix* [*simp*]: $\text{suffix } m \ (\text{suffix } k \ x) = \text{suffix } (k+m) \ x$
 $\langle \text{proof} \rangle$

lemma *subsequence-append*: $\text{prefix } (i + j) \ w = \text{prefix } i \ w \ @ \ (w \ [i \rightarrow i + j])$
 $\langle \text{proof} \rangle$

lemma *subsequence-drop*[*simp*]: $\text{drop } i \ (w \ [j \rightarrow k]) = w \ [j + i \rightarrow k]$
 $\langle \text{proof} \rangle$

lemma *subsequence-empty*[*simp*]: $w \ [i \rightarrow j] = [] \iff j \leq i$
 $\langle \text{proof} \rangle$

lemma *subsequence-length*[*simp*]: $\text{length } (\text{subsequence } w \ i \ j) = j - i$
 $\langle \text{proof} \rangle$

lemma *subsequence-nth*[*simp*]: $k < j - i \implies (w \ [i \rightarrow j]) ! k = w \ (i + k)$
 $\langle \text{proof} \rangle$

lemma *subseq-to-zero*[*simp*]: $w[i \rightarrow 0] = []$
 $\langle \text{proof} \rangle$

lemma *subseq-to-smaller*[*simp*]: $i \geq j \implies w[i \rightarrow j] = []$
 $\langle \text{proof} \rangle$

lemma *subseq-to-Suc*[*simp*]: $i \leq j \implies w \ [i \rightarrow \text{Suc } j] = w \ [i \rightarrow j] \ @ \ [w \ j]$
 $\langle \text{proof} \rangle$

lemma *subsequence-singleton*[*simp*]: $w \ [i \rightarrow \text{Suc } i] = [w \ i]$
 $\langle \text{proof} \rangle$

lemma *subsequence-prefix-suffix*: $\text{prefix } (j - i) \ (\text{suffix } i \ w) = w \ [i \rightarrow j]$
 $\langle \text{proof} \rangle$

lemma *prefix-suffix*: $x = \text{prefix } n \ x \frown (\text{suffix } n \ x)$
 $\langle \text{proof} \rangle$

declare *prefix-suffix*[*symmetric, simp*]

lemma *word-split*: **obtains** $v_1 \ v_2$ **where** $v = v_1 \frown v_2$ *length* $v_1 = k$
 $\langle \text{proof} \rangle$

lemma *set-subsequence*[*simp*]: $\text{set } (w[i \rightarrow j]) = w\{i..<j\}$

$\langle \text{proof} \rangle$

lemma *subsequence-take[simp]*: $\text{take } i \ (w \ [j \rightarrow k]) = w \ [j \rightarrow \min (j + i) \ k]$
 $\langle \text{proof} \rangle$

lemma *subsequence-shift[simp]*: $(\text{suffix } i \ w) \ [j \rightarrow k] = w \ [i + j \rightarrow i + k]$
 $\langle \text{proof} \rangle$

lemma *suffix-subseq-join[simp]*: $i \leq j \implies v \ [i \rightarrow j] \frown \text{suffix } j \ v = \text{suffix } i \ v$
 $\langle \text{proof} \rangle$

lemma *prefix-conc-fst[simp]*:
 assumes $j \leq \text{length } w$
 shows $\text{prefix } j \ (w \frown w') = \text{take } j \ w$
 $\langle \text{proof} \rangle$

lemma *prefix-conc-snd[simp]*:
 assumes $n \geq \text{length } u$
 shows $\text{prefix } n \ (u \frown v) = u \ @ \ \text{prefix } (n - \text{length } u) \ v$
 $\langle \text{proof} \rangle$

lemma *prefix-conc-length[simp]*: $\text{prefix } (\text{length } w) \ (w \frown w') = w$
 $\langle \text{proof} \rangle$

lemma *suffix-conc-fst[simp]*:
 assumes $n \leq \text{length } u$
 shows $\text{suffix } n \ (u \frown v) = \text{drop } n \ u \frown v$
 $\langle \text{proof} \rangle$

lemma *suffix-conc-snd[simp]*:
 assumes $n \geq \text{length } u$
 shows $\text{suffix } n \ (u \frown v) = \text{suffix } (n - \text{length } u) \ v$
 $\langle \text{proof} \rangle$

lemma *suffix-conc-length[simp]*: $\text{suffix } (\text{length } w) \ (w \frown w') = w'$
 $\langle \text{proof} \rangle$

lemma *concat-eq[iff]*:
 assumes $\text{length } v_1 = \text{length } v_2$
 shows $v_1 \frown u_1 = v_2 \frown u_2 \iff v_1 = v_2 \wedge u_1 = u_2$
 (is $?lhs \iff ?rhs$)
 $\langle \text{proof} \rangle$

lemma *same-concat-eq[iff]*: $u \frown v = u \frown w \iff v = w$
 $\langle \text{proof} \rangle$

lemma *comp-concat[simp]*: $f \circ u \frown v = \text{map } f \ u \frown (f \circ v)$
 $\langle \text{proof} \rangle$

66.3 Prepending

primrec *build* :: 'a $\Rightarrow$ 'a word $\Rightarrow$ 'a word (**infixr** $\langle \#\# \rangle$ 65)
where (a $\#\#$ w) 0 = a | (a $\#\#$ w) (Suc i) = w i

lemma *build-eq[iff]*: $a_1 \#\# w_1 = a_2 \#\# w_2 \longleftrightarrow a_1 = a_2 \wedge w_1 = w_2$
 $\langle proof \rangle$

lemma *build-cons[simp]*: $(a \# u) \frown v = a \#\# u \frown v$
 $\langle proof \rangle$

lemma *build-append[simp]*: $(w @ a \# u) \frown v = w \frown a \#\# u \frown v$
 $\langle proof \rangle$

lemma *build-first[simp]*: $w 0 \#\# \text{suffix } (Suc\ 0) w = w$
 $\langle proof \rangle$

lemma *build-split[intro]*: $w = w 0 \#\# \text{suffix } 1 w$
 $\langle proof \rangle$

lemma *build-range[simp]*: $\text{range } (a \#\# w) = \text{insert } a (\text{range } w)$
 $\langle proof \rangle$

lemma *suffix-singleton-suffix[simp]*: $w i \#\# \text{suffix } (Suc\ i) w = \text{suffix } i w$
 $\langle proof \rangle$

Find the first occurrence of a letter from a given set

lemma *word-first-split-set*:
assumes $A \cap \text{range } w \neq \{\}$
obtains $u\ a\ v$ **where** $w = u \frown [a] \frown v$ $A \cap \text{set } u = \{\}$ $a \in A$
 $\langle proof \rangle$

66.4 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 $\text{limit } x \equiv \{a . \exists_{\infty} n . x\ n = a\}$

lemma *limit-iff-frequent*: $a \in \text{limit } x \longleftrightarrow (\exists_{\infty} n . x\ n = a)$
 $\langle proof \rangle$

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*: $(a \in \text{limit } x) = \text{infinite } (x - \{a\})$
 $\langle proof \rangle$

lemma *two-in-limit-iff*:

$(\{a, b\} \subseteq \text{limit } x) =$
 $((\exists n. x \ n = a) \wedge (\forall n. x \ n = a \longrightarrow (\exists m > n. x \ m = b)) \wedge (\forall m. x \ m = b \longrightarrow$
 $(\exists n > m. x \ n = a)))$
 (is ?lhs = (?r1 $\wedge$?r2 $\wedge$?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 $\exists a. a \in \text{limit } x$
 <proof>

lemmas *limit-nonemptyE* = *limit-nonempty*[THEN *exE*]

lemma *limit-inter-INF*:

assumes *hyp*: $\text{limit } w \cap S \neq \{\}$
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 ($S \cap \text{range } w$)
shows $\exists a. a \in \text{limit } w \cap S$
 <proof>

lemma *fin-ex-inf-eq-limit*: $\text{finite } A \implies (\exists_{\infty} i. w \ i \in A) \longleftrightarrow \text{limit } w \cap A \neq \{\}$
 <proof>

lemma *limit-in-range-suffix*: $\text{limit } x \subseteq \text{range } (\text{suffix } k \ x)$
 <proof>

lemma *limit-in-range*: $\text{limit } r \subseteq \text{range } r$
 <proof>

lemmas *limit-in-range-suffixD* = *limit-in-range-suffix*[THEN *subsetD*]

lemma *limit-subset*: $\text{limit } f \subseteq f \text{ ‘ } \{n..\}$
 <proof>

theorem *limit-is-suffix*:

assumes *fin*: finite (range x)
shows $\exists k. \text{limit } x = \text{range } (\text{suffix } k \ x)$
 <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]: $\text{limit } (w \frown x) = \text{limit } x$
 ⟨proof⟩

theorem *limit-suffix* [simp]: $\text{limit } (\text{suffix } n \ x) = \text{limit } x$
 ⟨proof⟩

theorem *limit-iter* [simp]:
 assumes *nempty*: $0 < \text{length } w$
 shows $\text{limit } w^\omega = \text{set } w$
 ⟨proof⟩

lemma *limit-o* [simp]:
 assumes *a*: $a \in \text{limit } w$
 shows $f \ a \in \text{limit } (f \circ w)$
 ⟨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*: $\text{finite } (f^{-1} \{x\})$
 and *x*: $x \in \text{limit } (f \circ w)$
 shows $\exists a \in (f^{-1} \{x\}). a \in \text{limit } w$
 ⟨proof⟩

theorem *limit-inj* [simp]:
 assumes *inj*: $\text{inj } f$
 shows $\text{limit } (f \circ w) = f^{-1} (\text{limit } w)$
 ⟨proof⟩

lemma *limit-inter-empty*:
 assumes *fin*: $\text{finite } (\text{range } w)$
 assumes *hyp*: $\text{limit } w \cap S = \{\}$
 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 $\text{limit } r = \text{range } (\text{suffix } i \ r)$
 assumes $j \geq i$
 shows $\text{limit } r = \text{range } (\text{suffix } j \ r)$
 (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*⟩

66.5 Index sequences and piecewise definitions

A word can be defined piecewise: given a sequence of words $w_0, w_1, \dots$ and a strictly increasing sequence of integers $i_0, i_1, \dots$ 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

$$(w_0)_{i_0} \dots (w_0)_{i_1-1} (w_1)_{i_1} \dots (w_1)_{i_2-1} \dots$$

We prepare the field by proving some trivial facts about such sequences of indexes.

definition *idx-sequence* :: nat word $\Rightarrow$ bool
where *idx-sequence* *idx* $\equiv$ (*idx* 0 = 0) $\wedge$ ($\forall n. \text{idx } n < \text{idx } (\text{Suc } n)$)

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* $\leq$ *n*
shows *idx* *m* $\leq$ *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* $\in$ {*idx* *k* ..< *idx* (*Suc* *k*)}
 ⟨*proof*⟩

lemma *idx-sequence-interval*:

```

assumes isec: idx-sequence idx
shows  $\exists k. n \in \{idx\ k \ ..< \ idx\ (Suc\ k)\}$ 
  (is ?P n is  $\exists k. \ ?in\ n\ k$ )
<proof>

```

```

lemma idx-sequence-interval-unique:
assumes isec: idx-sequence idx
  and k:  $n \in \{idx\ k \ ..< \ idx\ (Suc\ k)\}$ 
  and m:  $n \in \{idx\ m \ ..< \ idx\ (Suc\ m)\}$ 
shows  $k = m$ 
<proof>

```

```

lemma idx-sequence-unique-interval:
assumes isec: idx-sequence idx
shows  $\exists! k. n \in \{idx\ k \ ..< \ idx\ (Suc\ k)\}$ 
<proof>

```

Now we can define the piecewise construction of a word using an index sequence.

```

definition merge :: 'a word word  $\Rightarrow$  nat word  $\Rightarrow$  'a word
  where merge ws idx  $\equiv \lambda n. \text{let } i = \text{THE } i. n \in \{idx\ i \ ..< \ idx\ (Suc\ i)\} \text{ in } ws\ i\ n$ 

```

```

lemma merge:
assumes idx: idx-sequence idx
  and n:  $n \in \{idx\ i \ ..< \ idx\ (Suc\ i)\}$ 
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:  $n \in \{idx\ i \ ..< \ idx\ (Suc\ i)\}$ 
shows merge ws idx (Suc n) = (if Suc n = idx (Suc i) then ws (Suc i) else ws i (Suc n))
<proof>

```

end

67 Combinator syntax for generic, open state monads (single-threaded monads)

```

theory Open-State-Syntax
imports Main
begin

```

```

context
  includes state-combinator-syntax
begin

```

67.1 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.

67.2 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 $\alpha, \beta, \gamma, \dots$ 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.

67.3 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

67.4 Do-syntax

nonterminal *sdo-binds* and *sdo-bind*

syntax

-sdo-block :: *sdo-binds* $\Rightarrow$ 'a (*exec* { //(2 -) // } [12] 62)
-sdo-bind :: [*pttrn*, 'a] $\Rightarrow$ *sdo-bind* ((- <-/-) 13)
-sdo-let :: [*pttrn*, 'a] $\Rightarrow$ *sdo-bind* ((2let - =/-) [1000, 13] 13)
-sdo-then :: 'a $\Rightarrow$ *sdo-bind* (- [14] 13)
-sdo-final :: 'a $\Rightarrow$ *sdo-binds* (-)
-sdo-cons :: [*sdo-bind*, *sdo-binds*] $\Rightarrow$ *sdo-binds* (-;/- [13, 12] 12)

syntax (*ASCII*)

-sdo-bind :: [*pttrn*, 'a] $\Rightarrow$ *sdo-bind* ((- <-/-) 13)

translations

-sdo-block (*-sdo-cons* (*-sdo-bind* *p t*) (*-sdo-final* *e*))
 $\equiv$ *CONST scomp t* ($\lambda p.$ *e*)
-sdo-block (*-sdo-cons* (*-sdo-then* *t*) (*-sdo-final* *e*))
 $\equiv$ *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

68 Canonical order on option type

```

theory Option-ord
imports Main
begin

```

```

unbundle lattice-syntax

```

```

instantiation option :: (preorder) preorder
begin

```

definition *less-eq-option* **where**

$$x \leq y \longleftrightarrow (\text{case } x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } y \Rightarrow x \leq y))$$

definition *less-option* **where**

$$x < y \longleftrightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } y \Rightarrow (\text{case } x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow x < y))$$

lemma *less-eq-option-None* [simp]: $\text{None} \leq x$
 <proof>

lemma *less-eq-option-None-code* [code]: $\text{None} \leq x \longleftrightarrow \text{True}$
 <proof>

lemma *less-eq-option-None-is-None*: $x \leq \text{None} \implies x = \text{None}$
 <proof>

lemma *less-eq-option-Some-None* [simp, code]: $\text{Some } x \leq \text{None} \longleftrightarrow \text{False}$
 <proof>

lemma *less-eq-option-Some* [simp, code]: $\text{Some } x \leq \text{Some } y \longleftrightarrow x \leq y$
 <proof>

lemma *less-option-None* [simp, code]: $x < \text{None} \longleftrightarrow \text{False}$

<proof>

lemma *less-option-None-is-Some*: $\text{None} < x \implies \exists z. x = \text{Some } z$
<proof>

lemma *less-option-None-Some* [simp]: $\text{None} < \text{Some } x$
<proof>

lemma *less-option-None-Some-code* [code]: $\text{None} < \text{Some } x \longleftrightarrow \text{True}$
<proof>

lemma *less-option-Some* [simp, code]: $\text{Some } x < \text{Some } y \longleftrightarrow x < y$
<proof>

instance
<proof>

end

instance *option* :: (order) order
<proof>

instance *option* :: (linorder) linorder
<proof>

instantiation *option* :: (order) order-bot
begin

definition *bot-option* **where** $\perp = \text{None}$

instance
<proof>

end

instantiation *option* :: (order-top) order-top
begin

definition *top-option* **where** $\top = \text{Some } \top$

instance
<proof>

end

instance *option* :: (wellorder) wellorder
<proof>

instantiation *option* :: (inf) inf

begin

definition *inf-option* **where**

$x \sqcap y = (\text{case } x \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } y \Rightarrow \text{Some } (x \sqcap y)))$

lemma *inf-None-1* [*simp*, *code*]: $\text{None} \sqcap y = \text{None}$
 $\langle \text{proof} \rangle$

lemma *inf-None-2* [*simp*, *code*]: $x \sqcap \text{None} = \text{None}$
 $\langle \text{proof} \rangle$

lemma *inf-Some* [*simp*, *code*]: $\text{Some } x \sqcap \text{Some } y = \text{Some } (x \sqcap y)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instantiation *option* :: (*sup*) *sup*
begin

definition *sup-option* **where**

$x \sqcup y = (\text{case } x \text{ of } \text{None} \Rightarrow y \mid \text{Some } x' \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow x \mid \text{Some } y \Rightarrow \text{Some } (x' \sqcup y)))$

lemma *sup-None-1* [*simp*, *code*]: $\text{None} \sqcup y = y$
 $\langle \text{proof} \rangle$

lemma *sup-None-2* [*simp*, *code*]: $x \sqcup \text{None} = x$
 $\langle \text{proof} \rangle$

lemma *sup-Some* [*simp*, *code*]: $\text{Some } x \sqcup \text{Some } y = \text{Some } (x \sqcup y)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instance *option* :: (*semilattice-inf*) *semilattice-inf*
 $\langle \text{proof} \rangle$

instance *option* :: (*semilattice-sup*) *semilattice-sup*
 $\langle \text{proof} \rangle$

instance *option* :: (*lattice*) *lattice* $\langle \text{proof} \rangle$

instance *option* :: (*lattice*) *bounded-lattice-bot* $\langle \text{proof} \rangle$

instance *option* :: (*bounded-lattice-top*) *bounded-lattice-top* $\langle$ *proof* $\rangle$

instance *option* :: (*bounded-lattice-top*) *bounded-lattice* $\langle$ *proof* $\rangle$

instance *option* :: (*distrib-lattice*) *distrib-lattice*
 $\langle$ *proof* $\rangle$

instantiation *option* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-option* :: 'a *option set* $\Rightarrow$ 'a *option* **where**
 $\bigcap A = (\text{if } \text{None} \in A \text{ then } \text{None} \text{ else } \text{Some } (\bigcap \text{Option.these } A))$

lemma *None-in-Inf* [*simp*]: $\text{None} \in A \implies \bigcap A = \text{None}$
 $\langle$ *proof* $\rangle$

definition *Sup-option* :: 'a *option set* $\Rightarrow$ 'a *option* **where**
 $\bigcup A = (\text{if } A = \{\} \vee A = \{\text{None}\} \text{ then } \text{None} \text{ else } \text{Some } (\bigcup \text{Option.these } A))$

lemma *empty-Sup* [*simp*]: $\bigcup \{\} = \text{None}$
 $\langle$ *proof* $\rangle$

lemma *singleton-None-Sup* [*simp*]: $\bigcup \{\text{None}\} = \text{None}$
 $\langle$ *proof* $\rangle$

instance
 $\langle$ *proof* $\rangle$

end

lemma *Some-Inf*:
 $\text{Some } (\bigcap A) = \bigcap (\text{Some } ` A)$
 $\langle$ *proof* $\rangle$

lemma *Some-Sup*:
 $A \neq \{\} \implies \text{Some } (\bigcup A) = \bigcup (\text{Some } ` A)$
 $\langle$ *proof* $\rangle$

lemma *Some-INF*:
 $\text{Some } (\bigcap x \in A. f x) = (\bigcap x \in A. \text{Some } (f x))$
 $\langle$ *proof* $\rangle$

lemma *Some-SUP*:
 $A \neq \{\} \implies \text{Some } (\bigcup x \in A. f x) = (\bigcup x \in A. \text{Some } (f x))$
 $\langle$ *proof* $\rangle$

lemma *option-Inf-Sup*: $\bigcap (\text{Sup } ` A) \leq \bigcup (\text{Inf } ` \{f ` A \mid f. \forall Y \in A. f Y \in Y\})$
for $A :: ('a :: \text{complete-distrib-lattice } \text{option}) \text{ set set}$
 $\langle$ *proof* $\rangle$

```

instance option :: (complete-distrib-lattice) complete-distrib-lattice
  ⟨proof⟩

instance option :: (complete-linorder) complete-linorder ⟨proof⟩

unbundle no-lattice-syntax

end

```

69 Futures and parallel lists for code generated towards Isabelle/ML

```

theory Parallel
imports Main
begin

```

69.1 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

```

69.2 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*]:
 $exists\ P\ xs \longleftrightarrow (\exists x \in set\ xs.\ P\ x)$
 ⟨*proof*⟩

code-printing
constant *map* $\rightarrow (Eval)\ Par'-List.map$
| **constant** *forall* $\rightarrow (Eval)\ Par'-List.forall$
| **constant** *exists* $\rightarrow (Eval)\ Par'-List.exists$

code-reserved *Eval Par-List*

hide-const (**open**) *fork join map exists forall*

end

70 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.

70.1 Definition

consts

as :: 'a $\Rightarrow$ 'a $\Rightarrow$ 'a
fake-quant :: ('a $\Rightarrow$ prop) $\Rightarrow$ prop

lemma *let-cong-unfolding*: $M = N \Longrightarrow f\ N = g\ N \Longrightarrow \text{Let}\ M\ f = \text{Let}\ N\ g$
 <proof>

translations $P <= \text{CONST}\ \text{fake-quant}\ (\lambda x. P)$

<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*

70.2 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* $x\ y = y + y$
 $\langle proof \rangle$

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* $(y \# y' \# ys) = (y \# y' \# ys) @ (y' \# ys) @ (y' \# ys)$
 $\langle proof \rangle$

$\langle ML \rangle$

end

end

71 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 $f(x + (1::'b)) = f\ x$ instead of $f(x + (1::'b) * (1::'b)) = f\ x$ etc.

locale *periodic-fun* =
fixes $f :: ('a :: \{ring-1\}) \Rightarrow 'b$ **and** $g\ gm :: 'a \Rightarrow 'a \Rightarrow 'a$ **and** $g1\ gn1 :: 'a \Rightarrow 'a$
assumes *plus-1*: $f(g1\ x) = f\ x$
assumes *periodic-arg-plus-0*: $g\ x\ 0 = x$
assumes *periodic-arg-plus-distrib*: $g\ x\ (of-int\ (m + n)) = g\ (g\ x\ (of-int\ n))\ (of-int\ m)$
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*: $f(g\ x\ (of-nat\ n)) = f\ x$
 $\langle proof \rangle$

lemma *minus-of-nat*: $f(gm\ x\ (of-nat\ n)) = f\ x$
 $\langle proof \rangle$

lemma *plus-of-int*: $f(g\ x\ (of-int\ n)) = f\ x$
 $\langle proof \rangle$

lemma *minus-of-int*: $f (gm\ x\ (of-int\ n)) = f\ x$
 $\langle proof \rangle$

lemma *plus-numeral*: $f (g\ x\ (numeral\ n)) = f\ x$
 $\langle proof \rangle$

lemma *minus-numeral*: $f (gm\ x\ (numeral\ n)) = f\ x$
 $\langle proof \rangle$

lemma *minus-1*: $f (gn1\ x) = f\ x$
 $\langle proof \rangle$

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 $f :: ('a :: \{ring-1\}) \Rightarrow 'b$ **and** $period :: 'a$
assumes *plus-period*: $f (x + period) = f\ x$
begin
sublocale *periodic-fun* $f\ \lambda z\ x. z + x * period\ \lambda z\ x. z - x * period$
 $\lambda z. z + period\ \lambda z. z - period$
 $\langle proof \rangle$
end

Specialised case of the *periodic-fun* locale for period 1. Gives lemmas f
 $(x - (1 :: 'b)) = f\ x$ etc.

locale *periodic-fun-simple'* =
fixes $f :: ('a :: \{ring-1\}) \Rightarrow 'b$
assumes *plus-period*: $f (x + 1) = f\ x$
begin
sublocale *periodic-fun* $f\ \lambda z\ x. z + x\ \lambda z\ x. z - x\ \lambda z. z + 1\ \lambda z. z - 1$
 $\langle proof \rangle$

lemma *of-nat*: $f (of-nat\ n) = f\ 0\ \langle proof \rangle$

lemma *uminus-of-nat*: $f (-of-nat\ n) = f\ 0\ \langle proof \rangle$

lemma *of-int*: $f (of-int\ n) = f\ 0\ \langle proof \rangle$

lemma *uminus-of-int*: $f (-of-int\ n) = f\ 0\ \langle proof \rangle$

lemma *of-numeral*: $f (numeral\ n) = f\ 0\ \langle proof \rangle$

lemma *of-neg-numeral*: $f (-numeral\ n) = f\ 0\ \langle proof \rangle$

lemma *of-1*: $f\ 1 = f\ 0\ \langle proof \rangle$

lemma *of-neg-1*: $f (-1) = f\ 0\ \langle proof \rangle$

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*: $\sin ((z :: 'a :: \{\text{real-normed-field}, \text{banach}\}) + \text{of-real } \pi) = -\sin z$
 <proof>

lemma *cos-plus-pi*: $\cos ((z :: 'a :: \{\text{real-normed-field}, \text{banach}\}) + \text{of-real } \pi) = -\cos z$
 <proof>

interpretation *sin*: *periodic-fun-simple* $\sin 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 <proof>

interpretation *cos*: *periodic-fun-simple* $\cos 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 <proof>

interpretation *tan*: *periodic-fun-simple* $\tan 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 <proof>

interpretation *cot*: *periodic-fun-simple* $\cot 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 <proof>

end

72 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view

theory *Poly-Mapping*
imports *Groups-Big-Fun Fun-Lexorder More-List*
begin

72.1 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 $f :: 'b \Rightarrow 'a :: \text{mult-zero}$
 assumes $\text{finite } \{a. f a \neq 0\}$
 shows $\text{finite } \{a. g a * f a \neq 0\}$
 <proof>

lemma *finite-mult-not-eq-zero-rightI*:
 fixes $f :: 'b \Rightarrow 'a :: \text{mult-zero}$
 assumes $\text{finite } \{a. f a \neq 0\}$
 shows $\text{finite } \{a. f a * g a \neq 0\}$
 <proof>

lemma *finite-mult-not-eq-zero-prodI*:

fixes $f\ g :: 'a \Rightarrow 'b::\text{semiring-0}$
 assumes $\text{finite } \{a. f\ a \neq 0\}$ (is finite ?F)
 assumes $\text{finite } \{b. g\ b \neq 0\}$ (is finite ?G)
 shows $\text{finite } \{(a, b). f\ a * g\ b \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-not-eq-zero-sumI*:

fixes $f\ g :: 'a::\text{monoid-add} \Rightarrow 'b::\text{semiring-0}$
 assumes $\text{finite } \{a. f\ a \neq 0\}$ (is finite ?F)
 assumes $\text{finite } \{b. g\ b \neq 0\}$ (is finite ?G)
 shows $\text{finite } \{a + b \mid a\ b. f\ a \neq 0 \wedge g\ b \neq 0\}$ (is finite ?FG)
 $\langle \text{proof} \rangle$

lemma *finite-mult-not-eq-zero-sumI*:

fixes $f\ g :: 'a::\text{monoid-add} \Rightarrow 'b::\text{semiring-0}$
 assumes $\text{finite } \{a. f\ a \neq 0\}$
 assumes $\text{finite } \{b. g\ b \neq 0\}$
 shows $\text{finite } \{a + b \mid a\ b. f\ a * g\ b \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-Sum-any-not-eq-zero-weakenI*:

assumes $\text{finite } \{a. \exists b. f\ a\ b \neq 0\}$
 shows $\text{finite } \{a. \text{Sum-any } (f\ a) \neq 0\}$
 $\langle \text{proof} \rangle$

context *zero*

begin

definition $\text{when} :: 'a \Rightarrow \text{bool} \Rightarrow 'a$ (infixl *when* 20)

where

$(a\ \text{when}\ P) = (\text{if } P\ \text{then } a\ \text{else } 0)$

Case distinctions always complicate matters, particularly when nested. The (*when*) operation allows to minimise these if $0::'a$ is the false-case value and makes proof obligations much more readable.

lemma *when [simp]*:

$P \Longrightarrow (a\ \text{when}\ P) = a$
 $\neg P \Longrightarrow (a\ \text{when}\ P) = 0$
 $\langle \text{proof} \rangle$

lemma *when-simps [simp]*:

$(a\ \text{when}\ \text{True}) = a$
 $(a\ \text{when}\ \text{False}) = 0$
 $\langle \text{proof} \rangle$

lemma *when-cong*:

assumes $P \longleftrightarrow Q$

and $Q \implies a = b$
shows $(a \text{ when } P) = (b \text{ when } Q)$
 $\langle \text{proof} \rangle$

lemma *zero-when* [simp]:
 $(0 \text{ when } P) = 0$
 $\langle \text{proof} \rangle$

lemma *when-when*:
 $(a \text{ when } P \text{ when } Q) = (a \text{ when } P \wedge Q)$
 $\langle \text{proof} \rangle$

lemma *when-commute*:
 $(a \text{ when } Q \text{ when } P) = (a \text{ when } P \text{ when } Q)$
 $\langle \text{proof} \rangle$

lemma *when-neq-zero* [simp]:
 $(a \text{ when } P) \neq 0 \longleftrightarrow P \wedge a \neq 0$
 $\langle \text{proof} \rangle$

end

context *monoid-add*
begin

lemma *when-add-distrib*:
 $(a + b \text{ when } P) = (a \text{ when } P) + (b \text{ when } P)$
 $\langle \text{proof} \rangle$

end

context *semiring-1*
begin

lemma *zero-power-eq*:
 $0 \wedge n = (1 \text{ when } n = 0)$
 $\langle \text{proof} \rangle$

end

context *comm-monoid-add*
begin

lemma *Sum-any-when-equal* [simp]:
 $(\sum a. (f a \text{ when } a = b)) = f b$
 $\langle \text{proof} \rangle$

lemma *Sum-any-when-equal'* [simp]:
 $(\sum a. (f a \text{ when } b = a)) = f b$

$\langle proof \rangle$

lemma *Sum-any-when-independent:*

$$(\sum a. g \ a \ \text{when } P) = ((\sum a. g \ a) \ \text{when } P)$$

$\langle proof \rangle$

lemma *Sum-any-when-dependent-prod-right:*

$$(\sum (a, b). g \ a \ \text{when } b = h \ a) = (\sum a. g \ a)$$

$\langle proof \rangle$

lemma *Sum-any-when-dependent-prod-left:*

$$(\sum (a, b). g \ b \ \text{when } a = h \ b) = (\sum b. g \ b)$$

$\langle proof \rangle$

end

context *cancel-comm-monoid-add*

begin

lemma *when-diff-distrib:*

$$(a - b \ \text{when } P) = (a \ \text{when } P) - (b \ \text{when } P)$$

$\langle proof \rangle$

end

context *group-add*

begin

lemma *when-uminus-distrib:*

$$(- \ a \ \text{when } P) = - \ (a \ \text{when } P)$$

$\langle proof \rangle$

end

context *mult-zero*

begin

lemma *mult-when:*

$$a * (b \ \text{when } P) = (a * b \ \text{when } P)$$

$\langle proof \rangle$

lemma *when-mult:*

$$(a \ \text{when } P) * b = (a * b \ \text{when } P)$$

$\langle proof \rangle$

end

72.2 Type definition

The following type is of central importance:

```
typedef (overloaded) ('a, 'b) poly-mapping ((-  $\Rightarrow_0$  /-) [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}  $\implies$  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  $\bigwedge k. \text{lookup } f \ k = \text{lookup } g \ k$ 
  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

$$\text{lookup} :: ('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow 'b$$

$$\text{Abs-poly-mapping} :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow_0 'b$$

satisfying

$$\text{Abs-poly-mapping} (\text{lookup } ?x) = ?x$$

$$\text{finite } \{x. ?f \ x \neq (0 :: ?'b)\} \implies \text{lookup } (\text{Abs-poly-mapping } ?f) = ?f$$

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 [?].

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.

72.3 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 $\lambda k. 0$
 $\langle proof \rangle$

instance $\langle proof \rangle$

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* ($\lambda x. \text{lookup}\ (pp\ x)\ i$) X
 ⟨proof⟩

instantiation *poly-mapping* :: (type, cancel-comm-monoid-add) cancel-comm-monoid-add
begin

lift-definition *minus-poly-mapping* :: ($'a \Rightarrow_0 'b$) $\Rightarrow$ ($'a \Rightarrow_0 'b$) $\Rightarrow$ $'a \Rightarrow_0 'b$
is $\lambda f1\ f2\ k. f1\ k - f2\ k$
 ⟨proof⟩

instance
 ⟨proof⟩

end

instantiation *poly-mapping* :: (type, ab-group-add) ab-group-add
begin

lift-definition *uminus-poly-mapping* :: ($'a \Rightarrow_0 'b$) $\Rightarrow$ $'a \Rightarrow_0 'b$
is *uminus*
 ⟨proof⟩

instance

$\langle proof \rangle$

end

lemma *lookup-uminus* [simp]:

$lookup\ (-f)\ k = -\ lookup\ f\ k$

$\langle proof \rangle$

lemma *lookup-minus*:

$lookup\ (f - g)\ k = lookup\ f\ k - lookup\ g\ k$

$\langle proof \rangle$

72.4 Multiplicative structure

instantiation *poly-mapping* :: (zero, zero-neq-one) zero-neq-one

begin

lift-definition *one-poly-mapping* :: 'a $\Rightarrow_0$ 'b

is $\lambda k. 1$ when $k = 0$

$\langle proof \rangle$

instance

$\langle proof \rangle$

end

lemma *lookup-one*:

$lookup\ 1\ k = (1\ \text{when}\ k = 0)$

$\langle proof \rangle$

lemma *lookup-one-zero* [simp]:

$lookup\ 1\ 0 = 1$

$\langle proof \rangle$

definition *prod-fun* :: ('a $\Rightarrow$ 'b) $\Rightarrow$ ('a $\Rightarrow$ 'b) $\Rightarrow$ 'a::monoid-add $\Rightarrow$ 'b::semiring-0

where

$prod-fun\ f1\ f2\ k = (\sum l. f1\ l * (\sum q. (f2\ q\ \text{when}\ k = l + q)))$

lemma *prod-fun-unfold-prod*:

fixes $f\ g :: 'a :: monoid-add \Rightarrow 'b :: semiring-0$

assumes *fin-f*: *finite* {a. $f\ a \neq 0$ }

assumes *fin-g*: *finite* {b. $g\ b \neq 0$ }

shows $prod-fun\ f\ g\ k = (\sum (a, b). f\ a * g\ b\ \text{when}\ k = a + b)$

$\langle proof \rangle$

lemma *finite-prod-fun*:

fixes $f1\ f2 :: 'a :: \text{monoid-add} \Rightarrow 'b :: \text{semiring-0}$
assumes $\text{fin1}: \text{finite } \{l. f1\ l \neq 0\}$
and $\text{fin2}: \text{finite } \{q. f2\ q \neq 0\}$
shows $\text{finite } \{k. \text{prod-fun } f1\ f2\ k \neq 0\}$
 $\langle \text{proof} \rangle$

instantiation $\text{poly-mapping} :: (\text{monoid-add}, \text{semiring-0}) \text{ semiring-0}$
begin

lift-definition $\text{times-poly-mapping} :: ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow_0 'b$
is prod-fun
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma lookup-mult :
 $\text{lookup } (f * g) \ k = (\sum l. \text{lookup } f \ l * (\sum q. \text{lookup } g \ q \text{ when } k = l + q))$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{comm-monoid-add}, \text{comm-semiring-0}) \text{ comm-semiring-0}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{monoid-add}, \text{semiring-0-cancel}) \text{ semiring-0-cancel}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{comm-monoid-add}, \text{comm-semiring-0-cancel}) \text{ comm-semiring-0-cancel}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{monoid-add}, \text{semiring-1}) \text{ semiring-1}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{comm-monoid-add}, \text{comm-semiring-1}) \text{ comm-semiring-1}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{monoid-add}, \text{semiring-1-cancel}) \text{ semiring-1-cancel}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{monoid-add}, \text{ring}) \text{ ring}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{comm-monoid-add}, \text{comm-ring}) \text{ comm-ring}$
 $\langle \text{proof} \rangle$

instance $\text{poly-mapping} :: (\text{monoid-add}, \text{ring-1}) \text{ ring-1}$
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring-1*) *comm-ring-1*
 ⟨*proof*⟩

72.5 Single-point mappings

lift-definition *single* :: 'a $\Rightarrow$ 'b $\Rightarrow$ 'a $\Rightarrow_0$ 'b::zero
is $\lambda k\ v\ k'.\ (v\ \text{when}\ k = k')$
 ⟨*proof*⟩

lemma *inj-single* [*iff*]:
 $\text{inj}\ (single\ k)$
 ⟨*proof*⟩

lemma *lookup-single*:
 $\text{lookup}\ (single\ k\ v)\ k' = (v\ \text{when}\ k = k')$
 ⟨*proof*⟩

lemma *lookup-single-eq* [*simp*]:
 $\text{lookup}\ (single\ k\ v)\ k = v$
 ⟨*proof*⟩

lemma *lookup-single-not-eq*:
 $k \neq k' \Longrightarrow \text{lookup}\ (single\ k\ v)\ k' = 0$
 ⟨*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*]:
 $single\ 0\ (\text{numeral}\ n) = \text{numeral}\ n$
 ⟨*proof*⟩

lemma *lookup-numeral*:

lookup (numeral n) k = (numeral n when k = 0)
 $\langle \text{proof} \rangle$

lemma *single-of-nat [simp]*:

single 0 (of-nat n) = of-nat n
 $\langle \text{proof} \rangle$

lemma *lookup-of-nat*:

lookup (of-nat n) k = (of-nat n when k = 0)
 $\langle \text{proof} \rangle$

lemma *of-nat-single*:

of-nat = single 0 $\circ$ of-nat
 $\langle \text{proof} \rangle$

lemma *mult-single*:

*single k a * single l b = single (k + l) (a * b)*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (monoid-add, semiring-char-0) semiring-char-0

$\langle \text{proof} \rangle$

instance *poly-mapping* :: (monoid-add, ring-char-0) ring-char-0

$\langle \text{proof} \rangle$

lemma *single-of-int [simp]*:

single 0 (of-int k) = of-int k
 $\langle \text{proof} \rangle$

lemma *lookup-of-int*:

lookup (of-int l) k = (of-int l when k = 0)
 $\langle \text{proof} \rangle$

72.6 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

$\langle \text{proof} \rangle$

instance *poly-mapping* :: ({ordered-cancel-comm-monoid-add, linorder}, ring-no-zero-divisors)

ring-no-zero-divisors

$\langle \text{proof} \rangle$

instance *poly-mapping* :: ({ordered-cancel-comm-monoid-add, linorder}, ring-1-no-zero-divisors)

ring-1-no-zero-divisors

$\langle proof \rangle$

instance *poly-mapping* :: ($\{ordered-cancel-comm-monoid-add, linorder\}, idom$) *idom*
 $\langle proof \rangle$

72.7 Mapping order

instantiation *poly-mapping* :: (*linorder*, $\{zero, linorder\}$) *linorder*
begin

lift-definition *less-poly-mapping* :: ($'a \Rightarrow_0 'b$) $\Rightarrow$ ($'a \Rightarrow_0 'b$) $\Rightarrow$ *bool*
is *less-fun*
 $\langle proof \rangle$

lift-definition *less-eq-poly-mapping* :: ($'a \Rightarrow_0 'b$) $\Rightarrow$ ($'a \Rightarrow_0 'b$) $\Rightarrow$ *bool*
is $\lambda f g. less-fun f g \vee f = g$
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

instance *poly-mapping* :: (*linorder*, $\{ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le, linorder\}$) *ordered-ab-semigroup-add*
 $\langle proof \rangle$

instance *poly-mapping* :: (*linorder*, $\{ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le, cancel-comm-monoid-add, linorder\}$) *linordered-cancel-ab-semigroup-add*
 $\langle proof \rangle$

instance *poly-mapping* :: (*linorder*, $\{ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le, cancel-comm-monoid-add, linorder\}$) *ordered-comm-monoid-add*
 $\langle proof \rangle$

instance *poly-mapping* :: (*linorder*, $\{ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le, cancel-comm-monoid-add, linorder\}$) *ordered-cancel-comm-monoid-add*
 $\langle proof \rangle$

instance *poly-mapping* :: (*linorder*, *linordered-ab-group-add*) *linordered-ab-group-add*
 $\langle proof \rangle$

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.

72.8 Fundamental mapping notions

lift-definition *keys* :: ($'a \Rightarrow_0 'b::zero$) $\Rightarrow$ *'a set*
is $\lambda f. \{k. f k \neq 0\}$ $\langle proof \rangle$

lift-definition $\text{range} :: ('a \Rightarrow_0 'b :: \text{zero}) \Rightarrow 'b \text{ set}$
is $\lambda f :: 'a \Rightarrow 'b. \text{Set.range } f - \{0\} \langle \text{proof} \rangle$

lemma finite-keys $[\text{simp}]$:
 $\text{finite } (\text{keys } f)$
 $\langle \text{proof} \rangle$

lemma $\text{not-in-keys-iff-lookup-eq-zero}$:
 $k \notin \text{keys } f \longleftrightarrow \text{lookup } f \ k = 0$
 $\langle \text{proof} \rangle$

lemma $\text{lookup-not-eq-zero-eq-in-keys}$:
 $\text{lookup } f \ k \neq 0 \longleftrightarrow k \in \text{keys } f$
 $\langle \text{proof} \rangle$

lemma $\text{lookup-eq-zero-in-keys-contradict}$ $[\text{dest}]$:
 $\text{lookup } f \ k = 0 \implies \neg k \in \text{keys } f$
 $\langle \text{proof} \rangle$

lemma finite-range $[\text{simp}]$: $\text{finite } (\text{Poly-Mapping.range } p)$
 $\langle \text{proof} \rangle$

lemma $\text{in-keys-lookup-in-range}$ $[\text{simp}]$:
 $k \in \text{keys } f \implies \text{lookup } f \ k \in \text{range } f$
 $\langle \text{proof} \rangle$

lemma in-keys-iff : $x \in (\text{keys } s) = (\text{lookup } s \ x \neq 0)$
 $\langle \text{proof} \rangle$

lemma keys-zero $[\text{simp}]$:
 $\text{keys } 0 = \{\}$
 $\langle \text{proof} \rangle$

lemma range-zero $[\text{simp}]$:
 $\text{range } 0 = \{\}$
 $\langle \text{proof} \rangle$

lemma keys-add :
 $\text{keys } (f + g) \subseteq \text{keys } f \cup \text{keys } g$
 $\langle \text{proof} \rangle$

lemma keys-one $[\text{simp}]$:
 $\text{keys } 1 = \{0\}$
 $\langle \text{proof} \rangle$

lemma range-one $[\text{simp}]$:
 $\text{range } 1 = \{1\}$
 $\langle \text{proof} \rangle$

lemma *keys-single* [simp]:

$keys (single\ k\ v) = (if\ v = 0\ then\ \{\}\ else\ \{k\})$
 $\langle proof \rangle$

lemma *range-single* [simp]:

$range (single\ k\ v) = (if\ v = 0\ then\ \{\}\ else\ \{v\})$
 $\langle proof \rangle$

lemma *keys-mult*:

$keys (f * g) \subseteq \{a + b \mid a\ b.\ a \in keys\ f \wedge b \in keys\ g\}$
 $\langle proof \rangle$

lemma *setsum-keys-plus-distrib*:

assumes *hom-0*: $\bigwedge k. f\ k\ 0 = 0$

and *hom-plus*: $\bigwedge k. k \in Poly-Mapping.keys\ p \cup Poly-Mapping.keys\ q \implies f\ k$
 $(Poly-Mapping.lookup\ p\ k + Poly-Mapping.lookup\ q\ k) = f\ k\ (Poly-Mapping.lookup\ p\ k) + f\ k\ (Poly-Mapping.lookup\ q\ k)$

shows

$(\sum_{k \in Poly-Mapping.keys\ (p + q)}. f\ k\ (Poly-Mapping.lookup\ (p + q)\ k)) =$
 $(\sum_{k \in Poly-Mapping.keys\ p}. f\ k\ (Poly-Mapping.lookup\ p\ k)) +$
 $(\sum_{k \in Poly-Mapping.keys\ q}. f\ k\ (Poly-Mapping.lookup\ q\ k))$
 $(is\ ?lhs = ?p + ?q)$

$\langle proof \rangle$

72.9 Degree

definition *degree* :: $(nat \Rightarrow_0 'a::zero) \Rightarrow nat$

where

$degree\ f = Max\ (insert\ 0\ (Suc\ ` keys\ f))$

lemma *degree-zero* [simp]:

$degree\ 0 = 0$
 $\langle proof \rangle$

lemma *degree-one* [simp]:

$degree\ 1 = 1$
 $\langle proof \rangle$

lemma *degree-single-zero* [simp]:

$degree (single\ k\ 0) = 0$
 $\langle proof \rangle$

lemma *degree-single-not-zero* [simp]:

$v \neq 0 \implies degree (single\ k\ v) = Suc\ k$
 $\langle proof \rangle$

lemma *degree-zero-iff* [simp]:

$degree\ f = 0 \longleftrightarrow f = 0$

<proof>

lemma *degree-greater-zero-in-keys:*

assumes $0 < \text{degree } f$

shows $\text{degree } f - 1 \in \text{keys } f$

<proof>

lemma *in-keys-less-degree:*

$n \in \text{keys } f \implies n < \text{degree } f$

<proof>

lemma *beyond-degree-lookup-zero:*

$\text{degree } f \leq n \implies \text{lookup } f \ n = 0$

<proof>

lemma *degree-add:*

$\text{degree } (f + g) \leq \max (\text{degree } f) (\text{Poly-Mapping.degree } g)$

<proof>

lemma *sorted-list-of-set-keys:*

$\text{sorted-list-of-set } (\text{keys } f) = \text{filter } (\lambda k. k \in \text{keys } f) [0..<\text{degree } f] \text{ (is - = ?r)}$

<proof>

72.10 Inductive structure

lift-definition *update* :: $'a \Rightarrow 'b \Rightarrow ('a \Rightarrow_0 'b::\text{zero}) \Rightarrow 'a \Rightarrow_0 'b$

is $\lambda k \ v \ f. f(k := v)$

<proof>

lemma *update-induct* [case-names const update]:

assumes *const'*: $P \ 0$

assumes *update'*: $\bigwedge f \ a \ b. a \notin \text{keys } f \implies b \neq 0 \implies P \ f \implies P \ (\text{update } a \ b \ f)$

shows $P \ f$

<proof>

lemma *lookup-update:*

$\text{lookup } (\text{update } k \ v \ f) \ k' = (\text{if } k = k' \text{ then } v \text{ else } \text{lookup } f \ k')$

<proof>

lemma *keys-update:*

$\text{keys } (\text{update } k \ v \ f) = (\text{if } v = 0 \text{ then } \text{keys } f - \{k\} \text{ else } \text{insert } k \ (\text{keys } f))$

<proof>

72.11 Quasi-functorial structure

lift-definition *map* :: $('b::\text{zero} \Rightarrow 'c::\text{zero})$

$\Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c::\text{zero})$

is $\lambda g \ f \ k. g \ (f \ k) \text{ when } f \ k \neq 0$

<proof>

```

context
  fixes  $f :: 'b \Rightarrow 'a$ 
  assumes  $\text{inj-}f: \text{inj } f$ 
begin

lift-definition  $\text{map-key} :: ('a \Rightarrow_0 'c::\text{zero}) \Rightarrow 'b \Rightarrow_0 'c$ 
  is  $\lambda p. p \circ f$ 
 $\langle \text{proof} \rangle$ 

end

lemma  $\text{map-key-compose}$ :
  assumes  $[\text{transfer-rule}]: \text{inj } f \text{ inj } g$ 
  shows  $\text{map-key } f (\text{map-key } g \ p) = \text{map-key } (g \circ f) \ p$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{map-key-id}$ :
   $\text{map-key } (\lambda x. x) \ p = p$ 
 $\langle \text{proof} \rangle$ 

context
  fixes  $f :: 'a \Rightarrow 'b$ 
  assumes  $\text{inj-}f \ [\text{transfer-rule}]: \text{inj } f$ 
begin

lemma  $\text{map-key-map}$ :
   $\text{map-key } f (\text{map } g \ p) = \text{map } g (\text{map-key } f \ p)$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{map-key-plus}$ :
   $\text{map-key } f (p + q) = \text{map-key } f \ p + \text{map-key } f \ q$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{keys-map-key}$ :
   $\text{keys } (\text{map-key } f \ p) = f - ' \text{keys } p$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{map-key-zero} \ [\text{simp}]$ :
   $\text{map-key } f \ 0 = 0$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{map-key-single} \ [\text{simp}]$ :
   $\text{map-key } f (\text{single } (f \ k) \ v) = \text{single } k \ v$ 
 $\langle \text{proof} \rangle$ 

end

lemma  $\text{mult-map-scale-conv-mult}$ :  $\text{map } ((*) \ s) \ p = \text{single } 0 \ s * p$ 
 $\langle \text{proof} \rangle$ 

```

lemma *map-single* [simp]:

$(c = 0 \implies f\ 0 = 0) \implies \text{map } f\ (\text{single } x\ c) = \text{single } x\ (f\ c)$
 ⟨proof⟩

lemma *map-eq-zero-iff*: $\text{map } f\ p = 0 \longleftrightarrow (\forall k \in \text{keys } p. f\ (\text{lookup } p\ k) = 0)$

⟨proof⟩

72.12 Canonical dense representation of $\text{nat} \Rightarrow_0 'a$

abbreviation *no-trailing-zeros* :: $'a :: \text{zero list} \Rightarrow \text{bool}$

where

$\text{no-trailing-zeros} \equiv \text{no-trailing } ((=)\ 0)$

lift-definition *nth* :: $'a\ \text{list} \Rightarrow (\text{nat} \Rightarrow_0 'a::\text{zero})$

is *nth-default* 0

⟨proof⟩

The opposite direction is directly specified on (later) type *nat-mapping*.

lemma *nth-Nil* [simp]:

$\text{nth } [] = 0$

⟨proof⟩

lemma *nth-singleton* [simp]:

$\text{nth } [v] = \text{single } 0\ v$

⟨proof⟩

lemma *nth-replicate* [simp]:

$\text{nth } (\text{replicate } n\ 0\ @\ [v]) = \text{single } n\ v$

⟨proof⟩

lemma *nth-strip-while* [simp]:

$\text{nth } (\text{strip-while } ((=)\ 0)\ xs) = \text{nth } xs$

⟨proof⟩

lemma *nth-strip-while'* [simp]:

$\text{nth } (\text{strip-while } (\lambda k. k = 0)\ xs) = \text{nth } xs$

⟨proof⟩

lemma *nth-eq-iff*:

$\text{nth } xs = \text{nth } ys \longleftrightarrow \text{strip-while } (\text{HOL.eq } 0)\ xs = \text{strip-while } (\text{HOL.eq } 0)\ ys$

⟨proof⟩

lemma *lookup-nth* [simp]:

$\text{lookup } (\text{nth } xs) = \text{nth-default } 0\ xs$

⟨proof⟩

lemma *keys-nth* [simp]:

$\text{keys } (\text{nth } xs) = \text{fst } ' \{ (n, v) \in \text{set } (\text{enumerate } 0\ xs). v \neq 0 \}$

⟨proof⟩

lemma *range-nth* [simp]:
 $\text{range } (\text{nth } xs) = \text{set } xs - \{0\}$
 ⟨proof⟩

lemma *degree-nth*:
 $\text{no-trailing-zeros } xs \implies \text{degree } (\text{nth } xs) = \text{length } xs$
 ⟨proof⟩

lemma *nth-trailing-zeros* [simp]:
 $\text{nth } (xs @ \text{replicate } n \ 0) = \text{nth } xs$
 ⟨proof⟩

lemma *nth-idem*:
 $\text{nth } (\text{List.map } (\text{lookup } f) \ [0..<\text{degree } f]) = f$
 ⟨proof⟩

lemma *nth-idem-bound*:
 assumes $\text{degree } f \leq n$
 shows $\text{nth } (\text{List.map } (\text{lookup } f) \ [0..<n]) = f$
 ⟨proof⟩

72.13 Canonical sparse representation of $'a \Rightarrow_0 'b$

lift-definition *the-value* :: $('a \times 'b) \text{ list} \Rightarrow 'a \Rightarrow_0 'b::\text{zero}$
 is $\lambda xs \ k. \text{case map-of } xs \ k \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } v \Rightarrow v$
 ⟨proof⟩

definition *items* :: $('a::\text{linorder} \Rightarrow_0 'b::\text{zero}) \Rightarrow ('a \times 'b) \text{ list}$
where
 $\text{items } f = \text{List.map } (\lambda k. (k, \text{lookup } f \ k)) \ (\text{sorted-list-of-set } (\text{keys } f))$

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]:
 $\text{the-value } (\text{items } f) = f$
 ⟨proof⟩

lemma *lookup-the-value*:
 $\text{lookup } (\text{the-value } xs) \ k = (\text{case map-of } xs \ k \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } v \Rightarrow v)$
 ⟨proof⟩

lemma *items-the-value*:
 assumes $\text{sorted } (\text{List.map fst } xs)$ **and** $\text{distinct } (\text{List.map fst } xs)$ **and** $0 \notin \text{snd } \text{'}$
 $\text{set } xs$
 shows $\text{items } (\text{the-value } xs) = xs$
 ⟨proof⟩

lemma *the-value-Nil* [simp]:

the-value [] = 0
 ⟨proof⟩

lemma *the-value-Cons* [simp]:

the-value (x # xs) = update (fst x) (snd x) (*the-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) ∈ set (*items* f) $\longleftrightarrow$ k ∈ keys f ∧ lookup f k = v
 ⟨proof⟩

72.14 Size estimation

context

fixes f :: 'a ⇒ nat
 and g :: 'b :: zero ⇒ nat
begin

definition *poly-mapping-size* :: ('a ⇒₀ 'b) ⇒ nat

where

poly-mapping-size m = g 0 + (∑ k ∈ keys m. Suc (f k + g (lookup m k)))

lemma *poly-mapping-size-0* [simp]:

poly-mapping-size 0 = g 0
 ⟨proof⟩

lemma *poly-mapping-size-single* [simp]:

poly-mapping-size (single k v) = (if v = 0 then g 0 else g 0 + f k + g v + 1)
 ⟨proof⟩

lemma *keys-less-poly-mapping-size*:

k ∈ keys m $\implies$ f k + g (lookup m k) < *poly-mapping-size* m
 ⟨proof⟩

lemma *lookup-le-poly-mapping-size*:

$g \text{ (lookup } m \text{ } k) \leq \text{poly-mapping-size } m$
 $\langle \text{proof} \rangle$

lemma *poly-mapping-size-estimation*:

$k \in \text{keys } m \implies y \leq f \text{ } k + g \text{ (lookup } m \text{ } k) \implies y < \text{poly-mapping-size } m$
 $\langle \text{proof} \rangle$

lemma *poly-mapping-size-estimation2*:

assumes $v \in \text{range } m$ **and** $y \leq g \text{ } v$

shows $y < \text{poly-mapping-size } m$

$\langle \text{proof} \rangle$

end

lemma *poly-mapping-size-one* [simp]:

$\text{poly-mapping-size } f \text{ } g \text{ } 1 = g \text{ } 0 + f \text{ } 0 + g \text{ } 1 + 1$

$\langle \text{proof} \rangle$

lemma *poly-mapping-size-cong* [fundef-cong]:

$m = m' \implies g \text{ } 0 = g' \text{ } 0 \implies (\bigwedge k. k \in \text{keys } m' \implies f \text{ } k = f' \text{ } k)$

$\implies (\bigwedge v. v \in \text{range } m' \implies g \text{ } v = g' \text{ } v)$

$\implies \text{poly-mapping-size } f \text{ } g \text{ } m = \text{poly-mapping-size } f' \text{ } g' \text{ } m'$

$\langle \text{proof} \rangle$

instantiation *poly-mapping* :: (type, zero) size

begin

definition *size* = *poly-mapping-size* ($\lambda \cdot. 0$) ($\lambda \cdot. 0$)

instance $\langle \text{proof} \rangle$

end

72.15 Further mapping operations and properties

It is like in algebra: there are many definitions, some are also used

lift-definition *mapp* ::

$('a \Rightarrow 'b :: \text{zero} \Rightarrow 'c :: \text{zero}) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c)$

is $\lambda f \text{ } p \text{ } k. (\text{if } k \in \text{keys } p \text{ then } f \text{ } k \text{ (lookup } p \text{ } k) \text{ else } 0)$

$\langle \text{proof} \rangle$

lemma *mapp-cong* [fundef-cong]:

$\llbracket m = m'; \bigwedge k. k \in \text{keys } m' \implies f \text{ } k \text{ (lookup } m' \text{ } k) = f' \text{ } k \text{ (lookup } m' \text{ } k) \rrbracket$

$\implies \text{mapp } f \text{ } m = \text{mapp } f' \text{ } m'$

$\langle \text{proof} \rangle$

lemma *lookup-mapp*:

$\text{lookup } (\text{mapp } f \text{ } p) \text{ } k = (f \text{ } k \text{ (lookup } p \text{ } k) \text{ when } k \in \text{keys } p)$

$\langle \text{proof} \rangle$

lemma *keys-mapp-subset*: $\text{keys } (\text{mapp } f \ p) \subseteq \text{keys } p$
 $\langle \text{proof} \rangle$

72.16 Free Abelian Groups Over a Type

abbreviation *frag-of* :: $'a \Rightarrow 'a \Rightarrow_0 \text{int}$
where *frag-of* $c \equiv \text{Poly-Mapping.single } c \ (1::\text{int})$

lemma *lookup-frag-of* [*simp*]:
 $\text{Poly-Mapping.lookup}(\text{frag-of } c) = (\lambda x. \text{if } x = c \text{ then } 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *frag-of-nonzero* [*simp*]: $\text{frag-of } a \neq 0$
 $\langle \text{proof} \rangle$

definition *frag-cmul* :: $\text{int} \Rightarrow ('a \Rightarrow_0 \text{int}) \Rightarrow ('a \Rightarrow_0 \text{int})$
where *frag-cmul* $c \ a = \text{Abs-poly-mapping } (\lambda x. c * \text{Poly-Mapping.lookup } a \ x)$

lemma *frag-cmul-zero* [*simp*]: $\text{frag-cmul } 0 \ x = 0$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-zero2* [*simp*]: $\text{frag-cmul } c \ 0 = 0$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-one* [*simp*]: $\text{frag-cmul } 1 \ x = x$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-minus-one* [*simp*]: $\text{frag-cmul } (-1) \ x = -x$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-cmul* [*simp*]: $\text{frag-cmul } c \ (\text{frag-cmul } d \ x) = \text{frag-cmul } (c*d) \ x$
 $\langle \text{proof} \rangle$

lemma *lookup-frag-cmul* [*simp*]: $\text{poly-mapping.lookup } (\text{frag-cmul } c \ x) \ i = c * \text{poly-mapping.lookup } x \ i$
 $\langle \text{proof} \rangle$

lemma *minus-frag-cmul* [*simp*]: $-\text{frag-cmul } k \ x = \text{frag-cmul } (-k) \ x$
 $\langle \text{proof} \rangle$

lemma *keys-frag-of*: $\text{Poly-Mapping.keys}(\text{frag-of } a) = \{a\}$
 $\langle \text{proof} \rangle$

lemma *finite-cmul-nonzero*: $\text{finite } \{x. c * \text{Poly-Mapping.lookup } a \ x \neq (0::\text{int})\}$
 $\langle \text{proof} \rangle$

lemma *keys-cmul*: $\text{Poly-Mapping.keys}(\text{frag-cmul } c \ a) \subseteq \text{Poly-Mapping.keys } a$
 $\langle \text{proof} \rangle$

lemma *keys-cmul-iff* [iff]: $i \in \text{Poly-Mapping.keys} (\text{frag-cmul } c \ x) \longleftrightarrow i \in \text{Poly-Mapping.keys } x \wedge c \neq 0$
 ⟨proof⟩

lemma *keys-minus* [simp]: $\text{Poly-Mapping.keys}(-a) = \text{Poly-Mapping.keys } a$
 ⟨proof⟩

lemma *keys-diff*:
 $\text{Poly-Mapping.keys}(a - b) \subseteq \text{Poly-Mapping.keys } a \cup \text{Poly-Mapping.keys } b$
 ⟨proof⟩

lemma *keys-eq-empty* [simp]: $\text{Poly-Mapping.keys } c = \{\} \longleftrightarrow c = 0$
 ⟨proof⟩

lemma *frag-cmul-eq-0-iff* [simp]: $\text{frag-cmul } k \ c = 0 \longleftrightarrow k=0 \vee c=0$
 ⟨proof⟩

lemma *frag-of-eq*: $\text{frag-of } x = \text{frag-of } y \longleftrightarrow x = y$
 ⟨proof⟩

lemma *frag-cmul-distrib*: $\text{frag-cmul } (c+d) \ a = \text{frag-cmul } c \ a + \text{frag-cmul } d \ a$
 ⟨proof⟩

lemma *frag-cmul-distrib2*: $\text{frag-cmul } c \ (a+b) = \text{frag-cmul } c \ a + \text{frag-cmul } c \ b$
 ⟨proof⟩

lemma *frag-cmul-diff-distrib*: $\text{frag-cmul } (a - b) \ c = \text{frag-cmul } a \ c - \text{frag-cmul } b \ c$
 ⟨proof⟩

lemma *frag-cmul-sum*:
 $\text{frag-cmul } a \ (\text{sum } b \ I) = (\sum i \in I. \text{frag-cmul } a \ (b \ i))$
 ⟨proof⟩

lemma *keys-sum*: $\text{Poly-Mapping.keys}(\text{sum } b \ I) \subseteq (\bigcup i \in I. \text{Poly-Mapping.keys}(b \ i))$
 ⟨proof⟩

definition *frag-extend* :: $('b \Rightarrow 'a \Rightarrow_0 \text{int}) \Rightarrow ('b \Rightarrow_0 \text{int}) \Rightarrow 'a \Rightarrow_0 \text{int}$
where $\text{frag-extend } b \ x \equiv (\sum i \in \text{Poly-Mapping.keys } x. \text{frag-cmul } (\text{Poly-Mapping.lookup } x \ i) \ (b \ i))$

lemma *frag-extend-0* [simp]: $\text{frag-extend } b \ 0 = 0$
 ⟨proof⟩

lemma *frag-extend-of* [simp]: $\text{frag-extend } f \ (\text{frag-of } a) = f \ a$
 ⟨proof⟩

lemma *frag-extend-cmul*:

$$\text{frag-extend } f \text{ (frag-cmul } c \text{ } x) = \text{frag-cmul } c \text{ (frag-extend } f \text{ } x)$$

<proof>

lemma *frag-extend-minus*:

$$\text{frag-extend } f \text{ (} - x \text{)} = - \text{(frag-extend } f \text{ } x)$$

<proof>

lemma *frag-extend-add*:

$$\text{frag-extend } f \text{ (} a+b \text{)} = \text{(frag-extend } f \text{ } a) + \text{(frag-extend } f \text{ } b)$$

<proof>

lemma *frag-extend-diff*:

$$\text{frag-extend } f \text{ (} a-b \text{)} = \text{(frag-extend } f \text{ } a) - \text{(frag-extend } f \text{ } b)$$

<proof>

lemma *frag-extend-sum*:

$$\text{finite } I \implies \text{frag-extend } f \text{ (} \sum_{i \in I} g \text{ } i \text{)} = \text{sum (frag-extend } f \text{ } o \text{ } g) \text{ } I$$

<proof>

lemma *frag-extend-eq*:

$$(\bigwedge f. f \in \text{Poly-Mapping.keys } c \implies g \text{ } f = h \text{ } f) \implies \text{frag-extend } g \text{ } c = \text{frag-extend } h \text{ } c$$

<proof>

lemma *frag-extend-eq-0*:

$$(\bigwedge x. x \in \text{Poly-Mapping.keys } c \implies f \text{ } x = 0) \implies \text{frag-extend } f \text{ } c = 0$$

<proof>

lemma *keys-frag-extend*: $\text{Poly-Mapping.keys}(\text{frag-extend } f \text{ } c) \subseteq (\bigcup x \in \text{Poly-Mapping.keys } c. \text{Poly-Mapping.keys}(f \text{ } x))$

<proof>

lemma *frag-expansion*: $a = \text{frag-extend frag-of } a$

<proof>

lemma *frag-closure-minus-cmul*:

assumes $P \text{ } 0$ **and** $P: \bigwedge x \ y. \llbracket P \text{ } x; P \text{ } y \rrbracket \implies P(x - y) \text{ } P \text{ } c$

shows $P(\text{frag-cmul } k \text{ } c)$

<proof>

lemma *frag-induction* [*consumes 1, case-names zero one diff*]:

assumes $\text{supp: Poly-Mapping.keys } c \subseteq S$

and $0: P \text{ } 0$ **and** *sing*: $\bigwedge x. x \in S \implies P(\text{frag-of } x)$

and *diff*: $\bigwedge a \ b. \llbracket P \text{ } a; P \text{ } b \rrbracket \implies P(a - b)$

shows $P \text{ } c$

<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 $c :: 'a \Rightarrow_0 \text{int}$
assumes *Poly-Mapping.keys* $c \subseteq S \cup T$
obtains d e **where** *Poly-Mapping.keys* $d \subseteq S$ *Poly-Mapping.keys* $e \subseteq T$ $d + e$
 = c
 ⟨*proof*⟩

hide-const (**open**) *lookup single update keys range map map-key degree nth the-value items foldr mapp*

end

73 Exponentiation by Squaring

theory *Power-By-Squaring*

imports *Main*

begin

context

fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$

begin

function *efficient-funpow* $:: 'a \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a$ **where**

efficient-funpow y x $0 = y$
 | *efficient-funpow* y x (*Suc* 0) = f x y
 | $n \neq 0 \implies \text{even } n \implies \text{efficient-funpow } y$ x $n = \text{efficient-funpow } y$ (f x x) ($n \text{ div } 2$)
 | $n \neq 1 \implies \text{odd } n \implies \text{efficient-funpow } y$ x $n = \text{efficient-funpow } (f$ x $y)$ (f x x) ($n \text{ div } 2$)
 ⟨*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) ($n \text{ div } 2$)
 else *efficient-funpow* (f x y) (f x x) ($n \text{ div } 2$))
 ⟨*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 $\widehat{\hspace{0.5em}}^n$) y

$\langle proof \rangle$

context *monoid-mult*
begin

lemma *power-by-squaring: efficient-funpow* $(*) (1 :: 'a) = (\wedge)$
 $\langle proof \rangle$

end

end

74 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 $((-/ \approx -) [51, 51] 50)$

lemma *equivD1*: $x \leq y$ **if** $x \approx y$
 $\langle proof \rangle$

lemma *equivD2*: $y \leq x$ **if** $x \approx y$
 $\langle proof \rangle$

lemma *equiv-refl* [*iff*]: $x \approx x$
 $\langle proof \rangle$

lemma *equiv-sym*: $x \approx y \longleftrightarrow y \approx x$
 $\langle proof \rangle$

lemma *equiv-trans*: $x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$
 $\langle proof \rangle$

lemma *equiv-antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
 $\langle proof \rangle$

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge \neg x \approx y$
 $\langle proof \rangle$

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x \approx y$
 $\langle \text{proof} \rangle$

lemma *le-imp-less-or-equiv*: $x \leq y \implies x < y \vee x \approx y$
 $\langle \text{proof} \rangle$

lemma *less-imp-not-equiv*: $x < y \implies \neg x \approx y$
 $\langle \text{proof} \rangle$

lemma *not-equiv-le-trans*: $\neg a \approx b \implies a \leq b \implies a < b$
 $\langle \text{proof} \rangle$

lemma *le-not-equiv-trans*: $a \leq b \implies \neg a \approx b \implies a < b$
 $\langle \text{proof} \rangle$

lemma *antisym-conv*: $y \leq x \implies x \leq y \longleftrightarrow x \approx y$
 $\langle \text{proof} \rangle$

end

$\langle ML \rangle$

end

75 Additive group operations on product types

theory *Product-Plus*
imports *Main*
begin

75.1 Operations

instantiation *prod* :: (*zero*, *zero*) *zero*
begin

definition *zero-prod-def*: $0 = (0, 0)$

instance $\langle \text{proof} \rangle$
end

instantiation *prod* :: (*plus*, *plus*) *plus*
begin

definition *plus-prod-def*:
 $x + y = (\text{fst } x + \text{fst } y, \text{snd } x + \text{snd } y)$

instance $\langle \text{proof} \rangle$
end

instantiation *prod* :: (*minus*, *minus*) *minus*
begin

definition *minus-prod-def*:
 $x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$

instance $\langle proof \rangle$
end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*
begin

definition *uminus-prod-def*:
 $- x = (-\ fst\ x, -\ snd\ x)$

instance $\langle proof \rangle$
end

lemma *fst-zero* [*simp*]: $fst\ 0 = 0$
 $\langle proof \rangle$

lemma *snd-zero* [*simp*]: $snd\ 0 = 0$
 $\langle proof \rangle$

lemma *fst-add* [*simp*]: $fst\ (x + y) = fst\ x + fst\ y$
 $\langle proof \rangle$

lemma *snd-add* [*simp*]: $snd\ (x + y) = snd\ x + snd\ y$
 $\langle proof \rangle$

lemma *fst-diff* [*simp*]: $fst\ (x - y) = fst\ x - fst\ y$
 $\langle proof \rangle$

lemma *snd-diff* [*simp*]: $snd\ (x - y) = snd\ x - snd\ y$
 $\langle proof \rangle$

lemma *fst-uminus* [*simp*]: $fst\ (-\ x) = -\ fst\ x$
 $\langle proof \rangle$

lemma *snd-uminus* [*simp*]: $snd\ (-\ x) = -\ snd\ x$
 $\langle proof \rangle$

lemma *add-Pair* [*simp*]: $(a, b) + (c, d) = (a + c, b + d)$
 $\langle proof \rangle$

lemma *diff-Pair* [*simp*]: $(a, b) - (c, d) = (a - c, b - d)$
 $\langle proof \rangle$

lemma *uminus-Pair* [*simp*, *code*]: $-(a, b) = (-a, -b)$
 ⟨*proof*⟩

75.2 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*: $\text{fst} (\sum x \in A. f\ x) = (\sum x \in A. \text{fst} (f\ x))$
 ⟨*proof*⟩

lemma *snd-sum*: $\text{snd} (\sum x \in A. f\ x) = (\sum x \in A. \text{snd} (f\ x))$
 ⟨*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

76 Roots of real quadratics

theory *Quadratic-Discriminant*
imports *Complex-Main*
begin

definition *discrim* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real* $\Rightarrow$ *real*
where *discrim a b c* $\equiv b^2 - 4 * a * c$

lemma *complete-square*:

$a \neq 0 \implies a * x^2 + b * x + c = 0 \iff (2 * a * x + b)^2 = \text{discrim } a \ b \ c$
 $\langle \text{proof} \rangle$

lemma *discriminant-negative*:

fixes *a b c x* :: *real*
assumes $a \neq 0$
and $\text{discrim } a \ b \ c < 0$
shows $a * x^2 + b * x + c \neq 0$
 $\langle \text{proof} \rangle$

lemma *plus-or-minus-sqrt*:

fixes *x y* :: *real*
assumes $y \geq 0$
shows $x^2 = y \iff x = \text{sqrt } y \vee x = - \text{sqrt } y$
 $\langle \text{proof} \rangle$

lemma *divide-non-zero*:

fixes *x y z* :: *real*
assumes $x \neq 0$
shows $x * y = z \iff y = z / x$
 $\langle \text{proof} \rangle$

lemma *discriminant-nonneg*:

fixes *a b c x* :: *real*
assumes $a \neq 0$
and $\text{discrim } a \ b \ c \geq 0$
shows $a * x^2 + b * x + c = 0 \iff$
 $x = (-b + \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a) \vee$
 $x = (-b - \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a)$
 $\langle \text{proof} \rangle$

lemma *discriminant-zero*:

fixes *a b c x* :: *real*
assumes $a \neq 0$
and $\text{discrim } a \ b \ c = 0$
shows $a * x^2 + b * x + c = 0 \iff x = -b / (2 * a)$
 $\langle \text{proof} \rangle$

theorem *discriminant-iff*:

fixes *a b c x* :: *real*
assumes $a \neq 0$
shows $a * x^2 + b * x + c = 0 \iff$
 $\text{discrim } a \ b \ c \geq 0 \wedge$
 $(x = (-b + \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a) \vee$
 $x = (-b - \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a))$

$\langle proof \rangle$

lemma *discriminant-nonneg-ex*:

fixes $a\ b\ c :: real$

assumes $a \neq 0$

and $discrim\ a\ b\ c \geq 0$

shows $\exists x. a * x^2 + b * x + c = 0$

$\langle proof \rangle$

lemma *discriminant-pos-ex*:

fixes $a\ b\ c :: real$

assumes $a \neq 0$

and $discrim\ a\ b\ c > 0$

shows $\exists x\ y. x \neq y \wedge a * x^2 + b * x + c = 0 \wedge a * y^2 + b * y + c = 0$

$\langle proof \rangle$

lemma *discriminant-pos-distinct*:

fixes $a\ b\ c\ x :: real$

assumes $a \neq 0$

and $discrim\ a\ b\ c > 0$

shows $\exists y. x \neq y \wedge a * y^2 + b * y + c = 0$

$\langle proof \rangle$

lemma *Rats-solution-QE*:

assumes $a \in \mathbb{Q}\ b \in \mathbb{Q}\ a \neq 0$

and $a * x^2 + b * x + c = 0$

and $\sqrt{discrim\ a\ b\ c} \in \mathbb{Q}$

shows $x \in \mathbb{Q}$

$\langle proof \rangle$

lemma *Rats-solution-QE-converse*:

assumes $a \in \mathbb{Q}\ b \in \mathbb{Q}$

and $a * x^2 + b * x + c = 0$

and $x \in \mathbb{Q}$

shows $\sqrt{discrim\ a\ b\ c} \in \mathbb{Q}$

$\langle proof \rangle$

end

77 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

78 Quotient infrastructure for the set type

```
theory Quotient-Set
imports Quotient-Syntax
begin
```

78.1 Contravariant set map (vimage) and set relator, rules for the Quotient package

definition *rel-vset* $R\ xs\ ys \equiv \forall x\ y. R\ x\ y \longrightarrow x \in xs \longleftrightarrow y \in ys$

lemma *rel-vset-eq* [*id-simps*]:
 $rel-vset\ (=) = (=)$
<proof>

lemma *rel-vset-equivp*:
assumes $e: equivp\ R$
shows $rel-vset\ R\ xs\ ys \longleftrightarrow xs = ys \wedge (\forall x\ y. x \in xs \longrightarrow R\ x\ y \longrightarrow y \in xs)$
<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 $(rel-vset\ R == => rel-vset\ R == => rel-vset\ R)\ (\cup)\ (\cup)$
<proof>

```

lemma union-prs[quot-preserve]:
  assumes Quotient3 R Abs Rep
  shows  $((-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Rep)\ (\cup) = (\cup)$ 
   $\langle proof \rangle$ 

lemma diff-rsp[quot-respect]:
  assumes Quotient3 R Abs Rep
  shows  $(rel\text{-}vset\ R \implies rel\text{-}vset\ R \implies rel\text{-}vset\ R)\ (-)\ (-)$ 
   $\langle proof \rangle$ 

lemma diff-prs[quot-preserve]:
  assumes Quotient3 R Abs Rep
  shows  $((-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Rep)\ (-) = (-)$ 
   $\langle proof \rangle$ 

lemma inter-rsp[quot-respect]:
  assumes Quotient3 R Abs Rep
  shows  $(rel\text{-}vset\ R \implies rel\text{-}vset\ R \implies rel\text{-}vset\ R)\ (\cap)\ (\cap)$ 
   $\langle proof \rangle$ 

lemma inter-prs[quot-preserve]:
  assumes Quotient3 R Abs Rep
  shows  $((-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Rep)\ (\cap) = (\cap)$ 
   $\langle proof \rangle$ 

lemma mem-prs[quot-preserve]:
  assumes Quotient3 R Abs Rep
  shows  $(Rep \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow id)\ (\in) = (\in)$ 
   $\langle proof \rangle$ 

lemma mem-rsp[quot-respect]:
  shows  $(R \implies rel\text{-}vset\ R \implies (=))\ (\in)\ (\in)$ 
   $\langle proof \rangle$ 

end

```

79 Quotient infrastructure for the product type

```

theory Quotient-Product
imports Quotient-Syntax
begin

```

79.1 Rules for the Quotient package

```

lemma map-prod-id [id-simps]:
  shows  $map\text{-}prod\ id\ id = id$ 
   $\langle proof \rangle$ 

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

```

80 Quotient infrastructure for the option type

```

theory Quotient-Option
imports Quotient-Syntax
begin

```

80.1 Rules for the Quotient package

```

lemma rel-option-map1:
  rel-option R (map-option f x) y ⟷ rel-option (λx. R (f x)) x y

```

$\langle \text{proof} \rangle$

lemma *rel-option-map2*:

$\text{rel-option } R \ x \ (\text{map-option } f \ y) \longleftrightarrow \text{rel-option } (\lambda x \ y. \ R \ x \ (f \ y)) \ x \ y$

$\langle \text{proof} \rangle$

declare

map-option.id [*id-simps*]

option.rel-eq [*id-simps*]

lemma *reflp-rel-option*:

$\text{reflp } R \implies \text{reflp } (\text{rel-option } R)$

$\langle \text{proof} \rangle$

lemma *option-symp*:

$\text{symp } R \implies \text{symp } (\text{rel-option } R)$

$\langle \text{proof} \rangle$

lemma *option-transp*:

$\text{transp } R \implies \text{transp } (\text{rel-option } R)$

$\langle \text{proof} \rangle$

lemma *option-equivp* [*quot-equiv*]:

$\text{equivp } R \implies \text{equivp } (\text{rel-option } R)$

$\langle \text{proof} \rangle$

lemma *option-quotient* [*quot-thm*]:

assumes *Quotient3* *R Abs Rep*

shows *Quotient3* $(\text{rel-option } R) \ (\text{map-option } \text{Abs}) \ (\text{map-option } \text{Rep})$

$\langle \text{proof} \rangle$

declare $[[\text{mapQ3 } \text{option} = (\text{rel-option}, \text{option-quotient})]]$

lemma *option-None-rsp* [*quot-respect*]:

assumes *q: Quotient3 R Abs Rep*

shows *rel-option R None None*

$\langle \text{proof} \rangle$

lemma *option-Some-rsp* [*quot-respect*]:

assumes *q: Quotient3 R Abs Rep*

shows $(R \implies \text{rel-option } R) \ \text{Some} \ \text{Some}$

$\langle \text{proof} \rangle$

lemma *option-None-prs* [*quot-preserve*]:

assumes *q: Quotient3 R Abs Rep*

shows *map-option Abs None = None*

$\langle \text{proof} \rangle$

lemma *option-Some-prs* [*quot-preserve*]:

```

assumes  $q$ : Quotient3  $R$   $Abs$   $Rep$ 
shows ( $Rep \dashv\dashv\dashv> map-option\ Abs$ )  $Some = Some$ 
   $\langle proof \rangle$ 

end

```

81 Quotient infrastructure for the list type

```

theory Quotient-List
imports Quotient-Set Quotient-Product Quotient-Option
begin

```

81.1 Rules for the Quotient package

```

lemma map-id [id-simps]:
   $map\ id = id$ 
   $\langle proof \rangle$ 

```

```

lemma list-all2-eq [id-simps]:
   $list-all2\ (=) = (=)$ 
   $\langle proof \rangle$ 

```

```

lemma reflp-list-all2:
  assumes reflp  $R$ 
  shows reflp ( $list-all2\ R$ )
   $\langle proof \rangle$ 

```

```

lemma list-symp:
  assumes symp  $R$ 
  shows symp ( $list-all2\ R$ )
   $\langle proof \rangle$ 

```

```

lemma list-transp:
  assumes transp  $R$ 
  shows transp ( $list-all2\ R$ )
   $\langle proof \rangle$ 

```

```

lemma list-equivp [quot-equiv]:
   $equivp\ R \implies equivp\ (list-all2\ R)$ 
   $\langle proof \rangle$ 

```

```

lemma list-quotient3 [quot-thm]:
  assumes Quotient3  $R$   $Abs$   $Rep$ 
  shows Quotient3 ( $list-all2\ R$ ) ( $map\ Abs$ ) ( $map\ Rep$ )
   $\langle proof \rangle$ 

```

```

declare [[mapQ3 list = ( $list-all2$ ,  $list-quotient3$ )]]

```

```

lemma cons-prs [quot-preserve]:

```

assumes q : *Quotient3* R Abs Rep
shows $(Rep \dashv\dashv\dashv (map\ Rep) \dashv\dashv\dashv (map\ Abs))\ (\#) = (\#)$
 $\langle proof \rangle$

lemma *cons-rsp* [*quot-respect*]:
assumes q : *Quotient3* R Abs Rep
shows $(R \implies list\text{-}all2\ R \implies list\text{-}all2\ R)\ (\#)\ (\#)$
 $\langle proof \rangle$

lemma *nil-prs* [*quot-preserve*]:
assumes q : *Quotient3* R Abs Rep
shows $map\ Abs\ [] = []$
 $\langle proof \rangle$

lemma *nil-rsp* [*quot-respect*]:
assumes q : *Quotient3* R Abs Rep
shows $list\text{-}all2\ R\ []\ []$
 $\langle proof \rangle$

lemma *map-prs-aux*:
assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $(map\ abs2)\ (map\ ((abs1 \dashv\dashv\dashv rep2)\ f)\ (map\ rep1\ l)) = map\ f\ l$
 $\langle proof \rangle$

lemma *map-prs* [*quot-preserve*]:
assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $((abs1 \dashv\dashv\dashv rep2) \dashv\dashv\dashv (map\ rep1) \dashv\dashv\dashv (map\ abs2))\ map = map$
and $((abs1 \dashv\dashv\dashv id) \dashv\dashv\dashv map\ rep1 \dashv\dashv\dashv id)\ map = map$
 $\langle proof \rangle$

lemma *map-rsp* [*quot-respect*]:
assumes $q1$: *Quotient3* $R1$ $Abs1$ $Rep1$
and $q2$: *Quotient3* $R2$ $Abs2$ $Rep2$
shows $((R1 \implies R2) \implies (list\text{-}all2\ R1 \implies list\text{-}all2\ R2))\ map\ map$
and $((R1 \implies (=)) \implies (list\text{-}all2\ R1 \implies (=))\ map\ map$
 $\langle proof \rangle$

lemma *foldr-prs-aux*:
assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $abs2\ (foldr\ ((abs1 \dashv\dashv\dashv abs2 \dashv\dashv\dashv rep2)\ f)\ (map\ rep1\ l)\ (rep2\ e))$
 $= foldr\ f\ l\ e$
 $\langle proof \rangle$

lemma *foldr-prs* [*quot-preserve*]:
assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$

shows $((abs1 \dashrightarrow abs2 \dashrightarrow rep2) \dashrightarrow (map\ rep1) \dashrightarrow rep2 \dashrightarrow abs2) \text{ foldr} = \text{foldr}$
 ⟨proof⟩

lemma *foldl-prs-aux*:

assumes $a: \text{Quotient3 } R1\ abs1\ rep1$
and $b: \text{Quotient3 } R2\ abs2\ rep2$
shows $abs1\ (\text{foldl } ((abs1 \dashrightarrow abs2 \dashrightarrow rep1)\ f)\ (rep1\ e)\ (map\ rep2\ l)) = \text{foldl } f\ e\ l$
 ⟨proof⟩

lemma *foldl-prs [quot-preserve]*:

assumes $a: \text{Quotient3 } R1\ abs1\ rep1$
and $b: \text{Quotient3 } R2\ abs2\ rep2$
shows $((abs1 \dashrightarrow abs2 \dashrightarrow rep1) \dashrightarrow rep1 \dashrightarrow (map\ rep2) \dashrightarrow abs1) \text{ foldl} = \text{foldl}$
 ⟨proof⟩

lemma *foldl-rsp[quot-respect]*:

assumes $q1: \text{Quotient3 } R1\ Abs1\ Rep1$
and $q2: \text{Quotient3 } R2\ Abs2\ Rep2$
shows $((R1 \implies R2 \implies R1) \implies R1 \implies \text{list-all2 } R2 \implies R1)$
 foldl foldl
 ⟨proof⟩

lemma *foldr-rsp[quot-respect]*:

assumes $q1: \text{Quotient3 } R1\ Abs1\ Rep1$
and $q2: \text{Quotient3 } R2\ Abs2\ Rep2$
shows $((R1 \implies R2 \implies R2) \implies \text{list-all2 } R1 \implies R2 \implies R2)$
 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 $l1: \text{list-all2 } R\ x\ y$
and $l2: \text{list-all2 } R\ a\ b$
shows $\text{list-all2 } S\ x\ a = \text{list-all2 } T\ y\ b$
 ⟨proof⟩

lemma *[quot-respect]*:

$((R \implies R \implies (=)) \implies \text{list-all2 } R \implies \text{list-all2 } R \implies (=))$
 $\text{list-all2 list-all2}$
 ⟨proof⟩

lemma *[quot-preserve]*:

assumes $a: \text{Quotient3 } R\ abs1\ rep1$
shows $((abs1 \dashrightarrow abs1 \dashrightarrow id) \dashrightarrow map\ rep1 \dashrightarrow map\ rep1 \dashrightarrow id) \text{ list-all2} = \text{list-all2}$
 ⟨proof⟩

lemma *[quot-preserve]*:
assumes a : *Quotient3* R *abs1* *rep1*
shows $(\text{list-all2 } ((\text{rep1} \dashrightarrow \text{rep1} \dashrightarrow \text{id}) R) l m) = (l = m)$
 $\langle \text{proof} \rangle$

lemma *list-all2-find-element*:
assumes a : $x \in \text{set } a$
and b : *list-all2* R a b
shows $\exists y. (y \in \text{set } b \wedge R x y)$
 $\langle \text{proof} \rangle$

lemma *list-all2-refl*:
assumes a : $\bigwedge x y. R x y = (R x = R y)$
shows *list-all2* R x x
 $\langle \text{proof} \rangle$

end

82 Quotient infrastructure for the sum type

theory *Quotient-Sum*
imports *Quotient-Syntax*
begin

82.1 Rules for the Quotient package

lemma *rel-sum-map1*:
 $\text{rel-sum } R1 \ R2 \ (\text{map-sum } f1 \ f2 \ x) \ y \longleftrightarrow \text{rel-sum } (\lambda x. R1 \ (f1 \ x)) \ (\lambda x. R2 \ (f2 \ x))$
 $x \ y$
 $\langle \text{proof} \rangle$

lemma *rel-sum-map2*:
 $\text{rel-sum } R1 \ R2 \ x \ (\text{map-sum } f1 \ f2 \ y) \longleftrightarrow \text{rel-sum } (\lambda x \ y. R1 \ x \ (f1 \ y)) \ (\lambda x \ y. R2 \ x \ (f2 \ y))$
 $x \ y$
 $\langle \text{proof} \rangle$

lemma *map-sum-id [id-simps]*:
 $\text{map-sum } id \ id = id$
 $\langle \text{proof} \rangle$

lemma *rel-sum-eq [id-simps]*:
 $\text{rel-sum } (=) \ (=) = (=)$
 $\langle \text{proof} \rangle$

lemma *reflp-rel-sum*:
 $\text{reflp } R1 \implies \text{reflp } R2 \implies \text{reflp } (\text{rel-sum } R1 \ R2)$
 $\langle \text{proof} \rangle$

lemma *sum-symp*:

symp R1 $\implies$ symp R2 $\implies$ symp (rel-sum R1 R2)
<proof>

lemma *sum-transp*:

transp R1 $\implies$ transp R2 $\implies$ transp (rel-sum R1 R2)
<proof>

lemma *sum-equivp [quot-equiv]*:

equivp R1 $\implies$ equivp R2 $\implies$ 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 $\implies\implies\implies$ 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 $\implies\implies\implies$ 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 $\dashrightarrow$ 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 $\dashrightarrow$ map-sum Abs1 Abs2) Inr = Inr*
<proof>

end

83 Quotient types

theory *Quotient-Type*

```
imports Main
begin
```

We introduce the notion of quotient types over equivalence relations via type classes.

83.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a \Rightarrow 'a \Rightarrow \text{bool}$.

```
class equiv =
  fixes eqv :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infixl  $\sim$  50)

class equiv = eqv +
  assumes equiv-refl [intro]:  $x \sim x$ 
  and equiv-trans [trans]:  $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$ 
  and equiv-sym [sym]:  $x \sim y \Longrightarrow y \sim x$ 
begin

lemma equiv-not-sym [sym]:  $\neg x \sim y \Longrightarrow \neg y \sim x$ 
  <proof>

lemma not-equiv-trans1 [trans]:  $\neg x \sim y \Longrightarrow y \sim z \Longrightarrow \neg x \sim z$ 
  <proof>

lemma not-equiv-trans2 [trans]:  $x \sim y \Longrightarrow \neg y \sim z \Longrightarrow \neg x \sim z$ 
  <proof>

end

The quotient type 'a quot consists of all equivalence classes over elements
of the base type 'a.

definition (in eqv) quot =  $\{\{x. a \sim x\} \mid a. \text{True}\}$ 

typedef (overloaded) 'a quot = quot :: 'a::eqv set set
  <proof>

lemma quotI [intro]:  $\{x. a \sim x\} \in \text{quot}$ 
  <proof>

lemma quotE [elim]:
  assumes  $R \in \text{quot}$ 
  obtains  $a$  where  $R = \{x. a \sim x\}$ 
  <proof>
```

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

```
definition class :: 'a::equiv  $\Rightarrow$  'a quot ([_])
  where  $[a] = \text{Abs-quot } \{x. a \sim x\}$ 
```

theorem *quot-exhaust*: $\exists a. A = \lfloor a \rfloor$
 $\langle \text{proof} \rangle$

lemma *quot-cases* [*cases type: quot*]:
obtains *a* **where** $A = \lfloor a \rfloor$
 $\langle \text{proof} \rangle$

83.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $\lfloor a \rfloor = \lfloor b \rfloor \longleftrightarrow a \sim b$
 $\langle \text{proof} \rangle$

83.3 Picking representing elements

definition *pick* :: $'a::\text{equiv quot} \Rightarrow 'a$
where *pick* $A = (\text{SOME } a. A = \lfloor a \rfloor)$

theorem *pick-equiv* [*intro*]: *pick* $\lfloor a \rfloor \sim a$
 $\langle \text{proof} \rangle$

theorem *pick-inverse* [*intro*]: $\lfloor \text{pick } A \rfloor = A$
 $\langle \text{proof} \rangle$

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 $\text{eq}: \bigwedge X Y. P X Y \Longrightarrow f X Y \equiv g (\text{pick } X) (\text{pick } Y)$
and $\text{cong}: \bigwedge x x' y y'. \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$
 $\Longrightarrow P \lfloor x \rfloor \lfloor y \rfloor \Longrightarrow P \lfloor x' \rfloor \lfloor y' \rfloor \Longrightarrow g x y = g x' y'$
and $P: P \lfloor a \rfloor \lfloor b \rfloor$
shows $f \lfloor a \rfloor \lfloor b \rfloor = g a b$
 $\langle \text{proof} \rangle$

theorem *quot-function*:
assumes $\bigwedge X Y. f X Y \equiv g (\text{pick } X) (\text{pick } Y)$
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 $f \lfloor a \rfloor \lfloor b \rfloor = g a b$
 $\langle \text{proof} \rangle$

theorem *quot-function'*:
 $(\bigwedge X Y. f X Y \equiv g (\text{pick } X) (\text{pick } Y)) \Longrightarrow$
 $(\bigwedge x x' y y'. x \sim x' \Longrightarrow y \sim y' \Longrightarrow g x y = g x' y') \Longrightarrow$
 $f \lfloor a \rfloor \lfloor b \rfloor = g a b$
 $\langle \text{proof} \rangle$

end

84 Ramsey’s Theorem

```
theory Ramsey
  imports Infinite-Set FuncSet
begin
```

84.1 Preliminary definitions

abbreviation *strict-sorted* :: ‘a::linorder list $\Rightarrow$ bool **where**
strict-sorted $\equiv$ sorted-wrt ($<$)

84.1.1 The n -element subsets of a set A

definition *nsets* :: ‘a set, nat] $\Rightarrow$ ‘a set set (([$\cdot$]) [0,999] 999)
where *nsets* $A\ n \equiv \{N. N \subseteq A \wedge \text{finite } N \wedge \text{card } N = n\}$

lemma *nsets-mono*: $A \subseteq B \implies \text{nsets } A\ n \subseteq \text{nsets } B\ n$
 $\langle \text{proof} \rangle$

lemma *nsets-Pi-contr*: $A' \subseteq A \implies \text{Pi } ([A]^n)\ B \subseteq \text{Pi } ([A']^n)\ B$
 $\langle \text{proof} \rangle$

lemma *nsets-2-eq*: $\text{nsets } A\ 2 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \{\{x, y\}\})$
 $\langle \text{proof} \rangle$

lemma *nsets-doubleton-2-eq* [simp]: $[\{x, y\}]^2 = (\text{if } x=y \text{ then } \{\} \text{ else } \{\{x, y\}\})$
 $\langle \text{proof} \rangle$

lemma *doubleton-in-nsets-2* [simp]: $\{x, y\} \in [A]^2 \longleftrightarrow x \in A \wedge y \in A \wedge x \neq y$
 $\langle \text{proof} \rangle$

lemma *nsets-3-eq*: $\text{nsets } A\ 3 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \bigcup z \in A - \{x, y\}. \{\{x, y, z\}\})$
 $\langle \text{proof} \rangle$

lemma *nsets-4-eq*: $[A]^4 = (\bigcup u \in A. \bigcup x \in A - \{u\}. \bigcup y \in A - \{u, x\}. \bigcup z \in A - \{u, x, y\}. \{\{u, x, y, z\}\})$
 (is - = ?rhs)
 $\langle \text{proof} \rangle$

lemma *nsets-disjoint-2*:
 $X \cap Y = \{\} \implies [X \cup Y]^2 = [X]^2 \cup [Y]^2 \cup (\bigcup x \in X. \bigcup y \in Y. \{\{x, y\}\})$
 $\langle \text{proof} \rangle$

lemma *ordered-nsets-2-eq*:
fixes $A :: 'a::linorder \text{ set}$
shows $\text{nsets } A\ 2 = \{\{x, y\} \mid x \neq y. x \in A \wedge y \in A \wedge x < y\}$
 (is - = ?rhs)
 $\langle \text{proof} \rangle$

lemma *ordered-nsets-3-eq*:

fixes $A :: 'a::linorder\ set$
shows $nsets\ A\ \mathcal{P} = \{\{x,y,z\} \mid x\ y\ z.\ x \in A \wedge y \in A \wedge z \in A \wedge x < y \wedge y < z\}$
 (is - = ?rhs)
 $\langle proof \rangle$

lemma *ordered-nsets-4-eq*:
fixes $A :: 'a::linorder\ set$
shows $[A]^4 = \{U.\ \exists u\ x\ y\ z.\ U = \{u,x,y,z\} \wedge u \in A \wedge x \in A \wedge y \in A \wedge z \in A$
 $\wedge u < x \wedge x < y \wedge y < z\}$
 (is - = Collect ?RHS)
 $\langle proof \rangle$

lemma *ordered-nsets-5-eq*:
fixes $A :: 'a::linorder\ set$
shows $[A]^5 = \{U.\ \exists u\ v\ x\ y\ z.\ U = \{u,v,x,y,z\} \wedge u \in A \wedge v \in A \wedge x \in A \wedge y$
 $\in A \wedge z \in A \wedge u < v \wedge v < x \wedge x < y \wedge y < z\}$
 (is - = Collect ?RHS)
 $\langle proof \rangle$

lemma *binomial-eq-nsets*: $n\ choose\ k = card\ (nsets\ \{0..<n\}\ k)$
 $\langle proof \rangle$

lemma *nsets-eq-empty-iff*: $nsets\ A\ r = \{\} \longleftrightarrow finite\ A \wedge card\ A < r$
 $\langle proof \rangle$

lemma *nsets-eq-empty*: $\llbracket finite\ A; card\ A < r \rrbracket \Longrightarrow nsets\ A\ r = \{\}$
 $\langle proof \rangle$

lemma *nsets-empty-iff*: $nsets\ \{\}\ r = (if\ r=0\ then\ \{\{\}\}\ else\ \{\})$
 $\langle proof \rangle$

lemma *nsets-singleton-iff*: $nsets\ \{a\}\ r = (if\ r=0\ then\ \{\{\}\}\ else\ if\ r=1\ then\ \{\{a\}\}$
 $else\ \{\})$
 $\langle proof \rangle$

lemma *nsets-self [simp]*: $nsets\ \{..<m\}\ m = \{\{..<m\}\}$
 $\langle proof \rangle$

lemma *nsets-zero [simp]*: $nsets\ A\ 0 = \{\{\}\}$
 $\langle proof \rangle$

lemma *nsets-one*: $nsets\ A\ (Suc\ 0) = (\lambda x.\ \{x\})\ 'A$
 $\langle proof \rangle$

lemma *inj-on-nsets*:
assumes *inj-on* $f\ A$
shows *inj-on* $(\lambda X.\ f\ 'X)\ ([A]^n)$
 $\langle proof \rangle$

lemma *bij-betw-nsets*:
assumes *bij-betw* f A B
shows *bij-betw* $(\lambda X. f \text{ ‘ } X)$ $([A]^n)$ $([B]^n)$
 $\langle proof \rangle$

lemma *nset-image-obtains*:
assumes $X \in [f \text{ ‘ } A]^k$ *inj-on* f A
obtains Y **where** $Y \in [A]^k$ $X = f \text{ ‘ } Y$
 $\langle proof \rangle$

lemma *nsets-image-funcset*:
assumes $g \in S \rightarrow T$ **and** *inj-on* g S
shows $(\lambda X. g \text{ ‘ } X) \in [S]^k \rightarrow [T]^k$
 $\langle proof \rangle$

lemma *nsets-compose-image-funcset*:
assumes $f: f \in [T]^k \rightarrow D$ **and** $g \in S \rightarrow T$ **and** *inj-on* g S
shows $f \circ (\lambda X. g \text{ ‘ } X) \in [S]^k \rightarrow D$
 $\langle proof \rangle$

84.1.2 Partition predicates

definition *partn* :: ‘*a set* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ ‘*b set* $\Rightarrow$ *bool*
where *partn* β α γ $\delta \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \delta. \exists H \in \text{nsets } \beta \alpha. \exists \xi \in \delta. f \text{ ‘ } (\text{nsets } H \gamma) \subseteq \{\xi\}$

definition *partn-lst* :: ‘*a set* $\Rightarrow$ *nat list* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where *partn-lst* β α $\gamma \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \{..<\text{length } \alpha\}.$
 $\exists i < \text{length } \alpha. \exists H \in \text{nsets } \beta (\alpha!i). f \text{ ‘ } (\text{nsets } H \gamma) \subseteq \{i\}$

lemma *partn-lst-greater-resource*:
fixes $M::\text{nat}$
assumes $M: \text{partn-lst } \{..<M\} \alpha \gamma$ **and** $M \leq N$
shows *partn-lst* $\{..<N\} \alpha \gamma$
 $\langle proof \rangle$

lemma *partn-lst-fewer-colours*:
assumes *major*: *partn-lst* β $(n\#\alpha)$ γ **and** $n \geq \gamma$
shows *partn-lst* β α γ
 $\langle proof \rangle$

lemma *partn-lst-eq-partn*: *partn-lst* $\{..<n\} [m,m] \text{ } 2 = \text{partn } \{..<n\} m \text{ } 2 \{..<2::\text{nat}\}$
 $\langle proof \rangle$

84.2 Finite versions of Ramsey’s theorem

To distinguish the finite and infinite ones, lower and upper case names are used.

84.2.1 Trivial cases

Vacuous, since we are dealing with 0-sets!

lemma *ramsey0*: $\exists N::nat. \text{partn-lst } \{..<N\} [q1, q2] \ 0$
 $\langle \text{proof} \rangle$

Just the pigeon hole principle, since we are dealing with 1-sets

lemma *ramsey1*: $\exists N::nat. \text{partn-lst } \{..<N\} [q0, q1] \ 1$
 $\langle \text{proof} \rangle$

84.2.2 Ramsey’s theorem with two colours and arbitrary exponents (hypergraph version)

proposition *ramsey2-full*: $\exists N::nat. \text{partn-lst } \{..<N\} [q1, q2] \ r$
 $\langle \text{proof} \rangle$

84.2.3 Full Ramsey’s theorem with multiple colours and arbitrary exponents

theorem *ramsey-full*: $\exists N::nat. \text{partn-lst } \{..<N\} \ q \ s \ r$
 $\langle \text{proof} \rangle$

84.2.4 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* $V \ E \longleftrightarrow (\forall v \in V. \forall w \in V. v \neq w \longrightarrow \{v, w\} \in E)$

definition *indep* $V \ E \longleftrightarrow (\forall v \in V. \forall w \in V. v \neq w \longrightarrow \{v, w\} \notin E)$

lemma *ramsey2*:

$\exists r \geq 1. \forall (V::'a \text{ set}) (E::'a \text{ set set}). \text{finite } V \wedge \text{card } V \geq r \longrightarrow$

$(\exists R \subseteq V. \text{card } R = m \wedge \text{clique } R \ E \vee \text{card } R = n \wedge \text{indep } R \ E)$

$\langle \text{proof} \rangle$

84.3 Preliminaries

84.3.1 “Axiom” of Dependent Choice

primrec *choice* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow \text{nat} \Rightarrow 'a$

where — An integer-indexed chain of choices

choice-0: $\text{choice } P \ r \ 0 = (\text{SOME } x. P \ x)$

| *choice-Suc*: $\text{choice } P \ r \ (\text{Suc } n) = (\text{SOME } y. P \ y \wedge (\text{choice } P \ r \ n, y) \in r)$

lemma *choice-n*:

assumes $P0$: $P \ x0$

and *Pstep*: $\bigwedge x. P \ x \Longrightarrow \exists y. P \ y \wedge (x, y) \in r$

shows $P \ (\text{choice } P \ r \ n)$

$\langle \text{proof} \rangle$

lemma *dependent-choice*:

assumes *trans*: $\text{trans } r$
and $P0: P\ x0$
and $Pstep: \bigwedge x. P\ x \implies \exists y. P\ y \wedge (x, y) \in r$
obtains $f :: \text{nat} \Rightarrow 'a$ **where** $\bigwedge n. P\ (f\ n)$ **and** $\bigwedge n\ m. n < m \implies (f\ n, f\ m) \in r$
 $\langle \text{proof} \rangle$

84.3.2 Partition functions

definition $\text{part-fn} :: \text{nat} \Rightarrow \text{nat} \Rightarrow 'a\ \text{set} \Rightarrow ('a\ \text{set} \Rightarrow \text{nat}) \Rightarrow \text{bool}$

— the function f partitions the r -subsets of the typically infinite set Y into s distinct categories.

where $\text{part-fn } r\ s\ Y\ f \longleftrightarrow (f \in \text{nsets } Y\ r \rightarrow \{..<s\})$

For induction, we decrease the value of r in partitions.

lemma *part-fn-Suc-imp-part-fn*:

$\llbracket \text{infinite } Y; \text{part-fn } (\text{Suc } r)\ s\ Y\ f; y \in Y \rrbracket \implies \text{part-fn } r\ s\ (Y - \{y\})\ (\lambda u. f\ (\text{insert } y\ u))$
 $\langle \text{proof} \rangle$

lemma *part-fn-subset*: $\text{part-fn } r\ s\ YY\ f \implies Y \subseteq YY \implies \text{part-fn } r\ s\ Y\ f$
 $\langle \text{proof} \rangle$

84.4 Ramsey’s Theorem: Infinitary Version

lemma *Ramsey-induction*:

fixes $s\ r :: \text{nat}$
and $YY :: 'a\ \text{set}$
and $f :: 'a\ \text{set} \Rightarrow \text{nat}$
assumes $\text{infinite } YY\ \text{part-fn } r\ s\ YY\ f$
shows $\exists Y'\ t'. Y' \subseteq YY \wedge \text{infinite } Y' \wedge t' < s \wedge (\forall X. X \subseteq Y' \wedge \text{finite } X \wedge \text{card } X = r \longrightarrow f\ X = t')$
 $\langle \text{proof} \rangle$

theorem *Ramsey*:

fixes $s\ r :: \text{nat}$
and $Z :: 'a\ \text{set}$
and $f :: 'a\ \text{set} \Rightarrow \text{nat}$
shows
 $\llbracket \text{infinite } Z;$
 $\forall X. X \subseteq Z \wedge \text{finite } X \wedge \text{card } X = r \longrightarrow f\ X < s \rrbracket$
 $\implies \exists Y\ t. Y \subseteq Z \wedge \text{infinite } Y \wedge t < s$
 $\wedge (\forall X. X \subseteq Y \wedge \text{finite } X \wedge \text{card } X = r \longrightarrow f\ X = t)$
 $\langle \text{proof} \rangle$

corollary *Ramsey2*:

fixes $s :: \text{nat}$
and $Z :: 'a\ \text{set}$
and $f :: 'a\ \text{set} \Rightarrow \text{nat}$

assumes $\text{inf}Z$: $\text{infinite } Z$
and part : $\forall x \in Z. \forall y \in Z. x \neq y \longrightarrow f \{x, y\} < s$
shows $\exists Y t. Y \subseteq Z \wedge \text{infinite } Y \wedge t < s \wedge (\forall x \in Y. \forall y \in Y. x \neq y \longrightarrow f \{x, y\} = t)$
 $\langle \text{proof} \rangle$

corollary Ramsey-nsets :
fixes $f :: 'a \text{ set} \Rightarrow \text{nat}$
assumes $\text{infinite } Z f \text{ ' nsets } Z r \subseteq \{..<s\}$
obtains $Y t \text{ where } Y \subseteq Z \text{ infinite } Y t < s f \text{ ' nsets } Y r \subseteq \{t\}$
 $\langle \text{proof} \rangle$

84.5 Disjunctive Well-Foundedness

An application of Ramsey’s theorem to program termination. See [3].

definition $\text{disj-wf} :: ('a \times 'a) \text{ set} \Rightarrow \text{bool}$
where $\text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i < n. \text{wf } (T i)) \wedge r = (\bigcup i < n. T i))$

definition $\text{transition-idx} :: (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow ('a \times 'a) \text{ set}) \Rightarrow \text{nat set} \Rightarrow \text{nat}$
where $\text{transition-idx } s T A = (\text{LEAST } k. \exists i j. A = \{i, j\} \wedge i < j \wedge (s j, s i) \in T k)$

lemma $\text{transition-idx-less}$:
assumes $i < j (s j, s i) \in T k k < n$
shows $\text{transition-idx } s T \{i, j\} < n$
 $\langle \text{proof} \rangle$

lemma transition-idx-in :
assumes $i < j (s j, s i) \in T k$
shows $(s j, s i) \in T (\text{transition-idx } s T \{i, j\})$
 $\langle \text{proof} \rangle$

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma disj-wf : $\text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i < n. \text{wf}(T i)) \wedge r \subseteq (\bigcup i < n. T i))$
 $\langle \text{proof} \rangle$

theorem $\text{trans-disj-wf-implies-wf}$:
assumes $\text{trans } r$
and $\text{disj-wf } r$
shows $\text{wf } r$
 $\langle \text{proof} \rangle$

end

85 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 ≡ λx. 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
```

86 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'(-)))⟩)

translations LENGTH('a) ⇐
  CONST len-of (CONST Pure.type :: 'a itself)
```

⟨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]:
   $LENGTH('a) \neq 0$ 
   $\langle proof \rangle$ 

end

instantiation num0 and num1 :: len0
begin

definition len-num0:  $len-of \ (- :: num0 \ itself) = 0$ 
definition len-num1:  $len-of \ (- :: num1 \ itself) = 1$ 

instance  $\langle proof \rangle$ 

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  $\langle proof \rangle$ 

end

lemmas len-of-numeral-defs [simp] = len-num0 len-num1 len-bit0 len-bit1

instance num1 :: len
   $\langle proof \rangle$ 
instance bit0 :: (len) len
   $\langle proof \rangle$ 
instance bit1 :: (len0) len
   $\langle proof \rangle$ 

instantiation Enum.finite-1 :: len
begin

definition
   $len-of-finite-1 \ (x :: Enum.finite-1 \ itself) \equiv (1 :: nat)$ 

instance
   $\langle proof \rangle$ 

```

end

instantiation *Enum.finite-2* :: *len*
begin

definition

len-of-finite-2 (*x* :: *Enum.finite-2* itself) $\equiv$ (*2* :: *nat*)

instance

<proof>

end

instantiation *Enum.finite-3* :: *len*
begin

definition

len-of-finite-3 (*x* :: *Enum.finite-3* itself) $\equiv$ (*4* :: *nat*)

instance

<proof>

end

lemma *length-not-greater-eq-2-iff* [*simp*]:

$\langle \neg 2 \leq \text{LENGTH}('a::\text{len}) \longleftrightarrow \text{LENGTH}('a) = 1 \rangle$

<proof>

context *linordered-idom*

begin

lemma *two-less-eq-exp-length* [*simp*]:

$\langle 2 \leq 2 \wedge \text{LENGTH}('b::\text{len}) \rangle$

<proof>

end

lemma *less-eq-decr-length-iff* [*simp*]:

$\langle n \leq \text{LENGTH}('a::\text{len}) - \text{Suc } 0 \longleftrightarrow n < \text{LENGTH}('a) \rangle$

<proof>

lemma *decr-length-less-iff* [*simp*]:

$\langle \text{LENGTH}('a::\text{len}) - \text{Suc } 0 < n \longleftrightarrow \text{LENGTH}('a) \leq n \rangle$

<proof>

end

87 Saturated arithmetic

```
theory Saturated
imports Numeral-Type Type-Length
begin
```

87.1 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  $\implies$  m = n
  <proof>
```

```
lemma sat-eq-iff:
  m = n  $\longleftrightarrow$  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  $\Rightarrow$  '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)  $\leq$  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

$$\text{less-eq-sat-def: } x \leq y \longleftrightarrow \text{nat-of } x \leq \text{nat-of } y$$
definition

$$\text{less-sat-def: } x < y \longleftrightarrow \text{nat-of } x < \text{nat-of } y$$
instance

$$\langle \text{proof} \rangle$$
end

instantiation $\text{sat} :: (\text{len}) \{ \text{minus}, \text{comm-semiring-1} \}$
begin

definition

$$0 = \text{Abs-sat}' 0$$
definition

$$1 = \text{Abs-sat}' 1$$

lemma nat-of-zero-sat [*simp, code abstract*]:

$$\text{nat-of } 0 = 0$$

$$\langle \text{proof} \rangle$$

lemma nat-of-one-sat [*simp, code abstract*]:

$$\text{nat-of } 1 = \min 1 (\text{LENGTH}('a))$$

$$\langle \text{proof} \rangle$$
definition

$$x + y = \text{Abs-sat}' (\text{nat-of } x + \text{nat-of } y)$$

lemma nat-of-plus-sat [*simp, code abstract*]:

$$\text{nat-of } (x + y) = \min (\text{nat-of } x + \text{nat-of } y) (\text{LENGTH}('a))$$

$$\langle \text{proof} \rangle$$
definition

$$x - y = \text{Abs-sat}' (\text{nat-of } x - \text{nat-of } y)$$

lemma nat-of-minus-sat [*simp, code abstract*]:

$$\text{nat-of } (x - y) = \text{nat-of } x - \text{nat-of } y$$

$$\langle \text{proof} \rangle$$
definition

$$x * y = \text{Abs-sat}' (\text{nat-of } x * \text{nat-of } y)$$

lemma nat-of-times-sat [*simp, code abstract*]:

$$\text{nat-of } (x * y) = \min (\text{nat-of } x * \text{nat-of } y) (\text{LENGTH}('a))$$

$$\langle \text{proof} \rangle$$

instance

$\langle proof \rangle$

end

instantiation $sat :: (len) \text{ ordered-comm-semiring}$
begin

instance

$\langle proof \rangle$

end

lemma $Abs\text{-}sat'\text{-}eq\text{-}of\text{-}nat$: $Abs\text{-}sat' \ n = of\text{-}nat \ n$
 $\langle proof \rangle$

abbreviation $Sat :: nat \Rightarrow 'a::len \ sat$ **where**
 $Sat \equiv of\text{-}nat$

lemma $nat\text{-}of\text{-}Sat$ $[simp]$:
 $nat\text{-}of \ (Sat \ n :: ('a::len) \ sat) = min \ (LENGTH('a)) \ n$
 $\langle proof \rangle$

lemma $[code\text{-}abbrev]$:
 $of\text{-}nat \ (numeral \ k) = (numeral \ k :: 'a::len \ sat)$
 $\langle proof \rangle$

context

begin

qualified definition $sat\text{-}of\text{-}nat :: nat \Rightarrow ('a::len) \ sat$
where $[code\text{-}abbrev]$: $sat\text{-}of\text{-}nat = of\text{-}nat$

lemma $[code \ abstract]$:
 $nat\text{-}of \ (sat\text{-}of\text{-}nat \ n :: ('a::len) \ sat) = min \ (LENGTH('a)) \ n$
 $\langle proof \rangle$

end

instance $sat :: (len) \ finite$
 $\langle proof \rangle$

instantiation $sat :: (len) \ equal$
begin

definition $HOL.equal \ A \ B \longleftrightarrow nat\text{-}of \ A = nat\text{-}of \ B$

instance

$\langle proof \rangle$


```

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⟩

end

```

88 Set Idioms

```

theory Set-Idioms
imports Countable-Set

```

```

begin

```

88.1 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 ≡ λS. ∃U. P U ∧ U ⊆ Collect Q ∧ ⋃U = S

```

definition *intersection-of* :: ('a set set $\Rightarrow$ bool) $\Rightarrow$ ('a set $\Rightarrow$ bool) $\Rightarrow$ 'a set $\Rightarrow$ bool
 (infixr *intersection'-of* 60)
where *P intersection-of* *Q* $\equiv \lambda S. \exists \mathcal{U}. P \mathcal{U} \wedge \mathcal{U} \subseteq \text{Collect } Q \wedge \bigcap \mathcal{U} = S$

definition *arbitrary*:: 'a set set $\Rightarrow$ bool **where** *arbitrary* *U* $\equiv \text{True}$

lemma *union-of-inc*: $\llbracket P \{S\}; Q \ S \rrbracket \Longrightarrow (P \text{ union-of } Q) \ S$
 $\langle \text{proof} \rangle$

lemma *intersection-of-inc*:
 $\llbracket P \{S\}; Q \ S \rrbracket \Longrightarrow (P \text{ intersection-of } Q) \ S$
 $\langle \text{proof} \rangle$

lemma *union-of-mono*:
 $\llbracket (P \text{ union-of } Q) \ S; \bigwedge x. Q \ x \Longrightarrow Q' \ x \rrbracket \Longrightarrow (P \text{ union-of } Q') \ S$
 $\langle \text{proof} \rangle$

lemma *intersection-of-mono*:
 $\llbracket (P \text{ intersection-of } Q) \ S; \bigwedge x. Q \ x \Longrightarrow Q' \ x \rrbracket \Longrightarrow (P \text{ intersection-of } Q') \ S$
 $\langle \text{proof} \rangle$

lemma *all-union-of*:
 $(\forall S. (P \text{ union-of } Q) \ S \longrightarrow R \ S) \longleftrightarrow (\forall T. P \ T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcup T))$
 $\langle \text{proof} \rangle$

lemma *all-intersection-of*:
 $(\forall S. (P \text{ intersection-of } Q) \ S \longrightarrow R \ S) \longleftrightarrow (\forall T. P \ T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcap T))$
 $\langle \text{proof} \rangle$

lemma *intersection-ofE*:
 $\llbracket (P \text{ intersection-of } Q) \ S; \bigwedge T. \llbracket P \ T; T \subseteq \text{Collect } Q \rrbracket \Longrightarrow R(\bigcap T) \rrbracket \Longrightarrow R \ S$
 $\langle \text{proof} \rangle$

lemma *union-of-empty*:
 $P \ \{\} \Longrightarrow (P \text{ union-of } Q) \ \{\}$
 $\langle \text{proof} \rangle$

lemma *intersection-of-empty*:
 $P \ \{\} \Longrightarrow (P \text{ intersection-of } Q) \ \text{UNIV}$
 $\langle \text{proof} \rangle$

The arbitrary and finite cases

lemma *arbitrary-union-of-alt*:
 $(\text{arbitrary union-of } Q) \ S \longleftrightarrow (\forall x \in S. \exists U. Q \ U \wedge x \in U \wedge U \subseteq S)$
 (is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *arbitrary-union-of-empty* [simp]: $(\text{arbitrary union-of } P) \ \{\}$
 $\langle \text{proof} \rangle$

lemma *arbitrary-intersection-of-empty* [simp]:
 $(\text{arbitrary intersection-of } P) \ \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *arbitrary-union-of-inc*:
 $P \ S \implies (\text{arbitrary union-of } P) \ S$
 $\langle \text{proof} \rangle$

lemma *arbitrary-intersection-of-inc*:
 $P \ S \implies (\text{arbitrary intersection-of } P) \ S$
 $\langle \text{proof} \rangle$

lemma *arbitrary-union-of-complement*:
 $(\text{arbitrary union-of } P) \ S \longleftrightarrow (\text{arbitrary intersection-of } (\lambda S. P(- S))) \ (- S)$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *arbitrary-intersection-of-complement*:
 $(\text{arbitrary intersection-of } P) \ S \longleftrightarrow (\text{arbitrary union-of } (\lambda S. P(- S))) \ (- S)$
 $\langle \text{proof} \rangle$

lemma *arbitrary-union-of-idempot* [simp]:
 $\text{arbitrary union-of arbitrary union-of } P = \text{arbitrary union-of } P$
 $\langle \text{proof} \rangle$

lemma *arbitrary-intersection-of-idempot*:
 $\text{arbitrary intersection-of arbitrary intersection-of } P = \text{arbitrary intersection-of } P$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *arbitrary-union-of-Union*:
 $(\bigwedge S. S \in \mathcal{U} \implies (\text{arbitrary union-of } P) \ S) \implies (\text{arbitrary union-of } P) \ (\bigcup \mathcal{U})$
 $\langle \text{proof} \rangle$

lemma *arbitrary-union-of-Un*:
 $\llbracket (\text{arbitrary union-of } P) \ S; (\text{arbitrary union-of } P) \ T \rrbracket$
 $\implies (\text{arbitrary union-of } P) \ (S \cup T)$
 $\langle \text{proof} \rangle$

lemma *arbitrary-intersection-of-Inter*:
 $(\bigwedge S. S \in \mathcal{U} \implies (\text{arbitrary intersection-of } P) \ S) \implies (\text{arbitrary intersection-of } P) \ (\bigcap \mathcal{U})$
 $\langle \text{proof} \rangle$

lemma *arbitrary-intersection-of-Int*:
 $\llbracket (\text{arbitrary intersection-of } P) \ S; (\text{arbitrary intersection-of } P) \ T \rrbracket$

$$\begin{array}{c} \implies (\text{arbitrary intersection-of } P) (S \cap T) \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-union-of-Int-eq*:

$$\begin{array}{c} (\forall S T. (\text{arbitrary union-of } P) S \wedge (\text{arbitrary union-of } P) T \\ \quad \longrightarrow (\text{arbitrary union-of } P) (S \cap T)) \\ \longleftrightarrow (\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary union-of } P) (S \cap T)) \quad (\text{is } ?lhs = ?rhs) \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-intersection-of-Un-eq*:

$$\begin{array}{c} (\forall S T. (\text{arbitrary intersection-of } P) S \wedge (\text{arbitrary intersection-of } P) T \\ \quad \longrightarrow (\text{arbitrary intersection-of } P) (S \cup T)) \longleftrightarrow \\ (\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary intersection-of } P) (S \cup T)) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-empty [simp]*: $(\text{finite union-of } P) \{\}$
 $\langle \text{proof} \rangle$

lemma *finite-intersection-of-empty [simp]*: $(\text{finite intersection-of } P) UNIV$
 $\langle \text{proof} \rangle$

lemma *finite-union-of-inc*:

$$\begin{array}{c} P S \implies (\text{finite union-of } P) S \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-intersection-of-inc*:

$$\begin{array}{c} P S \implies (\text{finite intersection-of } P) S \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-complement*:

$$\begin{array}{c} (\text{finite union-of } P) S \longleftrightarrow (\text{finite intersection-of } (\lambda S. P(- S))) (- S) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-intersection-of-complement*:

$$\begin{array}{c} (\text{finite intersection-of } P) S \longleftrightarrow (\text{finite union-of } (\lambda S. P(- S))) (- S) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-idempot [simp]*:

$$\begin{array}{c} \text{finite union-of finite union-of } P = \text{finite union-of } P \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-intersection-of-idempot [simp]*:

$$\begin{array}{c} \text{finite intersection-of finite intersection-of } P = \text{finite intersection-of } P \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-Union*:

$$\begin{array}{c} \llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{finite union-of } P) S \rrbracket \implies (\text{finite union-of } P) (\bigcup \mathcal{U}) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-Un:*

$$\llbracket (\text{finite union-of } P) \ S; (\text{finite union-of } P) \ T \rrbracket \Longrightarrow (\text{finite union-of } P) \ (S \cup T)$$

<proof>

lemma *finite-intersection-of-Inter:*

$$\llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \Longrightarrow (\text{finite intersection-of } P) \ S \rrbracket \Longrightarrow (\text{finite intersection-of } P) \ (\bigcap \mathcal{U})$$

<proof>

lemma *finite-intersection-of-Int:*

$$\llbracket (\text{finite intersection-of } P) \ S; (\text{finite intersection-of } P) \ T \rrbracket \Longrightarrow (\text{finite intersection-of } P) \ (S \cap T)$$

<proof>

lemma *finite-union-of-Int-eq:*

$$\begin{aligned} & (\forall S \ T. (\text{finite union-of } P) \ S \wedge (\text{finite union-of } P) \ T \longrightarrow (\text{finite union-of } P) \ (S \cap T)) \\ & \longleftrightarrow (\forall S \ T. P \ S \wedge P \ T \longrightarrow (\text{finite union-of } P) \ (S \cap T)) \\ & \text{(is ?lhs = ?rhs)} \\ & \text{<proof>} \end{aligned}$$

lemma *finite-intersection-of-Un-eq:*

$$\begin{aligned} & (\forall S \ T. (\text{finite intersection-of } P) \ S \wedge (\text{finite intersection-of } P) \ T \\ & \longrightarrow (\text{finite intersection-of } P) \ (S \cup T)) \longleftrightarrow \\ & (\forall S \ T. P \ S \wedge P \ T \longrightarrow (\text{finite intersection-of } P) \ (S \cup T)) \\ & \text{<proof>} \end{aligned}$$

abbreviation *finite'* :: 'a set $\Rightarrow$ bool

where *finite'* $A \equiv \text{finite } A \wedge A \neq \{\}$

lemma *finite'-intersection-of-Int:*

$$\llbracket (\text{finite}' \text{ intersection-of } P) \ S; (\text{finite}' \text{ intersection-of } P) \ T \rrbracket \Longrightarrow (\text{finite}' \text{ intersection-of } P) \ (S \cap T)$$

<proof>

lemma *finite'-intersection-of-inc:*

$$P \ S \Longrightarrow (\text{finite}' \text{ intersection-of } P) \ S$$

<proof>

88.2 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 $\Rightarrow$ bool, 'a set, 'a set] $\Rightarrow$ bool (**infixl** *relative'-to* 55)

where $P \text{ relative-to } S \equiv \lambda T. \exists U. P \ U \wedge S \cap U = T$

lemma *relative-to-UNIV* [simp]: $(P \text{ relative-to } \text{UNIV}) S \longleftrightarrow P S$
 $\langle \text{proof} \rangle$

lemma *relative-to-imp-subset*:
 $(P \text{ relative-to } S) T \implies T \subseteq S$
 $\langle \text{proof} \rangle$

lemma *all-relative-to*: $(\forall S. (P \text{ relative-to } U) S \longrightarrow Q S) \longleftrightarrow (\forall S. P S \longrightarrow Q(U \cap S))$
 $\langle \text{proof} \rangle$

lemma *relative-toE*: $\llbracket (P \text{ relative-to } U) S; \bigwedge S. P S \implies Q(U \cap S) \rrbracket \implies Q S$
 $\langle \text{proof} \rangle$

lemma *relative-to-inc*:
 $P S \implies (P \text{ relative-to } U) (U \cap S)$
 $\langle \text{proof} \rangle$

lemma *relative-to-relative-to* [simp]:
 $P \text{ relative-to } S \text{ relative-to } T = P \text{ relative-to } (S \cap T)$
 $\langle \text{proof} \rangle$

lemma *relative-to-compl*:
 $S \subseteq U \implies ((P \text{ relative-to } U) (U - S) \longleftrightarrow ((\lambda c. P(- c)) \text{ relative-to } U) S)$
 $\langle \text{proof} \rangle$

lemma *relative-to-subset*:
 $S \subseteq T \wedge P S \implies (P \text{ relative-to } T) S$
 $\langle \text{proof} \rangle$

lemma *relative-to-subset-trans*:
 $(P \text{ relative-to } U) S \wedge S \subseteq T \wedge T \subseteq U \implies (P \text{ relative-to } T) S$
 $\langle \text{proof} \rangle$

lemma *relative-to-mono*:
 $\llbracket (P \text{ relative-to } U) S; \bigwedge S. P S \implies Q S \rrbracket \implies (Q \text{ relative-to } U) S$
 $\langle \text{proof} \rangle$

lemma *relative-to-subset-inc*: $\llbracket S \subseteq U; P S \rrbracket \implies (P \text{ relative-to } U) S$
 $\langle \text{proof} \rangle$

lemma *relative-to-Int*:
 $\llbracket (P \text{ relative-to } S) C; (P \text{ relative-to } S) D; \bigwedge X Y. \llbracket P X; P Y \rrbracket \implies P(X \cap Y) \rrbracket$
 $\implies (P \text{ relative-to } S) (C \cap D)$
 $\langle \text{proof} \rangle$

lemma *relative-to-Un*:
 $\llbracket (P \text{ relative-to } S) C; (P \text{ relative-to } S) D; \bigwedge X Y. \llbracket P X; P Y \rrbracket \implies P(X \cup Y) \rrbracket$
 $\implies (P \text{ relative-to } S) (C \cup D)$

$\langle \text{proof} \rangle$

lemma *arbitrary-union-of-relative-to*:

$((\text{arbitrary union-of } P) \text{ relative-to } U) = (\text{arbitrary union-of } (P \text{ relative-to } U))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *finite-union-of-relative-to*:

$((\text{finite union-of } P) \text{ relative-to } U) = (\text{finite union-of } (P \text{ relative-to } U))$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

lemma *countable-union-of-relative-to*:

$((\text{countable union-of } P) \text{ relative-to } U) = (\text{countable union-of } (P \text{ relative-to } U))$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *arbitrary-intersection-of-relative-to*:

$((\text{arbitrary intersection-of } P) \text{ relative-to } U) = ((\text{arbitrary intersection-of } (P \text{ relative-to } U)) \text{ relative-to } U)$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

lemma *finite-intersection-of-relative-to*:

$((\text{finite intersection-of } P) \text{ relative-to } U) = ((\text{finite intersection-of } (P \text{ relative-to } U)) \text{ relative-to } U)$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

lemma *countable-intersection-of-relative-to*:

$((\text{countable intersection-of } P) \text{ relative-to } U) = ((\text{countable intersection-of } (P \text{ relative-to } U)) \text{ relative-to } U)$ **(is ?lhs = ?rhs)**
 $\langle \text{proof} \rangle$

end

88.3 Signed division: negative results rounded towards zero rather than minus infinity.

theory *Signed-Division*

imports *Main*

begin

class *signed-division* =

fixes *signed-divide* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixl** *sdiv* 70)
and *signed-modulo* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixl** *smod* 70)

instantiation *int* :: *signed-division*

begin

definition *signed-divide-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ sdiv } l = \text{sgn } k * \text{sgn } l * (|k| \text{ div } |l|) \rangle$ **for** $k \ l :: \text{int}$

definition *signed-modulo-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ smod } l = k - (k \text{ sdiv } l) * l \rangle$ **for** $k \ l :: \text{int}$

instance $\langle \text{proof} \rangle$

end

lemma *int-sdiv-simps* [simp]:
 $(a :: \text{int}) \text{ sdiv } 1 = a$
 $(a :: \text{int}) \text{ sdiv } 0 = 0$
 $(a :: \text{int}) \text{ sdiv } -1 = -a$
 $\langle \text{proof} \rangle$

lemma *sgn-div-eq-sgn-mult*:
 $a \text{ div } b \neq 0 \implies \text{sgn } ((a :: \text{int}) \text{ div } b) = \text{sgn } (a * b)$
 $\langle \text{proof} \rangle$

lemma *sgn-sdiv-eq-sgn-mult*:
 $a \text{ sdiv } b \neq 0 \implies \text{sgn } ((a :: \text{int}) \text{ sdiv } b) = \text{sgn } (a * b)$
 $\langle \text{proof} \rangle$

lemma *int-sdiv-same-is-1* [simp]:
 $a \neq 0 \implies ((a :: \text{int}) \text{ sdiv } b = a) = (b = 1)$
 $\langle \text{proof} \rangle$

lemma *int-sdiv-negated-is-minus1* [simp]:
 $a \neq 0 \implies ((a :: \text{int}) \text{ sdiv } b = -a) = (b = -1)$
 $\langle \text{proof} \rangle$

lemma *sdiv-int-range*:
 $(a :: \text{int}) \text{ sdiv } b \in \{ - (abs \ a) .. (abs \ a) \}$
 $\langle \text{proof} \rangle$

lemma *sdiv-int-div-0* [simp]:
 $(x :: \text{int}) \text{ sdiv } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *sdiv-int-0-div* [simp]:
 $0 \text{ sdiv } (x :: \text{int}) = 0$
 $\langle \text{proof} \rangle$

lemma *smod-int-alt-def*:
 $(a :: \text{int}) \text{ smod } b = \text{sgn } (a) * (abs \ a \text{ mod } abs \ b)$
 $\langle \text{proof} \rangle$

lemma *smod-int-range*:

$b \neq 0 \implies (a :: \text{int}) \text{ smod } b \in \{ - \text{abs } b + 1 .. \text{abs } b - 1 \}$
 ⟨proof⟩

lemma *smod-int-compares*:

$\llbracket 0 \leq a; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b < b$
 $\llbracket 0 \leq a; 0 < b \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; 0 < b \rrbracket \implies -b < (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b < -b$
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$
 $\llbracket a \leq 0; b < 0 \rrbracket \implies b \leq (a :: \text{int}) \text{ smod } b$
 ⟨proof⟩

lemma *smod-int-mod-0* [simp]:

$x \text{ smod } (0 :: \text{int}) = x$
 ⟨proof⟩

lemma *smod-int-0-mod* [simp]:

$0 \text{ smod } (x :: \text{int}) = 0$
 ⟨proof⟩

lemma *smod-mod-positive*:

$\llbracket 0 \leq (a :: \text{int}); 0 \leq b \rrbracket \implies a \text{ smod } b = a \text{ mod } b$
 ⟨proof⟩

lemma *minus-sdiv-eq* [simp]:

$\langle -k \text{ sdiv } l = - (k \text{ sdiv } l) \rangle \text{ for } k \ l :: \text{int}$
 ⟨proof⟩

lemma *sdiv-minus-eq* [simp]:

$\langle k \text{ sdiv } -l = - (k \text{ sdiv } l) \rangle \text{ for } k \ l :: \text{int}$
 ⟨proof⟩

lemma *sdiv-int-numeral-numeral* [simp]:

$\langle \text{numeral } m \text{ sdiv numeral } n = \text{numeral } m \text{ div } (\text{numeral } n :: \text{int}) \rangle$
 ⟨proof⟩

lemma *minus-smod-eq* [simp]:

$\langle -k \text{ smod } l = - (k \text{ smod } l) \rangle \text{ for } k \ l :: \text{int}$
 ⟨proof⟩

lemma *smod-minus-eq* [simp]:

$\langle k \text{ smod } -l = k \text{ smod } l \rangle \text{ for } k \ l :: \text{int}$
 ⟨proof⟩

lemma *smod-int-numeral-numeral* [simp]:

$\langle \text{numeral } m \text{ smod numeral } n = \text{numeral } m \text{ mod } (\text{numeral } n :: \text{int}) \rangle$
 ⟨proof⟩

end

89 State monad

theory *State-Monad*
imports *Monad-Syntax*
begin

datatype (*'s*, *'a*) *state* = *State* (*run-state*: *'s* $\Rightarrow$ (*'a* $\times$ *'s*))

lemma *set-state-iff*: $x \in \text{set-state } m \iff (\exists s\ s'. \text{run-state } m\ s = (x, s'))$
 $\langle \text{proof} \rangle$

lemma *pred-stateI*[*intro*]:
assumes $\bigwedge a\ s\ s'. \text{run-state } m\ s = (a, s') \implies P\ a$
shows *pred-state* *P* *m*
 $\langle \text{proof} \rangle$

lemma *pred-stateD*[*dest*]:
assumes *pred-state* *P* *m* *run-state* *m* *s* = (*a*, *s'*)
shows *P* *a*
 $\langle \text{proof} \rangle$

lemma *pred-state-run-state*: *pred-state* *P* *m* $\implies P$ (*fst* (*run-state* *m* *s*))
 $\langle \text{proof} \rangle$

definition *state-io-rel* :: (*'s* $\Rightarrow$ *'s* $\Rightarrow$ *bool*) $\Rightarrow$ (*'s*, *'a*) *state* $\Rightarrow$ *bool* **where**
state-io-rel *P* *m* = ($\forall s. P\ s\ (\text{snd } (\text{run-state } m\ s))$)

lemma *state-io-relI*[*intro*]:
assumes $\bigwedge a\ s\ s'. \text{run-state } m\ s = (a, s') \implies P\ s\ s'$
shows *state-io-rel* *P* *m*
 $\langle \text{proof} \rangle$

lemma *state-io-relD*[*dest*]:
assumes *state-io-rel* *P* *m* *run-state* *m* *s* = (*a*, *s'*)
shows *P* *s* *s'*
 $\langle \text{proof} \rangle$

lemma *state-io-rel-mono*[*mono*]: $P \leq Q \implies \text{state-io-rel } P \leq \text{state-io-rel } Q$
 $\langle \text{proof} \rangle$

lemma *state-ext*:
assumes $\bigwedge s. \text{run-state } m\ s = \text{run-state } n\ s$
shows *m* = *n*
 $\langle \text{proof} \rangle$

context **begin**

qualified definition $\text{return} :: 'a \Rightarrow ('s, 'a) \text{ state}$ **where**
 $\text{return } a = \text{State } (\text{Pair } a)$

lemma $\text{run-state-return}[\text{simp}]$: $\text{run-state } (\text{return } x) s = (x, s)$
 $\langle \text{proof} \rangle$ **definition** $\text{ap} :: ('s, 'a \Rightarrow 'b) \text{ state} \Rightarrow ('s, 'a) \text{ state} \Rightarrow ('s, 'b) \text{ state}$ **where**
 $\text{ap } f x = \text{State } (\lambda s. \text{case run-state } f s \text{ of } (g, s') \Rightarrow \text{case run-state } x s' \text{ of } (y, s'') \Rightarrow (g y, s''))$

lemma $\text{run-state-ap}[\text{simp}]$:
 $\text{run-state } (\text{ap } f x) s = (\text{case run-state } f s \text{ of } (g, s') \Rightarrow \text{case run-state } x s' \text{ of } (y, s'') \Rightarrow (g y, s''))$
 $\langle \text{proof} \rangle$ **definition** $\text{bind} :: ('s, 'a) \text{ state} \Rightarrow ('a \Rightarrow ('s, 'b) \text{ state}) \Rightarrow ('s, 'b) \text{ state}$ **where**
 $\text{bind } x f = \text{State } (\lambda s. \text{case run-state } x s \text{ of } (a, s') \Rightarrow \text{run-state } (f a) s')$

lemma $\text{run-state-bind}[\text{simp}]$:
 $\text{run-state } (\text{bind } x f) s = (\text{case run-state } x s \text{ of } (a, s') \Rightarrow \text{run-state } (f a) s')$
 $\langle \text{proof} \rangle$

ad hoc-overloading $\text{Monad-Syntax.bind bind}$

lemma $\text{bind-left-identity}[\text{simp}]$: $\text{bind } (\text{return } a) f = f a$
 $\langle \text{proof} \rangle$

lemma $\text{bind-right-identity}[\text{simp}]$: $\text{bind } m \text{return} = m$
 $\langle \text{proof} \rangle$

lemma $\text{bind-assoc}[\text{simp}]$: $\text{bind } (\text{bind } m f) g = \text{bind } m (\lambda x. \text{bind } (f x) g)$
 $\langle \text{proof} \rangle$

lemma $\text{bind-predI}[\text{intro}]$:
assumes $\text{pred-state } (\lambda x. \text{pred-state } P (f x)) m$
shows $\text{pred-state } P (\text{bind } m f)$
 $\langle \text{proof} \rangle$ **definition** $\text{get} :: ('s, 's) \text{ state}$ **where**
 $\text{get} = \text{State } (\lambda s. (s, s))$

lemma $\text{run-state-get}[\text{simp}]$: $\text{run-state } \text{get } s = (s, s)$
 $\langle \text{proof} \rangle$ **definition** $\text{set} :: 's \Rightarrow ('s, \text{unit}) \text{ state}$ **where**
 $\text{set } s' = \text{State } (\lambda -. ((), s'))$

lemma $\text{run-state-set}[\text{simp}]$: $\text{run-state } (\text{set } s') s = ((), s')$
 $\langle \text{proof} \rangle$

lemma $\text{get-set}[\text{simp}]$: $\text{bind } \text{get } \text{set} = \text{return } ()$
 $\langle \text{proof} \rangle$

lemma $\text{set-set}[\text{simp}]$: $\text{bind } (\text{set } s) (\lambda -. \text{set } s') = \text{set } s'$
 $\langle \text{proof} \rangle$

lemma *get-bind-set*[simp]: $\text{bind get } (\lambda s. \text{bind } (\text{set } s) (f s)) = \text{bind get } (\lambda s. f s ())$
 ⟨proof⟩

lemma *get-const*[simp]: $\text{bind get } (\lambda -. m) = m$
 ⟨proof⟩

fun *traverse-list* :: $('a \Rightarrow ('b, 'c) \text{ state}) \Rightarrow 'a \text{ list} \Rightarrow ('b, 'c \text{ list}) \text{ state}$ **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]: $\text{traverse-list } f (xs @ ys) = \text{do } \{$
 xs ← *traverse-list* *f* *xs*;
 ys ← *traverse-list* *f* *ys*;
 return (*xs* @ *ys*)
 }
 ⟨proof⟩

lemma *traverse-comp*[simp]: $\text{traverse-list } (g \circ f) xs = \text{traverse-list } g (\text{map } f xs)$
 ⟨proof⟩

abbreviation *mono-state* :: $('s::\text{preorder}, 'a) \text{ state} \Rightarrow \text{bool}$ **where**
mono-state $\equiv \text{state-io-rel } (\leq)$

abbreviation *strict-mono-state* :: $('s::\text{preorder}, 'a) \text{ state} \Rightarrow \text{bool}$ **where**
strict-mono-state $\equiv \text{state-io-rel } (<)$

corollary *strict-mono-implies-mono*: $\text{strict-mono-state } m \Longrightarrow \text{mono-state } m$
 ⟨proof⟩

lemma *return-mono*[simp, intro]: $\text{mono-state } (\text{return } x)$
 ⟨proof⟩

lemma *get-mono*[simp, intro]: $\text{mono-state } \text{get}$
 ⟨proof⟩

lemma *put-mono*:
 assumes $\bigwedge x. s' \geq x$
 shows $\text{mono-state } (\text{set } s')$
 ⟨proof⟩

lemma *map-mono*[intro]: $\text{mono-state } m \Longrightarrow \text{mono-state } (\text{map-state } f m)$
 ⟨proof⟩

lemma *map-strict-mono*[intro]: $\text{strict-mono-state } m \Longrightarrow \text{strict-mono-state } (\text{map-state } f m)$

$f\ m)$
 $\langle proof \rangle$

lemma *bind-mono-strong*:

assumes *mono-state* m

assumes $\bigwedge x\ s\ s'.\ run\_state\ m\ s = (x, s') \implies mono\_state\ (f\ x)$

shows *mono-state* $(bind\ m\ f)$

$\langle proof \rangle$

lemma *bind-strict-mono-strong1*:

assumes *mono-state* m

assumes $\bigwedge x\ s\ s'.\ run\_state\ m\ s = (x, s') \implies strict\_mono\_state\ (f\ x)$

shows *strict-mono-state* $(bind\ m\ f)$

$\langle proof \rangle$

lemma *bind-strict-mono-strong2*:

assumes *strict-mono-state* m

assumes $\bigwedge x\ s\ s'.\ run\_state\ m\ s = (x, s') \implies mono\_state\ (f\ x)$

shows *strict-mono-state* $(bind\ m\ f)$

$\langle proof \rangle$

corollary *bind-strict-mono-strong*:

assumes *strict-mono-state* m

assumes $\bigwedge x\ s\ s'.\ run\_state\ m\ s = (x, s') \implies strict\_mono\_state\ (f\ x)$

shows *strict-mono-state* $(bind\ m\ f)$

$\langle proof \rangle$ **definition** *update* $:: ('s \Rightarrow 's) \Rightarrow ('s, unit)\ state\ \mathbf{where}$
update $f = bind\ get\ (set \circ f)$

lemma *update-id[simp]*: *update* $(\lambda x. x) = return\ ()$

$\langle proof \rangle$

lemma *update-comp[simp]*: *bind* $(update\ f)\ (\lambda -. update\ g) = update\ (g \circ f)$

$\langle proof \rangle$

lemma *set-update[simp]*: *bind* $(set\ s)\ (\lambda -. update\ f) = set\ (f\ s)$

$\langle proof \rangle$

lemma *set-bind-update[simp]*: *bind* $(set\ s)\ (\lambda -. bind\ (update\ f)\ g) = bind\ (set\ (f\ s))\ g$

$\langle proof \rangle$

lemma *update-mono*:

assumes $\bigwedge x. x \leq f\ x$

shows *mono-state* $(update\ f)$

$\langle proof \rangle$

lemma *update-strict-mono*:

assumes $\bigwedge x. x < f\ x$

shows *strict-mono-state* $(update\ f)$

$\langle proof \rangle$

end

end

theory *Comparator*
imports *Main*
begin

90 Comparators on linear quasi-orders

90.1 Basic properties

datatype *comp* = *Less* | *Equiv* | *Greater*

locale *comparator* =
fixes *cmp* :: 'a $\Rightarrow$ 'a $\Rightarrow$ *comp*
assumes *refl* [*simp*]: $\bigwedge a. \text{cmp } a \ a = \text{Equiv}$
and *trans-equiv*: $\bigwedge a \ b \ c. \text{cmp } a \ b = \text{Equiv} \Longrightarrow \text{cmp } b \ c = \text{Equiv} \Longrightarrow \text{cmp } a \ c = \text{Equiv}$
assumes *trans-less*: $\text{cmp } a \ b = \text{Less} \Longrightarrow \text{cmp } b \ c = \text{Less} \Longrightarrow \text{cmp } a \ c = \text{Less}$
and *greater-iff-sym-less*: $\bigwedge b \ a. \text{cmp } b \ a = \text{Greater} \longleftrightarrow \text{cmp } a \ b = \text{Less}$
begin

Dual properties

lemma *trans-greater*:
 $\text{cmp } a \ c = \text{Greater} \text{ if } \text{cmp } a \ b = \text{Greater} \ \text{cmp } b \ c = \text{Greater}$
 $\langle proof \rangle$

lemma *less-iff-sym-greater*:
 $\text{cmp } b \ a = \text{Less} \longleftrightarrow \text{cmp } a \ b = \text{Greater}$
 $\langle proof \rangle$

The equivalence part

lemma *sym*:
 $\text{cmp } b \ a = \text{Equiv} \longleftrightarrow \text{cmp } a \ b = \text{Equiv}$
 $\langle proof \rangle$

lemma *reflp*:
 $\text{reflp } (\lambda a \ b. \text{cmp } a \ b = \text{Equiv})$
 $\langle proof \rangle$

lemma *symp*:
 $\text{symp } (\lambda a \ b. \text{cmp } a \ b = \text{Equiv})$
 $\langle proof \rangle$

lemma *transp*:

transp ($\lambda a\ b. \text{cmp } a\ b = \text{Equiv}$)
 $\langle \text{proof} \rangle$

lemma *equivp*:
equivp ($\lambda a\ b. \text{cmp } a\ b = \text{Equiv}$)
 $\langle \text{proof} \rangle$

The strict part

lemma *irreflp-less*:
irreflp ($\lambda a\ b. \text{cmp } a\ b = \text{Less}$)
 $\langle \text{proof} \rangle$

lemma *irreflp-greater*:
irreflp ($\lambda a\ b. \text{cmp } a\ b = \text{Greater}$)
 $\langle \text{proof} \rangle$

lemma *asym-less*:
 $\text{cmp } b\ a \neq \text{Less}$ **if** $\text{cmp } a\ b = \text{Less}$
 $\langle \text{proof} \rangle$

lemma *asym-greater*:
 $\text{cmp } b\ a \neq \text{Greater}$ **if** $\text{cmp } a\ b = \text{Greater}$
 $\langle \text{proof} \rangle$

lemma *asymmp-less*:
asymmp ($\lambda a\ b. \text{cmp } a\ b = \text{Less}$)
 $\langle \text{proof} \rangle$

lemma *asymmp-greater*:
asymmp ($\lambda a\ b. \text{cmp } a\ b = \text{Greater}$)
 $\langle \text{proof} \rangle$

lemma *trans-equiv-less*:
 $\text{cmp } a\ c = \text{Less}$ **if** $\text{cmp } a\ b = \text{Equiv}$ **and** $\text{cmp } b\ c = \text{Less}$
 $\langle \text{proof} \rangle$

lemma *trans-less-equiv*:
 $\text{cmp } a\ c = \text{Less}$ **if** $\text{cmp } a\ b = \text{Less}$ **and** $\text{cmp } b\ c = \text{Equiv}$
 $\langle \text{proof} \rangle$

lemma *trans-equiv-greater*:
 $\text{cmp } a\ c = \text{Greater}$ **if** $\text{cmp } a\ b = \text{Equiv}$ **and** $\text{cmp } b\ c = \text{Greater}$
 $\langle \text{proof} \rangle$

lemma *trans-greater-equiv*:
 $\text{cmp } a\ c = \text{Greater}$ **if** $\text{cmp } a\ b = \text{Greater}$ **and** $\text{cmp } b\ c = \text{Equiv}$
 $\langle \text{proof} \rangle$

lemma *transp-less*:

transp ($\lambda a\ b. \text{cmp } a\ b = \text{Less}$)
 $\langle \text{proof} \rangle$

lemma *transp-greater*:
transp ($\lambda a\ b. \text{cmp } a\ b = \text{Greater}$)
 $\langle \text{proof} \rangle$

The reflexive part

lemma *reflp-not-less*:
reflp ($\lambda a\ b. \text{cmp } a\ b \neq \text{Less}$)
 $\langle \text{proof} \rangle$

lemma *reflp-not-greater*:
reflp ($\lambda a\ b. \text{cmp } a\ b \neq \text{Greater}$)
 $\langle \text{proof} \rangle$

lemma *quasisym-not-less*:
 $\text{cmp } a\ b = \text{Equiv}$ **if** $\text{cmp } a\ b \neq \text{Less}$ **and** $\text{cmp } b\ a \neq \text{Less}$
 $\langle \text{proof} \rangle$

lemma *quasisym-not-greater*:
 $\text{cmp } a\ b = \text{Equiv}$ **if** $\text{cmp } a\ b \neq \text{Greater}$ **and** $\text{cmp } b\ a \neq \text{Greater}$
 $\langle \text{proof} \rangle$

lemma *trans-not-less*:
 $\text{cmp } a\ c \neq \text{Less}$ **if** $\text{cmp } a\ b \neq \text{Less}$ $\text{cmp } b\ c \neq \text{Less}$
 $\langle \text{proof} \rangle$

lemma *trans-not-greater*:
 $\text{cmp } a\ c \neq \text{Greater}$ **if** $\text{cmp } a\ b \neq \text{Greater}$ $\text{cmp } b\ c \neq \text{Greater}$
 $\langle \text{proof} \rangle$

lemma *transp-not-less*:
transp ($\lambda a\ b. \text{cmp } a\ b \neq \text{Less}$)
 $\langle \text{proof} \rangle$

lemma *transp-not-greater*:
transp ($\lambda a\ b. \text{cmp } a\ b \neq \text{Greater}$)
 $\langle \text{proof} \rangle$

Substitution under equivalences

lemma *equiv-subst-left*:
 $\text{cmp } z\ y = \text{comp} \longleftrightarrow \text{cmp } x\ y = \text{comp}$ **if** $\text{cmp } z\ x = \text{Equiv}$ **for** comp
 $\langle \text{proof} \rangle$

lemma *equiv-subst-right*:
 $\text{cmp } x\ z = \text{comp} \longleftrightarrow \text{cmp } x\ y = \text{comp}$ **if** $\text{cmp } z\ y = \text{Equiv}$ **for** comp
 $\langle \text{proof} \rangle$

end

typedef *'a comparator* = { *cmp* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ 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

A rudimentary quickcheck setup

instantiation *comparator* :: (*enum*) *equal*
begin

lift-definition *equal-comparator* :: *'a comparator* $\Rightarrow$ *'a comparator* $\Rightarrow$ *bool*
is λ *f g. $\forall x \in$ set Enum.enum. f* *x* = *g* *x* *<proof>*

instance
<proof>

end

lemma [*code*]:
HOL.equal *cmp1 cmp2* $\longleftrightarrow$ *Enum.enum-all* (λ *x. compare* *cmp1* *x* = *compare* *cmp2* *x*)
<proof>

lemma [*code nbe*]:
HOL.equal (*cmp* :: *'a::enum comparator*) *cmp* $\longleftrightarrow$ *True*
<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

90.2 Fundamental comparator combinators

lift-definition *reversed* :: 'a comparator ⇒ 'a comparator
is λ*cmp* *a* *b*. *cmp* *b* *a*
 ⟨proof⟩

lift-definition *key* :: ('b ⇒ 'a) ⇒ 'a comparator ⇒ 'b comparator
is λ*f* *cmp* *a* *b*. *cmp* (*f* *a*) (*f* *b*)
 ⟨proof⟩

90.3 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 = (λa 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

91 Stably sorted lists

abbreviation (input) *stable-segment* :: 'a comparator ⇒ 'a ⇒ 'a list ⇒ 'a list
 where *stable-segment cmp x ≡ filter (λy. 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 $\bigwedge y. xs = y \# ys \implies compare\ cmp\ x\ y \neq Greater$
 ⟨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 $x \in set\ xs$
 ⟨proof⟩

lemma *sorted-induct* [consumes 1, case-names Nil Cons, induct pred: sorted]:
 P xs if sorted cmp xs and P []
 and *: $\bigwedge x xs. sorted\ cmp\ xs \implies P\ xs$
 $\implies (\bigwedge y. y \in set\ xs \implies compare\ cmp\ x\ y \neq Greater) \implies P\ (x \# xs)$

<proof>

lemma *sorted-induct-remove1* [*consumes 1, case-names Nil minimum*]:

P xs if sorted cmp xs and P []
*and *: $\bigwedge x xs. \text{sorted cmp } xs \implies P (\text{remove1 } x \text{ } xs)$*
 $\implies x \in \text{set } xs \implies \text{hd } (\text{stable-segment cmp } x \text{ } xs) = x \implies (\bigwedge y. y \in \text{set } xs \implies$
compare cmp $x \ y \neq \text{Greater}$)
 $\implies P xs$
<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 $\Rightarrow$ '*a* $\Rightarrow$ '*a* list $\Rightarrow$ '*a* list

where *insort cmp y [] = [y]*
| insort cmp y (x # xs) = (if compare cmp y x $\neq \text{Greater}$
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 $\neq \text{Equiv}$
<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* $\wedge y. y \in \text{set } xs \implies \text{compare cmp } x y \neq \text{Greater}$
 <proof>

lemma *remove1-insort-not-same-eq* [simp]:

remove1 y (insort cmp x xs) = insort cmp x (remove1 y xs)
if *sorted cmp xs* $x \neq y$
 <proof>

lemma *insort-remove1-same-eq*:

insort cmp x (remove1 x xs) = xs
if *sorted cmp xs* **and** $x \in \text{set } xs$ **and** $\text{hd } (\text{stable-segment cmp } x xs) = x$
 <proof>

lemma *sorted-append-iff*:

sorted cmp (xs @ ys) $\longleftrightarrow$ sorted cmp xs $\wedge$ sorted cmp ys
 $\wedge (\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{compare cmp } x y \neq \text{Greater})$ (**is** $?P \longleftrightarrow ?R \wedge ?S \wedge ?Q$)
 <proof>

definition *sort* :: 'a comparator $\Rightarrow$ 'a list $\Rightarrow$ '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*: $\bigwedge y. y \in set\ ys \implies$
 $stable\_segment\ cmp\ y\ ys = stable\_segment\ cmp\ y\ xs$
 ⟨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 $\neg P\ x$
 ⟨proof⟩

lemma *filter-sort*:

$filter\ P (sort\ cmp\ xs) = sort\ cmp (filter\ P\ xs)$
 ⟨proof⟩

92 Alternative sorting algorithms

92.1 Quicksort

definition *quicksort* :: 'a comparator $\Rightarrow$ 'a list $\Rightarrow$ 'a list

where *quicksort-is-sort* [simp]: $quicksort = sort$

lemma *sort-by-quicksort*:

$sort = quicksort$
 ⟨proof⟩

lemma *sort-by-quicksort-rec*:

$sort\ cmp\ xs = sort\ cmp [x \leftarrow xs. compare\ cmp\ x\ (xs\ !\ (length\ xs\ div\ 2)) = Less]$
 @ *stable-segment* $cmp\ (xs\ !\ (length\ xs\ div\ 2))\ xs$
 @ $sort\ cmp [x \leftarrow xs. compare\ cmp\ x\ (xs\ !\ (length\ xs\ div\ 2)) = Greater]$ (**is** - =
 ?rhs)
 ⟨proof⟩

context
begin

qualified definition *partition* :: 'a comparator $\Rightarrow$ 'a $\Rightarrow$ 'a list $\Rightarrow$ 'a list $\times$ 'a list $\times$ 'a list

where *partition cmp pivot xs* =
 ($[x \leftarrow xs.$ compare *cmp* *x* *pivot* = *Less*], *stable-segment cmp pivot xs*, $[x \leftarrow xs.$
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 $\Rightarrow$ (*x* # *lts*, *eqs*, *gts*)
 | *Equiv* $\Rightarrow$ (*lts*, *x* # *eqs*, *gts*)
 | *Greater* $\Rightarrow$ (*lts*, *eqs*, *x* # *gts*))
 <proof>

lemma *quicksort-code* [code]:

quicksort cmp xs =
 (case *xs* of
 [] $\Rightarrow$ []
 | [*x*] $\Rightarrow$ *xs*
 | [*x*, *y*] $\Rightarrow$ (if compare *cmp* *x* *y* $\neq$ *Greater* then *xs* else [*y*, *x*])
 | - $\Rightarrow$
 let (*lts*, *eqs*, *gts*) = *partition cmp* (*xs* ! (*length xs* div 2)) *xs*
 in *quicksort cmp lts* @ *eqs* @ *quicksort cmp gts*)
 <proof>

end

92.2 Mergesort

definition *mergesort* :: 'a comparator $\Rightarrow$ 'a list $\Rightarrow$ '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 $\Rightarrow$ 'a list $\Rightarrow$ 'a list

where *merge [] ys* = *ys*
 | *merge xs []* = *xs*
 | *merge (x # xs) (y # ys)* = (if compare *cmp* *x* *y* = *Greater*

then $y \# \text{merge } (x \# xs) \text{ } ys \text{ else } x \# \text{merge } xs \text{ } (y \# ys)$
 $\langle \text{proof} \rangle$ **termination** $\langle \text{proof} \rangle$

lemma *mset-merge*:

$\text{mset } (\text{merge } xs \text{ } ys) = \text{mset } xs + \text{mset } ys$
 $\langle \text{proof} \rangle$

lemma *merge-eq-Cons-imp*:

$xs \neq [] \wedge z = \text{hd } xs \vee ys \neq [] \wedge z = \text{hd } ys$
if $\text{merge } xs \text{ } ys = z \# zs$
 $\langle \text{proof} \rangle$

lemma *filter-merge*:

$\text{filter } P \text{ } (\text{merge } xs \text{ } ys) = \text{merge } (\text{filter } P \text{ } xs) \text{ } (\text{filter } P \text{ } ys)$
if *sorted cmp xs and sorted cmp ys*
 $\langle \text{proof} \rangle$

lemma *sorted-merge*:

sorted cmp (merge xs ys) if sorted cmp xs and sorted cmp ys
 $\langle \text{proof} \rangle$

lemma *merge-eq-appendI*:

$\text{merge } xs \text{ } ys = xs @ ys$
if $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } ys \implies \text{compare cmp } x \text{ } y \neq \text{Greater}$
 $\langle \text{proof} \rangle$

lemma *merge-stable-segments*:

$\text{merge } (\text{stable-segment cmp } l \text{ } xs) \text{ } (\text{stable-segment cmp } l \text{ } ys) =$
 $\text{stable-segment cmp } l \text{ } xs @ \text{stable-segment cmp } l \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *sort-by-mergesort-rec*:

$\text{sort cmp } xs =$
 $\text{merge } (\text{sort cmp } (\text{take } (\text{length } xs \text{ div } 2) \text{ } xs))$
 $(\text{sort cmp } (\text{drop } (\text{length } xs \text{ div } 2) \text{ } xs)) \text{ } (\text{is - = ?rhs})$
 $\langle \text{proof} \rangle$

lemma *mergesort-code* [code]:

$\text{mergesort cmp } xs =$
 $(\text{case } xs \text{ of}$
 $\quad [] \Rightarrow []$
 $\quad | [x] \Rightarrow xs$
 $\quad | [x, y] \Rightarrow (\text{if compare cmp } x \text{ } y \neq \text{Greater then } xs \text{ else } [y, x])$
 $\quad | - \Rightarrow$
 $\quad \text{let}$
 $\quad \quad \text{half} = \text{length } xs \text{ div } 2;$
 $\quad \quad \text{ys} = \text{take half } xs;$
 $\quad \quad \text{zs} = \text{drop half } xs$
 $\quad \text{in merge (mergesort cmp } ys) \text{ } (\text{mergesort cmp } zs))$

$\langle proof \rangle$

end

end

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

theory *Sum-of-Squares*
imports *Complex-Main*
begin

$\langle ML \rangle$

end

94 A table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtrancl-path* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \Rightarrow \text{bool}$
for $r :: 'a \Rightarrow 'a \Rightarrow \text{bool}$

where

$\text{base: } rtrancl\text{-path } r\ x\ []\ x$
 $| \text{ step: } r\ x\ y \Longrightarrow rtrancl\text{-path } r\ y\ ys\ z \Longrightarrow rtrancl\text{-path } r\ x\ (y \# ys)\ z$

lemma *rtranclp-eq-rtrancl-path*: $r^{**}\ x\ y \longleftrightarrow (\exists xs. rtrancl\text{-path } r\ x\ xs\ y)$
 $\langle proof \rangle$

lemma *rtrancl-path-trans*:

assumes $xy: rtrancl\text{-path } r\ x\ xs\ y$
and $yz: rtrancl\text{-path } r\ y\ ys\ z$
shows $rtrancl\text{-path } r\ x\ (xs @ ys)\ z$ $\langle proof \rangle$

lemma *rtrancl-path-appendE*:

assumes $xz: rtrancl\text{-path } r\ x\ (xs @ y \# ys)\ z$
obtains $rtrancl\text{-path } r\ x\ (xs @ [y])\ y$ **and** $rtrancl\text{-path } r\ y\ ys\ z$
 $\langle proof \rangle$

lemma *rtrancl-path-distinct*:

assumes xy : $rtrancl\text{-}path\ r\ x\ xs\ y$
obtains xs' **where** $rtrancl\text{-}path\ r\ x\ xs'\ y$ **and** $distinct\ (x\ \# \ xs')$ **and** $set\ xs' \subseteq set\ xs$
 $\langle proof \rangle$

inductive $rtrancl\text{-}tab :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where
 $base$: $rtrancl\text{-}tab\ r\ xs\ x\ x$
 $| step$: $x \notin set\ xs \Longrightarrow r\ x\ y \Longrightarrow rtrancl\text{-}tab\ r\ (x\ \# \ xs)\ y\ z \Longrightarrow rtrancl\text{-}tab\ r\ xs\ x\ z$

lemma $rtrancl\text{-}path\text{-}imp\text{-}rtrancl\text{-}tab$:
assumes $path$: $rtrancl\text{-}path\ r\ x\ xs\ y$
and x : $distinct\ (x\ \# \ xs)$
and ys : $(\{x\} \cup set\ xs) \cap set\ ys = \{\}$
shows $rtrancl\text{-}tab\ r\ ys\ x\ y$
 $\langle proof \rangle$

lemma $rtrancl\text{-}tab\text{-}imp\text{-}rtrancl\text{-}path$:
assumes tab : $rtrancl\text{-}tab\ r\ ys\ x\ y$
obtains xs **where** $rtrancl\text{-}path\ r\ x\ xs\ y$
 $\langle proof \rangle$

lemma $rtranclp\text{-}eq\text{-}rtrancl\text{-}tab\text{-}nil$: $r^{**}\ x\ y \longleftrightarrow rtrancl\text{-}tab\ r\ []\ x\ y$
 $\langle proof \rangle$

declare $rtranclp\text{-}rtrancl\text{-}eq$ [code del]
declare $rtranclp\text{-}eq\text{-}rtrancl\text{-}tab\text{-}nil$ [THEN iffD2, code-pred-intro]

code-pred $rtranclp$
 $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}Range$: $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; z \in set\ xs \rrbracket \Longrightarrow Rangep\ R\ z$
 $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}Range\text{-}end$: $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; xs \neq [] \rrbracket \Longrightarrow Rangep\ R\ y$
 $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}nth$:
 $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; i < length\ xs \rrbracket \Longrightarrow R\ ((x\ \# \ xs)\ !\ i)\ (xs\ !\ i)$
 $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}last$: $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; xs \neq [] \rrbracket \Longrightarrow last\ xs = y$
 $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}mono$:
 $\llbracket rtrancl\text{-}path\ R\ x\ p\ y; \bigwedge x\ y. R\ x\ y \Longrightarrow S\ x\ y \rrbracket \Longrightarrow rtrancl\text{-}path\ S\ x\ p\ y$
 $\langle proof \rangle$

end

95 Binary Tree

```
theory Tree
imports Main
begin
```

```
datatype 'a tree =
  Leaf ⟨⟩ |
  Node 'a tree (value: 'a) 'a tree ((1⟨-, / -, / -⟩))
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 |
```

$\text{min-height } (\text{Node } l - r) = \min (\text{min-height } l) (\text{min-height } r) + 1$

fun *complete* :: 'a tree $\Rightarrow$ bool **where**
complete Leaf = True |
complete (Node l x r) = (height l = height r $\wedge$ *complete* l $\wedge$ *complete* r)

Almost complete:

definition *acomplete* :: 'a tree $\Rightarrow$ bool **where**
acomplete t = (height t - min-height t $\leq$ 1)

Weight balanced:

fun *wbalanced* :: 'a tree $\Rightarrow$ bool **where**
wbalanced Leaf = True |
wbalanced (Node l x r) = (abs(int(size l) - int(size r)) $\leq$ 1 $\wedge$ *wbalanced* l $\wedge$ *wbalanced* r)

Internal path length:

fun *ipl* :: 'a tree $\Rightarrow$ nat **where**
ipl Leaf = 0 |
ipl (Node l - r) = *ipl* l + size l + *ipl* r + size r

fun *preorder* :: 'a tree $\Rightarrow$ 'a list **where**
preorder $\langle \rangle$ = [] |
preorder $\langle l, x, r \rangle$ = x # *preorder* l @ *preorder* r

fun *inorder* :: 'a tree $\Rightarrow$ 'a list **where**
inorder $\langle \rangle$ = [] |
inorder $\langle l, x, r \rangle$ = *inorder* l @ [x] @ *inorder* r

A linear version avoiding append:

fun *inorder2* :: 'a tree $\Rightarrow$ 'a list $\Rightarrow$ 'a list **where**
inorder2 $\langle \rangle$ xs = xs |
inorder2 $\langle l, x, r \rangle$ xs = *inorder2* l (x # *inorder2* r xs)

fun *postorder* :: 'a tree $\Rightarrow$ 'a list **where**
postorder $\langle \rangle$ = [] |
postorder $\langle l, x, r \rangle$ = *postorder* l @ *postorder* r @ [x]

Binary Search Tree:

fun *bst-wrt* :: ('a $\Rightarrow$ 'a $\Rightarrow$ bool) $\Rightarrow$ 'a tree $\Rightarrow$ bool **where**
bst-wrt P $\langle \rangle$ $\longleftrightarrow$ True |
bst-wrt P $\langle l, a, r \rangle$ $\longleftrightarrow$
 ($\forall x \in \text{set-tree } l. P x a$) $\wedge$ ($\forall x \in \text{set-tree } r. P a x$) $\wedge$ *bst-wrt* P l $\wedge$ *bst-wrt* P r

abbreviation *bst* :: ('a::linorder) tree $\Rightarrow$ bool **where**
bst $\equiv$ *bst-wrt* (<)

fun (in *linorder*) *heap* :: 'a tree $\Rightarrow$ bool **where**
heap Leaf = True |

$heap\ (Node\ l\ m\ r) =$
 $((\forall x \in set-tree\ l \cup set-tree\ r.\ m \leq x) \wedge heap\ l \wedge heap\ r)$

95.1 *map-tree*

lemma *eq-map-tree-Leaf[simp]*: $map-tree\ f\ t = Leaf \longleftrightarrow t = Leaf$
 $\langle proof \rangle$

lemma *eq-Leaf-map-tree[simp]*: $Leaf = map-tree\ f\ t \longleftrightarrow t = Leaf$
 $\langle proof \rangle$

95.2 *size*

lemma *size1-size*: $size1\ t = size\ t + 1$
 $\langle proof \rangle$

lemma *size1-ge0[simp]*: $0 < size1\ t$
 $\langle proof \rangle$

lemma *eq-size-0[simp]*: $size\ t = 0 \longleftrightarrow t = Leaf$
 $\langle proof \rangle$

lemma *eq-0-size[simp]*: $0 = size\ t \longleftrightarrow t = Leaf$
 $\langle proof \rangle$

lemma *neg-Leaf-iff*: $(t \neq \langle \rangle) = (\exists l\ a\ r.\ t = \langle l, a, r \rangle)$
 $\langle proof \rangle$

lemma *size-map-tree[simp]*: $size\ (map-tree\ f\ t) = size\ t$
 $\langle proof \rangle$

lemma *size1-map-tree[simp]*: $size1\ (map-tree\ f\ t) = size1\ t$
 $\langle proof \rangle$

95.3 *set-tree*

lemma *eq-set-tree-empty[simp]*: $set-tree\ t = \{\} \longleftrightarrow t = Leaf$
 $\langle proof \rangle$

lemma *eq-empty-set-tree[simp]*: $\{\} = set-tree\ t \longleftrightarrow t = Leaf$
 $\langle proof \rangle$

lemma *finite-set-tree[simp]*: $finite(set-tree\ t)$
 $\langle proof \rangle$

95.4 *subtrees*

lemma *neg-subtrees-empty[simp]*: $subtrees\ t \neq \{\}$
 $\langle proof \rangle$

lemma *neq-empty-subtrees[simp]*: $\{\} \neq \text{subtrees } t$
 $\langle \text{proof} \rangle$

lemma *size-subtrees*: $s \in \text{subtrees } t \implies \text{size } s \leq \text{size } t$
 $\langle \text{proof} \rangle$

lemma *set-treeE*: $a \in \text{set-tree } t \implies \exists l r. \langle l, a, r \rangle \in \text{subtrees } t$
 $\langle \text{proof} \rangle$

lemma *Node-notin-subtrees-if[simp]*: $a \notin \text{set-tree } t \implies \text{Node } l a r \notin \text{subtrees } t$
 $\langle \text{proof} \rangle$

lemma *in-set-tree-if*: $\langle l, a, r \rangle \in \text{subtrees } t \implies a \in \text{set-tree } t$
 $\langle \text{proof} \rangle$

95.5 height and min-height

lemma *eq-height-0[simp]*: $\text{height } t = 0 \iff t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *eq-0-height[simp]*: $0 = \text{height } t \iff t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *height-map-tree[simp]*: $\text{height } (\text{map-tree } f t) = \text{height } t$
 $\langle \text{proof} \rangle$

lemma *height-le-size-tree*: $\text{height } t \leq \text{size } (t::'a \text{ tree})$
 $\langle \text{proof} \rangle$

lemma *size1-height*: $\text{size1 } t \leq 2 \wedge \text{height } (t::'a \text{ tree})$
 $\langle \text{proof} \rangle$

corollary *size-height*: $\text{size } t \leq 2 \wedge \text{height } (t::'a \text{ tree}) - 1$
 $\langle \text{proof} \rangle$

lemma *height-subtrees*: $s \in \text{subtrees } t \implies \text{height } s \leq \text{height } t$
 $\langle \text{proof} \rangle$

lemma *min-height-le-height*: $\text{min-height } t \leq \text{height } t$
 $\langle \text{proof} \rangle$

lemma *min-height-map-tree[simp]*: $\text{min-height } (\text{map-tree } f t) = \text{min-height } t$
 $\langle \text{proof} \rangle$

lemma *min-height-size1*: $2 \wedge \text{min-height } t \leq \text{size1 } t$
 $\langle \text{proof} \rangle$

95.6 *complete*

lemma *complete-iff-height*: $\text{complete } t \longleftrightarrow (\text{min-height } t = \text{height } t)$
 ⟨proof⟩

lemma *size1-if-complete*: $\text{complete } t \implies \text{size1 } t = 2^{\text{height } t}$
 ⟨proof⟩

lemma *size-if-complete*: $\text{complete } t \implies \text{size } t = 2^{\text{height } t} - 1$
 ⟨proof⟩

lemma *size1-height-if-incomplete*:
 $\neg \text{complete } t \implies \text{size1 } t < 2^{\text{height } t}$
 ⟨proof⟩

lemma *complete-iff-min-height*: $\text{complete } t \longleftrightarrow (\text{height } t = \text{min-height } t)$
 ⟨proof⟩

lemma *min-height-size1-if-incomplete*:
 $\neg \text{complete } t \implies 2^{\text{min-height } t} < \text{size1 } t$
 ⟨proof⟩

lemma *complete-if-size1-height*: $\text{size1 } t = 2^{\text{height } t} \implies \text{complete } t$
 ⟨proof⟩

lemma *complete-if-size1-min-height*: $\text{size1 } t = 2^{\text{min-height } t} \implies \text{complete } t$
 ⟨proof⟩

lemma *complete-iff-size1*: $\text{complete } t \longleftrightarrow \text{size1 } t = 2^{\text{height } t}$
 ⟨proof⟩

95.7 *acomplete*

lemma *acomplete-subtreeL*: $\text{acomplete } (\text{Node } l \ x \ r) \implies \text{acomplete } l$
 ⟨proof⟩

lemma *acomplete-subtreeR*: $\text{acomplete } (\text{Node } l \ x \ r) \implies \text{acomplete } r$
 ⟨proof⟩

lemma *acomplete-subtrees*: $\llbracket \text{acomplete } t; s \in \text{subtrees } t \rrbracket \implies \text{acomplete } s$
 ⟨proof⟩

Balanced trees have optimal height:

lemma *acomplete-optimal*:
fixes $t :: 'a \text{ tree}$ **and** $t' :: 'b \text{ tree}$
assumes $\text{acomplete } t$ $\text{size } t \leq \text{size } t'$ **shows** $\text{height } t \leq \text{height } t'$
 ⟨proof⟩

95.8 *wbalanced*

lemma *wbalanced-subtrees*: $\llbracket \text{wbalanced } t; s \in \text{subtrees } t \rrbracket \implies \text{wbalanced } s$
 $\langle \text{proof} \rangle$

95.9 *ipl*

The internal path length of a tree:

lemma *ipl-if-complete-int*:
 $\text{complete } t \implies \text{int}(\text{ipl } t) = (\text{int}(\text{height } t) - 2) * 2^{\text{height } t} + 2$
 $\langle \text{proof} \rangle$

95.10 List of entries

lemma *eq-inorder-Nil[simp]*: $\text{inorder } t = [] \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *eq-Nil-inorder[simp]*: $[] = \text{inorder } t \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *set-inorder[simp]*: $\text{set } (\text{inorder } t) = \text{set-tree } t$
 $\langle \text{proof} \rangle$

lemma *set-preorder[simp]*: $\text{set } (\text{preorder } t) = \text{set-tree } t$
 $\langle \text{proof} \rangle$

lemma *set-postorder[simp]*: $\text{set } (\text{postorder } t) = \text{set-tree } t$
 $\langle \text{proof} \rangle$

lemma *length-preorder[simp]*: $\text{length } (\text{preorder } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *length-inorder[simp]*: $\text{length } (\text{inorder } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *length-postorder[simp]*: $\text{length } (\text{postorder } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *preorder-map*: $\text{preorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{preorder } t)$
 $\langle \text{proof} \rangle$

lemma *inorder-map*: $\text{inorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{inorder } t)$
 $\langle \text{proof} \rangle$

lemma *postorder-map*: $\text{postorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{postorder } t)$
 $\langle \text{proof} \rangle$

lemma *inorder2-inorder*: $\text{inorder2 } t \ xs = \text{inorder } t \ @ \ xs$
 $\langle \text{proof} \rangle$

95.11 Binary Search Tree

lemma *bst-wrt-mono*: $(\bigwedge x y. P x y \implies Q x y) \implies \text{bst-wrt } P t \implies \text{bst-wrt } Q t$
 $\langle \text{proof} \rangle$

lemma *bst-wrt-le-if-bst*: $\text{bst } t \implies \text{bst-wrt } (\leq) t$
 $\langle \text{proof} \rangle$

lemma *bst-wrt-le-iff-sorted*: $\text{bst-wrt } (\leq) t \longleftrightarrow \text{sorted } (\text{inorder } t)$
 $\langle \text{proof} \rangle$

lemma *bst-iff-sorted-wrt-less*: $\text{bst } t \longleftrightarrow \text{sorted-wrt } (<) (\text{inorder } t)$
 $\langle \text{proof} \rangle$

95.12 heap**95.13 mirror**

lemma *mirror-Leaf[simp]*: $\text{mirror } t = \langle \rangle \longleftrightarrow t = \langle \rangle$
 $\langle \text{proof} \rangle$

lemma *Leaf-mirror[simp]*: $\langle \rangle = \text{mirror } t \longleftrightarrow t = \langle \rangle$
 $\langle \text{proof} \rangle$

lemma *size-mirror[simp]*: $\text{size}(\text{mirror } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *size1-mirror[simp]*: $\text{size1}(\text{mirror } t) = \text{size1 } t$
 $\langle \text{proof} \rangle$

lemma *height-mirror[simp]*: $\text{height}(\text{mirror } t) = \text{height } t$
 $\langle \text{proof} \rangle$

lemma *min-height-mirror [simp]*: $\text{min-height } (\text{mirror } t) = \text{min-height } t$
 $\langle \text{proof} \rangle$

lemma *ipl-mirror [simp]*: $\text{ipl } (\text{mirror } t) = \text{ipl } t$
 $\langle \text{proof} \rangle$

lemma *inorder-mirror*: $\text{inorder}(\text{mirror } t) = \text{rev}(\text{inorder } t)$
 $\langle \text{proof} \rangle$

lemma *map-mirror*: $\text{map-tree } f (\text{mirror } t) = \text{mirror } (\text{map-tree } f t)$
 $\langle \text{proof} \rangle$

lemma *mirror-mirror[simp]*: $\text{mirror}(\text{mirror } t) = t$
 $\langle \text{proof} \rangle$

end

96 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  $\Rightarrow$  'a multiset where
  mset-tree Leaf = {#} |
  mset-tree (Node l a r) = {#a#} + mset-tree l + mset-tree r
```

```
fun subtrees-mset :: 'a tree  $\Rightarrow$  '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 = {#}  $\longleftrightarrow$  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:  $x \in \#$  mset-tree t  $\longleftrightarrow$   $x \in$  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]:  $s \in \#$  subtrees-mset t  $\longleftrightarrow$   $s \in$  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:  $\log 2 \text{ (size1 } t) \leq \text{height } t$ 
  <proof>

```

```

lemma min-height-size1-log:  $\text{min-height } t \leq \log 2 \text{ (size1 } t)$ 
  <proof>

```

```

lemma size1-log-if-complete:  $\text{complete } t \implies \text{height } t = \log 2 \text{ (size1 } t)$ 
  <proof>

```

```

lemma min-height-size1-log-if-incomplete:
   $\neg \text{complete } t \implies \text{min-height } t < \log 2 \text{ (size1 } t)$ 
  <proof>

```

```

lemma min-height-acomplete: assumes acomplete t
shows  $\text{min-height } t = \text{nat}(\text{floor}(\log 2 \text{ (size1 } t)))$ 
  <proof>

```

```

lemma height-acomplete: assumes acomplete t
shows  $\text{height } t = \text{nat}(\text{ceiling}(\log 2 \text{ (size1 } t)))$ 
  <proof>

```

```

lemma acomplete-Node-if-wbal1:
assumes acomplete l acomplete r  $\text{size } l = \text{size } r + 1$ 
shows acomplete  $\langle l, x, r \rangle$ 
  <proof>

```

```

lemma acomplete-sym: acomplete  $\langle l, x, r \rangle \implies \text{acomplete } \langle r, y, l \rangle$ 
  <proof>

```

```

lemma acomplete-Node-if-wbal2:
assumes acomplete l acomplete r  $\text{abs}(\text{int}(\text{size } l) - \text{int}(\text{size } r)) \leq 1$ 
shows acomplete  $\langle l, x, r \rangle$ 
  <proof>

```

```

lemma acomplete-if-wbalanced: wbalanced t  $\implies \text{acomplete } t$ 
  <proof>

```

```

end

```

97 Unordered pairs

theory *Uprod* **imports** *Main* **begin**

typedef (*'a*, *'b*) *commute* = {*f* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'b*. $\forall x y. f\ x\ y = f\ y\ x$ }
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* $\Rightarrow$ *'b* $\Rightarrow$ *bool*) $\Rightarrow$ (*'c* $\Rightarrow$ *'d* $\Rightarrow$ *bool*) $\Rightarrow$ (*'a*, *'c*)
commute $\Rightarrow$ (*'b*, *'d*) *commute* $\Rightarrow$ *bool*
is $\lambda A\ B. A ==> A ==> B$ *<proof>*

end

definition *eq-upair* :: (*'a* $\times$ *'a*) $\Rightarrow$ (*'a* $\times$ *'a*) $\Rightarrow$ *bool*
where *eq-upair* = ($\lambda(a, b)\ (c, d). a = c \wedge b = d \vee a = d \wedge b = c$)

lemma *eq-upair-simps* [*simp*]:
eq-upair (a, b) (c, d) $\longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
<proof>

lemma *equivp-eq-upair*: *equivp eq-upair*
<proof>

quotient-type *'a uprod* = *'a* $\times$ *'a* / *eq-upair* *<proof>*

lift-definition *Upair* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'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 $\longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$*
<proof>

code-datatype *Upair*

lift-definition *case-uprod* :: (*'a*, *'b*) *commute* $\Rightarrow$ *'a uprod* $\Rightarrow$ *'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*) $\longleftrightarrow$ ($\forall a\ b. x = \text{Upair } a\ b \longrightarrow P\ (\text{apply-commute } f\ a\ b)$)
 ⟨*proof*⟩

lemma *uprod-split-asm*: *P* (*case-uprod* *f x*) $\longleftrightarrow \neg (\exists a\ b. x = \text{Upair } a\ b \wedge \neg P\ (\text{apply-commute } f\ a\ b))$
 ⟨*proof*⟩

lift-definition *not-equal* :: ('*a*, *bool*) *commute* **is** ($\neq$) ⟨*proof*⟩

lemma *apply-not-equal* [*simp*]: *apply-commute* *not-equal* *x y* $\longleftrightarrow x \neq y$
 ⟨*proof*⟩

definition *proper-uprod* :: '*a uprod* $\Rightarrow$ *bool*
where *proper-uprod* = *case-uprod not-equal*

lemma *proper-uprod-simps* [*simp*, *code*]: *proper-uprod* (*Upair* *x y*) $\longleftrightarrow x \neq y$
 ⟨*proof*⟩

context **includes** *lifting-syntax* **begin**

private lemma *set-uprod-parametric'*:
 (*rel-prod* *A A* $\implies$ *rel-set* *A*) ($\lambda(a, b). \{a, b\}$) ($\lambda(a, b). \{a, b\}$)
 ⟨*proof*⟩

lift-definition *set-uprod* :: '*a uprod* $\Rightarrow$ '*a set* **is** $\lambda(a, b). \{a, b\}$
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* $\implies$ *B*) $\implies$ *rel-prod* *A A* $\implies$ *rel-prod* *B B*) ($\lambda f. \text{map-prod } f\ f$) ($\lambda f. \text{map-prod } f\ f$)
 ⟨*proof*⟩

lift-definition *map-uprod* :: ('*a* $\Rightarrow$ '*b*) $\Rightarrow$ '*a uprod* $\Rightarrow$ '*b uprod* **is** $\lambda f. \text{map-prod } f\ f$
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* $\implies$ *B* $\implies$ (=)) $\implies$ *rel-prod* *A A* $\implies$ *rel-prod* *B B* $\implies$ (=))
 (=)

$(\lambda R (a, b) (c, d). R a c \wedge R b d \vee R a d \wedge R b c) (\lambda R (a, b) (c, d). R a c \wedge R b d \vee R a d \wedge R b c)$
 $\langle proof \rangle$

lift-definition *rel-uprod* :: ('a $\Rightarrow$ 'b $\Rightarrow$ bool) $\Rightarrow$ 'a uprod $\Rightarrow$ 'b uprod $\Rightarrow$ bool
is $\lambda R (a, b) (c, d). R a c \wedge R b d \vee R a d \wedge R b c$ **parametric** *rel-uprod-transfer'*
 $\langle proof \rangle$

lemma *rel-uprod-simps* [*simp*, *code*]:
 $rel-uprod R (Upair a b) (Upair c d) \longleftrightarrow R a c \wedge R b d \vee R a d \wedge R b c$
 $\langle proof \rangle$

lemma *Upair-parametric* [*transfer-rule*]: ($A ==> A ==> rel-uprod A$) *Upair*
 $\langle proof \rangle$

lemma *case-uprod-parametric* [*transfer-rule*]:
 $(rel-commute A B ==> rel-uprod A ==> B) case-uprod case-uprod$
 $\langle proof \rangle$

end

bnf *uprod*: 'a uprod
 map: map-uprod
 sets: set-uprod
 bd: natLeq
 rel: rel-uprod
 $\langle proof \rangle$

lemma *pred-uprod-code* [*simp*, *code*]: $pred-uprod P (Upair x y) \longleftrightarrow P x \wedge P y$
 $\langle proof \rangle$

instantiation *uprod* :: (equal) equal **begin**

definition *equal-uprod* :: 'a uprod $\Rightarrow$ 'a uprod $\Rightarrow$ bool
where *equal-uprod* = (=)

lemma *equal-uprod-code* [*code*]:
 $HOL.equal (Upair x y) (Upair z u) \longleftrightarrow x = z \wedge y = u \vee x = u \wedge y = z$
 $\langle proof \rangle$

instance $\langle proof \rangle$
end

quickcheck-generator *uprod constructors*: *Upair*

lemma *UNIV-uprod*: $UNIV = (\lambda x. Upair x x) \text{ ‘ } UNIV \cup (\lambda(x, y). Upair x y) \text{ ‘ }$
 $Sigma UNIV (\lambda x. UNIV - \{x\})$
 $\langle proof \rangle$

context begin

private lift-definition *upair-inv* :: 'a uprod $\Rightarrow$ 'a

is $\lambda(x, y). \text{if } x = y \text{ then } x \text{ else undefined}$ $\langle \text{proof} \rangle$

lemma *finite-UNIV-prod* [simp]:

finite (UNIV :: 'a uprod set) $\longleftrightarrow$ *finite* (UNIV :: 'a set) (**is** ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

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 -)
 $\langle \text{proof} \rangle$

end

98 A type of finite bit strings

theory *Word*

imports

HOL-Library.Type-Length

begin

98.1 Preliminaries

lemma *signed-take-bit-decr-length-iff*:

$\langle \text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - \text{Suc } 0) \ k = \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \ l \rangle$
 $\longleftrightarrow \text{take-bit } \text{LENGTH}('a) \ k = \text{take-bit } \text{LENGTH}('a) \ l \rangle$
 $\langle \text{proof} \rangle$

98.2 Fundamentals

98.2.1 Type definition

quotient-type (overloaded) 'a word = int / $\langle \lambda k \ l. \text{take-bit } \text{LENGTH}('a) \ k = \text{take-bit } \text{LENGTH}('a::\text{len}) \ l \rangle$
morphisms *rep* *Word* $\langle \text{proof} \rangle$

hide-const (**open**) *rep* — only for foundational purpose

hide-const (**open**) *Word* — only for code generation

98.2.2 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) (λk. 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⟩

```

98.2.3 Basic tool setup

⟨ML⟩

98.2.4 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* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$
is $\langle \lambda k \ l. \text{take-bit } LENGTH('a) \ k = \text{take-bit } LENGTH('a) \ l \rangle$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma [*code*]:
 $\langle HOL.equal \ v \ w \longleftrightarrow HOL.equal \ (Word.the-int \ v) \ (Word.the-int \ w) \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle Word.the-int \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle Word.the-int \ 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle Word.the-int \ (v + w) = \text{take-bit } LENGTH('a) \ (Word.the-int \ v + Word.the-int \ w) \rangle$
for $v \ w :: \langle 'a::len \ word \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle Word.the-int \ (- \ w) = (\text{let } k = Word.the-int \ w \text{ in if } w = 0 \text{ then } 0 \text{ else } 2^{\wedge} LENGTH('a) - k) \rangle$
for $w :: \langle 'a::len \ word \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle Word.the-int \ (v - w) = \text{take-bit } LENGTH('a) \ (Word.the-int \ v - Word.the-int \ w) \rangle$
for $v \ w :: \langle 'a::len \ word \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle Word.the-int \ (v * w) = \text{take-bit } LENGTH('a) \ (Word.the-int \ v * Word.the-int \ w) \rangle$
for $v \ w :: \langle 'a::len \ word \rangle$
 $\langle \text{proof} \rangle$

98.2.5 Basic conversions

abbreviation *word-of-nat* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \rangle$

where $\langle \text{word-of-nat} \equiv \text{of-nat} \rangle$

abbreviation *word-of-int* :: $\langle \text{int} \Rightarrow 'a::\text{len word} \rangle$

where $\langle \text{word-of-int} \equiv \text{of-int} \rangle$

lemma *word-of-nat-eq-iff*:

$\langle \text{word-of-nat } m = (\text{word-of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a) } m = \text{take-bit LENGTH('a) } n \rangle$

$\langle \text{proof} \rangle$

lemma *word-of-int-eq-iff*:

$\langle \text{word-of-int } k = (\text{word-of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a) } k = \text{take-bit LENGTH('a) } l \rangle$

$\langle \text{proof} \rangle$

lemma *word-of-nat-eq-0-iff*:

$\langle \text{word-of-nat } n = (0 :: 'a::\text{len word}) \longleftrightarrow 2 \wedge \text{LENGTH('a) } \text{dvd } n \rangle$

$\langle \text{proof} \rangle$

lemma *word-of-int-eq-0-iff*:

$\langle \text{word-of-int } k = (0 :: 'a::\text{len word}) \longleftrightarrow 2 \wedge \text{LENGTH('a) } \text{dvd } k \rangle$

$\langle \text{proof} \rangle$

context *semiring-1*

begin

lift-definition *unsigned* :: $\langle 'b::\text{len word} \Rightarrow 'a \rangle$

is $\langle \text{of-nat} \circ \text{nat} \circ \text{take-bit LENGTH('b)} \rangle$

$\langle \text{proof} \rangle$

lemma *unsigned-0 [simp]*:

$\langle \text{unsigned } 0 = 0 \rangle$

$\langle \text{proof} \rangle$

lemma *unsigned-1 [simp]*:

$\langle \text{unsigned } 1 = 1 \rangle$

$\langle \text{proof} \rangle$

lemma *unsigned-numeral [simp]*:

$\langle \text{unsigned } (\text{numeral } n :: 'b::\text{len word}) = \text{of-nat } (\text{take-bit LENGTH('b) } (\text{numeral } n)) \rangle$

$\langle \text{proof} \rangle$

lemma *unsigned-neg-numeral [simp]*:

$\langle \text{unsigned } (- \text{numeral } n :: 'b::\text{len word}) = \text{of-nat } (\text{nat } (\text{take-bit LENGTH('b) } (- \text{numeral } n))) \rangle$

$\langle \text{proof} \rangle$

end

context *semiring-1*
begin

lemma *unsigned-of-nat*:
 $\langle \text{unsigned} (\text{word-of-nat } n :: 'b::\text{len word}) = \text{of-nat} (\text{take-bit LENGTH}('b) n) \rangle$
 $\langle \text{proof} \rangle$

lemma *unsigned-of-int*:
 $\langle \text{unsigned} (\text{word-of-int } k :: 'b::\text{len word}) = \text{of-nat} (\text{nat} (\text{take-bit LENGTH}('b) k)) \rangle$
 $\langle \text{proof} \rangle$

end

context *semiring-char-0*
begin

lemma *unsigned-word-eqI*:
 $\langle v = w \rangle \text{ if } \langle \text{unsigned } v = \text{unsigned } w \rangle$
 $\langle \text{proof} \rangle$

lemma *word-eq-iff-unsigned*:
 $\langle v = w \longleftrightarrow \text{unsigned } v = \text{unsigned } w \rangle$
 $\langle \text{proof} \rangle$

lemma *inj-unsigned* [simp]:
 $\langle \text{inj unsigned} \rangle$
 $\langle \text{proof} \rangle$

lemma *unsigned-eq-0-iff*:
 $\langle \text{unsigned } w = 0 \longleftrightarrow w = 0 \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-1*
begin

lift-definition *signed* :: $\langle 'b::\text{len word} \Rightarrow 'a \rangle$
is $\langle \text{of-int} \circ \text{signed-take-bit} (\text{LENGTH}('b) - \text{Suc } 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-0* [simp]:
 $\langle \text{signed } 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-1* [simp]:

$\langle \text{signed } (1 :: 'b::\text{len word}) = (\text{if } \text{LENGTH}('b) = 1 \text{ then } - 1 \text{ else } 1) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-minus-1* [simp]:
 $\langle \text{signed } (- 1 :: 'b::\text{len word}) = - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-numeral* [simp]:
 $\langle \text{signed } (\text{numeral } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - 1) (\text{numeral } n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-neg-numeral* [simp]:
 $\langle \text{signed } (- \text{numeral } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - 1) (- \text{numeral } n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-of-nat*:
 $\langle \text{signed } (\text{word-of-nat } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - \text{Suc } 0) (\text{int } n)) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-of-int*:
 $\langle \text{signed } (\text{word-of-int } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - \text{Suc } 0) n) \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-char-0*
begin

lemma *signed-word-eqI*:
 $\langle v = w \rangle \text{ if } \langle \text{signed } v = \text{signed } w \rangle$
 $\langle \text{proof} \rangle$

lemma *word-eq-iff-signed*:
 $\langle v = w \iff \text{signed } v = \text{signed } w \rangle$
 $\langle \text{proof} \rangle$

lemma *inj-signed* [simp]:
 $\langle \text{inj signed} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-eq-0-iff*:
 $\langle \text{signed } w = 0 \iff w = 0 \rangle$
 $\langle \text{proof} \rangle$

end

abbreviation *unat* :: $\langle 'a::\text{len word} \Rightarrow \text{nat} \rangle$
where $\langle \text{unat} \equiv \text{unsigned} \rangle$

abbreviation *uint* :: $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$
where $\langle \text{uint} \equiv \text{unsigned} \rangle$

abbreviation *sint* :: $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$
where $\langle \text{sint} \equiv \text{signed} \rangle$

abbreviation *ucast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
where $\langle \text{ucast} \equiv \text{unsigned} \rangle$

abbreviation *scast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
where $\langle \text{scast} \equiv \text{signed} \rangle$

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
 $\langle (\text{pcr-word} ==> (=)) (\text{nat} \circ \text{take-bit LENGTH}('a)) (\text{unat} :: 'a::\text{len word} \Rightarrow \text{nat}) \rangle$
 $\langle \text{proof} \rangle$

lemma [*transfer-rule*]:
 $\langle (\text{pcr-word} ==> (=)) (\text{take-bit LENGTH}('a)) (\text{uint} :: 'a::\text{len word} \Rightarrow \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma [*transfer-rule*]:
 $\langle (\text{pcr-word} ==> (=)) (\text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0)) (\text{sint} :: 'a::\text{len word} \Rightarrow \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma [*transfer-rule*]:
 $\langle (\text{pcr-word} ==> \text{pcr-word}) (\text{take-bit LENGTH}('a)) (\text{ucast} :: 'a::\text{len word} \Rightarrow 'b::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

lemma [*transfer-rule*]:
 $\langle (\text{pcr-word} ==> \text{pcr-word}) (\text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0)) (\text{scast} :: 'a::\text{len word} \Rightarrow 'b::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

end

lemma *of-nat-unat* [*simp*]:
 $\langle \text{of-nat} (\text{unat } w) = \text{unsigned } w \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-uint* [simp]:
 $\langle \text{of-int } (\text{uint } w) = \text{unsigned } w \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-sint* [simp]:
 $\langle \text{of-int } (\text{sint } a) = \text{signed } a \rangle$
 $\langle \text{proof} \rangle$

lemma *nat-uint-eq* [simp]:
 $\langle \text{nat } (\text{uint } w) = \text{unat } w \rangle$
 $\langle \text{proof} \rangle$

lemma *sgn-uint-eq* [simp]:
 $\langle \text{sgn } (\text{uint } w) = \text{of-bool } (w \neq 0) \rangle$
 $\langle \text{proof} \rangle$

Aliases only for code generation

context
begin

qualified lift-definition *of-int* :: $\langle \text{int} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \text{take-bit } \text{LENGTH}('a) \rangle \langle \text{proof} \rangle$ **lift-definition** *of-nat* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \text{int} \circ \text{take-bit } \text{LENGTH}('a) \rangle \langle \text{proof} \rangle$ **lift-definition** *the-nat* :: $\langle 'a::\text{len word} \Rightarrow \text{nat} \rangle$
is $\langle \text{nat} \circ \text{take-bit } \text{LENGTH}('a) \rangle \langle \text{proof} \rangle$ **lift-definition** *the-signed-int* :: $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$
is $\langle \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle \langle \text{proof} \rangle$ **lift-definition** *cast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
is $\langle \text{take-bit } \text{LENGTH}('a) \rangle \langle \text{proof} \rangle$ **lift-definition** *signed-cast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
is $\langle \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle \langle \text{proof} \rangle$

end

lemma [code-abbrev, simp]:
 $\langle \text{Word.the-int} = \text{uint} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{Word.of-int } k :: 'a::\text{len word}) = \text{take-bit } \text{LENGTH}('a) \ k \rangle$
 $\langle \text{proof} \rangle$

lemma [code-abbrev, simp]:
 $\langle \text{Word.of-int} = \text{word-of-int} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{Word.of-nat } n :: 'a::\text{len word}) = \text{take-bit } \text{LENGTH}('a) \ (\text{int } n) \rangle$

⟨*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* = *signed-take-bit* (*LENGTH*('a) − *Suc* 0) (*Word.the-int* *w*)⟩
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*]:

$\langle \text{signed } w = \text{of-int } (\text{Word.the-signed-int } w) \rangle$
 $\langle \text{proof} \rangle$

98.2.6 Basic ordering

instantiation $\text{word} :: (\text{len}) \text{ linorder}$
begin

lift-definition $\text{less-eq-word} :: 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool}$
is $\lambda a b. \text{take-bit LENGTH}('a) a \leq \text{take-bit LENGTH}('a) b$
 $\langle \text{proof} \rangle$

lift-definition $\text{less-word} :: 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool}$
is $\lambda a b. \text{take-bit LENGTH}('a) a < \text{take-bit LENGTH}('a) b$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

interpretation $\text{word-order}: \text{ordering-top } \langle (\leq) \rangle \langle (<) \rangle \langle - 1 :: 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

interpretation $\text{word-coorder}: \text{ordering-top } \langle (\geq) \rangle \langle (>) \rangle \langle 0 :: 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-of-nat-less-eq-iff}$:
 $\langle \text{word-of-nat } m \leq (\text{word-of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH}('a) m$
 $\leq \text{take-bit LENGTH}('a) n \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-of-int-less-eq-iff}$:
 $\langle \text{word-of-int } k \leq (\text{word-of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH}('a) k \leq$
 $\text{take-bit LENGTH}('a) l \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-of-nat-less-iff}$:
 $\langle \text{word-of-nat } m < (\text{word-of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH}('a) m$
 $< \text{take-bit LENGTH}('a) n \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-of-int-less-iff}$:
 $\langle \text{word-of-int } k < (\text{word-of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH}('a) k <$
 $\text{take-bit LENGTH}('a) l \rangle$
 $\langle \text{proof} \rangle$

lemma word-le-def [code]:
 $a \leq b \longleftrightarrow \text{uint } a \leq \text{uint } b$

⟨proof⟩

lemma *word-less-def* [code]:
 $a < b \longleftrightarrow \text{uint } a < \text{uint } b$
 ⟨proof⟩

lemma *word-greater-zero-iff*:
 $\langle a > 0 \longleftrightarrow a \neq 0 \rangle \text{ for } a :: \langle 'a::\text{len word} \rangle$
 ⟨proof⟩

lemma *of-nat-word-less-eq-iff*:
 $\langle \text{of-nat } m \leq (\text{of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit } \text{LENGTH}('a) \text{ } m \leq \text{take-bit } \text{LENGTH}('a) \text{ } n \rangle$
 ⟨proof⟩

lemma *of-nat-word-less-iff*:
 $\langle \text{of-nat } m < (\text{of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit } \text{LENGTH}('a) \text{ } m < \text{take-bit } \text{LENGTH}('a) \text{ } n \rangle$
 ⟨proof⟩

lemma *of-int-word-less-eq-iff*:
 $\langle \text{of-int } k \leq (\text{of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit } \text{LENGTH}('a) \text{ } k \leq \text{take-bit } \text{LENGTH}('a) \text{ } l \rangle$
 ⟨proof⟩

lemma *of-int-word-less-iff*:
 $\langle \text{of-int } k < (\text{of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit } \text{LENGTH}('a) \text{ } k < \text{take-bit } \text{LENGTH}('a) \text{ } l \rangle$
 ⟨proof⟩

98.3 Enumeration

lemma *inj-on-word-of-nat*:
 $\langle \text{inj-on } (\text{word-of-nat} :: \text{nat} \Rightarrow 'a::\text{len word}) \{0..<2 \wedge \text{LENGTH}('a)\} \rangle$
 ⟨proof⟩

lemma *UNIV-word-eq-word-of-nat*:
 $\langle (\text{UNIV} :: 'a::\text{len word set}) = \text{word-of-nat } \{0..<2 \wedge \text{LENGTH}('a)\} \text{ (is } \langle - = ?A \rangle) \rangle$
 ⟨proof⟩

instantiation *word* :: (len) enum
begin

definition *enum-word* :: ⟨'a word list⟩
where $\langle \text{enum-word} = \text{map word-of-nat } [0..<2 \wedge \text{LENGTH}('a)] \rangle$

definition *enum-all-word* :: ⟨('a word $\Rightarrow$ bool) $\Rightarrow$ bool⟩
where $\langle \text{enum-all-word} = \text{Ball UNIV} \rangle$

definition *enum-ex-word* :: $\langle 'a \text{ word} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$
where $\langle \text{enum-ex-word} = \text{Bex UNIV} \rangle$

lemma [code]:
 $\langle \text{Enum.enum-all } P \longleftrightarrow \text{Ball UNIV } P \rangle$
 $\langle \text{Enum.enum-ex } P \longleftrightarrow \text{Bex UNIV } P \rangle$ **for** $P :: \langle 'a \text{ word} \Rightarrow \text{bool} \rangle$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

98.4 Bit-wise operations

instantiation *word* :: (len) *semiring-modulo*
begin

lift-definition *divide-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda a b. \text{take-bit LENGTH('a) } a \text{ div take-bit LENGTH('a) } b \rangle$
 $\langle \text{proof} \rangle$

lift-definition *modulo-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda a b. \text{take-bit LENGTH('a) } a \text{ mod take-bit LENGTH('a) } b \rangle$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instance *word* :: (len) *semiring-parity*
 $\langle \text{proof} \rangle$

lemma *word-bit-induct* [case-names zero even odd]:
 $\langle P a \rangle$ **if** *word-zero*: $\langle P 0 \rangle$
and *word-even*: $\langle \bigwedge a. P a \implies 0 < a \implies a < 2^\wedge(\text{LENGTH('a)} - \text{Suc } 0) \implies P (2 * a) \rangle$
and *word-odd*: $\langle \bigwedge a. P a \implies a < 2^\wedge(\text{LENGTH('a)} - \text{Suc } 0) \implies P (1 + 2 * a) \rangle$
for P **and** $a :: \langle 'a :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-half-eq*:
 $\langle (\text{of-bool } b + a * 2) \text{ div } 2 = a \rangle$
if $\langle a < 2^\wedge(\text{LENGTH('a)} - \text{Suc } 0) \rangle$
for $a :: \langle 'a :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *even-mult-exp-div-word-iff*:

$\langle \text{even } (a * 2^m \text{ div } 2^n) \longleftrightarrow \neg ($
 $m \leq n \wedge$
 $n < \text{LENGTH}('a) \wedge \text{odd } (a \text{ div } 2^{(n-m)})) \rangle$ **for** $a :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

instantiation *word* :: (len) *semiring-bits*
begin

lift-definition *bit-word* :: $\langle 'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$

is $\langle \lambda k n. n < \text{LENGTH}('a) \wedge \text{bit } k n \rangle$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *bit-word-eqI*:

$\langle a = b \rangle$ **if** $\langle \bigwedge n. n < \text{LENGTH}('a) \Longrightarrow \text{bit } a n \longleftrightarrow \text{bit } b n \rangle$
for $a b :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-imp-le-length*:

$\langle n < \text{LENGTH}('a) \rangle$ **if** $\langle \text{bit } w n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *not-bit-length* [simp]:

$\langle \neg \text{bit } w \text{ LENGTH}('a) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *finite-bit-word* [simp]:

$\langle \text{finite } \{n. \text{bit } w n\} \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-word-iff* [simp]:

$\langle \text{bit } (\text{numeral } w :: 'a::\text{len word}) n$
 $\longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } (\text{numeral } w :: \text{int}) n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-neg-numeral-word-iff* [simp]:

$\langle \text{bit } (- \text{numeral } w :: 'a::\text{len word}) n$
 $\longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } (- \text{numeral } w :: \text{int}) n \rangle$
 $\langle \text{proof} \rangle$

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 ⟨λn. drop-bit n ∘ take-bit LENGTH('a)⟩
  ⟨proof⟩

lift-definition take-bit-word :: ⟨nat ⇒ 'a word ⇒ 'a word⟩
  is ⟨λn. take-bit (min LENGTH('a) n)⟩
  ⟨proof⟩

instance ⟨proof⟩

end

lemma [code]:

```

$\langle \text{push-bit } n \ w = w * 2^{\wedge n} \ \mathbf{for} \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{Word.the-int } w) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{take-bit } n \ w) = (\text{if } n < \text{LENGTH}('a::\text{len}) \text{ then } \text{take-bit } n \ (\text{Word.the-int } w) \text{ else } \text{Word.the-int } w) \rangle$
 $\mathbf{for} \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code-abbrev]:
 $\langle \text{push-bit } n \ 1 = (2 :: 'a::\text{len word})^{\wedge n} \rangle$
 $\langle \text{proof} \rangle$

context
 $\mathbf{includes} \ \text{bit-operations-syntax}$
begin

lemma [code]:
 $\langle \text{NOT } w = \text{Word.of-int } (\text{NOT } (\text{Word.the-int } w)) \rangle$
 $\mathbf{for} \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (v \ \text{AND } w) = \text{Word.the-int } v \ \text{AND } \text{Word.the-int } w \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (v \ \text{OR } w) = \text{Word.the-int } v \ \text{OR } \text{Word.the-int } w \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (v \ \text{XOR } w) = \text{Word.the-int } v \ \text{XOR } \text{Word.the-int } w \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{mask } n :: 'a::\text{len word}) = \text{mask } (\text{min } \text{LENGTH}('a) \ n) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{set-bit } n \ w = w \ \text{OR } \text{push-bit } n \ 1 \rangle \ \mathbf{for} \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{unset-bit } n \ w = w \ \text{AND } \text{NOT } (\text{push-bit } n \ 1) \rangle \ \mathbf{for} \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *[code]*:
 $\langle \text{flip-bit } n \ w = w \text{ XOR } \text{push-bit } n \ 1 \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

context
includes *lifting-syntax*
begin

lemma *set-bit-word-transfer [transfer-rule]*:
 $\langle ((=) == => \text{pcr-word} == => \text{pcr-word}) \text{ set-bit set-bit} \rangle$
 $\langle \text{proof} \rangle$

lemma *unset-bit-word-transfer [transfer-rule]*:
 $\langle ((=) == => \text{pcr-word} == => \text{pcr-word}) \text{ unset-bit unset-bit} \rangle$
 $\langle \text{proof} \rangle$

lemma *flip-bit-word-transfer [transfer-rule]*:
 $\langle ((=) == => \text{pcr-word} == => \text{pcr-word}) \text{ flip-bit flip-bit} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-word-transfer [transfer-rule]*:
 $\langle ((=) == => \text{pcr-word} == => \text{pcr-word})$
 $(\lambda n \ k. \text{signed-take-bit } n \ (\text{take-bit } \text{LENGTH}('a::\text{len}) \ k))$
 $(\text{signed-take-bit} :: \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}) \rangle$
 $\langle \text{proof} \rangle$

end

end

98.5 Conversions including casts

98.5.1 Generic unsigned conversion

context *semiring-bits*
begin

lemma *bit-unsigned-iff [bit-simps]*:
 $\langle \text{bit } (\text{unsigned } w) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } w \ n \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

end

lemma *possible-bit-word[simp]*:
 $\langle \text{possible-bit } \text{TYPE}(('a :: \text{len}) \text{ word}) \ m \longleftrightarrow m < \text{LENGTH}('a) \rangle$
 $\langle \text{proof} \rangle$

context *semiring-bit-operations*

begin

lemma *unsigned-minus-1-eq-mask*:

$\langle \text{unsigned } (-1 :: 'b::\text{len word}) = \text{mask } \text{LENGTH}('b) \rangle$
 $\langle \text{proof} \rangle$

lemma *unsigned-push-bit-eq*:

$\langle \text{unsigned } (\text{push-bit } n \ w) = \text{take-bit } \text{LENGTH}('b) \ (\text{push-bit } n \ (\text{unsigned } w)) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *unsigned-take-bit-eq*:

$\langle \text{unsigned } (\text{take-bit } n \ w) = \text{take-bit } n \ (\text{unsigned } w) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

end

context *unique-euclidean-semiring-with-bit-operations*

begin

lemma *unsigned-drop-bit-eq*:

$\langle \text{unsigned } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{take-bit } \text{LENGTH}('b) \ (\text{unsigned } w)) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

end

lemma *ucast-drop-bit-eq*:

$\langle \text{ucast } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{ucast } w :: 'b::\text{len word}) \rangle$
if $\langle \text{LENGTH}('a) \leq \text{LENGTH}('b) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

context *semiring-bit-operations*

begin

context

includes *bit-operations-syntax*

begin

lemma *unsigned-and-eq*:

$\langle \text{unsigned } (v \ \text{AND} \ w) = \text{unsigned } v \ \text{AND} \ \text{unsigned } w \rangle$
for $v \ w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *unsigned-or-eq*:

$\langle \text{unsigned } (v \ \text{OR} \ w) = \text{unsigned } v \ \text{OR} \ \text{unsigned } w \rangle$
for $v \ w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$


```

lemma unsigned-xor-eq:
   $\langle \text{unsigned } (v \text{ XOR } w) = \text{unsigned } v \text{ XOR } \text{unsigned } w \rangle$ 
  for  $v \ w :: \langle 'b::\text{len word} \rangle$ 
   $\langle \text{proof} \rangle$ 

end

end

context ring-bit-operations
begin

context
  includes bit-operations-syntax
begin

lemma unsigned-not-eq:
   $\langle \text{unsigned } (\text{NOT } w) = \text{take-bit } \text{LENGTH}('b) (\text{NOT } (\text{unsigned } w)) \rangle$ 
  for  $w :: \langle 'b::\text{len word} \rangle$ 
   $\langle \text{proof} \rangle$ 

end

end

context unique-euclidean-semiring-numeral
begin

lemma unsigned-greater-eq [simp]:
   $\langle 0 \leq \text{unsigned } w \rangle$  for  $w :: \langle 'b::\text{len word} \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma unsigned-less [simp]:
   $\langle \text{unsigned } w < 2^{\text{LENGTH}('b)} \rangle$  for  $w :: \langle 'b::\text{len word} \rangle$ 
   $\langle \text{proof} \rangle$ 

end

context linordered-semidom
begin

lemma word-less-eq-iff-unsigned:
   $a \leq b \longleftrightarrow \text{unsigned } a \leq \text{unsigned } b$ 
   $\langle \text{proof} \rangle$ 

lemma word-less-iff-unsigned:
   $a < b \longleftrightarrow \text{unsigned } a < \text{unsigned } b$ 
   $\langle \text{proof} \rangle$ 

```

end

98.5.2 Generic signed conversion

context *ring-bit-operations*
begin

lemma *bit-signed-iff* [*bit-simps*]:

$\langle \text{bit } (\text{signed } w) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } w \ (\text{min } (\text{LENGTH}('b) - \text{Suc } 0) \ n) \rangle$
for $w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-push-bit-eq*:

$\langle \text{signed } (\text{push-bit } n \ w) = \text{signed-take-bit } (\text{LENGTH}('b) - \text{Suc } 0) \ (\text{push-bit } n \ (\text{signed } w :: 'a)) \rangle$
for $w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-eq*:

$\langle \text{signed } (\text{take-bit } n \ w) = (\text{if } n < \text{LENGTH}('b) \text{ then } \text{take-bit } n \ (\text{signed } w) \text{ else } \text{signed } w) \rangle$
for $w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

context
includes *bit-operations-syntax*
begin

lemma *signed-not-eq*:

$\langle \text{signed } (\text{NOT } w) = \text{signed-take-bit } \text{LENGTH}('b) \ (\text{NOT } (\text{signed } w)) \rangle$
for $w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-and-eq*:

$\langle \text{signed } (v \ \text{AND } w) = \text{signed } v \ \text{AND } \text{signed } w \rangle$
for $v \ w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-or-eq*:

$\langle \text{signed } (v \ \text{OR } w) = \text{signed } v \ \text{OR } \text{signed } w \rangle$
for $v \ w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-xor-eq*:

$\langle \text{signed } (v \ \text{XOR } w) = \text{signed } v \ \text{XOR } \text{signed } w \rangle$
for $v \ w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

end

end

98.5.3 More

lemma *sint-greater-eq*:

$\langle - (2^{\wedge} (LENGTH('a) - Suc\ 0)) \leq sint\ w \rangle$ **for** $w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *sint-less*:

$\langle sint\ w < 2^{\wedge} (LENGTH('a) - Suc\ 0) \rangle$ **for** $w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *unat-div-distrib*:

$\langle unat\ (v\ div\ w) = unat\ v\ div\ unat\ w \rangle$
 $\langle proof \rangle$

lemma *unat-mod-distrib*:

$\langle unat\ (v\ mod\ w) = unat\ v\ mod\ unat\ w \rangle$
 $\langle proof \rangle$

lemma *uint-div-distrib*:

$\langle uint\ (v\ div\ w) = uint\ v\ div\ uint\ w \rangle$
 $\langle proof \rangle$

lemma *unat-drop-bit-eq*:

$\langle unat\ (drop-bit\ n\ w) = drop-bit\ n\ (unat\ w) \rangle$
 $\langle proof \rangle$

lemma *uint-mod-distrib*:

$\langle uint\ (v\ mod\ w) = uint\ v\ mod\ uint\ w \rangle$
 $\langle proof \rangle$

context *semiring-bit-operations*

begin

lemma *unsigned-ucast-eq*:

$\langle unsigned\ (ucast\ w :: 'c::len\ word) = take-bit\ LENGTH('c)\ (unsigned\ w) \rangle$
for $w :: \langle 'b::len\ word \rangle$
 $\langle proof \rangle$

end

context *ring-bit-operations*

begin

lemma *signed-ucast-eq*:

$\langle \text{signed } (\text{ucast } w :: 'c::\text{len word}) = \text{signed-take-bit } (\text{LENGTH}('c) - \text{Suc } 0) (\text{unsigned } w) \rangle$
for $w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-scast-eq*:

$\langle \text{signed } (\text{scast } w :: 'c::\text{len word}) = \text{signed-take-bit } (\text{LENGTH}('c) - \text{Suc } 0) (\text{signed } w) \rangle$
for $w :: 'b::\text{len word}$
 $\langle \text{proof} \rangle$

end

lemma *uint-nonnegative*: $0 \leq \text{uint } w$

$\langle \text{proof} \rangle$

lemma *uint-bounded*: $\text{uint } w < 2^{\text{LENGTH}('a)}$

for $w :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-idem*: $\text{uint } w \bmod 2^{\text{LENGTH}('a)} = \text{uint } w$

for $w :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *word-uint-eqI*: $\text{uint } a = \text{uint } b \implies a = b$

$\langle \text{proof} \rangle$

lemma *word-uint-eq-iff*: $a = b \iff \text{uint } a = \text{uint } b$

$\langle \text{proof} \rangle$

lemma *uint-word-of-int-eq*:

$\langle \text{uint } (\text{word-of-int } k :: 'a::\text{len word}) = \text{take-bit } \text{LENGTH}('a) \ k \rangle$

$\langle \text{proof} \rangle$

lemma *uint-word-of-int*: $\text{uint } (\text{word-of-int } k :: 'a::\text{len word}) = k \bmod 2^{\text{LENGTH}('a)}$

$\langle \text{proof} \rangle$

lemma *word-of-int-uint*: $\text{word-of-int } (\text{uint } w) = w$

$\langle \text{proof} \rangle$

lemma *word-div-def* [code]:

$a \text{ div } b = \text{word-of-int } (\text{uint } a \text{ div } \text{uint } b)$

$\langle \text{proof} \rangle$

lemma *word-mod-def* [code]:

$a \bmod b = \text{word-of-int } (\text{uint } a \bmod \text{uint } b)$

$\langle \text{proof} \rangle$

lemma *split-word-all*: $(\bigwedge x::'a::\text{len word}. \text{PROP } P \ x) \equiv (\bigwedge x. \text{PROP } P \ (\text{word-of-int } x))$

$x))$
 $\langle proof \rangle$

lemma *sint-uint*:
 $\langle sint\ w = signed-take-bit\ (LENGTH('a) - Suc\ 0)\ (uint\ w) \rangle$
for $w :: 'a::len\ word$
 $\langle proof \rangle$

lemma *unat-eq-nat-uint*:
 $\langle unat\ w = nat\ (uint\ w) \rangle$
 $\langle proof \rangle$

lemma *ucast-eq*:
 $\langle ucast\ w = word-of-int\ (uint\ w) \rangle$
 $\langle proof \rangle$

lemma *scast-eq*:
 $\langle scast\ w = word-of-int\ (sint\ w) \rangle$
 $\langle proof \rangle$

lemma *uint-0-eq*:
 $\langle uint\ 0 = 0 \rangle$
 $\langle proof \rangle$

lemma *uint-1-eq*:
 $\langle uint\ 1 = 1 \rangle$
 $\langle proof \rangle$

lemma *word-m1-wi*: $- 1 = word-of-int\ (- 1)$
 $\langle proof \rangle$

lemma *uint-0-iff*: $uint\ x = 0 \longleftrightarrow x = 0$
 $\langle proof \rangle$

lemma *unat-0-iff*: $unat\ x = 0 \longleftrightarrow x = 0$
 $\langle proof \rangle$

lemma *unat-0*: $unat\ 0 = 0$
 $\langle proof \rangle$

lemma *unat-gt-0*: $0 < unat\ x \longleftrightarrow x \neq 0$
 $\langle proof \rangle$

lemma *ucast-0*: $ucast\ 0 = 0$
 $\langle proof \rangle$

lemma *sint-0*: $sint\ 0 = 0$
 $\langle proof \rangle$

```

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 $\langle \lambda-. \text{LENGTH}('a) \leq \text{LENGTH}('b) \rangle \langle \text{proof} \rangle$

lift-definition $\text{is-down} :: \langle ('a::\text{len word} \Rightarrow 'b::\text{len word}) \Rightarrow \text{bool} \rangle$
is $\langle \lambda-. \text{LENGTH}('a) \geq \text{LENGTH}('b) \rangle \langle \text{proof} \rangle$

lemma is-up-eq :

$\langle \text{is-up } f \longleftrightarrow \text{source-size } f \leq \text{target-size } f \rangle$
for $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma is-down-eq :

$\langle \text{is-down } f \longleftrightarrow \text{target-size } f \leq \text{source-size } f \rangle$
for $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lift-definition $\text{word-int-case} :: \langle (\text{int} \Rightarrow 'b) \Rightarrow 'a::\text{len word} \Rightarrow 'b \rangle$
is $\langle \lambda f. f \circ \text{take-bit LENGTH}('a) \rangle \langle \text{proof} \rangle$

lemma $\text{word-int-case-eq-uint}$ $[\text{code}]$:

$\langle \text{word-int-case } f \ w = f \ (\text{uint } w) \rangle$
 $\langle \text{proof} \rangle$

translations

$\text{case } x \text{ of } XCONST \text{ of-int } y \Rightarrow b \Rightarrow CONST \text{ word-int-case } (\lambda y. b) \ x$
 $\text{case } x \text{ of } (XCONST \text{ of-int } :: 'a) \ y \Rightarrow b \rightarrow CONST \text{ word-int-case } (\lambda y. b) \ x$

98.6 Arithmetic operations

lemma div-word-self :

$\langle w \text{ div } w = 1 \rangle$ **if** $\langle w \neq 0 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma mod-word-self $[\text{simp}]$:

$\langle w \text{ mod } w = 0 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma div-word-less :

$\langle w \text{ div } v = 0 \rangle$ **if** $\langle w < v \rangle$ **for** $w \ v :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma mod-word-less :

$\langle w \text{ mod } v = w \rangle$ **if** $\langle w < v \rangle$ **for** $w \ v :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma div-word-one $[\text{simp}]$:

$\langle 1 \text{ div } w = \text{of-bool } (w = 1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma mod-word-one $[\text{simp}]$:

$\langle 1 \bmod w = 1 - w * \text{of-bool } (w = 1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *div-word-by-minus-1-eq* [simp]:
 $\langle w \text{ div } - 1 = \text{of-bool } (w = - 1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *mod-word-by-minus-1-eq* [simp]:
 $\langle w \bmod - 1 = w * \text{of-bool } (w < - 1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

Legacy theorems:

lemma *word-add-def* [code]:
 $a + b = \text{word-of-int } (\text{uint } a + \text{uint } b)$
 $\langle \text{proof} \rangle$

lemma *word-sub-wi* [code]:
 $a - b = \text{word-of-int } (\text{uint } a - \text{uint } b)$
 $\langle \text{proof} \rangle$

lemma *word-mult-def* [code]:
 $a * b = \text{word-of-int } (\text{uint } a * \text{uint } b)$
 $\langle \text{proof} \rangle$

lemma *word-minus-def* [code]:
 $- a = \text{word-of-int } (- \text{uint } a)$
 $\langle \text{proof} \rangle$

lemma *word-0-wi*:
 $0 = \text{word-of-int } 0$
 $\langle \text{proof} \rangle$

lemma *word-1-wi*:
 $1 = \text{word-of-int } 1$
 $\langle \text{proof} \rangle$

lift-definition *word-succ* :: $'a::\text{len word} \Rightarrow 'a \text{ word}$ **is** $\lambda x. x + 1$
 $\langle \text{proof} \rangle$

lift-definition *word-pred* :: $'a::\text{len word} \Rightarrow 'a \text{ word}$ **is** $\lambda x. x - 1$
 $\langle \text{proof} \rangle$

lemma *word-succ-alt* [code]:
 $\text{word-succ } a = \text{word-of-int } (\text{uint } a + 1)$
 $\langle \text{proof} \rangle$

lemma *word-pred-alt* [code]:
 $\text{word-pred } a = \text{word-of-int } (\text{uint } a - 1)$
 $\langle \text{proof} \rangle$

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*: $\text{word-of-int } a + \text{word-of-int } b = \text{word-of-int } (a + b)$
and *wi-hom-sub*: $\text{word-of-int } a - \text{word-of-int } b = \text{word-of-int } (a - b)$
and *wi-hom-mult*: $\text{word-of-int } a * \text{word-of-int } b = \text{word-of-int } (a * b)$
and *wi-hom-neg*: $-\text{word-of-int } a = \text{word-of-int } (-a)$
and *wi-hom-succ*: $\text{word-succ } (\text{word-of-int } a) = \text{word-of-int } (a + 1)$
and *wi-hom-pred*: $\text{word-pred } (\text{word-of-int } a) = \text{word-of-int } (a - 1)$
 $\langle \text{proof} \rangle$

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*:
 $\langle 2 * a = 0 \longleftrightarrow a = 0 \vee a = 2 \wedge (\text{LENGTH}('a) - \text{Suc } 0) \rangle$
for $a :: \langle 'a :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

98.7 Ordering

lift-definition *word-sle* :: $\langle 'a :: \text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$
is $\langle \lambda k l. \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) k \leq \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) l \rangle$
 $\langle \text{proof} \rangle$

lift-definition *word-sless* :: $\langle 'a :: \text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$
is $\langle \lambda k l. \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) k < \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) l \rangle$
 $\langle \text{proof} \rangle$

notation
word-sle $(\langle '(\leq s') \rangle)$ **and**
word-sle $((-/ \leq s -) \text{ [51, 51] } 50)$ **and**
word-sless $(\langle '(< s') \rangle)$ **and**
word-sless $((-/ < s -) \text{ [51, 51] } 50)$

notation (*input*)
word-sle $((-/ \leq s -) \text{ [51, 51] } 50)$

lemma *word-sle-eq* [*code*]:
 $\langle a \leq s b \longleftrightarrow \text{sint } a \leq \text{sint } b \rangle$

$\langle \text{proof} \rangle$

lemma [code]:

$\langle a <_s b \iff \text{sint } a < \text{sint } b \rangle$

$\langle \text{proof} \rangle$

lemma *signed-ordering*: $\langle \text{ordering word-sle word-sless} \rangle$

$\langle \text{proof} \rangle$

lemma *signed-linorder*: $\langle \text{class.linorder word-sle word-sless} \rangle$

$\langle \text{proof} \rangle$

interpretation *signed*: *linorder word-sle word-sless*

$\langle \text{proof} \rangle$

lemma *word-sless-eq*:

$\langle x <_s y \iff x \leq_s y \wedge x \neq y \rangle$

$\langle \text{proof} \rangle$

lemma *word-less-alt*: $a < b \iff \text{uint } a < \text{uint } b$

$\langle \text{proof} \rangle$

lemma *word-zero-le* [simp]: $0 \leq y$

for $y :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *word-m1-ge* [simp] : $\text{word-pred } 0 \geq y$

$\langle \text{proof} \rangle$

lemma *word-n1-ge* [simp]: $y \leq -1$

for $y :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemmas *word-not-simps* [simp] =

word-zero-le [THEN leD] *word-m1-ge* [THEN leD] *word-n1-ge* [THEN leD]

lemma *word-gt-0*: $0 < y \iff 0 \neq y$

for $y :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemmas *word-gt-0-no* [simp] = *word-gt-0* [of numeral y] **for** y

lemma *word-sless-alt*: $a <_s b \iff \text{sint } a < \text{sint } b$

$\langle \text{proof} \rangle$

lemma *word-le-nat-alt*: $a \leq b \iff \text{unat } a \leq \text{unat } b$

$\langle \text{proof} \rangle$

lemma *word-less-nat-alt*: $a < b \iff \text{unat } a < \text{unat } b$

$\langle \text{proof} \rangle$

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1*]

instance *word* :: (*len*) *wellorder*

$\langle \text{proof} \rangle$

lemma *wi-less*:

$(\text{word-of-int } n < (\text{word-of-int } m :: 'a::\text{len word})) =$
 $(n \bmod 2 \wedge \text{LENGTH}('a) < m \bmod 2 \wedge \text{LENGTH}('a))$
 $\langle \text{proof} \rangle$

lemma *wi-le*:

$(\text{word-of-int } n \leq (\text{word-of-int } m :: 'a::\text{len word})) =$
 $(n \bmod 2 \wedge \text{LENGTH}('a) \leq m \bmod 2 \wedge \text{LENGTH}('a))$
 $\langle \text{proof} \rangle$

98.8 Bit-wise operations

context

includes *bit-operations-syntax*

begin

lemma *uint-take-bit-eq*:

$\langle \text{uint } (\text{take-bit } n \ w) = \text{take-bit } n \ (\text{uint } w) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-word-eq-self*:

$\langle \text{take-bit } n \ w = w \rangle \text{ if } \langle \text{LENGTH}('a) \leq n \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-length-eq* [*simp*]:

$\langle \text{take-bit } \text{LENGTH}('a) \ w = w \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-of-int-iff*:

$\langle \text{bit } (\text{word-of-int } k :: 'a::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } k \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-uint-iff*:

$\langle \text{bit } (\text{uint } w) \ n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-sint-iff*:

$\langle \text{bit } (\text{sint } w) \ n \longleftrightarrow n \geq \text{LENGTH}('a) \wedge \text{bit } w \ (\text{LENGTH}('a) - 1) \vee \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-ucast-iff*:

$\langle \text{bit } (\text{ucast } w :: 'b::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}('a) \wedge n < \text{LENGTH}('b) \wedge \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-scast-iff*:

$\langle \text{bit } (\text{scast } w :: 'b::\text{len word}) \ n \longleftrightarrow$
 $n < \text{LENGTH}('b) \wedge (\text{bit } w \ n \vee \text{LENGTH}('a) \leq n \wedge \text{bit } w \ (\text{LENGTH}('a) -$
 $\text{Suc } 0)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-iff-drop-bit-and* [code]:

$\langle \text{bit } a \ n \longleftrightarrow \text{drop-bit } n \ a \ \text{AND } 1 = 1 \rangle$ **for** $a :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma

word-not-def: $\text{NOT } (a :: 'a::\text{len word}) = \text{word-of-int } (\text{NOT } (\text{uint } a))$
and *word-and-def*: $(a :: 'a \text{ word}) \ \text{AND } b = \text{word-of-int } (\text{uint } a \ \text{AND } \text{uint } b)$
and *word-or-def*: $(a :: 'a \text{ word}) \ \text{OR } b = \text{word-of-int } (\text{uint } a \ \text{OR } \text{uint } b)$
and *word-xor-def*: $(a :: 'a \text{ word}) \ \text{XOR } b = \text{word-of-int } (\text{uint } a \ \text{XOR } \text{uint } b)$
 $\langle \text{proof} \rangle$

definition *even-word* :: $\langle 'a::\text{len word} \Rightarrow \text{bool} \rangle$

where [code-abbrev]: $\langle \text{even-word} = \text{even} \rangle$

lemma *even-word-iff* [code]:

$\langle \text{even-word } a \longleftrightarrow a \ \text{AND } 1 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *map-bit-range-eq-if-take-bit-eq*:

$\langle \text{map } (\text{bit } k) \ [0..<n] = \text{map } (\text{bit } l) \ [0..<n] \rangle$
if $\langle \text{take-bit } n \ k = \text{take-bit } n \ l \rangle$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma

take-bit-word-Bit0-eq [simp]: $\langle \text{take-bit } (\text{numeral } n) \ (\text{numeral } (\text{num.Bit0 } m)) ::$
 $'a::\text{len word})$
 $= 2 * \text{take-bit } (\text{pred-numeral } n) \ (\text{numeral } m) \rangle$ (**is** ?P)
and *take-bit-word-Bit1-eq* [simp]: $\langle \text{take-bit } (\text{numeral } n) \ (\text{numeral } (\text{num.Bit1 } m))$
 $:: 'a::\text{len word})$
 $= 1 + 2 * \text{take-bit } (\text{pred-numeral } n) \ (\text{numeral } m) \rangle$ (**is** ?Q)
and *take-bit-word-minus-Bit0-eq* [simp]: $\langle \text{take-bit } (\text{numeral } n) \ (- \text{numeral } (\text{num.Bit0}$
 $m)) :: 'a::\text{len word})$
 $= 2 * \text{take-bit } (\text{pred-numeral } n) \ (- \text{numeral } m) \rangle$ (**is** ?R)
and *take-bit-word-minus-Bit1-eq* [simp]: $\langle \text{take-bit } (\text{numeral } n) \ (- \text{numeral } (\text{num.Bit1}$
 $m)) :: 'a::\text{len word})$
 $= 1 + 2 * \text{take-bit } (\text{pred-numeral } n) \ (- \text{numeral } (\text{Num.inc } m)) \rangle$ (**is** ?S)

$\langle \text{proof} \rangle$

98.9 More shift operations

lift-definition *signed-drop-bit* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \lambda n. \text{drop-bit } n \circ \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-signed-drop-bit-iff* [bit-simps]:
 $\langle \text{bit } (\text{signed-drop-bit } m \ w) \ n \longleftrightarrow \text{bit } w \ (\text{if } \text{LENGTH}('a) - m \leq n \wedge n < \text{LENGTH}('a) \text{ then } \text{LENGTH}('a) - 1 \text{ else } m + n) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{signed-drop-bit } n \ w) = \text{take-bit } \text{LENGTH}('a) \ (\text{drop-bit } n \ (\text{Word.the-signed-int } w)) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-of-0* [simp]:
 $\langle \text{signed-drop-bit } n \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-of-minus-1* [simp]:
 $\langle \text{signed-drop-bit } n \ (-1) = -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-signed-drop-bit* [simp]:
 $\langle \text{signed-drop-bit } m \ (\text{signed-drop-bit } n \ w) = \text{signed-drop-bit } (m + n) \ w \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-0* [simp]:
 $\langle \text{signed-drop-bit } 0 \ w = w \rangle$
 $\langle \text{proof} \rangle$

lemma *sint-signed-drop-bit-eq*:
 $\langle \text{sint } (\text{signed-drop-bit } n \ w) = \text{drop-bit } n \ (\text{sint } w) \rangle$
 $\langle \text{proof} \rangle$

98.10 Rotation

lift-definition *word-rotr* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \lambda n \ k. \text{concat-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a))$
 $(\text{drop-bit } (n \bmod \text{LENGTH}('a)) \ (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (n \bmod \text{LENGTH}('a)) \ k) \rangle$
 $\langle \text{proof} \rangle$

lift-definition *word-rotl* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$

is $\langle \lambda n k. \text{concat-bit } (n \bmod \text{LENGTH}('a))$
 $(\text{drop-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) k))$
 $(\text{take-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) k) \rangle$
 $\langle \text{proof} \rangle$

lift-definition $\text{word-roti} :: \langle \text{int} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \lambda r k. \text{concat-bit } (\text{LENGTH}('a) - \text{nat } (r \bmod \text{int } \text{LENGTH}('a)))$
 $(\text{drop-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) (\text{take-bit } \text{LENGTH}('a) k))$
 $(\text{take-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) k) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-rotl-eq-word-rotr}$ [code]:
 $\langle \text{word-rotl } n = (\text{word-rotr } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) :: 'a::\text{len word}$
 $\Rightarrow 'a \text{ word}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-roti-eq-word-rotr-word-rotl}$ [code]:
 $\langle \text{word-roti } i w =$
 $(\text{if } i \geq 0 \text{ then word-rotr } (\text{nat } i) w \text{ else word-rotl } (\text{nat } (- i)) w) \rangle$
 $\langle \text{proof} \rangle$

lemma bit-word-rotr-iff [bit-simps]:
 $\langle \text{bit } (\text{word-rotr } m w) n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } w ((n + m) \bmod \text{LENGTH}('a)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma bit-word-rotl-iff [bit-simps]:
 $\langle \text{bit } (\text{word-rotl } m w) n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } w ((n + (\text{LENGTH}('a) - m \bmod \text{LENGTH}('a))) \bmod$
 $\text{LENGTH}('a)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma bit-word-roti-iff [bit-simps]:
 $\langle \text{bit } (\text{word-roti } k w) n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } w (\text{nat } ((\text{int } n + k) \bmod \text{int } \text{LENGTH}('a))) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma uint-word-rotr-eq :
 $\langle \text{uint } (\text{word-rotr } n w) = \text{concat-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a))$
 $(\text{drop-bit } (n \bmod \text{LENGTH}('a)) (\text{uint } w))$
 $(\text{uint } (\text{take-bit } (n \bmod \text{LENGTH}('a)) w)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{word-rotr } n w) = \text{concat-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a))$

```

    (drop-bit (n mod LENGTH('a)) (Word.the-int w))
    (Word.the-int (take-bit (n mod LENGTH('a)) w))
  for w :: ⟨'a::len word⟩
  ⟨proof⟩

```

98.11 Split and cat operations

lift-definition *word-cat* :: ⟨'a::len word ⇒ 'b::len word ⇒ 'c::len word⟩
is ⟨λk l. concat-bit LENGTH('b) l (take-bit LENGTH('a) k)⟩
 ⟨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⟩

98.12 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 $w :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *wi-bintr*:
 $LENGTH('a::len) \leq n \implies$
 $\text{word-of-int } (\text{take-bit } n \ w) = (\text{word-of-int } w :: 'a \text{ word})$
 $\langle \text{proof} \rangle$

lemma *word-numeral-alt*: $\text{numeral } b = \text{word-of-int } (\text{numeral } b)$
 $\langle \text{proof} \rangle$

declare *word-numeral-alt* [*symmetric, code-abbrev*]

lemma *word-neg-numeral-alt*: $- \text{numeral } b = \text{word-of-int } (- \text{numeral } b)$
 $\langle \text{proof} \rangle$

declare *word-neg-numeral-alt* [*symmetric, code-abbrev*]

lemma *uint-bintrunc* [*simp*]:
 $\text{uint } (\text{numeral } bin :: 'a \text{ word}) =$
 $\text{take-bit } (LENGTH('a::len)) (\text{numeral } bin)$
 $\langle \text{proof} \rangle$

lemma *uint-bintrunc-neg* [*simp*]:
 $\text{uint } (- \text{numeral } bin :: 'a \text{ word}) = \text{take-bit } (LENGTH('a::len)) (- \text{numeral } bin)$
 $\langle \text{proof} \rangle$

lemma *sint-sbintrunc* [*simp*]:
 $\text{sint } (\text{numeral } bin :: 'a \text{ word}) = \text{signed-take-bit } (LENGTH('a::len) - 1) (\text{numeral } bin)$
 $\langle \text{proof} \rangle$

lemma *sint-sbintrunc-neg* [*simp*]:
 $\text{sint } (- \text{numeral } bin :: 'a \text{ word}) = \text{signed-take-bit } (LENGTH('a::len) - 1) (- \text{numeral } bin)$
 $\langle \text{proof} \rangle$

lemma *unat-bintrunc* [*simp*]:
 $\text{unat } (\text{numeral } bin :: 'a::len \text{ word}) = \text{nat } (\text{take-bit } (LENGTH('a)) (\text{numeral } bin))$
 $\langle \text{proof} \rangle$

lemma *unat-bintrunc-neg* [*simp*]:
 $\text{unat } (- \text{numeral } bin :: 'a::len \text{ word}) = \text{nat } (\text{take-bit } (LENGTH('a)) (- \text{numeral } bin))$
 $\langle \text{proof} \rangle$

lemma *size-0-eq*: $\text{size } w = 0 \implies v = w$
for $v \ w :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *uint-ge-0* [iff]: $0 \leq \text{uint } x$
 ⟨proof⟩

lemma *uint-lt2p* [iff]: $\text{uint } x < 2^{\wedge} \text{LENGTH}('a)$
 for $x :: 'a::\text{len word}$
 ⟨proof⟩

lemma *sint-ge*: $-(2^{\wedge} (\text{LENGTH}('a) - 1)) \leq \text{sint } x$
 for $x :: 'a::\text{len word}$
 ⟨proof⟩

lemma *sint-lt*: $\text{sint } x < 2^{\wedge} (\text{LENGTH}('a) - 1)$
 for $x :: 'a::\text{len word}$
 ⟨proof⟩

lemma *uint-m2p-neg*: $\text{uint } x - 2^{\wedge} \text{LENGTH}('a) < 0$
 for $x :: 'a::\text{len word}$
 ⟨proof⟩

lemma *uint-m2p-not-non-neg*: $\neg 0 \leq \text{uint } x - 2^{\wedge} \text{LENGTH}('a)$
 for $x :: 'a::\text{len word}$
 ⟨proof⟩

lemma *lt2p-lem*: $\text{LENGTH}('a) \leq n \implies \text{uint } w < 2^{\wedge} n$
 for $w :: 'a::\text{len word}$
 ⟨proof⟩

lemma *uint-le-0-iff* [simp]: $\text{uint } x \leq 0 \longleftrightarrow \text{uint } x = 0$
 ⟨proof⟩

lemma *uint-nat*: $\text{uint } w = \text{int } (\text{unat } w)$
 ⟨proof⟩

lemma *uint-numeral*: $\text{uint } (\text{numeral } b :: 'a::\text{len word}) = \text{numeral } b \bmod 2^{\wedge} \text{LENGTH}('a)$
 ⟨proof⟩

lemma *uint-neg-numeral*: $\text{uint } (- \text{numeral } b :: 'a::\text{len word}) = - \text{numeral } b \bmod 2^{\wedge} \text{LENGTH}('a)$
 ⟨proof⟩

lemma *unat-numeral*: $\text{unat } (\text{numeral } b :: 'a::\text{len word}) = \text{numeral } b \bmod 2^{\wedge} \text{LENGTH}('a)$
 ⟨proof⟩

lemma *sint-numeral*:
 $\text{sint } (\text{numeral } b :: 'a::\text{len word}) =$
 $(\text{numeral } b + 2^{\wedge} (\text{LENGTH}('a) - 1)) \bmod 2^{\wedge} \text{LENGTH}('a) - 2^{\wedge} (\text{LENGTH}('a) - 1)$

– 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-of-int-neg-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*

$\langle \text{proof} \rangle$

lemma *word-unat-eq-iff*:
 $\langle v = w \longleftrightarrow \text{unat } v = \text{unat } w \rangle$
for $v \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

98.13 Testing bits

lemma *bin-nth-uint-imp*: $\text{bit } (\text{uint } w) \ n \implies n < \text{LENGTH}('a)$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *bin-nth-sint*:
 $\text{LENGTH}('a) \leq n \implies$
 $\text{bit } (\text{sint } w) \ n = \text{bit } (\text{sint } w) \ (\text{LENGTH}('a) - 1)$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *num-of-bintr'*:
 $\text{take-bit } (\text{LENGTH}('a::\text{len})) \ (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$
 $\langle \text{proof} \rangle$

lemma *num-of-sbintr'*:
 $\text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - 1) \ (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$
 $\langle \text{proof} \rangle$

lemma *num-abs-bintr*:
 $(\text{numeral } x :: 'a \text{ word}) =$
 $\text{word-of-int } (\text{take-bit } (\text{LENGTH}('a::\text{len})) \ (\text{numeral } x))$
 $\langle \text{proof} \rangle$

lemma *num-abs-sbintr*:
 $(\text{numeral } x :: 'a \text{ word}) =$
 $\text{word-of-int } (\text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - 1) \ (\text{numeral } x))$
 $\langle \text{proof} \rangle$

cast – note, no arg for new length, as it’s determined by type of result,
thus in $\text{cast } w = w$, the type means cast to length of w !

lemma *bit-ucast-iff*:
 $\langle \text{bit } (\text{ucast } a :: 'a::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}('a::\text{len}) \wedge \text{bit } a \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-id [simp]*: $\text{ucast } w = w$
 $\langle \text{proof} \rangle$

lemma *scast-id [simp]*: $\text{scast } w = w$

⟨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 ⟷ LENGTH('b) ≤ LENGTH('a)

for c :: 'a::len word ⇒ 'b::len word
 ⟨proof⟩

lemma *is-up*: is-up c ⟷ LENGTH('a) ≤ LENGTH('b)

for c :: 'a::len word ⇒ 'b::len word
 ⟨proof⟩

lemma *is-up-down*:

⟨is-up c ⟷ is-down d⟩
 for c :: 'a::len word ⇒ 'b::len word
 and d :: 'b::len word ⇒ 'a::len word
 ⟨proof⟩

context

fixes dummy-types :: 'a::len × 'b::len

begin

private abbreviation (input) *UCAST* :: 'a::len word ⇒ 'b::len word

where ⟨UCAST == ucast⟩

private abbreviation (input) *SCAST* :: 'a::len word ⇒ 'b::len word

where ⟨SCAST == scast⟩

lemma *down-cast-same*:

⟨UCAST = scast⟩ if ⟨is-down UCAST⟩

$\langle \text{proof} \rangle$

lemma *sint-up-scast*:

$\langle \text{sint} (\text{SCAST } w) = \text{sint } w \rangle$ **if** $\langle \text{is-up SCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-up-ucast*:

$\langle \text{uint} (\text{UCAST } w) = \text{uint } w \rangle$ **if** $\langle \text{is-up UCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-up-ucast*:

$\langle \text{ucast} (\text{UCAST } w) = \text{ucast } w \rangle$ **if** $\langle \text{is-up UCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-up-ucast-id*:

$\langle \text{ucast} (\text{UCAST } w) = w \rangle$ **if** $\langle \text{is-up UCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *scast-up-scast*:

$\langle \text{scast} (\text{SCAST } w) = \text{scast } w \rangle$ **if** $\langle \text{is-up SCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *scast-up-scast-id*:

$\langle \text{scast} (\text{SCAST } w) = w \rangle$ **if** $\langle \text{is-up SCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *isduu*:

$\langle \text{is-up UCAST} \rangle$ **if** $\langle \text{is-down } d \rangle$
for $d :: \langle 'b \text{ word} \Rightarrow 'a \text{ word} \rangle$
 $\langle \text{proof} \rangle$

lemma *isdus*:

$\langle \text{is-up SCAST} \rangle$ **if** $\langle \text{is-down } d \rangle$
for $d :: \langle 'b \text{ word} \Rightarrow 'a \text{ word} \rangle$
 $\langle \text{proof} \rangle$

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*:

$\langle \text{surj} (\text{ucast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$ **if** $\langle \text{is-up UCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *up-scast-surj*:

$\langle \text{surj} (\text{scast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$ **if** $\langle \text{is-up SCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *down-ucast-inj*:

$\langle \text{inj-on UCAST } A \rangle$ **if** $\langle \text{is-down} (\text{ucast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$

⟨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⟩

98.14 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 $\langle -\ 1 \rangle \langle -\ \text{numeral } b \rangle$, 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]:
 $\langle \text{signed-drop-bit } n\ (1 :: 'a::\text{len word}) \rangle = \text{of\_bool}\ (LENGTH('a) = 1 \vee n = 0)$
 $\langle \text{proof} \rangle$

lemma *take-bit-word-beyond-length-eq*:
 $\langle \text{take-bit } n\ w = w \rangle$ **if** $\langle LENGTH('a) \leq n \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

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'*: $\text{size } w = 0 \implies w = v$
for $v\ w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

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*: $\text{mask } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *mask-Suc-0*: $\text{mask } (\text{Suc } 0) = 1$
 $\langle \text{proof} \rangle$

lemma *bin-last-bintrunc*: $\text{odd } (\text{take-bit } l\ n) \longleftrightarrow l > 0 \wedge \text{odd } n$
 $\langle \text{proof} \rangle$

lemma *push-bit-word-beyond* [simp]:
 $\langle \text{push-bit } n\ w = 0 \rangle$ **if** $\langle LENGTH('a) \leq n \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-word-beyond* [simp]:
 $\langle \text{drop-bit } n\ w = 0 \rangle$ **if** $\langle LENGTH('a) \leq n \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-beyond*:

$\langle \text{signed-drop-bit } n \ w = (\text{if bit } w \ (\text{LENGTH}('a) - \text{Suc } 0) \text{ then } - 1 \text{ else } 0) \rangle$
if $\langle \text{LENGTH}('a) \leq n \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-minus-numeral-word* [simp]:
 $\langle \text{take-bit (numeral } m) \ (- \text{ numeral } n :: 'a::\text{len word}) =$
 $(\text{case take-bit-num (numeral } m) \ n \text{ of None } \Rightarrow 0 \mid \text{Some } q \Rightarrow \text{take-bit (numeral } m) \ (2 \wedge \text{ numeral } m - \text{ numeral } q)) \rangle$ (**is** $\langle ?lhs = ?rhs \rangle$)
 $\langle \text{proof} \rangle$

lemma *of-nat-inverse*:
 $\langle \text{word-of-nat } r = a \implies r < 2 \wedge \text{LENGTH}('a) \implies \text{unat } a = r \rangle$
for $a :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

98.15 Transferring goals from words to ints

lemma *word-ths*:
shows *word-succ-p1*: $\text{word-succ } a = a + 1$
and *word-pred-m1*: $\text{word-pred } a = a - 1$
and *word-pred-succ*: $\text{word-pred } (\text{word-succ } a) = a$
and *word-succ-pred*: $\text{word-succ } (\text{word-pred } a) = a$
and *word-mult-succ*: $\text{word-succ } a * b = b + a * b$
 $\langle \text{proof} \rangle$

lemma *uint-cong*: $x = y \implies \text{uint } x = \text{uint } y$
 $\langle \text{proof} \rangle$

lemma *uint-word-ariths*:
fixes $a \ b :: 'a::\text{len word}$
shows $\text{uint } (a + b) = (\text{uint } a + \text{uint } b) \bmod 2 \wedge \text{LENGTH}('a::\text{len})$
and $\text{uint } (a - b) = (\text{uint } a - \text{uint } b) \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (a * b) = \text{uint } a * \text{uint } b \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (- a) = - \text{uint } a \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (\text{word-succ } a) = (\text{uint } a + 1) \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (\text{word-pred } a) = (\text{uint } a - 1) \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (0 :: 'a \text{ word}) = 0 \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (1 :: 'a \text{ word}) = 1 \bmod 2 \wedge \text{LENGTH}('a)$
 $\langle \text{proof} \rangle$

lemma *uint-word-arith-bintrs*:
fixes $a \ b :: 'a::\text{len word}$
shows $\text{uint } (a + b) = \text{take-bit } (\text{LENGTH}('a)) \ (\text{uint } a + \text{uint } b)$
and $\text{uint } (a - b) = \text{take-bit } (\text{LENGTH}('a)) \ (\text{uint } a - \text{uint } b)$
and $\text{uint } (a * b) = \text{take-bit } (\text{LENGTH}('a)) \ (\text{uint } a * \text{uint } b)$
and $\text{uint } (- a) = \text{take-bit } (\text{LENGTH}('a)) \ (- \text{uint } a)$
and $\text{uint } (\text{word-succ } a) = \text{take-bit } (\text{LENGTH}('a)) \ (\text{uint } a + 1)$
and $\text{uint } (\text{word-pred } a) = \text{take-bit } (\text{LENGTH}('a)) \ (\text{uint } a - 1)$
and $\text{uint } (0 :: 'a \text{ word}) = \text{take-bit } (\text{LENGTH}('a)) \ 0$

and $\text{uint } (1 :: 'a \text{ word}) = \text{take-bit } (\text{LENGTH}('a)) \ 1$
 $\langle \text{proof} \rangle$

lemma *sint-word-ariths*:

fixes $a \ b :: 'a::\text{len word}$

shows $\text{sint } (a + b) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a + \text{sint } b)$
and $\text{sint } (a - b) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a - \text{sint } b)$
and $\text{sint } (a * b) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a * \text{sint } b)$
and $\text{sint } (- a) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (- \text{sint } a)$
and $\text{sint } (\text{word-succ } a) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a + 1)$
and $\text{sint } (\text{word-pred } a) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a - 1)$
and $\text{sint } (0 :: 'a \text{ word}) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) \ 0$
and $\text{sint } (1 :: 'a \text{ word}) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) \ 1$
 $\langle \text{proof} \rangle$

lemma *word-pred-0-n1*: $\text{word-pred } 0 = \text{word-of-int } (- \ 1)$
 $\langle \text{proof} \rangle$

lemma *succ-pred-no [simp]*:

$\text{word-succ } (\text{numeral } w) = \text{numeral } w + 1$
 $\text{word-pred } (\text{numeral } w) = \text{numeral } w - 1$
 $\text{word-succ } (- \text{numeral } w) = - \text{numeral } w + 1$
 $\text{word-pred } (- \text{numeral } w) = - \text{numeral } w - 1$
 $\langle \text{proof} \rangle$

lemma *word-sp-01 [simp]*:

$\text{word-succ } (- \ 1) = 0 \wedge \text{word-succ } 0 = 1 \wedge \text{word-pred } 0 = - \ 1 \wedge \text{word-pred } 1 = 0$
 $\langle \text{proof} \rangle$

lemma *word-of-int-Ex*: $\exists y. x = \text{word-of-int } y$
 $\langle \text{proof} \rangle$

98.16 Order on fixed-length words

lift-definition $\text{udvd} :: \langle 'a::\text{len word} \Rightarrow 'a::\text{len word} \Rightarrow \text{bool} \rangle$ (**infixl** $\langle \text{udvd} \rangle \ 50$)
is $\langle \lambda k \ l. \text{take-bit } \text{LENGTH}('a) \ k \ \text{dvd} \ \text{take-bit } \text{LENGTH}('a) \ l \rangle$ $\langle \text{proof} \rangle$

lemma *udvd-iff-dvd*:

$\langle x \ \text{udvd} \ y \longleftrightarrow \text{unat } x \ \text{dvd} \ \text{unat } y \rangle$
 $\langle \text{proof} \rangle$

lemma *udvd-iff-dvd-int*:

$\langle v \ \text{udvd} \ w \longleftrightarrow \text{uint } v \ \text{dvd} \ \text{uint } w \rangle$
 $\langle \text{proof} \rangle$

lemma *udvdI [intro]*:

$\langle v \ \text{udvd} \ w \rangle$ **if** $\langle \text{unat } w = \text{unat } v * \text{unat } u \rangle$
 $\langle \text{proof} \rangle$

lemma *udvdE* [*elim*]:
 fixes $v\ w :: \langle 'a::len\ word \rangle$
 assumes $\langle v\ udvd\ w \rangle$
 obtains $u :: \langle 'a\ word \rangle$ **where** $\langle unat\ w = unat\ v * unat\ u \rangle$
 $\langle proof \rangle$

lemma *udvd-imp-mod-eq-0*:
 $\langle w\ mod\ v = 0 \rangle$ **if** $\langle v\ udvd\ w \rangle$
 $\langle proof \rangle$

lemma *mod-eq-0-imp-udvd* [*intro?*]:
 $\langle v\ udvd\ w \rangle$ **if** $\langle w\ mod\ v = 0 \rangle$
 $\langle proof \rangle$

lemma *udvd-imp-dvd*:
 $\langle v\ dvd\ w \rangle$ **if** $\langle v\ udvd\ w \rangle$ **for** $v\ w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *exp-dvd-iff-exp-udvd*:
 $\langle 2^{\wedge n}\ dvd\ w \longleftrightarrow 2^{\wedge n}\ udvd\ w \rangle$ **for** $v\ w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *udvd-nat-alt*:
 $\langle a\ udvd\ b \longleftrightarrow (\exists n. unat\ b = n * unat\ a) \rangle$
 $\langle proof \rangle$

lemma *udvd-unfold-int*:
 $\langle a\ udvd\ b \longleftrightarrow (\exists n \geq 0. uint\ b = n * uint\ a) \rangle$
 $\langle proof \rangle$

lemma *unat-minus-one*:
 $\langle unat\ (w - 1) = unat\ w - 1 \rangle$ **if** $\langle w \neq 0 \rangle$
 $\langle proof \rangle$

lemma *measure-unat*: $p \neq 0 \implies unat\ (p - 1) < unat\ p$
 $\langle proof \rangle$

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*]: $uint\ x - uint\ y < 2^{\wedge LENGTH('a)}$
for $x :: 'a::len\ word$ **and** $y :: 'b::len\ word$
 $\langle proof \rangle$

98.17 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:
 $(uint\ x + uint\ y < 2^{\wedge LENGTH('a)}) =$

$(uint\ (x + y) = uint\ x + uint\ y)$
for $x\ y :: 'a::len\ word$
 $\langle proof \rangle$

lemma *uint-mult-lem*:
 $(uint\ x * uint\ y < 2 \wedge LENGTH('a)) =$
 $(uint\ (x * y) = uint\ x * uint\ y)$
for $x\ y :: 'a::len\ word$
 $\langle proof \rangle$

lemma *uint-sub-lem*: $uint\ x \geq uint\ y \longleftrightarrow uint\ (x - y) = uint\ x - uint\ y$
 $\langle proof \rangle$

lemma *uint-add-le*: $uint\ (x + y) \leq uint\ x + uint\ y$
 $\langle proof \rangle$

lemma *uint-sub-ge*: $uint\ (x - y) \geq uint\ x - uint\ y$
 $\langle proof \rangle$

lemma *int-mod-ge*: $\langle a \leq a\ mod\ n \rangle$ **if** $\langle a < n \rangle \langle 0 < n \rangle$
for $a\ n :: int$
 $\langle proof \rangle$

lemma *mod-add-if-z*:
 $\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$
 $(x + y)\ mod\ z = (if\ x + y < z\ then\ x + y\ else\ x + y - z)$
for $x\ y\ z :: int$
 $\langle proof \rangle$

lemma *uint-plus-if'*:
 $uint\ (a + b) =$
 $(if\ uint\ a + uint\ b < 2 \wedge LENGTH('a)\ then\ uint\ a + uint\ b$
 $\quad else\ uint\ a + uint\ b - 2 \wedge LENGTH('a))$
for $a\ b :: 'a::len\ word$
 $\langle proof \rangle$

lemma *mod-sub-if-z*:
 $\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$
 $(x - y)\ mod\ z = (if\ y \leq x\ then\ x - y\ else\ x - y + z)$
for $x\ y\ z :: int$
 $\langle proof \rangle$

lemma *uint-sub-if'*:
 $uint\ (a - b) =$
 $(if\ uint\ b \leq uint\ a\ then\ uint\ a - uint\ b$
 $\quad else\ uint\ a - uint\ b + 2 \wedge LENGTH('a))$
for $a\ b :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-of-int-inverse*:

word-of-int $r = a \implies 0 \leq r \implies r < 2^{\wedge LENGTH('a)} \implies uint\ a = r$

for $a :: 'a::len\ word$

$\langle proof \rangle$

lemma *unat-split*: $P\ (unat\ x) \longleftrightarrow (\forall n. of\_nat\ n = x \wedge n < 2^{\wedge LENGTH('a)} \longrightarrow P\ n)$

for $x :: 'a::len\ word$

$\langle proof \rangle$

lemma *unat-split-asm*: $P\ (unat\ x) \longleftrightarrow (\nexists n. of\_nat\ n = x \wedge n < 2^{\wedge LENGTH('a)} \wedge \neg P\ n)$

for $x :: 'a::len\ word$

$\langle proof \rangle$

lemma *un-ui-le*:

$\langle unat\ a \leq unat\ b \longleftrightarrow uint\ a \leq uint\ b \rangle$

$\langle proof \rangle$

lemma *unat-plus-if'*:

$\langle unat\ (a + b) =$

$(if\ unat\ a + unat\ b < 2^{\wedge LENGTH('a)}$

$then\ unat\ a + unat\ b$

$else\ unat\ a + unat\ b - 2^{\wedge LENGTH('a)}) \rangle$ **for** $a\ b :: \langle 'a::len\ word \rangle$

$\langle proof \rangle$

lemma *unat-sub-if-size*:

$unat\ (x - y) =$

$(if\ unat\ y \leq unat\ x$

$then\ unat\ x - unat\ y$

$else\ unat\ x + 2^{\wedge size\ x} - unat\ y)$

$\langle proof \rangle$

lemmas *unat-sub-if' = unat-sub-if-size* [unfolded word-size]

lemma *uint-split*:

$P\ (uint\ x) = (\forall i. word\_of\_int\ i = x \wedge 0 \leq i \wedge i < 2^{\wedge LENGTH('a)} \longrightarrow P\ i)$

for $x :: 'a::len\ word$

$\langle proof \rangle$

lemma *uint-split-asm*:

$P\ (uint\ x) = (\nexists i. word\_of\_int\ i = x \wedge 0 \leq i \wedge i < 2^{\wedge LENGTH('a)} \wedge \neg P\ i)$

for $x :: 'a::len\ word$

$\langle proof \rangle$

98.18 Some proof tool support

lemma *power-False-cong*: $False \implies a^{\wedge} b = c^{\wedge} d$

$\langle proof \rangle$

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*
 $\langle ML \rangle$

98.19 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*: $x \leq x + y \longleftrightarrow \text{uint } x + \text{uint } y < 2^{\text{size } x}$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *no-olen-add* = *no-plus-overflow-uint-size* [*unfolded word-size*]

lemma *no-ulen-sub*: $x \geq x - y \longleftrightarrow \text{uint } y \leq \text{uint } x$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *no-olen-add'*: $x \leq y + x \longleftrightarrow \text{uint } y + \text{uint } x < 2^{\text{LENGTH}('a)}$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *olen-add-equiv* = *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*: $x \neq 0 \implies 1 < x \longleftrightarrow 0 < x - 1$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-le-sub1*: $x \neq 0 \implies 1 \leq x \longleftrightarrow 0 \leq x - 1$
for $x :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *sub-wrap-lt*: $x < x - z \longleftrightarrow x < z$
for $x\ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *sub-wrap*: $x \leq x - z \longleftrightarrow z = 0 \vee x < z$
for $x\ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *plus-minus-not-NULL-ab*: $x \leq ab - c \implies c \leq ab \implies c \neq 0 \implies x + c \neq 0$
for $x\ ab\ c :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *plus-minus-no-overflow-ab*: $x \leq ab - c \implies c \leq ab \implies x \leq x + c$
for $x\ ab\ c :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *le-minus'*: $a + c \leq b \implies a \leq a + c \implies c \leq b - a$
for $a\ b\ c :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *le-plus'*: $a \leq b \implies c \leq b - a \implies a + c \leq b$
for $a\ b\ c :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *le-plus* = *le-plus'* [rotated]

lemmas *le-minus* = *leD* [THEN *thin-rl*, THEN *le-minus'*]

lemma *word-plus-mono-right*: $y \leq z \implies x \leq x + z \implies x + y \leq x + z$
for $x\ y\ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-less-minus-cancel*: $y - x < z - x \implies x \leq z \implies y < z$
for $x\ y\ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-less-minus-mono-left*: $y < z \implies x \leq y \implies y - x < z - x$
for $x\ y\ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-less-minus-mono*: $a < c \implies d < b \implies a - b < a \implies c - d < c \implies a - b < c - d$
for $a\ b\ c\ d :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-le-minus-cancel*: $y - x \leq z - x \implies x \leq z \implies y \leq z$

for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-le-minus-mono-left*: $y \leq z \implies x \leq y \implies y - x \leq z - x$
for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-le-minus-mono*:
 $a \leq c \implies d \leq b \implies a - b \leq c - d \implies a - b \leq c - d$
for $a\ b\ c\ d :: 'a::len\ word$
 $\langle proof \rangle$

lemma *plus-le-left-cancel-wrap*: $x + y' < x \implies x + y < x \implies x + y' < x + y$
 $\longleftrightarrow y' < y$
for $x\ y\ y' :: 'a::len\ word$
 $\langle proof \rangle$

lemma *plus-le-left-cancel-nowrap*: $x \leq x + y' \implies x \leq x + y \implies x + y' < x + y$
 $\longleftrightarrow y' < y$
for $x\ y\ y' :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-plus-mono-right2*: $a \leq a + b \implies c \leq b \implies a \leq a + c$
for $a\ b\ c :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-less-add-right*: $x < y - z \implies z \leq y \implies x + z < y$
for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-less-sub-right*: $x < y + z \implies y \leq x \implies x - y < z$
for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-le-plus-either*: $x \leq y \vee x \leq z \implies y \leq y + z \implies x \leq y + z$
for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-less-nowrapI*: $x < z - k \implies k \leq z \implies 0 < k \implies x < x + k$
for $x\ z\ k :: 'a::len\ word$
 $\langle proof \rangle$

lemma *inc-le*: $i < m \implies i + 1 \leq m$
for $i\ m :: 'a::len\ word$
 $\langle proof \rangle$

lemma *inc-i*: $1 \leq i \implies i < m \implies 1 \leq i + 1 \wedge i + 1 \leq m$
for $i\ m :: 'a::len\ word$
 $\langle proof \rangle$

lemma *udvd-incr-lem*:

$up < uq \implies up = ua + n * uint\ K \implies$
 $uq = ua + n' * uint\ K \implies up + uint\ K \leq uq$
 $\langle proof \rangle$

lemma *udvd-incr'*:

$p < q \implies uint\ p = ua + n * uint\ K \implies$
 $uint\ q = ua + n' * uint\ K \implies p + K \leq q$
 $\langle proof \rangle$

lemma *udvd-decr'*:

assumes $p < q$ $uint\ p = ua + n * uint\ K$ $uint\ q = ua + n' * uint\ K$
shows $uint\ q = ua + n' * uint\ K \implies p \leq q - K$
 $\langle proof \rangle$

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'*: $xy < k \implies z\ udvd\ xy \implies z\ udvd\ k \implies xy \leq k - z$
 $\langle proof \rangle$

lemma *udvd-incr2-K*:

$p < a + s \implies a \leq a + s \implies K\ udvd\ s \implies K\ udvd\ p - a \implies a \leq p \implies$
 $0 < K \implies p \leq p + K \wedge p + K \leq a + s$
 $\langle proof \rangle$

98.20 Arithmetic type class instantiations

lemmas *word-le-0-iff* [*simp*] =

word-zero-le [*THEN leD*, *THEN antisym-conv1*]

lemma *word-of-int-nat*: $0 \leq x \implies word\ of\ int\ x = of\ nat\ (nat\ x)$

$\langle proof \rangle$

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*]:

$iszero\ (numeral\ bin :: 'a::len\ word) =$
 $iszero\ (take\ bit\ LENGTH('a)\ (numeral\ bin :: int))$
 $\langle proof \rangle$

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [*simp*] =

eq-numeral-iff-iszero [**where** $'a='a::len\ word$]

98.21 Word and nat

lemma *word-nchotomy*: $\forall w :: 'a::len\ word. \exists n. w = of\ nat\ n \wedge n < 2 \wedge LENGTH('a)$

<proof>

lemma *of-nat-eq*: $of\text{-}nat\ n = w \longleftrightarrow (\exists q. n = unat\ w + q * 2 \wedge LENGTH('a))$
for $w :: 'a::len\ word$
<proof>

lemma *of-nat-eq-size*: $of\text{-}nat\ n = w \longleftrightarrow (\exists q. n = unat\ w + q * 2 \wedge size\ w)$
<proof>

lemma *of-nat-0*: $of\text{-}nat\ m = (0 :: 'a::len\ word) \longleftrightarrow (\exists q. m = q * 2 \wedge LENGTH('a))$
<proof>

lemma *of-nat-2p [simp]*: $of\text{-}nat\ (2 \wedge LENGTH('a)) = (0 :: 'a::len\ word)$
<proof>

lemma *of-nat-gt-0*: $of\text{-}nat\ k \neq 0 \implies 0 < k$
<proof>

lemma *of-nat-neq-0*: $0 < k \implies k < 2 \wedge LENGTH('a::len) \implies of\text{-}nat\ k \neq (0 :: 'a\ word)$
<proof>

lemma *Abs-fnat-hom-add*: $of\text{-}nat\ a + of\text{-}nat\ b = of\text{-}nat\ (a + b)$
<proof>

lemma *Abs-fnat-hom-mult*: $of\text{-}nat\ a * of\text{-}nat\ b = (of\text{-}nat\ (a * b) :: 'a::len\ word)$
<proof>

lemma *Abs-fnat-hom-Suc*: $word\text{-}succ\ (of\text{-}nat\ a) = of\text{-}nat\ (Suc\ a)$
<proof>

lemma *Abs-fnat-hom-0*: $(0 :: 'a::len\ word) = of\text{-}nat\ 0$
<proof>

lemma *Abs-fnat-hom-1*: $(1 :: 'a::len\ word) = of\text{-}nat\ (Suc\ 0)$
<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*: $a + b = of\text{-}nat\ (unat\ a + unat\ b)$
<proof>

lemma *word-arith-nat-mult*: $a * b = of\text{-}nat\ (unat\ a * unat\ b)$
<proof>

lemma *word-arith-nat-Suc*: $word\text{-}succ\ a = of\text{-}nat\ (Suc\ (unat\ a))$
<proof>

lemma *word-arith-nat-div*: $a \text{ div } b = \text{of-nat } (\text{unat } a \text{ div unat } b)$
 $\langle \text{proof} \rangle$

lemma *word-arith-nat-mod*: $a \text{ mod } b = \text{of-nat } (\text{unat } a \text{ mod unat } b)$
 $\langle \text{proof} \rangle$

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-cong*: $x = y \implies \text{unat } x = \text{unat } y$
 $\langle \text{proof} \rangle$

lemma *unat-of-nat*:
 $\langle \text{unat } (\text{word-of-nat } x :: 'a::\text{len word}) = x \text{ mod } 2^{\wedge} \text{LENGTH}('a) \rangle$
 $\langle \text{proof} \rangle$

lemmas *unat-word-ariths* = *word-arith-nat-defs*
 $[\text{THEN trans } [\text{OF unat-cong unat-of-nat}]]$

lemmas *word-sub-less-iff* = *word-sub-le-iff*
 $[\text{unfolded linorder-not-less } [\text{symmetric}]] \text{ Not-eq-iff}]$

lemma *unat-add-lem*:
 $\text{unat } x + \text{unat } y < 2^{\wedge} \text{LENGTH}('a) \longleftrightarrow \text{unat } (x + y) = \text{unat } x + \text{unat } y$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *unat-mult-lem*:
 $\text{unat } x * \text{unat } y < 2^{\wedge} \text{LENGTH}('a) \longleftrightarrow \text{unat } (x * y) = \text{unat } x * \text{unat } y$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *le-no-overflow*: $x \leq b \implies a \leq a + b \implies x \leq a + b$
for $a \ b \ x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *uint-div*:
 $\langle \text{uint } (x \text{ div } y) = \text{uint } x \text{ div uint } y \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-mod*:
 $\langle \text{uint } (x \text{ mod } y) = \text{uint } x \text{ mod uint } y \rangle$
 $\langle \text{proof} \rangle$

lemma *no-plus-overflow-unat-size*: $x \leq x + y \longleftrightarrow \text{unat } x + \text{unat } y < 2^{\wedge} \text{size } x$

```

for  $x\ y :: 'a::len\ word$ 
   $\langle proof \rangle$ 

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:  $\llbracket 0 < y; unat\ x * unat\ y < 2^{\wedge} LENGTH('a) \rrbracket \implies x * y$ 
   $div\ y = x$ 
  for  $x\ y :: 'a::len\ word$ 
   $\langle proof \rangle$ 

lemma div-lt':  $i \leq k\ div\ x \implies unat\ i * unat\ x < 2^{\wedge} LENGTH('a)$ 
  for  $i\ k\ x :: 'a::len\ word$ 
   $\langle proof \rangle$ 

lemmas div-lt'' = order-less-imp-le [THEN div-lt']

lemma div-lt-mult:  $\llbracket i < k\ div\ x; 0 < x \rrbracket \implies i * x < k$ 
  for  $i\ k\ x :: 'a::len\ word$ 
   $\langle proof \rangle$ 

lemma div-le-mult:  $\llbracket i \leq k\ div\ x; 0 < x \rrbracket \implies i * x \leq k$ 
  for  $i\ k\ x :: 'a::len\ word$ 
   $\langle proof \rangle$ 

lemma div-lt-uint':  $i \leq k\ div\ x \implies uint\ i * uint\ x < 2^{\wedge} LENGTH('a)$ 
  for  $i\ k\ x :: 'a::len\ word$ 
   $\langle proof \rangle$ 

lemmas div-lt-uint'' = order-less-imp-le [THEN div-lt-uint']

lemma word-le-exists':  $x \leq y \implies \exists z. y = x + z \wedge uint\ x + uint\ z < 2^{\wedge} LENGTH('a)$ 
  for  $x\ y\ z :: 'a::len\ word$ 
   $\langle proof \rangle$ 

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-voi*:
 $\langle y \leq \text{unat } z \implies \text{unat } (\text{word-of-nat } y :: 'a \text{ word}) = y \rangle$
for $z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *thd* = *times-div-less-eq-dividend*

lemmas *uno-simps* [THEN *le-unat-voi*] = *mod-le-divisor div-le-dividend*

lemma *word-mod-div-equality*: $(n \text{ div } b) * b + (n \text{ mod } b) = n$
for $n\ b :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-div-mult-le*: $a \text{ div } b * b \leq a$
for $a\ b :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-mod-less-divisor*: $0 < n \implies m \text{ mod } n < n$
for $m\ n :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-of-int-power-hom*: $\text{word-of-int } a \wedge n = (\text{word-of-int } (a \wedge n) :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemma *word-arith-power-alt*: $a \wedge n = (\text{word-of-int } (\text{uint } a \wedge n) :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemma *unatSuc*: $1 + n \neq 0 \implies \text{unat } (1 + n) = \text{Suc } (\text{unat } n)$
for $n :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

98.22 Cardinality, finiteness of set of words

lemma *inj-on-word-of-int*: $\langle \text{inj-on } (\text{word-of-int} :: \text{int} \Rightarrow 'a \text{ word}) \{0..<2 \wedge \text{LENGTH}('a::\text{len})\} \rangle$
 $\langle \text{proof} \rangle$

lemma *range-uint*: $\langle \text{range } (\text{uint} :: 'a \text{ word} \Rightarrow \text{int}) = \{0..<2 \wedge \text{LENGTH}('a::\text{len})\} \rangle$
 $\langle \text{proof} \rangle$

lemma *UNIV-eq*: $\langle (\text{UNIV} :: 'a \text{ word set}) = \text{word-of-int } ' \{0..<2 \wedge \text{LENGTH}('a::\text{len})\} \rangle$
 $\langle \text{proof} \rangle$

lemma *card-word*: $\text{CARD}('a \text{ word}) = 2 \wedge \text{LENGTH}('a::\text{len})$
 $\langle \text{proof} \rangle$

lemma *card-word-size*: $CARD('a \text{ word}) = 2^{\wedge \text{size } x}$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$
end

instance *word* :: $(\text{len}) \text{ finite}$
 $\langle \text{proof} \rangle$

98.23 Bitwise Operations on Words

context
includes *bit-operations-syntax*
begin

lemma *word-wi-log-defs*:
 $NOT (\text{word-of-int } a) = \text{word-of-int } (NOT \ a)$
 $\text{word-of-int } a \text{ AND } \text{word-of-int } b = \text{word-of-int } (a \text{ AND } b)$
 $\text{word-of-int } a \text{ OR } \text{word-of-int } b = \text{word-of-int } (a \text{ OR } b)$
 $\text{word-of-int } a \text{ XOR } \text{word-of-int } b = \text{word-of-int } (a \text{ XOR } b)$
 $\langle \text{proof} \rangle$

lemma *word-no-log-defs* [simp]:
 $NOT (\text{numeral } a) = \text{word-of-int } (NOT (\text{numeral } a))$
 $NOT (\neg \text{numeral } a) = \text{word-of-int } (NOT (\neg \text{numeral } a))$
 $\text{numeral } a \text{ AND } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ AND } \text{numeral } b)$
 $\text{numeral } a \text{ AND } \neg \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ AND } \neg \text{numeral } b)$
 $\neg \text{numeral } a \text{ AND } \text{numeral } b = \text{word-of-int } (\neg \text{numeral } a \text{ AND } \text{numeral } b)$
 $\neg \text{numeral } a \text{ AND } \neg \text{numeral } b = \text{word-of-int } (\neg \text{numeral } a \text{ AND } \neg \text{numeral } b)$
 $\text{numeral } a \text{ OR } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ OR } \text{numeral } b)$
 $\text{numeral } a \text{ OR } \neg \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ OR } \neg \text{numeral } b)$
 $\neg \text{numeral } a \text{ OR } \text{numeral } b = \text{word-of-int } (\neg \text{numeral } a \text{ OR } \text{numeral } b)$
 $\neg \text{numeral } a \text{ OR } \neg \text{numeral } b = \text{word-of-int } (\neg \text{numeral } a \text{ OR } \neg \text{numeral } b)$
 $\text{numeral } a \text{ XOR } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ XOR } \text{numeral } b)$
 $\text{numeral } a \text{ XOR } \neg \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ XOR } \neg \text{numeral } b)$
 $\neg \text{numeral } a \text{ XOR } \text{numeral } b = \text{word-of-int } (\neg \text{numeral } a \text{ XOR } \text{numeral } b)$
 $\neg \text{numeral } a \text{ XOR } \neg \text{numeral } b = \text{word-of-int } (\neg \text{numeral } a \text{ XOR } \neg \text{numeral } b)$
 $\langle \text{proof} \rangle$

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [simp]:
 $NOT (1::'a::\text{len word}) = \neg 1$
 $1 \text{ AND } \text{numeral } b = \text{word-of-int } (1 \text{ AND } \text{numeral } b)$
 $1 \text{ AND } \neg \text{numeral } b = \text{word-of-int } (1 \text{ AND } \neg \text{numeral } b)$
 $\text{numeral } a \text{ AND } 1 = \text{word-of-int } (\text{numeral } a \text{ AND } 1)$
 $\neg \text{numeral } a \text{ AND } 1 = \text{word-of-int } (\neg \text{numeral } a \text{ AND } 1)$
 $1 \text{ OR } \text{numeral } b = \text{word-of-int } (1 \text{ OR } \text{numeral } b)$
 $1 \text{ OR } \neg \text{numeral } b = \text{word-of-int } (1 \text{ OR } \neg \text{numeral } b)$
 $\text{numeral } a \text{ OR } 1 = \text{word-of-int } (\text{numeral } a \text{ OR } 1)$
 $\neg \text{numeral } a \text{ OR } 1 = \text{word-of-int } (\neg \text{numeral } a \text{ OR } 1)$

$1 \text{ XOR numeral } b = \text{word-of-int } (1 \text{ XOR numeral } b)$
 $1 \text{ XOR } - \text{ numeral } b = \text{word-of-int } (1 \text{ XOR } - \text{ numeral } b)$
 $\text{numeral } a \text{ XOR } 1 = \text{word-of-int } (\text{numeral } a \text{ XOR } 1)$
 $- \text{ numeral } a \text{ XOR } 1 = \text{word-of-int } (- \text{ numeral } a \text{ XOR } 1)$
 $\langle \text{proof} \rangle$

Special cases for when one of the arguments equals -1.

lemma *word-bitwise-m1-simps* [simp]:

$\text{NOT } (-1 :: 'a :: \text{len word}) = 0$
 $(-1 :: 'a :: \text{len word}) \text{ AND } x = x$
 $x \text{ AND } (-1 :: 'a :: \text{len word}) = x$
 $(-1 :: 'a :: \text{len word}) \text{ OR } x = -1$
 $x \text{ OR } (-1 :: 'a :: \text{len word}) = -1$
 $(-1 :: 'a :: \text{len word}) \text{ XOR } x = \text{NOT } x$
 $x \text{ XOR } (-1 :: 'a :: \text{len word}) = \text{NOT } x$
 $\langle \text{proof} \rangle$

lemma *word-of-int-not-numeral-eq* [simp]:

$\langle (\text{word-of-int } (\text{NOT } (\text{numeral bin})) :: 'a :: \text{len word}) = - \text{ numeral bin } - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-and*:

$\langle \text{uint } (x \text{ AND } y) = \text{uint } x \text{ AND } \text{uint } y \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-or*:

$\langle \text{uint } (x \text{ OR } y) = \text{uint } x \text{ OR } \text{uint } y \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-xor*:

$\langle \text{uint } (x \text{ XOR } y) = \text{uint } x \text{ XOR } \text{uint } y \rangle$
 $\langle \text{proof} \rangle$

lemmas *bwsimps* =

wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:

$(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
for $x :: 'a :: \text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-bw-comms*:

$x \text{ AND } y = y \text{ AND } x$
 $x \text{ OR } y = y \text{ OR } x$
 $x \text{ XOR } y = y \text{ XOR } x$
for $x :: 'a :: \text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-bw-lcs*:

$y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
for $x :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *word-log-esimps*:

$x \text{ AND } 0 = 0$
 $x \text{ AND } -1 = x$
 $x \text{ OR } 0 = x$
 $x \text{ OR } -1 = -1$
 $x \text{ XOR } 0 = x$
 $x \text{ XOR } -1 = \text{NOT } x$
 $0 \text{ AND } x = 0$
 $-1 \text{ AND } x = x$
 $0 \text{ OR } x = x$
 $-1 \text{ OR } x = -1$
 $0 \text{ XOR } x = x$
 $-1 \text{ XOR } x = \text{NOT } x$
for $x :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *word-not-dist*:

$\text{NOT } (x \text{ OR } y) = \text{NOT } x \text{ AND } \text{NOT } y$
 $\text{NOT } (x \text{ AND } y) = \text{NOT } x \text{ OR } \text{NOT } y$
for $x :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *word-bw-same*:

$x \text{ AND } x = x$
 $x \text{ OR } x = x$
 $x \text{ XOR } x = 0$
for $x :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *word-ao-absorbs* [*simp*]:

$x \text{ AND } (y \text{ OR } x) = x$
 $x \text{ OR } y \text{ AND } x = x$
 $x \text{ AND } (x \text{ OR } y) = x$
 $y \text{ AND } x \text{ OR } x = x$
 $(y \text{ OR } x) \text{ AND } x = x$
 $x \text{ OR } x \text{ AND } y = x$
 $(x \text{ OR } y) \text{ AND } x = x$
 $x \text{ AND } y \text{ OR } x = x$
for $x :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *word-not-not* [*simp*]: $NOT (NOT x) = x$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-ao-dist*: $(x OR y) AND z = x AND z OR y AND z$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-oa-dist*: $x AND y OR z = (x OR z) AND (y OR z)$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-add-not* [*simp*]: $x + NOT x = -1$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-plus-and-or* [*simp*]: $(x AND y) + (x OR y) = x + y$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *leoa*: $w = x OR y \implies y = w AND y$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *leao*: $w' = x' AND y' \implies x' = x' OR w'$
for $x' :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-ao-equiv*: $w = w OR w' \longleftrightarrow w' = w AND w'$
for $w w' :: 'a::len\ word$
 $\langle proof \rangle$

lemma *le-word-or2*: $x \leq x OR y$
for $x y :: 'a::len\ word$
 $\langle proof \rangle$

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*]:
 $\langle bit\ (horner-sum\ of-bool\ (2 :: 'a::len\ word)\ bs)\ n$
 $\longleftrightarrow n < \min\ LENGTH('a)\ (length\ bs) \wedge bs\ !\ n \rangle$
 $\langle proof \rangle$

definition *word-reverse* :: $\langle 'a::len\ word \Rightarrow 'a\ word \rangle$
where $\langle word-reverse\ w = horner-sum\ of-bool\ 2\ (rev\ (map\ (bit\ w)\ [0..<LENGTH('a)])) \rangle$

lemma *bit-word-reverse-iff* [*bit-simps*]:

$\langle \text{bit } (\text{word-reverse } w) \ n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } w \ (\text{LENGTH}('a) - \text{Suc } n) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-rev-rev* [simp] : $\text{word-reverse } (\text{word-reverse } w) = w$
 $\langle \text{proof} \rangle$

lemma *word-rev-gal*: $\text{word-reverse } w = u \implies \text{word-reverse } u = w$
 $\langle \text{proof} \rangle$

lemma *word-rev-gal'*: $u = \text{word-reverse } w \implies w = \text{word-reverse } u$
 $\langle \text{proof} \rangle$

lemma *uint-2p*: $(0 :: 'a::\text{len word}) < 2 \wedge n \implies \text{uint } (2 \wedge n :: 'a::\text{len word}) = 2 \wedge n$
 $\langle \text{proof} \rangle$

lemma *word-of-int-2p*: $(\text{word-of-int } (2 \wedge n) :: 'a::\text{len word}) = 2 \wedge n$
 $\langle \text{proof} \rangle$

98.23.1 shift functions in terms of lists of bools

lemma *drop-bit-word-numeral* [simp]:
 $\langle \text{drop-bit } (\text{numeral } n) \ (\text{numeral } k) =$
 $(\text{word-of-int } (\text{drop-bit } (\text{numeral } n) \ (\text{take-bit } \text{LENGTH}('a) \ (\text{numeral } k))) :: 'a::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-word-Suc-numeral* [simp]:
 $\langle \text{drop-bit } (\text{Suc } n) \ (\text{numeral } k) =$
 $(\text{word-of-int } (\text{drop-bit } (\text{Suc } n) \ (\text{take-bit } \text{LENGTH}('a) \ (\text{numeral } k))) :: 'a::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-word-minus-numeral* [simp]:
 $\langle \text{drop-bit } (\text{numeral } n) \ (- \text{numeral } k) =$
 $(\text{word-of-int } (\text{drop-bit } (\text{numeral } n) \ (\text{take-bit } \text{LENGTH}('a) \ (- \text{numeral } k))) :: 'a::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-word-Suc-minus-numeral* [simp]:
 $\langle \text{drop-bit } (\text{Suc } n) \ (- \text{numeral } k) =$
 $(\text{word-of-int } (\text{drop-bit } (\text{Suc } n) \ (\text{take-bit } \text{LENGTH}('a) \ (- \text{numeral } k))) :: 'a::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-word-numeral* [simp]:
 $\langle \text{signed-drop-bit } (\text{numeral } n) \ (\text{numeral } k) =$
 $(\text{word-of-int } (\text{drop-bit } (\text{numeral } n) \ (\text{signed-take-bit } (\text{LENGTH}('a) - 1) \ (\text{numeral } k))) :: 'a::\text{len word}) \rangle$

⟨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*: $\llbracket \text{set } xs \subseteq \{\text{False}\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\vee) \text{ } xs$
 $ys = ys$

⟨proof⟩

lemma *align-lem-or*:

assumes $\text{length } xs = n + m$ $\text{length } ys = n + m$

and $\text{drop } m \text{ } xs = \text{replicate } n \text{ False take } m \text{ } ys = \text{replicate } m \text{ False}$

shows $\text{map2 } (\vee) \text{ } xs \text{ } ys = \text{take } m \text{ } xs @ \text{drop } m \text{ } ys$

⟨proof⟩

lemma *False-map2-and*: $\llbracket \text{set } xs \subseteq \{\text{False}\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\wedge)$
 $xs \text{ } ys = xs$

⟨proof⟩

lemma *align-lem-and*:

assumes $\text{length } xs = n + m$ $\text{length } ys = n + m$

and $\text{drop } m \text{ } xs = \text{replicate } n \text{ False take } m \text{ } ys = \text{replicate } m \text{ False}$

shows $\text{map2 } (\wedge) \text{ } xs \text{ } ys = \text{replicate } (n + m) \text{ False}$

⟨proof⟩

98.23.2 Mask

lemma *minus-1-eq-mask*:

⟨− 1 = (mask LENGTH('a) :: 'a::len word)⟩

⟨proof⟩

lemma *mask-eq-decr-exp*:

⟨mask n = 2ⁿ − (1 :: 'a::len word)⟩

⟨proof⟩

lemma *mask-Suc-rec*:

$\langle \text{mask } (\text{Suc } n) = 2 * \text{mask } n + (1 :: 'a::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

context
begin

qualified lemma *bit-mask-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{mask } m :: 'a::\text{len word}) \ n \longleftrightarrow n < \min \text{LENGTH}('a) \ m \rangle$
 $\langle \text{proof} \rangle$

end

lemma *mask-bin*: $\text{mask } n = \text{word-of-int } (\text{take-bit } n \ (- \ 1))$
 $\langle \text{proof} \rangle$

lemma *and-mask-bintr*: $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \ (\text{uint } w))$
 $\langle \text{proof} \rangle$

lemma *and-mask-wi*: $\text{word-of-int } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \ i)$
 $\langle \text{proof} \rangle$

lemma *and-mask-wi'*:
 $\text{word-of-int } i \text{ AND } \text{mask } n = (\text{word-of-int } (\text{take-bit } (\min \text{LENGTH}('a) \ n) \ i) :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemma *and-mask-no*: $\text{numeral } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \ (\text{numeral } i))$
 $\langle \text{proof} \rangle$

lemma *and-mask-mod-2p*: $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{uint } w \text{ mod } 2^n)$
 $\langle \text{proof} \rangle$

lemma *uint-mask-eq*:
 $\langle \text{uint } (\text{mask } n :: 'a::\text{len word}) = \text{mask } (\min \text{LENGTH}('a) \ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *and-mask-lt-2p*: $\text{uint } (w \text{ AND } \text{mask } n) < 2^n$
 $\langle \text{proof} \rangle$

lemma *mask-eq-iff*: $w \text{ AND } \text{mask } n = w \longleftrightarrow \text{uint } w < 2^n$
 $\langle \text{proof} \rangle$

lemma *and-mask-dvd*: $2^n \text{ dvd } \text{uint } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$
 $\langle \text{proof} \rangle$

lemma *and-mask-dvd-nat*: $2^n \text{ dvd } \text{unat } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$
 $\langle \text{proof} \rangle$

lemma *word-2p-lem*: $n < \text{size } w \implies w < 2 \wedge n = (\text{uint } w < 2 \wedge n)$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *less-mask-eq*:
fixes $x :: 'a::\text{len word}$
assumes $x < 2 \wedge n$ **shows** $x \text{ AND mask } n = x$
 $\langle \text{proof} \rangle$

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*: $n < \text{size } x \implies x \text{ AND mask } n < 2 \wedge n$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-mod-2p-is-mask* [*OF* *refl*]: $c = 2 \wedge n \implies c > 0 \implies x \text{ mod } c = x$
AND *mask* n
for $c \ x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *mask-egs*:
 $(a \text{ AND mask } n) + b \text{ AND mask } n = a + b \text{ AND mask } n$
 $a + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$
 $(a \text{ AND mask } n) - b \text{ AND mask } n = a - b \text{ AND mask } n$
 $a - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$
 $a * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$
 $(b \text{ AND mask } n) * a \text{ AND mask } n = b * a \text{ AND mask } n$
 $(a \text{ AND mask } n) + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$
 $(a \text{ AND mask } n) - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$
 $(a \text{ AND mask } n) * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$
 $-(a \text{ AND mask } n) \text{ AND mask } n = -a \text{ AND mask } n$
 $\text{word-succ } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-succ } a \text{ AND mask } n$
 $\text{word-pred } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-pred } a \text{ AND mask } n$
 $\langle \text{proof} \rangle$

lemma *mask-power-eq*: $(x \text{ AND mask } n) \wedge^k \text{ AND mask } n = x \wedge^k \text{ AND mask } n$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *mask-full* [*simp*]: $\text{mask LENGTH}('a) = (-1 :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

98.23.3 Slices

definition *slice1* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
where $\langle \text{slice1 } n \ w = (\text{if } n < \text{LENGTH}('a)$
 $\text{then } \text{ucast } (\text{drop-bit } (\text{LENGTH}('a) - n) \ w)$

else push-bit ($n - \text{LENGTH}('a)$) (*ucast* w) $\rangle$

lemma *bit-slice1-iff* [*bit-simps*]:

$\langle \text{bit } (\text{slice1 } m \ w :: 'b::\text{len word}) \ n \longleftrightarrow m - \text{LENGTH}('a) \leq n \wedge n < \min$
 $\text{LENGTH}('b) \ m$
 $\wedge \text{bit } w \ (n + (\text{LENGTH}('a) - m) - (m - \text{LENGTH}('a))) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

definition *slice* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$

where $\langle \text{slice } n = \text{slice1 } (\text{LENGTH}('a) - n) \rangle$

lemma *bit-slice-iff* [*bit-simps*]:

$\langle \text{bit } (\text{slice } m \ w :: 'b::\text{len word}) \ n \longleftrightarrow n < \min \text{LENGTH}('b) \ (\text{LENGTH}('a) -$
 $m) \wedge \text{bit } w \ (n + \text{LENGTH}('a) - (\text{LENGTH}('a) - m)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *slice1-0* [*simp*] : $\text{slice1 } n \ 0 = 0$

$\langle \text{proof} \rangle$

lemma *slice-0* [*simp*] : $\text{slice } n \ 0 = 0$

$\langle \text{proof} \rangle$

lemma *ucast-slice1*: $\text{ucast } w = \text{slice1 } (\text{size } w) \ w$

$\langle \text{proof} \rangle$

lemma *ucast-slice*: $\text{ucast } w = \text{slice } 0 \ w$

$\langle \text{proof} \rangle$

lemma *slice-id*: $\text{slice } 0 \ t = t$

$\langle \text{proof} \rangle$

lemma *rev-slice1*:

$\langle \text{slice1 } n \ (\text{word-reverse } w :: 'b::\text{len word}) = \text{word-reverse } (\text{slice1 } k \ w :: 'a::\text{len}$
 $\text{word}) \rangle$
if $\langle n + k = \text{LENGTH}('a) + \text{LENGTH}('b) \rangle$
 $\langle \text{proof} \rangle$

lemma *rev-slice*:

$n + k + \text{LENGTH}('a::\text{len}) = \text{LENGTH}('b::\text{len}) \implies$
 $\text{slice } n \ (\text{word-reverse } (w::'b \ \text{word})) = \text{word-reverse } (\text{slice } k \ w :: 'a \ \text{word})$
 $\langle \text{proof} \rangle$

98.23.4 Recast

definition *recast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$

where $\langle \text{recast} = \text{slice1 } \text{LENGTH}('b) \rangle$

lemma *bit-revcast-iff* [*bit-simps*]:

$\langle \text{bit } (\text{revcast } w :: 'b::\text{len word}) \ n \longleftrightarrow \text{LENGTH}('b) - \text{LENGTH}('a) \leq n \wedge n < \text{LENGTH}('b) \wedge \text{bit } w \ (n + (\text{LENGTH}('a) - \text{LENGTH}('b)) - (\text{LENGTH}('b) - \text{LENGTH}('a))) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *revcast-slice1* [*OF refl*]: $\text{rc} = \text{revcast } w \implies \text{slice1 } (\text{size } \text{rc}) \ w = \text{rc}$
 $\langle \text{proof} \rangle$

lemma *revcast-rev-ucast* [*OF refl refl refl*]:

$\text{cs} = [\text{rc}, \text{uc}] \implies \text{rc} = \text{revcast } (\text{word-reverse } w) \implies \text{uc} = \text{ucast } w \implies \text{rc} = \text{word-reverse } \text{uc}$
 $\langle \text{proof} \rangle$

lemma *revcast-ucast*: $\text{revcast } w = \text{word-reverse } (\text{ucast } (\text{word-reverse } w))$
 $\langle \text{proof} \rangle$

lemma *ucast-revcast*: $\text{ucast } w = \text{word-reverse } (\text{revcast } (\text{word-reverse } w))$
 $\langle \text{proof} \rangle$

lemma *ucast-rev-revcast*: $\text{ucast } (\text{word-reverse } w) = \text{word-reverse } (\text{revcast } w)$
 $\langle \text{proof} \rangle$

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*]]

98.24 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*: $\text{word-cat } a \ b = b$
 $\langle \text{proof} \rangle$

lemma *word-cat-split-alt*: $\llbracket \text{size } w \leq \text{size } u + \text{size } v; \text{word-split } w = (u, v) \rrbracket \implies \text{word-cat } u \ v = w$
 $\langle \text{proof} \rangle$

lemmas *word-cat-split-size* = *sym* [*THEN* [2] *word-cat-split-alt* [*symmetric*]]

98.24.1 Split and slice

lemma *split-slices*:

assumes $\text{word-split } w = (u, v)$
shows $u = \text{slice } (\text{size } v) \ w \wedge v = \text{slice } 0 \ w$
 $\langle \text{proof} \rangle$

lemma *slice-cat1* [*OF refl*]:
 $\llbracket wc = \text{word-cat } a \ b; \text{size } a + \text{size } b \leq \text{size } wc \rrbracket \implies \text{slice } (\text{size } b) \ wc = a$
 $\langle \text{proof} \rangle$

lemmas *slice-cat2* = *trans* [*OF slice-id word-cat-id*]

lemma *cat-slices*:
 $\llbracket a = \text{slice } n \ c; \ b = \text{slice } 0 \ c; \ n = \text{size } b; \text{size } c \leq \text{size } a + \text{size } b \rrbracket \implies \text{word-cat } a \ b = c$
 $\langle \text{proof} \rangle$

lemma *word-split-cat-alt*:
assumes $w = \text{word-cat } u \ v$ **and** *size*: $\text{size } u + \text{size } v \leq \text{size } w$
shows $\text{word-split } w = (u, v)$
 $\langle \text{proof} \rangle$

lemma *horner-sum-uint-exp-Cons-eg*:
 $\langle \text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \ (w \# \text{ws}) =$
 $\text{concat-bit LENGTH}('a) \ (\text{uint } w) \ (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \ \text{ws}) \rangle$
for $\text{ws} :: \langle 'a :: \text{len word list} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-horner-sum-uint-exp-iff*:
 $\langle \text{bit } (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \ \text{ws}) \ n \longleftrightarrow$
 $n \text{ div LENGTH}('a) < \text{length } \text{ws} \wedge \text{bit } (\text{ws} ! (n \text{ div LENGTH}('a))) \ (n \text{ mod}$
 $\text{LENGTH}('a)) \rangle$
for $\text{ws} :: \langle 'a :: \text{len word list} \rangle$
 $\langle \text{proof} \rangle$

98.25 Rotation

lemma *word-rotr-word-rotr-eg*: $\langle \text{word-rotr } m \ (\text{word-rotr } n \ w) = \text{word-rotr } (m + n) \ w \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rot-lem*: $\llbracket l + k = d + k \text{ mod } l; \ n < l \rrbracket \implies ((d + n) \text{ mod } l) = n$ **for** $l :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *word-rot-rl* [*simp*]: $\langle \text{word-rotl } k \ (\text{word-rotr } k \ v) = v \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rot-lr* [*simp*]: $\langle \text{word-rotr } k \ (\text{word-rotl } k \ v) = v \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rot-gal*:

$\langle \text{word-rotl } n \ v = w \longleftrightarrow \text{word-rotl } n \ w = v \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rot-gal'*:

$\langle w = \text{word-rotl } n \ v \longleftrightarrow v = \text{word-rotl } n \ w \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rotl-rev*:

$\langle \text{word-rotl } n \ w = \text{word-reverse } (\text{word-rotl } n \ (\text{word-reverse } w)) \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rotl-0 [simp]*: $\text{word-rotl } 0 \ w = w$

$\langle \text{proof} \rangle$

lemma *word-rotl-add*: $\text{word-rotl } (m + n) \ w = \text{word-rotl } m \ (\text{word-rotl } n \ w)$

$\langle \text{proof} \rangle$

lemma *word-rotl-conv-mod'*:

$\text{word-rotl } n \ w = \text{word-rotl } (n \bmod \text{int } (\text{size } w)) \ w$
 $\langle \text{proof} \rangle$

lemmas *word-rotl-conv-mod = word-rotl-conv-mod' [unfolded word-size]*

end

98.25.1 "Word rotation commutes with bit-wise operations

locale *word-rotate*

begin

context

includes *bit-operations-syntax*

begin

lemma *word-rot-logs*:

$\text{word-rotl } n \ (\text{NOT } v) = \text{NOT } (\text{word-rotl } n \ v)$
 $\text{word-rotl } n \ (\text{NOT } v) = \text{NOT } (\text{word-rotl } n \ v)$
 $\text{word-rotl } n \ (x \ \text{AND } y) = \text{word-rotl } n \ x \ \text{AND } \text{word-rotl } n \ y$
 $\text{word-rotl } n \ (x \ \text{AND } y) = \text{word-rotl } n \ x \ \text{AND } \text{word-rotl } n \ y$
 $\text{word-rotl } n \ (x \ \text{OR } y) = \text{word-rotl } n \ x \ \text{OR } \text{word-rotl } n \ y$
 $\text{word-rotl } n \ (x \ \text{OR } y) = \text{word-rotl } n \ x \ \text{OR } \text{word-rotl } n \ y$
 $\text{word-rotl } n \ (x \ \text{XOR } y) = \text{word-rotl } n \ x \ \text{XOR } \text{word-rotl } n \ y$
 $\text{word-rotl } n \ (x \ \text{XOR } y) = \text{word-rotl } n \ x \ \text{XOR } \text{word-rotl } n \ y$
 $\langle \text{proof} \rangle$

end

end

lemmas *word-rot-logs* = *word-rotate.word-rot-logs*

lemma *word-rotx-0* [*simp*] : *word-rotr i 0 = 0* $\wedge$ *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*]

98.26 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** $0 \leq n$ **and** $n < 2^{\text{LENGTH}('a)}$
 ⟨*proof*⟩

lemma *word-nat-cases* [*cases type: word*]:
fixes *x* :: '*a*::len word
obtains *n* **where** *x* = *of-nat n* **and** $n < 2^{\text{LENGTH}('a)}$
 ⟨*proof*⟩

lemma *max-word-max* [*intro!*]:
 $\langle n \leq -1 \rangle$ **for** *n* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-of-int-2p-len*: *word-of-int* ($2^{\text{LENGTH}('a)}$) = (*0*::'*a*::len word)
 ⟨*proof*⟩

lemma *word-pow-0*: ($2^{\text{LENGTH}('a)}$) $\wedge$ *LENGTH*('a) = 0
 ⟨*proof*⟩

lemma *max-word-wrap*:
 $\langle x + 1 = 0 \implies x = -1 \rangle$ **for** *x* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-and-max*:
 $\langle x \text{ AND } -1 = x \rangle$ **for** *x* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-or-max*:
 $\langle x \text{ OR } -1 = -1 \rangle$ **for** *x* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-ao-dist2*: $x \text{ AND } (y \text{ OR } z) = x \text{ AND } y \text{ OR } x \text{ AND } z$
for $x \ y \ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-oa-dist2*: $x \text{ OR } y \text{ AND } z = (x \text{ OR } y) \text{ AND } (x \text{ OR } z)$
for $x \ y \ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-and-not* [simp]: $x \text{ AND NOT } x = 0$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-or-not* [simp]:
 $\langle x \text{ OR NOT } x = -1 \rangle$ **for** $x :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-xor-and-or*: $x \text{ XOR } y = x \text{ AND NOT } y \text{ OR NOT } x \text{ AND } y$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *uint-lt-0* [simp]: $\text{uint } x < 0 = \text{False}$
 $\langle \text{proof} \rangle$

lemma *word-less-1* [simp]: $x < 1 \longleftrightarrow x = 0$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *uint-plus-if-size*:
 $\text{uint } (x + y) =$
 $(\text{if } \text{uint } x + \text{uint } y < 2^{\text{size } x}$
 $\text{then } \text{uint } x + \text{uint } y$
 $\text{else } \text{uint } x + \text{uint } y - 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemma *unat-plus-if-size*:
 $\text{unat } (x + y) =$
 $(\text{if } \text{unat } x + \text{unat } y < 2^{\text{size } x}$
 $\text{then } \text{unat } x + \text{unat } y$
 $\text{else } \text{unat } x + \text{unat } y - 2^{\text{size } x})$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-neq-0-conv*: $w \neq 0 \longleftrightarrow 0 < w$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *max-lt*: $\text{unat } (\max a \ b \ \text{div } c) = \text{unat } (\max a \ b) \ \text{div } \text{unat } c$
for $c :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-sub-if-size*:

$\text{uint } (x - y) =$
 $(\text{if } \text{uint } y \leq \text{uint } x$
 $\text{then } \text{uint } x - \text{uint } y$
 $\text{else } \text{uint } x - \text{uint } y + 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemma *unat-sub*:

$\langle \text{unat } (a - b) = \text{unat } a - \text{unat } b \rangle$
if $\langle b \leq a \rangle$
 $\langle \text{proof} \rangle$

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*: $\text{word-of-int } (2^{\text{LENGTH('a)}} - i) = (\text{word-of-int } (-i))::'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *word-of-int-inj*:

$\langle (\text{word-of-int } x :: 'a::\text{len word}) = \text{word-of-int } y \longleftrightarrow x = y \rangle$
if $\langle 0 \leq x \wedge x < 2^{\text{LENGTH('a)}} \rangle \langle 0 \leq y \wedge y < 2^{\text{LENGTH('a)}} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-le-less-eq*: $x \leq y \longleftrightarrow x = y \vee x < y$

for $x \ y :: 'z::\text{len word}$

$\langle \text{proof} \rangle$

lemma *mod-plus-cong*:

fixes $b \ b' :: \text{int}$
assumes $1: b = b'$
 and $2: x \text{ mod } b' = x' \text{ mod } b'$
 and $3: y \text{ mod } b' = y' \text{ mod } b'$
 and $4: x' + y' = z'$
shows $(x + y) \text{ mod } b = z' \text{ mod } b'$
 $\langle \text{proof} \rangle$

lemma *mod-minus-cong*:

fixes $b \ b' :: \text{int}$
assumes $b = b'$
 and $x \text{ mod } b' = x' \text{ mod } b'$
 and $y \text{ mod } b' = y' \text{ mod } b'$
 and $x' - y' = z'$
shows $(x - y) \text{ mod } b = z' \text{ mod } b'$
 $\langle \text{proof} \rangle$

lemma *word-induct-less* [case-names zero less]:

$\langle P\ m \rangle$ **if** zero: $\langle P\ 0 \rangle$ **and** less: $\langle \bigwedge n. n < m \implies P\ n \implies P\ (1 + n) \rangle$
for $m :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-induct*: $P\ 0 \implies (\bigwedge n. P\ n \implies P\ (1 + n)) \implies P\ m$
for $P :: 'a::len\ word \Rightarrow bool$
 $\langle proof \rangle$

lemma *word-induct2* [case-names zero suc, induct type]: $P\ 0 \implies (\bigwedge n. 1 + n \neq 0 \implies P\ n \implies P\ (1 + n)) \implies P\ n$
for $P :: 'b::len\ word \Rightarrow bool$
 $\langle proof \rangle$

98.27 Recursion combinator for words

definition *word-rec* :: $'a \Rightarrow ('b::len\ word \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b\ word \Rightarrow 'a$
where *word-rec* forZero forSuc $n = rec\ nat\ forZero\ (forSuc \circ of\ nat)\ (unat\ n)$

lemma *word-rec-0* [simp]: $word\ rec\ z\ s\ 0 = z$
 $\langle proof \rangle$

lemma *word-rec-Suc* [simp]: $1 + n \neq 0 \implies word\ rec\ z\ s\ (1 + n) = s\ n\ (word\ rec\ z\ s\ n)$
for $n :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-rec-Pred*: $n \neq 0 \implies word\ rec\ z\ s\ n = s\ (n - 1)\ (word\ rec\ z\ s\ (n - 1))$
 $\langle proof \rangle$

lemma *word-rec-in*: $f\ (word\ rec\ z\ (\lambda\ -. f)\ n) = word\ rec\ (f\ z)\ (\lambda\ -. f)\ n$
 $\langle proof \rangle$

lemma *word-rec-in2*: $f\ n\ (word\ rec\ z\ f\ n) = word\ rec\ (f\ 0\ z)\ (f \circ (+)\ 1)\ n$
 $\langle proof \rangle$

lemma *word-rec-twice*:
 $m \leq n \implies word\ rec\ z\ f\ n = word\ rec\ (word\ rec\ z\ f\ (n - m))\ (f \circ (+)\ (n - m))\ m$
 $\langle proof \rangle$

lemma *word-rec-id*: $word\ rec\ z\ (\lambda\ -. id)\ n = z$
 $\langle proof \rangle$

lemma *word-rec-id-eq*: $(\bigwedge m. m < n \implies f\ m = id) \implies word\ rec\ z\ f\ n = z$
 $\langle proof \rangle$

lemma *word-rec-max*:
assumes $\forall m \geq n. m \neq -1 \longrightarrow f\ m = id$

shows $\text{word-rec } z \ f \ (-\ 1) = \text{word-rec } z \ f \ n$
 $\langle \text{proof} \rangle$

end

98.28 Tool support

$\langle ML \rangle$

end

99 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* = $\langle UNIV :: \text{bool set} \rangle \langle \text{proof} \rangle$

instantiation *bit* :: *zero-neq-one*
begin

definition *zero-bit* :: *bit*
where $\langle 0 = \text{Abs-bit False} \rangle$

definition *one-bit* :: *bit*
where $\langle 1 = \text{Abs-bit True} \rangle$

instance
 $\langle \text{proof} \rangle$

end

free-constructors *case-bit* **for** $\langle 0 :: \text{bit} \rangle \mid \langle 1 :: \text{bit} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-not-zero-iff* [*simp*]:
 $\langle a \neq 0 \longleftrightarrow a = 1 \rangle$ **for** $a :: \text{bit}$
 $\langle \text{proof} \rangle$

lemma *bit-not-one-iff* [*simp*]:
 $\langle a \neq 1 \longleftrightarrow a = 0 \rangle$ **for** $a :: \text{bit}$
 $\langle \text{proof} \rangle$

instantiation *bit* :: *semidom-modulo*
begin

definition *plus-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where $\langle a + b = \text{Abs-bit } (\text{Rep-bit } a \neq \text{Rep-bit } b) \rangle$

definition *minus-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where $[\text{simp}]: \langle \text{minus-bit} = \text{plus} \rangle$

definition *times-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where $\langle a * b = \text{Abs-bit } (\text{Rep-bit } a \wedge \text{Rep-bit } b) \rangle$

definition *divide-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where $[\text{simp}]: \langle \text{divide-bit} = \text{times} \rangle$

definition *modulo-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where $\langle a \text{ mod } b = \text{Abs-bit } (\text{Rep-bit } a \wedge \neg \text{Rep-bit } b) \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *bit-2-eq-0* $[\text{simp}]$:
 $\langle 2 = (0::\text{bit}) \rangle$
 $\langle \text{proof} \rangle$

instance *bit* :: *semiring-parity*
 $\langle \text{proof} \rangle$

lemma *Abs-bit-eq-of-bool* $[\text{code-abbrev}]$:
 $\langle \text{Abs-bit} = \text{of-bool} \rangle$
 $\langle \text{proof} \rangle$

lemma *Rep-bit-eq-odd*:
 $\langle \text{Rep-bit} = \text{odd} \rangle$
 $\langle \text{proof} \rangle$

lemma *Rep-bit-iff-odd* $[\text{code-abbrev}]$:
 $\langle \text{Rep-bit } b \longleftrightarrow \text{odd } b \rangle$
 $\langle \text{proof} \rangle$

lemma *Not-Rep-bit-iff-even* $[\text{code-abbrev}]$:
 $\langle \neg \text{Rep-bit } b \longleftrightarrow \text{even } b \rangle$
 $\langle \text{proof} \rangle$

lemma *Not-Not-Rep-bit* $[\text{code-unfold}]$:
 $\langle \neg \neg \text{Rep-bit } b \longleftrightarrow \text{Rep-bit } b \rangle$
 $\langle \text{proof} \rangle$

code-datatype $\langle 0::\text{bit} \rangle \langle 1::\text{bit} \rangle$

lemma *Abs-bit-code* [*code*]:

⟨*Abs-bit False* = 0⟩

⟨*Abs-bit True* = 1⟩

⟨*proof*⟩

lemma *Rep-bit-code* [*code*]:

⟨*Rep-bit 0* $\longleftrightarrow$ *False*⟩

⟨*Rep-bit 1* $\longleftrightarrow$ *True*⟩

⟨*proof*⟩

context *zero-neq-one*

begin

abbreviation *of-bit* :: ⟨*bit* $\Rightarrow$ 'a⟩

where ⟨*of-bit b* $\equiv$ *of-bool (odd b)*⟩

end

context

begin

qualified lemma *bit-eq-iff*:

⟨*a* = *b* $\longleftrightarrow$ (*even a* $\longleftrightarrow$ *even b*)⟩ **for** *a b* :: *bit*

⟨*proof*⟩

end

lemma *modulo-bit-unfold* [*simp*, *code*]:

⟨*a mod b* = *of-bool (odd a $\wedge$ even b)*⟩ **for** *a b* :: *bit*

⟨*proof*⟩

lemma *power-bit-unfold* [*simp*, *code*]:

⟨*a* $\wedge$ *n* = *of-bool (odd a $\vee$ n = 0)*⟩ **for** *a* :: *bit*

⟨*proof*⟩

instantiation *bit* :: *field*

begin

definition *uminus-bit* :: ⟨*bit* $\Rightarrow$ *bit*⟩

where [*simp*]: ⟨*uminus-bit* = *id*⟩

definition *inverse-bit* :: ⟨*bit* $\Rightarrow$ *bit*⟩

where [*simp*]: ⟨*inverse-bit* = *id*⟩

instance

⟨*proof*⟩

end

instantiation *bit* :: *semiring-bits*
begin

definition *bit-bit* :: $\langle \text{bit} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$
where [*simp*]: $\langle \text{bit-bit } b \ n \longleftrightarrow \text{odd } b \wedge n = 0 \rangle$

instance
 $\langle \text{proof} \rangle$

end

instantiation *bit* :: *ring-bit-operations*
begin

context
includes *bit-operations-syntax*
begin

definition *not-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{NOT } b = \text{of-bool } (\text{even } b) \rangle$ **for** *b* :: *bit*

definition *and-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \ \text{AND} \ c = \text{of-bool } (\text{odd } b \wedge \text{odd } c) \rangle$ **for** *b c* :: *bit*

definition *or-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \ \text{OR} \ c = \text{of-bool } (\text{odd } b \vee \text{odd } c) \rangle$ **for** *b c* :: *bit*

definition *xor-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \ \text{XOR} \ c = \text{of-bool } (\text{odd } b \neq \text{odd } c) \rangle$ **for** *b c* :: *bit*

definition *mask-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{mask } n = (\text{of-bool } (n > 0)) :: \text{bit} \rangle$

definition *set-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{set-bit } n \ b = \text{of-bool } (n = 0 \vee \text{odd } b) \rangle$ **for** *b* :: *bit*

definition *unset-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{unset-bit } n \ b = \text{of-bool } (n > 0 \wedge \text{odd } b) \rangle$ **for** *b* :: *bit*

definition *flip-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{flip-bit } n \ b = \text{of-bool } ((n = 0) \neq \text{odd } b) \rangle$ **for** *b* :: *bit*

definition *push-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{push-bit } n \ b = \text{of-bool } (\text{odd } b \wedge n = 0) \rangle$ **for** *b* :: *bit*

definition *drop-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{drop-bit } n \ b = \text{of-bool } (\text{odd } b \wedge n = 0) \rangle$ **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 w) = (0 :: bit)
  ⟨proof⟩

lemma bit-numeral-odd [simp]: numeral (Num.Bit1 w) = (1 :: bit)
  ⟨proof⟩

end

```

100 Pointwise order on product types

```

theory Product-Order
imports Product-Plus
begin

```

100.1 Pointwise ordering

```

instantiation prod :: (ord, ord) ord
begin

```

```

definition
   $x \leq y \longleftrightarrow \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$ 

```

```

definition
   $(x::'a \times 'b) < y \longleftrightarrow x \leq y \wedge \neg y \leq x$ 

```

```

instance ⟨proof⟩

```

```

end

```

```

lemma fst-mono:  $x \leq y \implies \text{fst } x \leq \text{fst } y$ 

```

$\langle proof \rangle$

lemma *snd-mono*: $x \leq y \implies snd\ x \leq snd\ y$
 $\langle proof \rangle$

lemma *Pair-mono*: $x \leq x' \implies y \leq y' \implies (x, y) \leq (x', y')$
 $\langle proof \rangle$

lemma *Pair-le [simp]*: $(a, b) \leq (c, d) \iff a \leq c \wedge b \leq d$
 $\langle proof \rangle$

instance *prod* :: (*preorder*, *preorder*) *preorder*
 $\langle proof \rangle$

instance *prod* :: (*order*, *order*) *order*
 $\langle proof \rangle$

100.2 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]*: $inf\ (a, b)\ (c, d) = (inf\ a\ c,\ inf\ b\ d)$
 $\langle proof \rangle$

lemma *fst-inf [simp]*: $fst\ (inf\ x\ y) = inf\ (fst\ x)\ (fst\ y)$
 $\langle proof \rangle$

lemma *snd-inf [simp]*: $snd\ (inf\ x\ y) = inf\ (snd\ x)\ (snd\ y)$
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

instance *prod* :: (*semilattice-inf*, *semilattice-inf*) *semilattice-inf*
 $\langle proof \rangle$

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]*: $sup\ (a, b)\ (c, d) = (sup\ a\ c,\ sup\ b\ d)$
 $\langle proof \rangle$

lemma *fst-sup* [*simp*]: $\text{fst } (\text{sup } x \ y) = \text{sup } (\text{fst } x) (\text{fst } y)$
 $\langle \text{proof} \rangle$

lemma *snd-sup* [*simp*]: $\text{snd } (\text{sup } x \ y) = \text{sup } (\text{snd } x) (\text{snd } y)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instance *prod* :: (*semilattice-sup*, *semilattice-sup*) *semilattice-sup*
 $\langle \text{proof} \rangle$

instance *prod* :: (*lattice*, *lattice*) *lattice* $\langle \text{proof} \rangle$

instance *prod* :: (*distrib-lattice*, *distrib-lattice*) *distrib-lattice*
 $\langle \text{proof} \rangle$

100.3 Top and bottom elements

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
 $\text{top} = (\text{top}, \text{top})$

instance $\langle \text{proof} \rangle$

end

lemma *fst-top* [*simp*]: $\text{fst } \text{top} = \text{top}$
 $\langle \text{proof} \rangle$

lemma *snd-top* [*simp*]: $\text{snd } \text{top} = \text{top}$
 $\langle \text{proof} \rangle$

lemma *Pair-top-top*: $(\text{top}, \text{top}) = \text{top}$
 $\langle \text{proof} \rangle$

instance *prod* :: (*order-top*, *order-top*) *order-top*
 $\langle \text{proof} \rangle$

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
 $\text{bot} = (\text{bot}, \text{bot})$

instance $\langle proof \rangle$

end

lemma *fst-bot* [*simp*]: *fst bot = bot*
 $\langle proof \rangle$

lemma *snd-bot* [*simp*]: *snd bot = bot*
 $\langle proof \rangle$

lemma *Pair-bot-bot*: $(bot, bot) = bot$
 $\langle proof \rangle$

instance *prod* :: $(order-bot, order-bot) \rightarrow order-bot$
 $\langle proof \rangle$

instance *prod* :: $(bounded-lattice, bounded-lattice) \rightarrow bounded-lattice$ $\langle proof \rangle$

instance *prod* :: $(boolean-algebra, boolean-algebra) \rightarrow boolean-algebra$
 $\langle proof \rangle$

100.4 Complete lattice operations

instantiation *prod* :: $(Inf, Inf) \rightarrow Inf$
begin

definition *Inf A* = $(INF\ x \in A. fst\ x, INF\ x \in A. snd\ x)$

instance $\langle proof \rangle$

end

instantiation *prod* :: $(Sup, Sup) \rightarrow Sup$
begin

definition *Sup A* = $(SUP\ x \in A. fst\ x, SUP\ x \in A. snd\ x)$

instance $\langle proof \rangle$

end

instance *prod* :: $(conditionally-complete-lattice, conditionally-complete-lattice) \rightarrow conditionally-complete-lattice$
 $\langle proof \rangle$

instance *prod* :: $(complete-lattice, complete-lattice) \rightarrow complete-lattice$
 $\langle proof \rangle$

lemma *fst-Inf*: *fst (Inf A) = (INF x ∈ A. fst x)*

<proof>

lemma *fst-INF*: $\text{fst } (\text{INF } x \in A. f x) = (\text{INF } x \in A. \text{fst } (f x))$
<proof>

lemma *fst-Sup*: $\text{fst } (\text{Sup } A) = (\text{SUP } x \in A. \text{fst } x)$
<proof>

lemma *fst-SUP*: $\text{fst } (\text{SUP } x \in A. f x) = (\text{SUP } x \in A. \text{fst } (f x))$
<proof>

lemma *snd-Inf*: $\text{snd } (\text{Inf } A) = (\text{INF } x \in A. \text{snd } x)$
<proof>

lemma *snd-INF*: $\text{snd } (\text{INF } x \in A. f x) = (\text{INF } x \in A. \text{snd } (f x))$
<proof>

lemma *snd-Sup*: $\text{snd } (\text{Sup } A) = (\text{SUP } x \in A. \text{snd } x)$
<proof>

lemma *snd-SUP*: $\text{snd } (\text{SUP } x \in A. f x) = (\text{SUP } x \in A. \text{snd } (f x))$
<proof>

lemma *INF-Pair*: $(\text{INF } x \in A. (f x, g x)) = (\text{INF } x \in A. f x, \text{INF } x \in A. g x)$
<proof>

lemma *SUP-Pair*: $(\text{SUP } x \in A. (f x, g x)) = (\text{SUP } x \in A. f x, \text{SUP } x \in A. g x)$
<proof>

Alternative formulations for set infima and suprema over the product of two complete lattices:

lemma *INF-prod-alt-def*:
 $\text{Inf } (f \text{ ‘ } A) = (\text{Inf } ((\text{fst} \circ f) \text{ ‘ } A), \text{Inf } ((\text{snd} \circ f) \text{ ‘ } A))$
<proof>

lemma *SUP-prod-alt-def*:
 $\text{Sup } (f \text{ ‘ } A) = (\text{Sup } ((\text{fst} \circ f) \text{ ‘ } A), \text{Sup } ((\text{snd} \circ f) \text{ ‘ } A))$
<proof>

100.5 Complete distributive lattices

instance *prod* :: (complete-distrib-lattice, complete-distrib-lattice) complete-distrib-lattice

<proof>

100.6 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 $F :: 'a::complete-lattice \times 'b::complete-lattice \Rightarrow 'a \times 'b$
assumes *mono F*
shows $lfp\ F = (lfp\ (\lambda x. fst\ (F\ (x, lfp\ (\lambda y. snd\ (F\ (x, y)))))),$
 $(lfp\ (\lambda y. snd\ (F\ (lfp\ (\lambda x. fst\ (F\ (x, lfp\ (\lambda y. snd\ (F\ (x, y))))), y))))$
(is $lfp\ F = (?x, ?y)$
 $\langle proof \rangle$

lemma *gfp-prod*:

fixes $F :: 'a::complete-lattice \times 'b::complete-lattice \Rightarrow 'a \times 'b$
assumes *mono F*
shows $gfp\ F = (gfp\ (\lambda x. fst\ (F\ (x, GFP\ (\lambda y. snd\ (F\ (x, y)))))),$
 $(gfp\ (\lambda y. snd\ (F\ (gfp\ (\lambda x. fst\ (F\ (x, GFP\ (\lambda y. snd\ (F\ (x, y))))), y))))$
(is $gfp\ F = (?x, ?y)$
 $\langle proof \rangle$

end

theory *Finite-Lattice*
imports *Product-Order*
begin

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*: $bot = Inf\text{-}fin\ UNIV$
assumes *top-def*: $top = Sup\text{-}fin\ UNIV$
assumes *Inf-def*: $Inf\ A = Finite\text{-}Set.fold\ inf\ top\ A$
assumes *Sup-def*: $Sup\ A = Finite\text{-}Set.fold\ sup\ bot\ A$

The definitional assumptions on the operators *bot* and *top* of class *finite-lattice-complete* ensure that they yield bottom and top.

lemma *finite-lattice-complete-bot-least*: $(bot::'a::finite-lattice-complete) \leq x$
 $\langle proof \rangle$

instance *finite-lattice-complete* $\subseteq$ *order-bot*
 $\langle proof \rangle$

lemma *finite-lattice-complete-top-greatest*: $(top::'a::finite-lattice-complete) \geq x$
 $\langle proof \rangle$

instance *finite-lattice-complete* $\subseteq$ *order-top*
 $\langle proof \rangle$

instance *finite-lattice-complete* $\subseteq$ *bounded-lattice* $\langle$ proof $\rangle$

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*: $\text{Inf } \{\} = (\text{top} :: 'a::\text{finite-lattice-complete})$
 $\langle$ proof $\rangle$

lemma *finite-lattice-complete-Sup-empty*: $\text{Sup } \{\} = (\text{bot} :: 'a::\text{finite-lattice-complete})$
 $\langle$ proof $\rangle$

lemma *finite-lattice-complete-Inf-insert*:
fixes $A :: 'a::\text{finite-lattice-complete}$ *set*
shows $\text{Inf } (\text{insert } x \ A) = \text{inf } x \ (\text{Inf } A)$
 $\langle$ proof $\rangle$

lemma *finite-lattice-complete-Sup-insert*:
fixes $A :: 'a::\text{finite-lattice-complete}$ *set*
shows $\text{Sup } (\text{insert } x \ A) = \text{sup } x \ (\text{Sup } A)$
 $\langle$ proof $\rangle$

lemma *finite-lattice-complete-Inf-lower*:
 $(x :: 'a::\text{finite-lattice-complete}) \in A \implies \text{Inf } A \leq x$
 $\langle$ proof $\rangle$

lemma *finite-lattice-complete-Inf-greatest*:
 $\forall x :: 'a::\text{finite-lattice-complete} \in A. z \leq x \implies z \leq \text{Inf } A$
 $\langle$ proof $\rangle$

lemma *finite-lattice-complete-Sup-upper*:
 $(x :: 'a::\text{finite-lattice-complete}) \in A \implies \text{Sup } A \geq x$
 $\langle$ proof $\rangle$

lemma *finite-lattice-complete-Sup-least*:
 $\forall x :: 'a::\text{finite-lattice-complete} \in A. z \geq x \implies z \geq \text{Sup } A$
 $\langle$ proof $\rangle$

instance *finite-lattice-complete* $\subseteq$ *complete-lattice*
 $\langle$ proof $\rangle$

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
 $(\text{bot} :: ('a::\text{finite-lattice-complete} \times 'b::\text{finite-lattice-complete})) =$
 Inf-fin UNIV
 $\langle$ proof $\rangle$

lemma *finite-top-prod*:
 $(\text{top} :: ('a::\text{finite-lattice-complete} \times 'b::\text{finite-lattice-complete})) =$
 Sup-fin UNIV
 $\langle$ proof $\rangle$

lemma *finite-Inf-prod*:

$Inf(A :: ('a::finite-lattice-complete \times 'b::finite-lattice-complete) set) =$
 $Finite-Set.fold\ inf\ top\ A$
 $\langle proof \rangle$

lemma *finite-Sup-prod*:

$Sup(A :: ('a::finite-lattice-complete \times 'b::finite-lattice-complete) set) =$
 $Finite-Set.fold\ sup\ bot\ A$
 $\langle proof \rangle$

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
 $\langle proof \rangle$

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

lemma *finite-bot-fun*: (*bot* :: (*a*::*finite* $\Rightarrow$ *b*::*finite-lattice-complete*)) = *Inf-fin UNIV*
 $\langle proof \rangle$

lemma *finite-top-fun*: (*top* :: (*a*::*finite* $\Rightarrow$ *b*::*finite-lattice-complete*)) = *Sup-fin UNIV*
 $\langle proof \rangle$

lemma *finite-Inf-fun*:

$Inf(A :: ('a::finite \Rightarrow 'b::finite-lattice-complete) set) =$
 $Finite-Set.fold\ inf\ top\ A$
 $\langle proof \rangle$

lemma *finite-Sup-fun*:

$Sup(A :: ('a::finite \Rightarrow 'b::finite-lattice-complete) set) =$
 $Finite-Set.fold\ sup\ bot\ A$
 $\langle proof \rangle$

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*
 $\langle proof \rangle$

100.7 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*:

$sup(x :: 'a::finite-distrib-lattice-complete) (Inf\ A) = (INF\ y \in A.\ sup\ x\ y)$
 $\langle proof \rangle$

lemma *finite-distrib-lattice-complete-inf-Sup*:

$inf(x :: 'a::finite-distrib-lattice-complete) (Sup\ A) = (SUP\ y \in A.\ inf\ x\ y)$

<proof>

```
context finite-distrib-lattice-complete
begin
subclass finite-distrib-lattice
<proof>
end
```

instance *finite-distrib-lattice-complete* $\subseteq$ *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>
```

100.8 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  $\leq$  y then x else y)
  assumes sup-def: sup x y = (if x  $\geq$  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  $\leq$  x
<proof>
```

```
lemma linorder-lattice-inf-le2: inf (x::'a::linorder-lattice) y  $\leq$  y
<proof>
```

```
lemma linorder-lattice-inf-greatest:
  (x::'a::linorder-lattice)  $\leq$  y  $\implies$  x  $\leq$  z  $\implies$  x  $\leq$  inf y z
<proof>
```

```
lemma linorder-lattice-sup-ge1: sup (x::'a::linorder-lattice) y  $\geq$  x
<proof>
```

lemma *linorder-lattice-sup-ge2*: $\text{sup } (x::'a::\text{linorder-lattice}) \ y \geq y$
 $\langle \text{proof} \rangle$

lemma *linorder-lattice-sup-least*:
 $(x::'a::\text{linorder-lattice}) \geq y \implies x \geq z \implies x \geq \text{sup } y \ z$
 $\langle \text{proof} \rangle$

lemma *linorder-lattice-sup-inf-distrib1*:
 $\text{sup } (x::'a::\text{linorder-lattice}) \ (\text{inf } y \ z) = \text{inf } (\text{sup } x \ y) \ (\text{sup } x \ z)$
 $\langle \text{proof} \rangle$

instance *linorder-lattice* $\subseteq$ *distrib-lattice*
 $\langle \text{proof} \rangle$

100.9 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* $\subseteq$ *complete-linorder* $\langle \text{proof} \rangle$

A (non-empty) finite linear order is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

instance *finite-linorder-complete* $\subseteq$ *finite-distrib-lattice-complete* $\langle \text{proof} \rangle$

end

101 Lexicographic order on lists

theory *List-Lexorder*
imports *Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def: $xs < ys \longleftrightarrow (xs, ys) \in \text{lexord } \{(u, v). u < v\}$

definition
list-le-def: $(xs :: \text{list}) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

instance $\langle \text{proof} \rangle$

end

instance *list* :: (*order*) *order*

$\langle proof \rangle$

instance *list* :: (*linorder*) *linorder*
 $\langle proof \rangle$

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition (*inf* :: 'a *list* $\Rightarrow$ -) = *min*

definition (*sup* :: 'a *list* $\Rightarrow$ -) = *max*

instance
 $\langle proof \rangle$

end

lemma *not-less-Nil* [*simp*]: $\neg x < []$
 $\langle proof \rangle$

lemma *Nil-less-Cons* [*simp*]: $[] < a \# x$
 $\langle proof \rangle$

lemma *Cons-less-Cons* [*simp*]: $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$
 $\langle proof \rangle$

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
 $\langle proof \rangle$

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
 $\langle proof \rangle$

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
 $\langle proof \rangle$

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = $[]$

instance
 $\langle proof \rangle$

end

lemma *less-list-code* [*code*]:
 $xs < ([] :: 'a :: \{equal, order\} list) \longleftrightarrow False$
 $[] < (x :: 'a :: \{equal, order\}) \# xs \longleftrightarrow True$
 $(x :: 'a :: \{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$

$\langle proof \rangle$

lemma *less-eq-list-code* [code]:

$x \# xs \leq ([::'a::\{equal, order\} list) \longleftrightarrow False$

$[] \leq (xs::'a::\{equal, order\} list) \longleftrightarrow True$

$(x::'a::\{equal, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$

$\langle proof \rangle$

end

102 Lexicographic order on lists

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 $\langle proof \rangle$

end

instance *list* :: (*order*) *order*

$\langle proof \rangle$

instance *list* :: (*linorder*) *linorder*

$\langle proof \rangle$

instantiation *list* :: (*linorder*) *distrib-lattice*

begin

definition (*inf* :: '*a* *list* $\Rightarrow$ -) = *min*

definition (*sup* :: '*a* *list* $\Rightarrow$ -) = *max*

instance

$\langle proof \rangle$

end

lemma *not-less-Nil* [*simp*]: $\neg x < []$

<proof>

lemma *Nil-less-Cons* [simp]: $[] < a \# x$
<proof>

lemma *Cons-less-Cons*: $a \# x < b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x < y)$
<proof>

lemma *le-Nil* [simp]: $x \leq [] \longleftrightarrow x = []$
<proof>

lemma *Nil-le-Cons* [simp]: $[] \leq x$
<proof>

lemma *Cons-le-Cons*: $a \# x \leq b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x \leq y)$
<proof>

instantiation *list* :: (order) order-bot
begin

definition *bot* = []

instance
<proof>

end

end

103 Prefix order on lists as order class instance

theory *Prefix-Order*
imports *Sublist*
begin

instantiation *list* :: (type) order
begin

definition $xs \leq ys \equiv \text{prefix } xs \text{ } ys$ **for** $xs \text{ } ys :: 'a \text{ list}$

definition $xs < ys \equiv xs \leq ys \wedge \neg (ys \leq xs)$ **for** $xs \text{ } ys :: 'a \text{ list}$

instance
<proof>

end

lemma *less-list-def'*: $xs < ys \longleftrightarrow \text{strict-prefix } xs \text{ } ys$ **for** $xs \text{ } ys :: 'a \text{ list}$

<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

104 Lexicographic order on product types

theory *Product-Lexorder*

imports *Main*

begin

instantiation *prod* :: (ord, ord) ord

begin

definition

$$x \leq y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$$

definition

$$x < y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x < \text{snd } y$$

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 $x < y \iff fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x < snd\ y$

$\langle proof \rangle$

instance *prod* :: (preorder, preorder) preorder

$\langle proof \rangle$

instance *prod* :: (order, order) order

$\langle proof \rangle$

instance *prod* :: (linorder, linorder) linorder

$\langle proof \rangle$

instantiation *prod* :: (linorder, linorder) distrib-lattice

begin

definition

$(inf :: 'a \times 'b \Rightarrow - \Rightarrow -) = min$

definition

$(sup :: 'a \times 'b \Rightarrow - \Rightarrow -) = max$

instance

$\langle proof \rangle$

end

instantiation *prod* :: (bot, bot) bot

begin

definition

$bot = (bot, bot)$

instance $\langle proof \rangle$

end

instance *prod* :: (order-bot, order-bot) order-bot

$\langle proof \rangle$

instantiation *prod* :: (top, top) top

begin

definition

$top = (top, top)$

instance $\langle proof \rangle$

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

105 Subsequence Ordering

theory *Subseq-Order*

imports *Sublist*

begin

This theory defines subsequence ordering on lists. A list *ys* is a subsequence of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

105.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *ord*

begin

definition $xs \leq ys \longleftrightarrow \text{subseq } xs \ ys$ **for** $xs \ ys :: 'a \ \text{list}$

definition $xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs$ **for** $xs \ ys :: 'a \ \text{list}$

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*]

lemmas *less-eq-list-drop* = *list-emb.list-emb-Cons* [*of* (=), *folded less-eq-list-def*]

lemmas *le-list-Cons2-iff* [*simp*, *code*] = *subseq-Cons2-iff* [*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*: $xs < ys \implies \text{length } xs < \text{length } ys$

$\langle proof \rangle$

lemma *less-list-empty* [simp]: $[] < xs \longleftrightarrow xs \neq []$
 $\langle proof \rangle$

lemma *less-list-below-empty* [simp]: $xs < [] \longleftrightarrow False$
 $\langle proof \rangle$

lemma *less-list-drop*: $xs < ys \implies xs < x \# ys$
 $\langle proof \rangle$

lemma *less-list-take-iff*: $x \# xs < x \# ys \longleftrightarrow xs < ys$
 $\langle proof \rangle$

lemma *less-list-drop-many*: $xs < ys \implies xs < zs @ ys$
 $\langle proof \rangle$

lemma *less-list-take-many-iff*: $zs @ xs < zs @ ys \longleftrightarrow xs < ys$
 $\langle proof \rangle$

lemma *less-list-rev-take*: $xs @ zs < ys @ zs \longleftrightarrow xs < ys$
 $\langle proof \rangle$

end

106 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 $\langle \dots \rangle$ 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

no-syntax

```

-constify      :: id => ident          (-)
-constify      :: longid => ident      (-)

-field-type     :: ident => type => field-type      ((2- ::/ -))
               :: field-type => field-types      (-)
-field-types    :: field-type => field-types => field-types  (-,/ -)
-record-type    :: field-types => type              ((3(|-|)))
-record-type-scheme :: field-types => type => type      ((3(|-/, (2... ::/ -)|)))

-field          :: ident => 'a => field              ((2- =/ -))
               :: field => fields                  (-)
-fields         :: field => fields => fields          (-,/ -)
-record         :: fields => 'a                    ((3(|-|)))
-record-scheme  :: fields => 'a => 'a                ((3(|-/, (2... =/ -)|)))

-field-update   :: ident => 'a => field-update        ((2- :=/ -))
               :: field-update => field-updates      (-)
-field-updates  :: field-update => field-updates => field-updates  (-,/ -)
-record-update  :: 'a => field-updates => 'b          (-/(3(|-|)) [900, 0] 900)

```

no-syntax (ASCII)

```

-record-type     :: field-types => type              ((3'(| - |')))
-record-type-scheme :: field-types => type => type      ((3'(| -,/ (2... ::/ -) |')))
-record          :: fields => 'a                    ((3'(| - |')))
-record-scheme   :: fields => 'a => 'a                ((3'(| -,/ (2... =/ -) |')))
-record-update   :: 'a => field-updates => 'b          (-/(3'(| - |')) [900, 0] 900)

```

nonterminal

field **and**
fields **and**
field-update **and**
field-updates

syntax

```

-constify      :: id => ident          (-)
-constify      :: longid => ident      (-)

-datatype-field  :: ident => 'a => field              ((2- =/ -))
                :: field => fields                  (-)
-datatype-fields :: field => fields => fields          (-,/ -)
-datatype-record :: fields => 'a                    ((3(|-|)))
-datatype-field-update :: ident => 'a => field-update  ((2- :=/ -))
                    :: field-update => field-updates  (-)
-datatype-field-updates :: field-update => field-updates => field-updates  (-,/ -)

```

-datatype-record-update :: 'a => field-updates => 'b
900) (-(3(|-)) [900, 0]

syntax (ASCII)

-datatype-record :: fields => 'a ((3'(| - |'))
-datatype-record-scheme :: fields => 'a => 'a ((3'(| -, / (2... = / -)
|'))
-datatype-record-update :: 'a => field-updates => 'b (-(3'(| - |')) [900,
0] 900)

named-theorems datatype-record-update

⟨ML⟩

end

107 Implementation of mappings with Association Lists

theory AList-Mapping
imports AList Mapping
begin

lift-definition Mapping :: ('a × 'b) list ⇒ ('a, 'b) mapping **is** map-of ⟨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* ($\lambda k. (k, f\ k)$) *ks*)
 ⟨proof⟩

lemma *bulkload-Mapping* [code]:
Mapping.bulkload vs = *Mapping* (*map* ($\lambda n. (n, vs\ !\ n)$) [*0..length vs*])
 ⟨proof⟩

lemma *equal-Mapping* [code]:
HOL.equal (*Mapping xs*) (*Mapping ys*) $\longleftrightarrow$
 (*let* *ks* = *map fst xs*; *ls* = *map fst ys*
 in $(\forall l \in \text{set } ls. l \in \text{set } ks) \wedge (\forall k \in \text{set } ks. k \in \text{set } ls \wedge \text{map-of } xs\ k = \text{map-of } ys\ k)$)
 ⟨proof⟩

lemma *map-values-Mapping* [code]:
Mapping.map-values f (*Mapping xs*) = *Mapping* (*map* ($\lambda(x,y). (x, f\ x\ y)$) *xs*)
for *f* :: '*c* $\Rightarrow$ '*a* $\Rightarrow$ '*b* **and** *xs* :: ('*c* $\times$ '*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*))
 ($\lambda x. \text{the } (\text{combine-options } (f\ x) (\text{map-of } xs\ x) (\text{map-of } ys\ x))$)
 ⟨proof⟩

lemma *combine-code* [code]:
Mapping.combine f (*Mapping xs*) (*Mapping ys*) =
Mapping.tabulate (*remdups* (*map fst xs* @ *map fst ys*))

($\lambda x.$ 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 $\Rightarrow$ None
 | Some $y \Rightarrow$ if P (x,y) then Some y else None)
 ⟨proof⟩

lemma filter-Mapping [code]:
 Mapping.filter P (Mapping xs) = Mapping (filter ($\lambda(k,v).$ P k v) (AList.clearjunk
 xs))
 ⟨proof⟩

lemma [code nbe]: HOL.equal ($x :: ('a, 'b)$ mapping) $x \longleftrightarrow$ True
 ⟨proof⟩

end

108 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).

108.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [code, code-unfold]:
 case-nat = ($\lambda f g n.$ if $n = 0$ then f else g ($n - 1$))
 ⟨proof⟩

108.2 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 $\bigwedge n. f \ (Suc \ n) \equiv h \ n$
assumes $f \ 0 \equiv g$
shows $f \ n \equiv \text{if } n = 0 \text{ then } g \text{ else } h \ (n - 1)$
 $\langle proof \rangle$

The rule above is built into a preprocessor that is plugged into the code generator.

$\langle ML \rangle$

108.3 One candidate which needs special treatment

lemma *take-bit-num-code* [code]:
 $\langle take-bit-num \ n \ Num.One =$
 $\quad (case \ n \ of \ 0 \Rightarrow None \mid Suc \ n \Rightarrow Some \ Num.One) \rangle$
 $\langle take-bit-num \ n \ (Num.Bit0 \ m) =$
 $\quad (case \ n \ of \ 0 \Rightarrow None \mid Suc \ n \Rightarrow (case \ take-bit-num \ n \ m \ of \ None \Rightarrow None \mid$
 $Some \ q \Rightarrow Some \ (Num.Bit0 \ q))) \rangle$
 $\langle take-bit-num \ n \ (Num.Bit1 \ m) =$
 $\quad (case \ n \ of \ 0 \Rightarrow None \mid Suc \ n \Rightarrow Some \ (case \ take-bit-num \ n \ m \ of \ None \Rightarrow$
 $Num.One \mid Some \ q \Rightarrow Num.Bit1 \ q)) \rangle$
 $\langle proof \rangle$

end

109 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.

109.1 Representation

code-datatype *0::nat nat-of-num*

lemma [code]:
 $num-of-nat \ 0 = Num.One$
 $num-of-nat \ (nat-of-num \ k) = k$
 $\langle proof \rangle$

lemma [code]:

$(1::nat) = \text{Numeral1}$
 $\langle \text{proof} \rangle$

lemma $[\text{code-abbrev}]$: $\text{Numeral1} = (1::nat)$
 $\langle \text{proof} \rangle$

lemma $[\text{code}]$:
 $\text{Suc } n = n + 1$
 $\langle \text{proof} \rangle$

109.2 Basic arithmetic

context
begin

declare $[[\text{code drop: plus} :: nat \Rightarrow -]]$

lemma $\text{plus-nat-code} [\text{code}]$:
 $\text{nat-of-num } k + \text{nat-of-num } l = \text{nat-of-num } (k + l)$
 $m + 0 = (m::nat)$
 $0 + n = (n::nat)$
 $\langle \text{proof} \rangle$

Bounded subtraction needs some auxiliary

qualified definition $\text{dup} :: nat \Rightarrow nat$ **where**
 $\text{dup } n = n + n$

lemma $\text{dup-code} [\text{code}]$:
 $\text{dup } 0 = 0$
 $\text{dup } (\text{nat-of-num } k) = \text{nat-of-num } (\text{Num.Bit0 } k)$
 $\langle \text{proof} \rangle$ **definition** $\text{sub} :: num \Rightarrow num \Rightarrow nat \text{ option}$ **where**
 $\text{sub } k \ l = (\text{if } k \geq l \text{ then } \text{Some } (\text{numeral } k - \text{numeral } l) \text{ else } \text{None})$

lemma $\text{sub-code} [\text{code}]$:
 $\text{sub } \text{Num.One } \text{Num.One} = \text{Some } 0$
 $\text{sub } (\text{Num.Bit0 } m) \ \text{Num.One} = \text{Some } (\text{nat-of-num } (\text{Num.BitM } m))$
 $\text{sub } (\text{Num.Bit1 } m) \ \text{Num.One} = \text{Some } (\text{nat-of-num } (\text{Num.Bit0 } m))$
 $\text{sub } \text{Num.One } (\text{Num.Bit0 } n) = \text{None}$
 $\text{sub } \text{Num.One } (\text{Num.Bit1 } n) = \text{None}$
 $\text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit0 } n) = \text{map-option } \text{dup } (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit1 } n) = \text{map-option } \text{dup } (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit0 } n) = \text{map-option } (\lambda q. \text{dup } q + 1) (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit1 } n) = (\text{case } \text{sub } m \ n \text{ of } \text{None} \Rightarrow \text{None}$
 $\quad | \text{Some } q \Rightarrow \text{if } q = 0 \text{ then } \text{None} \text{ else } \text{Some } (\text{dup } q - 1))$
 $\langle \text{proof} \rangle$

declare $[[\text{code drop: minus} :: nat \Rightarrow -]]$

lemma $\text{minus-nat-code} [\text{code}]$:
 $\text{nat-of-num } k - \text{nat-of-num } l = (\text{case } \text{sub } k \ l \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } j \Rightarrow j)$

$m - 0 = (m::nat)$
 $0 - n = (0::nat)$
 $\langle proof \rangle$

declare $[[code\ drop: times :: nat \Rightarrow -]]$

lemma *times-nat-code* $[code]$:
 $nat-of-num\ k * nat-of-num\ l = nat-of-num\ (k * l)$
 $m * 0 = (0::nat)$
 $0 * n = (0::nat)$
 $\langle proof \rangle$

declare $[[code\ drop: HOL.equal :: nat \Rightarrow -]]$

lemma *equal-nat-code* $[code]$:
 $HOL.equal\ 0\ (0::nat) \longleftrightarrow True$
 $HOL.equal\ 0\ (nat-of-num\ l) \longleftrightarrow False$
 $HOL.equal\ (nat-of-num\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (nat-of-num\ k)\ (nat-of-num\ l) \longleftrightarrow HOL.equal\ k\ l$
 $\langle proof \rangle$

lemma *equal-nat-refl* $[code\ nbe]$:
 $HOL.equal\ (n::nat)\ n \longleftrightarrow True$
 $\langle proof \rangle$

declare $[[code\ drop: less-eq :: nat \Rightarrow -]]$

lemma *less-eq-nat-code* $[code]$:
 $0 \leq (n::nat) \longleftrightarrow True$
 $nat-of-num\ k \leq 0 \longleftrightarrow False$
 $nat-of-num\ k \leq nat-of-num\ l \longleftrightarrow k \leq l$
 $\langle proof \rangle$

declare $[[code\ drop: less :: nat \Rightarrow -]]$

lemma *less-nat-code* $[code]$:
 $(m::nat) < 0 \longleftrightarrow False$
 $0 < nat-of-num\ l \longleftrightarrow True$
 $nat-of-num\ k < nat-of-num\ l \longleftrightarrow k < l$
 $\langle proof \rangle$

declare $[[code\ drop: Divides.divmod-nat]]$

lemma *divmod-nat-code* $[code]$:
 $Divides.divmod-nat\ (nat-of-num\ k)\ (nat-of-num\ l) = divmod\ k\ l$
 $Divides.divmod-nat\ m\ 0 = (0, m)$
 $Divides.divmod-nat\ 0\ n = (0, 0)$
 $\langle proof \rangle$

end

109.3 Conversions

declare $[[code\ drop:\ of\ nat]]$

lemma *of-nat-code* $[code]:$
 $of\ nat\ 0 = 0$
 $of\ nat\ (nat\ of\ num\ k) = numeral\ k$
 $\langle proof \rangle$

code-identifier

code-module *Code-Binary-Nat* $\rightarrow$
 $(SML)\ Arith\ \mathbf{and}\ (OCaml)\ Arith\ \mathbf{and}\ (Haskell)\ Arith$

end

110 Code generation of prolog programs

theory *Code-Prolog*
imports *Main*
keywords *values-prolog :: diag*
begin

$\langle ML \rangle$

111 Setup for Numerals

$\langle ML \rangle$

end

112 Implementation of integer numbers by target-language integers

theory *Code-Target-Int*
imports *Main*
begin

code-datatype *int-of-integer*

declare $[[code\ drop:\ integer\ of\ int]]$

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]:

$k + l = \text{int-of-integer } (\text{of-int } k + \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$-k = \text{int-of-integer } (-\text{of-int } k)$
 $\langle \text{proof} \rangle$

lemma [code]:

$k - l = \text{int-of-integer } (\text{of-int } k - \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$\text{Int.dup } k = \text{int-of-integer } (\text{Code-Numeral.dup } (\text{of-int } k))$
 $\langle \text{proof} \rangle$

declare [[code drop: Int.sub]]

lemma [code]:

$k * l = \text{int-of-integer } (\text{of-int } k * \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$k \text{ div } l = \text{int-of-integer } (\text{of-int } k \text{ div } \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$k \text{ mod } l = \text{int-of-integer } (\text{of-int } k \text{ mod } \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$\text{divmod } m \ n = \text{map-prod int-of-integer int-of-integer } (\text{divmod } m \ n)$
 $\langle \text{proof} \rangle$

lemma [code]:

$\text{HOL.equal } k \ l = \text{HOL.equal } (\text{of-int } k :: \text{integer}) \ (\text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$k \leq l \iff (\text{of-int } k :: \text{integer}) \leq \text{of-int } l$
 $\langle \text{proof} \rangle$

lemma [code]:

$k < l \iff (\text{of-int } k :: \text{integer}) < \text{of-int } l$
 $\langle \text{proof} \rangle$

declare [[code drop: gcd :: int $\Rightarrow$ - lcm :: int $\Rightarrow$ -]]

lemma *gcd-int-of-integer* [code]:
 $\text{gcd } (\text{int-of-integer } x) (\text{int-of-integer } y) = \text{int-of-integer } (\text{gcd } x \ y)$
 ⟨proof⟩

lemma *lcm-int-of-integer* [code]:
 $\text{lcm } (\text{int-of-integer } x) (\text{int-of-integer } y) = \text{int-of-integer } (\text{lcm } x \ y)$
 ⟨proof⟩

end

lemma (*in ring-1*) *of-int-code-if*:
 $\text{of-int } k = (\text{if } k = 0 \text{ then } 0$
 $\quad \text{else if } k < 0 \text{ then } - \text{of-int } (-k)$
 $\quad \text{else let}$
 $\quad \quad l = 2 * \text{of-int } (k \text{ div } 2);$
 $\quad \quad j = k \text{ mod } 2$
 $\quad \text{in if } j = 0 \text{ then } l \text{ else } l + 1)$
 ⟨proof⟩

declare *of-int-code-if* [code]

lemma [code]:
 $\text{nat} = \text{nat-of-integer} \circ \text{of-int}$
including *integer.lifting* ⟨proof⟩

definition *char-of-int* :: $\text{int} \Rightarrow \text{char}$
where [code-abbrev]: *char-of-int* = *char-of*

definition *int-of-char* :: $\text{char} \Rightarrow \text{int}$
where [code-abbrev]: *int-of-char* = *of-char*

lemma [code]:
 $\text{char-of-int} = \text{char-of-integer} \circ \text{integer-of-int}$
including *integer.lifting* ⟨proof⟩

lemma [code]:
 $\text{int-of-char} = \text{int-of-integer} \circ \text{integer-of-char}$
including *integer.lifting* ⟨proof⟩

code-identifier

code-module *Code-Target-Int* $\rightarrow$
 (*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.

code-printing

type-constructor *real* $\rightarrow$
 (*SML*) *real*
 and (*OCaml*) *float*

code-printing

constant *Ratreal* $\rightarrow$
 (*SML*) *error/ Bad constant: Ratreal*

code-printing

constant *0* :: *real* $\rightarrow$
 (*SML*) *0.0*
 and (*OCaml*) *0.0*

code-printing

constant *1* :: *real* $\rightarrow$
 (*SML*) *1.0*
 and (*OCaml*) *1.0*

code-printing

constant *HOL.equal* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool* $\rightarrow$
 (*SML*) *Real.==* ((-), (-))
 and (*OCaml*) *Pervasives.(=)*

code-printing

constant *Orderings.less-eq* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool* $\rightarrow$
 (*SML*) *Real.<=* ((-), (-))
 and (*OCaml*) *Pervasives.(<=)*

code-printing

constant *Orderings.less* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool* $\rightarrow$
 (*SML*) *Real.<* ((-), (-))
 and (*OCaml*) *Pervasives.(<)*

code-printing

constant *(+)* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real* $\rightarrow$
 (*SML*) *Real.+* ((-), (-))
 and (*OCaml*) *Pervasives.(+)*

code-printing

constant *(*)* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real* $\rightarrow$

(*SML*) *Real*.* ((-), (-))
 and (*OCaml*) *Pervasives*.(*.)

code-printing

constant (-) :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real* $\rightarrow$
 (*SML*) *Real*.- ((-), (-))
 and (*OCaml*) *Pervasives*.(-.)

code-printing

constant *uminus* :: *real* $\Rightarrow$ *real* $\rightarrow$
 (*SML*) *Real*.~
 and (*OCaml*) *Pervasives*.(~-.)

code-printing

constant (/) :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real* $\rightarrow$
 (*SML*) *Real*.'/ ((-), (-))
 and (*OCaml*) *Pervasives*.('/.)

code-printing

constant *HOL.equal* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool* $\rightarrow$
 (*SML*) *Real*.== ((-:*real*), (-))

code-printing

constant *sqrt* :: *real* $\Rightarrow$ *real* $\rightarrow$
 (*SML*) *Math*.*sqrt*
 and (*OCaml*) *Pervasives*.*sqrt*
declare *sqrt-def*[*code del*]

context**begin**

qualified definition *real-exp* :: *real* $\Rightarrow$ *real*

where *real-exp* = *exp*

lemma *exp-eq-real-exp* [*code-unfold*]: *exp* = *real-exp*
 ⟨*proof*⟩

end**code-printing**

constant *Code-Real-Approx-By-Float.real-exp* $\rightarrow$
 (*SML*) *Math*.*exp*
 and (*OCaml*) *Pervasives*.*exp*
declare *Code-Real-Approx-By-Float.real-exp-def*[*code del*]
declare *exp-def*[*code del*]

code-printing

constant *ln* $\rightarrow$
 (*SML*) *Math*.*ln*

```

    and (OCaml) Pervasives.ln
declare ln-real-def[code del]

code-printing
  constant cos  $\rightarrow$ 
    (SML) Math.cos
    and (OCaml) Pervasives.cos
declare cos-def[code del]

code-printing
  constant sin  $\rightarrow$ 
    (SML) Math.sin
    and (OCaml) Pervasives.sin
declare sin-def[code del]

code-printing
  constant pi  $\rightarrow$ 
    (SML) Math.pi
    and (OCaml) Pervasives.pi
declare pi-def[code del]

code-printing
  constant arctan  $\rightarrow$ 
    (SML) Math.atan
    and (OCaml) Pervasives.atan
declare arctan-def[code del]

code-printing
  constant arccos  $\rightarrow$ 
    (SML) Math.scos
    and (OCaml) Pervasives.acos
declare arccos-def[code del]

code-printing
  constant arcsin  $\rightarrow$ 
    (SML) Math.asin
    and (OCaml) Pervasives.asin
declare arcsin-def[code del]

definition real-of-integer :: integer  $\Rightarrow$  real
  where real-of-integer = of-int  $\circ$  int-of-integer

code-printing
  constant real-of-integer  $\rightarrow$ 
    (SML) Real.fromInt
    and (OCaml) Pervasives.float/ (Big'-int.to'-int (-))

context
begin

```

qualified definition *real-of-int* :: *int* $\Rightarrow$ *real*

where [*code-abbrev*]: *real-of-int* = *of-int*

lemma [*code*]: *real-of-int* = *real-of-integer* $\circ$ *integer-of-int*
 $\langle$ *proof* $\rangle$

lemma [*code-unfold del*]: 0 $\equiv$ (*of-rat* 0 :: *real*)
 $\langle$ *proof* $\rangle$

lemma [*code-unfold del*]: 1 $\equiv$ (*of-rat* 1 :: *real*)
 $\langle$ *proof* $\rangle$

lemma [*code-unfold del*]: *numeral k* $\equiv$ (*of-rat* (*numeral k*) :: *real*)
 $\langle$ *proof* $\rangle$

lemma [*code-unfold del*]: $-$ *numeral k* $\equiv$ (*of-rat* ($-$ *numeral k*) :: *real*)
 $\langle$ *proof* $\rangle$

end

code-printing

constant *Ratreal* $\rightarrow$ (*SML*)

definition *Realfract* :: *int* $\Rightarrow$ *int* $\Rightarrow$ *real*

where *Realfract p q* = *of-int p* / *of-int q*

code-datatype *Realfract*

code-printing

constant *Realfract* $\rightarrow$ (*SML*) *Real.fromInt* -/ '/' *Real.fromInt* -

lemma [*code*]: *Ratreal r* = *case-prod Realfract* (*quotient-of r*)
 $\langle$ *proof* $\rangle$

declare [[*code drop: HOL.equal* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool*

plus :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*

uminus :: *real* $\Rightarrow$ *real*

minus :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*

times :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*

divide :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real*

(<) :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool*

($\leq$) :: *real* $\Rightarrow$ *real* $\Rightarrow$ *bool*]]

lemma [*code*]: *inverse r* = 1 / *r* **for** *r* :: *real*
 $\langle$ *proof* $\rangle$

notepad

begin

```

    <proof>
  end

```

```

end

```

113 Implementation of natural numbers by target-language integers

```

theory Code-Target-Nat
imports Code-Abstract-Nat
begin

```

113.1 Implementation for *nat*

```

context
includes natural.lifting integer.lifting
begin

```

```

lift-definition Nat :: integer  $\Rightarrow$  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 $\text{natural} :: \text{num} \Rightarrow \text{nat}$
where $[\text{simp}]: \text{natural} = \text{nat-of-num}$

lemma $[\text{code-computation-unfold}]$:
 $\text{numeral} = \text{natural}$
 $\text{nat-of-num} = \text{natural}$
 $\langle \text{proof} \rangle$

end

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat} (\text{nat-of-num } n) = \text{integer-of-num } n$
 $\langle \text{proof} \rangle$

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat } 0 = 0$
 $\langle \text{proof} \rangle$

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat } 1 = 1$
 $\langle \text{proof} \rangle$

lemma $[\text{code}]$:
 $\text{Suc } n = n + 1$
 $\langle \text{proof} \rangle$

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat} (m + n) = \text{of-nat } m + \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat} (m - n) = \max 0 (\text{of-nat } m - \text{of-nat } n)$
 $\langle \text{proof} \rangle$

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat} (m * n) = \text{of-nat } m * \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat} (m \text{ div } n) = \text{of-nat } m \text{ div } \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma $[\text{code abstract}]$:
 $\text{integer-of-nat} (m \text{ mod } n) = \text{of-nat } m \text{ mod } \text{of-nat } n$
 $\langle \text{proof} \rangle$

context

includes *integer.lifting*

begin

lemma *divmod-nat-code* [code]:

Divides.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) = Divides.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*⟩

code-identifier
code-module *Code-Target-Nat* ↪
 (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

114 Implementation of natural and integer numbers by target-language integers

```

theory Code-Target-Numeral
imports Code-Target-Int Code-Target-Nat
begin

end

```

115 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]:
   $\langle \text{numeral } k = \text{Float } (\text{int-of-integer } (\text{Code-Numeral.positive } k)) \ 0 \rangle$ 
   $\langle - \text{numeral } k = \text{Float } (\text{int-of-integer } (\text{Code-Numeral.negative } k)) \ 0 \rangle$ 
   $\langle \text{proof} \rangle$ 

end

```

```

theory Complex-Order
  imports Complex-Main
begin

```

```

instantiation complex :: order begin

```

```

definition  $\langle x < y \longleftrightarrow \text{Re } x < \text{Re } y \wedge \text{Im } x = \text{Im } y \rangle$ 

```

```

definition  $\langle x \leq y \longleftrightarrow \text{Re } x \leq \text{Re } y \wedge \text{Im } x = \text{Im } y \rangle$ 

```

```

instance
   $\langle \text{proof} \rangle$ 
end

```

```

lemma nonnegative-complex-is-real:  $\langle (x::\text{complex}) \geq 0 \implies x \in \mathbb{R} \rangle$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma complex-is-real-iff-compare0:  $\langle (x::\text{complex}) \in \mathbb{R} \longleftrightarrow x \leq 0 \vee x \geq 0 \rangle$ 
   $\langle \text{proof} \rangle$ 

```

```

instance complex :: ordered-comm-ring
   $\langle \text{proof} \rangle$ 

```

```

instance complex :: ordered-real-vector
   $\langle \text{proof} \rangle$ 

```



```
instance complex :: ordered-cancel-comm-semiring
  <proof>
```

```
end
```

116 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.

116.1 Preliminaries

```
lemma distinct-map-fst-filter:
  distinct (map fst xs)  $\implies$  distinct (map fst (List.filter P xs))
  <proof>
```

116.2 Type ('key, 'value) alist

```
typedef ('key, 'value) alist = {xs :: ('key  $\times$  'value) list. (distinct  $\circ$  map fst) xs}
morphisms impl-of Alist
  <proof>
```

```
setup-lifting type-definition-alist
```

```
lemma alist-ext: impl-of xs = impl-of ys  $\implies$  xs = ys
  <proof>
```

```
lemma alist-eq-iff: xs = ys  $\longleftrightarrow$  impl-of xs = impl-of ys
  <proof>
```

```
lemma impl-of-distinct [simp, intro]: distinct (map fst (impl-of xs))
  <proof>
```

```
lemma Alist-impl-of [code abstype]: Alist (impl-of xs) = xs
  <proof>
```

116.3 Primitive operations

```
lift-definition lookup :: ('key, 'value) alist  $\Rightarrow$  'key  $\Rightarrow$  'value option is map-of
  <proof>
```

```
lift-definition empty :: ('key, 'value) alist is []
  <proof>
```

lift-definition *update* :: 'key $\Rightarrow$ 'value $\Rightarrow$ ('key, 'value) alist $\Rightarrow$ ('key, 'value) alist
is *AList.update*
 ⟨proof⟩

lift-definition *delete* :: 'key $\Rightarrow$ ('key, 'value) alist $\Rightarrow$ ('key, 'value) alist
is *AList.delete*
 ⟨proof⟩

lift-definition *map-entry* ::
 'key $\Rightarrow$ ('value $\Rightarrow$ 'value) $\Rightarrow$ ('key, 'value) alist $\Rightarrow$ ('key, 'value) alist
is *AList.map-entry*
 ⟨proof⟩

lift-definition *filter* :: ('key $\times$ 'value $\Rightarrow$ bool) $\Rightarrow$ ('key, 'value) alist $\Rightarrow$ ('key, 'value) alist
is *List.filter*
 ⟨proof⟩

lift-definition *map-default* ::
 'key $\Rightarrow$ 'value $\Rightarrow$ ('value $\Rightarrow$ 'value) $\Rightarrow$ ('key, 'value) alist $\Rightarrow$ ('key, 'value) alist
is *AList.map-default*
 ⟨proof⟩

116.4 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 $\implies$ lookup (update k1 v xs) k2 = Some v
 ⟨proof⟩

lemma *lookup-update-neq* [simp]:
k1 $\neq$ k2 $\implies$ lookup (update k1 v xs) k2 = lookup xs k2
 ⟨proof⟩

lemma *update-update-eq* [simp]:
k1 = k2 $\implies$ 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⟩

116.5 Further operations

116.5.1 Equality

instantiation *alist* :: (*equal*, *equal*) *equal*
begin

definition *HOL.equal* (*xs* :: ('a, 'b) *alist*) *ys* == *impl-of xs = impl-of ys*

instance
 ⟨*proof*⟩

end

116.5.2 Size

instantiation *alist* :: (*type*, *type*) *size*
begin

definition *size* (*al* :: ('a, 'b) *alist*) = *length (impl-of al)*

instance ⟨*proof*⟩

end

116.6 Quickcheck generators

context
includes *state-combinator-syntax 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*

```

○→
  (λv. random-aux-alist (i - 1) j ○→ (λa. 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

```

117 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

```

118 Multisets partially implemented by association lists

```

theory DAList-Multiset
imports Multiset DAList
begin

  Delete preexisting code equations

  declare [[code drop: {#} Multiset.is-empty add-mset
    plus :: 'a multiset ⇒ - minus :: 'a multiset ⇒ -
    inter-mset union-mset image-mset filter-mset count
    size :: - multiset ⇒ nat sum-mset prod-mset
    set-mset sorted-list-of-multiset subset-mset subseteq-mset
    equal-multiset-inst.equal-multiset]]

  Raw operations on lists

  definition join-raw ::
    ('key ⇒ 'val × 'val ⇒ 'val) ⇒
    ('key × 'val) list ⇒ ('key × 'val) list ⇒ ('key × 'val) list

```

where $\text{join-raw } f \text{ } xs \text{ } ys = \text{foldr } (\lambda(k, v). \text{map-default } k \text{ } v \text{ } (\lambda v'. f \text{ } k \text{ } (v', v))) \text{ } ys \text{ } xs$

lemma join-raw-Nil $[simp]$: $\text{join-raw } f \text{ } xs \text{ } [] = xs$
 $\langle \text{proof} \rangle$

lemma join-raw-Cons $[simp]$:
 $\text{join-raw } f \text{ } xs \text{ } ((k, v) \# ys) = \text{map-default } k \text{ } v \text{ } (\lambda v'. f \text{ } k \text{ } (v', v)) \text{ } (\text{join-raw } f \text{ } xs \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma map-of-join-raw :
assumes $\text{distinct } (\text{map fst } ys)$
shows $\text{map-of } (\text{join-raw } f \text{ } xs \text{ } ys) \text{ } x =$
 $(\text{case map-of } xs \text{ } x \text{ of}$
 $\text{None} \Rightarrow \text{map-of } ys \text{ } x$
 $| \text{Some } v \Rightarrow (\text{case map-of } ys \text{ } x \text{ of None} \Rightarrow \text{Some } v | \text{Some } v' \Rightarrow \text{Some } (f \text{ } x \text{ } (v, v'))))$
 $\langle \text{proof} \rangle$

lemma distinct-join-raw :
assumes $\text{distinct } (\text{map fst } xs)$
shows $\text{distinct } (\text{map fst } (\text{join-raw } f \text{ } xs \text{ } ys))$
 $\langle \text{proof} \rangle$

definition $\text{subtract-entries-raw } xs \text{ } ys = \text{foldr } (\lambda(k, v). \text{AList.map-entry } k \text{ } (\lambda v'. v' - v)) \text{ } ys \text{ } xs$

lemma $\text{map-of-subtract-entries-raw}$:
assumes $\text{distinct } (\text{map fst } ys)$
shows $\text{map-of } (\text{subtract-entries-raw } xs \text{ } ys) \text{ } x =$
 $(\text{case map-of } xs \text{ } x \text{ of}$
 $\text{None} \Rightarrow \text{None}$
 $| \text{Some } v \Rightarrow (\text{case map-of } ys \text{ } x \text{ of None} \Rightarrow \text{Some } v | \text{Some } v' \Rightarrow \text{Some } (v - v')))$
 $\langle \text{proof} \rangle$

lemma $\text{distinct-subtract-entries-raw}$:
assumes $\text{distinct } (\text{map fst } xs)$
shows $\text{distinct } (\text{map fst } (\text{subtract-entries-raw } xs \text{ } ys))$
 $\langle \text{proof} \rangle$

Operations on alists with distinct keys

lift-definition $\text{join} :: ('a \Rightarrow 'b \times 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{ alist} \Rightarrow ('a, 'b) \text{ alist} \Rightarrow ('a, 'b) \text{ alist}$
is join-raw
 $\langle \text{proof} \rangle$

lift-definition $\text{subtract-entries} :: ('a, ('b :: \text{minus})) \text{ alist} \Rightarrow ('a, 'b) \text{ alist} \Rightarrow ('a, 'b) \text{ alist}$
is $\text{subtract-entries-raw}$
 $\langle \text{proof} \rangle$

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) = (λy. 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 (λx. 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 (λx (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 *Mempty-Bag* [code]: {#} = Bag (DAList.empty)
 ⟨proof⟩

lift-definition *is-empty-Bag-impl* :: ('a, nat) alist ⇒ bool is
 λxs. list-all (λx. snd x = 0) xs ⟨proof⟩

lemma *is-empty-Bag* [code]: $\text{Multiset.is-empty } (\text{Bag } xs) \longleftrightarrow \text{is-empty-Bag-impl } xs$
 ⟨proof⟩

lemma *union-Bag* [code]: $\text{Bag } xs + \text{Bag } ys = \text{Bag } (\text{join } (\lambda x \ (n1, n2). n1 + n2) \ xs \ ys)$
 ⟨proof⟩

lemma *add-mset-Bag* [code]: $\text{add-mset } x \ (\text{Bag } xs) =$
 $\text{Bag } (\text{join } (\lambda x \ (n1, n2). n1 + n2) \ (\text{DAList.update } x \ 1 \ \text{DAList.empty}) \ xs)$
 ⟨proof⟩

lemma *minus-Bag* [code]: $\text{Bag } xs - \text{Bag } ys = \text{Bag } (\text{subtract-entries } xs \ ys)$
 ⟨proof⟩

lemma *filter-Bag* [code]: $\text{filter-mset } P \ (\text{Bag } xs) = \text{Bag } (\text{DAList.filter } (P \circ \text{fst}) \ xs)$
 ⟨proof⟩

lemma *mset-eq* [code]: $\text{HOL.equal } (m1 :: 'a :: \text{equal multiset}) \ m2 \longleftrightarrow m1 \subseteq\# m2 \wedge m2 \subseteq\# m1$
 ⟨proof⟩

By default the code for $<$ is $(xs < ys) = (xs \leq ys \wedge xs \neq ys)$. With equality implemented by $\leq$, this leads to three calls of $\leq$. Here is a more efficient version:

lemma *mset-less*[code]: $xs \subset\# (ys :: 'a \text{ multiset}) \longleftrightarrow xs \subseteq\# ys \wedge \neg ys \subseteq\# xs$
 ⟨proof⟩

lemma *mset-less-eq-Bag0*:
 $\text{Bag } xs \subseteq\# A \longleftrightarrow (\forall (x, n) \in \text{set } (\text{DAList.impl-of } xs). \text{count-of } (\text{DAList.impl-of } xs) \ x \leq \text{count } A \ x)$
 (is ?lhs $\longleftrightarrow$?rhs)
 ⟨proof⟩

lemma *mset-less-eq-Bag* [code]:
 $\text{Bag } xs \subseteq\# (A :: 'a \text{ multiset}) \longleftrightarrow (\forall (x, n) \in \text{set } (\text{DAList.impl-of } xs). n \leq \text{count } A \ x)$
 ⟨proof⟩

declare *inter-mset-def* [code]
declare *union-mset-def* [code]
declare *mset.simps* [code]

fun *fold-impl* :: $('a \Rightarrow \text{nat} \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow ('a \times \text{nat}) \text{ list} \Rightarrow 'b$
where
 $\text{fold-impl } fn \ e \ ((a, n) \# ms) = (\text{fold-impl } fn \ ((fn \ a \ n) \ e) \ ms)$
 $\mid \text{fold-impl } fn \ e \ [] = e$

context
begin

qualified definition $fold :: ('a \Rightarrow nat \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow ('a, nat) \text{ alist} \Rightarrow 'b$
 where $fold\ f\ e\ al = fold\_impl\ f\ e\ (DAList.impl\_of\ al)$

end

context *comp-fun-commute*
begin

lemma *DAList-Multiset-fold*:
 assumes $fn: \bigwedge a\ n\ x. fn\ a\ n\ x = (f\ a\ \frown n)\ x$
 shows $fold\_mset\ f\ e\ (Bag\ al) = DAList-Multiset.fold\ fn\ e\ al$
 $\langle proof \rangle$

end

context
begin

private lift-definition $single\_alist\_entry :: 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ alist} \text{ is } \lambda a\ b. [(a,$
 $b)]$
 $\langle proof \rangle$

lemma *image-mset-Bag* [code]:
 $image\_mset\ f\ (Bag\ ms) =$
 $DAList-Multiset.fold\ (\lambda a\ n\ m. Bag\ (single\_alist\_entry\ (f\ a)\ n) + m)\ \{\#\}\ ms$
 $\langle proof \rangle$

end

— we cannot use $\lambda a\ n. (+)\ (a * n)$ for folding, since $(*)$ is not defined in *comm-monoid-add*

lemma *sum-mset-Bag* [code]: $sum\_mset\ (Bag\ ms) = DAList-Multiset.fold\ (\lambda a\ n.$
 $((+)\ a)\ \frown n)\ 0\ ms$
 $\langle proof \rangle$

lemma *prod-mset-Bag* [code]: $prod\_mset\ (Bag\ ms) = DAList-Multiset.fold\ (\lambda a\ n.$
 $((*)\ a)\ \frown n)\ 1\ ms$
 $\langle proof \rangle$

lemma *size-fold*: $size\ A = fold\_mset\ (\lambda -. Suc)\ 0\ A\ (\text{is } - = fold\_mset\ ?f\ -)$
 $\langle proof \rangle$

lemma *size-Bag* [code]: $size\ (Bag\ ms) = DAList-Multiset.fold\ (\lambda a\ n. (+)\ n)\ 0\ ms$
 $\langle proof \rangle$

lemma *set-mset-fold*: $set\_mset\ A = fold\_mset\ insert\ \{\}\ A\ (\text{is } - = fold\_mset\ ?f\ -)$
 $\langle proof \rangle$

```

lemma set-mset-Bag[code]:
  set-mset (Bag ms) = DAList-Multiset.fold ( $\lambda a\ n.$  (if  $n = 0$  then ( $\lambda m.$  m) else
insert a)) {} ms
  <proof>

```

```

instantiation multiset :: (exhaustive) exhaustive
begin

```

```

definition exhaustive-multiset ::
  ('a multiset  $\Rightarrow$  ( $\text{bool} \times \text{term list}$ ) option)  $\Rightarrow$  natural  $\Rightarrow$  ( $\text{bool} \times \text{term list}$ ) option
  where exhaustive-multiset f i = Quickcheck-Exhaustive.exhaustive ( $\lambda xs.$  f (Bag
xs)) i

```

```

instance <proof>

```

```

end

```

```

end

```

119 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.

119.1 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>

```

119.2 Tree properties

119.2.1 Content of a tree

```

primrec entries :: ('a, 'b) rbt  $\Rightarrow$  ('a  $\times$  'b) list
where
  entries Empty = []
  | entries (Branch - l k v r) = entries l @ (k,v) # entries r

```

abbreviation $(input)$ $entry-in-tree :: 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow bool$

where

$entry-in-tree\ k\ v\ t \equiv (k, v) \in set\ (entries\ t)$

definition $keys :: ('a, 'b) rbt \Rightarrow 'a\ list$ **where**

$keys\ t = map\ fst\ (entries\ t)$

lemma $keys-simps$ $[simp, code]$:

$keys\ Empty = []$

$keys\ (Branch\ c\ l\ k\ v\ r) = keys\ l @ k \# keys\ r$

$\langle proof \rangle$

lemma $entry-in-tree-keys$:

assumes $(k, v) \in set\ (entries\ t)$

shows $k \in set\ (keys\ t)$

$\langle proof \rangle$

lemma $keys-entries$:

$k \in set\ (keys\ t) \longleftrightarrow (\exists v. (k, v) \in set\ (entries\ t))$

$\langle proof \rangle$

lemma $non-empty-rbt-keys$:

$t \neq rbt.Empty \Longrightarrow keys\ t \neq []$

$\langle proof \rangle$

119.2.2 Search tree properties

context ord **begin**

definition $rbt-less :: 'a \Rightarrow ('a, 'b) rbt \Rightarrow bool$

where

$rbt-less-prop: rbt-less\ k\ t \longleftrightarrow (\forall x \in set\ (keys\ t). x < k)$

abbreviation $rbt-less-symbol$ (**infix** $| \ll 50$)

where $t | \ll x \equiv rbt-less\ x\ t$

definition $rbt-greater :: 'a \Rightarrow ('a, 'b) rbt \Rightarrow bool$ (**infix** $\ll | 50$)

where

$rbt-greater-prop: rbt-greater\ k\ t = (\forall x \in set\ (keys\ t). k < x)$

lemma $rbt-less-simps$ $[simp]$:

$Empty | \ll k = True$

$Branch\ c\ lt\ kt\ v\ rt | \ll k \longleftrightarrow kt < k \wedge lt | \ll k \wedge rt | \ll k$

$\langle proof \rangle$

lemma $rbt-greater-simps$ $[simp]$:

$k \ll | Empty = True$

$k \ll | (Branch\ c\ lt\ kt\ v\ rt) \longleftrightarrow k < kt \wedge k \ll | lt \wedge k \ll | rt$

$\langle \text{proof} \rangle$

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*: $l \mid\!\!\ll u \implies u \leq v \implies l \mid\!\!\ll v$

and *rbt-less-trans*: $t \mid\!\!\ll x \implies x < y \implies t \mid\!\!\ll y$

and *rbt-greater-eq-trans*: $u \leq v \implies v \ll\!\!\mid r \implies u \ll\!\!\mid r$

and *rbt-greater-trans*: $x < y \implies y \ll\!\!\mid t \implies x \ll\!\!\mid t$

$\langle \text{proof} \rangle$

primrec *rbt-sorted* :: $('a, 'b) \text{ rbt} \Rightarrow \text{bool}$

where

rbt-sorted *Empty* = *True*

$\mid \text{rbt-sorted } (\text{Branch } c \ l \ k \ v \ r) = (l \mid\!\!\ll k \wedge k \ll\!\!\mid r \wedge \text{rbt-sorted } l \wedge \text{rbt-sorted } r)$

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:

rbt-sorted $t \implies \text{List.sorted } (\text{map fst } (\text{entries } t))$

$\langle \text{proof} \rangle$

lemma *distinct-entries*:

rbt-sorted $t \implies \text{distinct } (\text{map fst } (\text{entries } t))$

$\langle \text{proof} \rangle$

lemma *distinct-keys*:

rbt-sorted $t \implies \text{distinct } (\text{keys } t)$

$\langle \text{proof} \rangle$

119.2.3 Tree lookup

primrec (*in ord*) *rbt-lookup* :: $('a, 'b) \text{ rbt} \Rightarrow 'a \rightarrow 'b$

where

rbt-lookup *Empty* $k = \text{None}$

$\mid \text{rbt-lookup } (\text{Branch } - \ l \ x \ y \ r) \ k =$

$(\text{if } k < x \text{ then } \text{rbt-lookup } l \ k \text{ else if } x < k \text{ then } \text{rbt-lookup } r \ k \text{ else } \text{Some } y)$

lemma *rbt-lookup-keys*: *rbt-sorted* $t \implies \text{dom } (\text{rbt-lookup } t) = \text{set } (\text{keys } t)$

$\langle \text{proof} \rangle$

lemma *dom-rbt-lookup-Branch*:

rbt-sorted $(\text{Branch } c \ t1 \ k \ v \ t2) \implies$

$\text{dom } (\text{rbt-lookup } (\text{Branch } c \ t1 \ k \ v \ t2))$

$= \text{Set.insert } k \ (\text{dom } (\text{rbt-lookup } t1) \cup \text{dom } (\text{rbt-lookup } t2))$
 $\langle \text{proof} \rangle$

lemma *finite-dom-rbt-lookup* [*simp*, *intro!*]: *finite* (*dom* (*rbt-lookup* *t*))
 $\langle \text{proof} \rangle$

end

context *ord* **begin**

lemma *rbt-lookup-rbt-less*[*simp*]: $t \mid \ll k \implies \text{rbt-lookup } t \ k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-greater*[*simp*]: $k \ll t \implies \text{rbt-lookup } t \ k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-Empty*: $\text{rbt-lookup } \text{Empty} = \text{Map.empty}$
 $\langle \text{proof} \rangle$

end

context *linorder* **begin**

lemma *map-of-entries*:
 $\text{rbt-sorted } t \implies \text{map-of } (\text{entries } t) = \text{rbt-lookup } t$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-in-tree*: $\text{rbt-sorted } t \implies \text{rbt-lookup } t \ k = \text{Some } v \longleftrightarrow (k, v) \in \text{set } (\text{entries } t)$
 $\langle \text{proof} \rangle$

lemma *set-entries-inject*:
assumes *rbt-sorted*: $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$
shows $\text{set } (\text{entries } t1) = \text{set } (\text{entries } t2) \longleftrightarrow \text{entries } t1 = \text{entries } t2$
 $\langle \text{proof} \rangle$

lemma *entries-eqI*:
assumes *rbt-sorted*: $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$
assumes *rbt-lookup*: $\text{rbt-lookup } t1 = \text{rbt-lookup } t2$
shows $\text{entries } t1 = \text{entries } t2$
 $\langle \text{proof} \rangle$

lemma *entries-rbt-lookup*:
assumes *rbt-sorted*: $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$
shows $\text{entries } t1 = \text{entries } t2 \longleftrightarrow \text{rbt-lookup } t1 = \text{rbt-lookup } t2$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-from-in-tree*:
assumes *rbt-sorted*: $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$

and $\bigwedge v. (k, v) \in \text{set } (\text{entries } t1) \longleftrightarrow (k, v) \in \text{set } (\text{entries } t2)$
 shows $\text{rbt-lookup } t1 \ k = \text{rbt-lookup } t2 \ k$
 $\langle \text{proof} \rangle$

end

119.2.4 Red-black properties

primrec $\text{color-of} :: ('a, 'b) \text{rbt} \Rightarrow \text{color}$

where

$\text{color-of } \text{Empty} = B$
 $| \text{color-of } (\text{Branch } c \ - \ - \ -) = c$

primrec $\text{bheight} :: ('a, 'b) \text{rbt} \Rightarrow \text{nat}$

where

$\text{bheight } \text{Empty} = 0$
 $| \text{bheight } (\text{Branch } c \ lt \ k \ v \ rt) = (\text{if } c = B \text{ then } \text{Suc } (\text{bheight } lt) \text{ else } \text{bheight } lt)$

primrec $\text{inv1} :: ('a, 'b) \text{rbt} \Rightarrow \text{bool}$

where

$\text{inv1 } \text{Empty} = \text{True}$
 $| \text{inv1 } (\text{Branch } c \ lt \ k \ v \ rt) \longleftrightarrow \text{inv1 } lt \wedge \text{inv1 } rt \wedge (c = B \vee \text{color-of } lt = B \wedge \text{color-of } rt = B)$

primrec $\text{inv1l} :: ('a, 'b) \text{rbt} \Rightarrow \text{bool}$ — Weaker version

where

$\text{inv1l } \text{Empty} = \text{True}$
 $| \text{inv1l } (\text{Branch } c \ l \ k \ v \ r) = (\text{inv1 } l \wedge \text{inv1 } r)$

lemma $[\text{simp}]: \text{inv1 } t \Longrightarrow \text{inv1l } t \langle \text{proof} \rangle$

primrec $\text{inv2} :: ('a, 'b) \text{rbt} \Rightarrow \text{bool}$

where

$\text{inv2 } \text{Empty} = \text{True}$
 $| \text{inv2 } (\text{Branch } c \ lt \ k \ v \ rt) = (\text{inv2 } lt \wedge \text{inv2 } rt \wedge \text{bheight } lt = \text{bheight } rt)$

context ord **begin**

definition $\text{is-rbt} :: ('a, 'b) \text{rbt} \Rightarrow \text{bool}$ **where**

$\text{is-rbt } t \longleftrightarrow \text{inv1 } t \wedge \text{inv2 } t \wedge \text{color-of } t = B \wedge \text{rbt-sorted } t$

lemma $\text{is-rbt-rbt-sorted} \ [\text{simp}]:$

$\text{is-rbt } t \Longrightarrow \text{rbt-sorted } t \langle \text{proof} \rangle$

theorem $\text{Empty-is-rbt} \ [\text{simp}]:$

$\text{is-rbt } \text{Empty} \langle \text{proof} \rangle$

end

119.3 Insertion

The function definitions are based on the book by Okasaki.

fun

$balance :: ('a, 'b) \text{ rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$

where

$balance \ (Branch \ R \ a \ w \ x \ b) \ s \ t \ (Branch \ R \ c \ y \ z \ d) = Branch \ R \ (Branch \ B \ a \ w \ x \ b) \ s \ t \ (Branch \ B \ c \ y \ z \ d) \mid$

$balance \ (Branch \ R \ (Branch \ R \ a \ w \ x \ b) \ s \ t \ c) \ y \ z \ d = Branch \ R \ (Branch \ B \ a \ w \ x \ b) \ s \ t \ (Branch \ B \ c \ y \ z \ d) \mid$

$balance \ (Branch \ R \ a \ w \ x \ (Branch \ R \ b \ s \ t \ c)) \ y \ z \ d = Branch \ R \ (Branch \ B \ a \ w \ x \ b) \ s \ t \ (Branch \ B \ c \ y \ z \ d) \mid$

$balance \ a \ w \ x \ (Branch \ R \ b \ s \ t \ (Branch \ R \ c \ y \ z \ d)) = Branch \ R \ (Branch \ B \ a \ w \ x \ b) \ s \ t \ (Branch \ B \ c \ y \ z \ d) \mid$

$balance \ a \ w \ x \ (Branch \ R \ (Branch \ R \ b \ s \ t \ c) \ y \ z \ d) = Branch \ R \ (Branch \ B \ a \ w \ x \ b) \ s \ t \ (Branch \ B \ c \ y \ z \ d) \mid$

$balance \ a \ s \ t \ b = Branch \ B \ a \ s \ t \ b$

lemma *balance-inv1*: $\llbracket inv1l \ l; \ inv1l \ r \rrbracket \Longrightarrow inv1 \ (balance \ l \ k \ v \ r)$

$\langle proof \rangle$

lemma *balance-bheight*: $bheight \ l = bheight \ r \Longrightarrow bheight \ (balance \ l \ k \ v \ r) = Suc \ (bheight \ l)$

$\langle proof \rangle$

lemma *balance-inv2*:

assumes $inv2 \ l \ inv2 \ r \ bheight \ l = bheight \ r$

shows $inv2 \ (balance \ l \ k \ v \ r)$

$\langle proof \rangle$

context *ord* **begin**

lemma *balance-rbt-greater[simp]*: $(v \ll \mid balance \ a \ k \ x \ b) = (v \ll \mid a \wedge v \ll \mid b \wedge v < k)$

$\langle proof \rangle$

lemma *balance-rbt-less[simp]*: $(balance \ a \ k \ x \ b \mid \ll v) = (a \mid \ll v \wedge b \mid \ll v \wedge k < v)$

$\langle proof \rangle$

end

lemma (**in** *linorder*) *balance-rbt-sorted*:

fixes $k :: 'a$

assumes $rbt\text{-sorted} \ l \ rbt\text{-sorted} \ r \ l \mid \ll k \ k \mid \ll r$

shows $rbt\text{-sorted} \ (balance \ l \ k \ v \ r)$

$\langle proof \rangle$

lemma *entries-balance* [simp]:

$entries \ (balance \ l \ k \ v \ r) = entries \ l \ @ \ (k, v) \ # \ entries \ r$

$\langle \text{proof} \rangle$

lemma *keys-balance* [simp]:

$\text{keys } (\text{balance } l \ k \ v \ r) = \text{keys } l \ @ \ k \ \# \ \text{keys } r$

$\langle \text{proof} \rangle$

lemma *balance-in-tree*:

$\text{entry-in-tree } k \ x \ (\text{balance } l \ v \ y \ r) \longleftrightarrow \text{entry-in-tree } k \ x \ l \vee k = v \wedge x = y \vee$
 $\text{entry-in-tree } k \ x \ r$

$\langle \text{proof} \rangle$

lemma (in *linorder*) *rbt-lookup-balance*[simp]:

fixes $k :: 'a$

assumes *rbt-sorted* l *rbt-sorted* r $l \mid \ll k \mid \ll r$

shows *rbt-lookup* $(\text{balance } l \ k \ v \ r) \ x = \text{rbt-lookup } (\text{Branch } B \ l \ k \ v \ r) \ x$

$\langle \text{proof} \rangle$

primrec *paint* :: $\text{color} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

where

$\text{paint } c \ \text{Empty} = \text{Empty}$

$\mid \text{paint } c \ (\text{Branch } l \ k \ v \ r) = \text{Branch } c \ l \ k \ v \ r$

lemma *paint-inv1l*[simp]: $\text{inv1l } t \Longrightarrow \text{inv1l } (\text{paint } c \ t) \langle \text{proof} \rangle$

lemma *paint-inv1*[simp]: $\text{inv1l } t \Longrightarrow \text{inv1 } (\text{paint } B \ t) \langle \text{proof} \rangle$

lemma *paint-inv2*[simp]: $\text{inv2 } t \Longrightarrow \text{inv2 } (\text{paint } c \ t) \langle \text{proof} \rangle$

lemma *paint-color-of*[simp]: $\text{color-of } (\text{paint } B \ t) = B \langle \text{proof} \rangle$

lemma *paint-in-tree*[simp]: $\text{entry-in-tree } k \ x \ (\text{paint } c \ t) = \text{entry-in-tree } k \ x \ t \langle \text{proof} \rangle$

context *ord* **begin**

lemma *paint-rbt-sorted*[simp]: $\text{rbt-sorted } t \Longrightarrow \text{rbt-sorted } (\text{paint } c \ t) \langle \text{proof} \rangle$

lemma *paint-rbt-lookup*[simp]: $\text{rbt-lookup } (\text{paint } c \ t) = \text{rbt-lookup } t \langle \text{proof} \rangle$

lemma *paint-rbt-greater*[simp]: $(v \ll \text{paint } c \ t) = (v \ll t) \langle \text{proof} \rangle$

lemma *paint-rbt-less*[simp]: $(\text{paint } c \ t \mid \ll v) = (t \mid \ll v) \langle \text{proof} \rangle$

fun

$\text{rbt-ins} :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

where

$\text{rbt-ins } f \ k \ v \ \text{Empty} = \text{Branch } R \ \text{Empty } k \ v \ \text{Empty} \mid$

$\text{rbt-ins } f \ k \ v \ (\text{Branch } B \ l \ x \ y \ r) = (\text{if } k < x \text{ then } \text{balance } (\text{rbt-ins } f \ k \ v \ l) \ x \ y \ r$
 $\text{else if } k > x \text{ then } \text{balance } l \ x \ y \ (\text{rbt-ins } f \ k \ v \ r)$
 $\text{else } \text{Branch } B \ l \ x \ (f \ k \ y \ v) \ r) \mid$

$\text{rbt-ins } f \ k \ v \ (\text{Branch } R \ l \ x \ y \ r) = (\text{if } k < x \text{ then } \text{Branch } R \ (\text{rbt-ins } f \ k \ v \ l) \ x \ y \ r$
 $\text{else if } k > x \text{ then } \text{Branch } R \ l \ x \ y \ (\text{rbt-ins } f \ k \ v \ r)$
 $\text{else } \text{Branch } R \ l \ x \ (f \ k \ y \ v) \ r)$

lemma *ins-inv1-inv2*:

assumes *inv1* t *inv2* t

shows *inv2* $(\text{rbt-ins } f \ k \ x \ t) \ \text{bheight } (\text{rbt-ins } f \ k \ x \ t) = \text{bheight } t$

color-of $t = B \implies \text{inv1 } (\text{rbt-ins } f \ k \ x \ t) \ \text{inv1l } (\text{rbt-ins } f \ k \ x \ t)$
 ⟨proof⟩

end

context *linorder* **begin**

lemma *ins-rbt-greater[simp]*: $(v \ll \text{rbt-ins } f \ (k :: 'a) \ x \ t) = (v \ll t \wedge k > v)$
 ⟨proof⟩

lemma *ins-rbt-less[simp]*: $(\text{rbt-ins } f \ k \ x \ t \mid \ll v) = (t \mid \ll v \wedge k < v)$
 ⟨proof⟩

lemma *ins-rbt-sorted[simp]*: $\text{rbt-sorted } t \implies \text{rbt-sorted } (\text{rbt-ins } f \ k \ x \ t)$
 ⟨proof⟩

lemma *keys-ins*: $\text{set } (\text{keys } (\text{rbt-ins } f \ k \ v \ t)) = \{ k \} \cup \text{set } (\text{keys } t)$
 ⟨proof⟩

lemma *rbt-lookup-ins*:

fixes $k :: 'a$

assumes *rbt-sorted* t

shows $\text{rbt-lookup } (\text{rbt-ins } f \ k \ v \ t) \ x = ((\text{rbt-lookup } t)(k \mid \rightarrow \text{case } \text{rbt-lookup } t \ k$
of None $\Rightarrow v$

$\mid \text{Some } w \Rightarrow f \ k \ w \ v)) \ x$

⟨proof⟩

end

context *ord* **begin**

definition *rbt-insert-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow$
 $('a, 'b) \text{rbt}$

where *rbt-insert-with-key* $f \ k \ v \ t = \text{paint } B \ (\text{rbt-ins } f \ k \ v \ t)$

definition *rbt-insertw-def*: *rbt-insert-with* $f = \text{rbt-insert-with-key } (\lambda \cdot. f)$

definition *rbt-insert* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**

rbt-insert $= \text{rbt-insert-with-key } (\lambda \cdot. \text{nv. nv})$

end

context *linorder* **begin**

lemma *rbt-insertwk-rbt-sorted*: $\text{rbt-sorted } t \implies \text{rbt-sorted } (\text{rbt-insert-with-key } f \ (k$
 $:: 'a) \ x \ t)$
 ⟨proof⟩

theorem *rbt-insertwk-is-rbt*:

assumes *inv*: *is-rbt* t

shows *is-rbt* $(\text{rbt-insert-with-key } f \ k \ x \ t)$

<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

119.4 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 $inv2\ (balance-left\ lt\ k\ v\ rt)$

$\langle proof \rangle$

lemma *balance-left-inv2-app*:

assumes $inv2\ lt\ inv2\ rt\ bheight\ lt + 1 = bheight\ rt\ color-of\ rt = B$

shows $inv2\ (balance-left\ lt\ k\ v\ rt)$

$bheight\ (balance-left\ lt\ k\ v\ rt) = bheight\ rt$

$\langle proof \rangle$

lemma *balance-left-inv1*: $\llbracket inv1\ l\ a; inv1\ b; color-of\ b = B \rrbracket \implies inv1\ (balance-left\ a\ k\ x\ b)$

$\langle proof \rangle$

lemma *balance-left-inv1l*: $\llbracket inv1\ l\ lt; inv1\ rt \rrbracket \implies inv1\ (balance-left\ lt\ k\ x\ rt)$

$\langle proof \rangle$

lemma (in *linorder*) *balance-left-rbt-sorted*:

$\llbracket rbt-sorted\ l; rbt-sorted\ r; rbt-less\ k\ l; k \ll r \rrbracket \implies rbt-sorted\ (balance-left\ l\ k\ v\ r)$

$\langle proof \rangle$

context *order* **begin**

lemma *balance-left-rbt-greater*:

fixes $k :: 'a$

assumes $k \ll a\ k \ll b\ k < x$

shows $k \ll balance-left\ a\ x\ t\ b$

$\langle proof \rangle$

lemma *balance-left-rbt-less*:

fixes $k :: 'a$

assumes $a \ll k\ b \ll k\ x < k$

shows $balance-left\ a\ x\ t\ b \ll k$

$\langle proof \rangle$

end

lemma *balance-left-in-tree*:

assumes $inv1\ l\ inv1\ r\ bheight\ l + 1 = bheight\ r$

shows $entry-in-tree\ k\ v\ (balance-left\ l\ a\ b\ r) = (entry-in-tree\ k\ v\ l \vee k = a \wedge v = b \vee entry-in-tree\ k\ v\ r)$

$\langle proof \rangle$

fun

balance-right :: $('a, 'b)\ rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

where

$\text{balance-right } a \ k \ x \ (\text{Branch } R \ b \ s \ y \ c) = \text{Branch } R \ a \ k \ x \ (\text{Branch } B \ b \ s \ y \ c) \mid$
 $\text{balance-right } (\text{Branch } B \ a \ k \ x \ b) \ s \ y \ bl = \text{balance } (\text{Branch } R \ a \ k \ x \ b) \ s \ y \ bl \mid$
 $\text{balance-right } (\text{Branch } R \ a \ k \ x \ (\text{Branch } B \ b \ s \ y \ c)) \ t \ z \ bl = \text{Branch } R \ (\text{balance}$
 $(\text{paint } R \ a) \ k \ x \ b) \ s \ y \ (\text{Branch } B \ c \ t \ z \ bl) \mid$
 $\text{balance-right } t \ k \ x \ s = \text{Empty}$

lemma *balance-right-inv2-with-inv1*:

assumes $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt = \text{bheight } rt + 1 \ \text{inv1 } lt$

shows $\text{inv2 } (\text{balance-right } lt \ k \ v \ rt) \wedge \text{bheight } (\text{balance-right } lt \ k \ v \ rt) = \text{bheight } lt$
 $\langle \text{proof} \rangle$

lemma *balance-right-inv1*: $\llbracket \text{inv1 } a; \text{inv1l } b; \text{color-of } a = B \rrbracket \Longrightarrow \text{inv1 } (\text{balance-right } a \ k \ x \ b)$
 $\langle \text{proof} \rangle$

lemma *balance-right-inv1l*: $\llbracket \text{inv1 } lt; \text{inv1l } rt \rrbracket \Longrightarrow \text{inv1l } (\text{balance-right } lt \ k \ x \ rt)$
 $\langle \text{proof} \rangle$

lemma (in *linorder*) *balance-right-rbt-sorted*:

$\llbracket \text{rbt-sorted } l; \text{rbt-sorted } r; \text{rbt-less } k \ l; k \ll r \rrbracket \Longrightarrow \text{rbt-sorted } (\text{balance-right } l \ k \ v \ r)$
 $\langle \text{proof} \rangle$

context *order* **begin**

lemma *balance-right-rbt-greater*:

fixes $k :: 'a$

assumes $k \ll a \ k \ll b \ k < x$

shows $k \ll \text{balance-right } a \ x \ t \ b$
 $\langle \text{proof} \rangle$

lemma *balance-right-rbt-less*:

fixes $k :: 'a$

assumes $a \ll k \ b \ll k \ x < k$

shows $\text{balance-right } a \ x \ t \ b \ll k$
 $\langle \text{proof} \rangle$

end

lemma *balance-right-in-tree*:

assumes $\text{inv1 } l \ \text{inv1l } r \ \text{bheight } l = \text{bheight } r + 1 \ \text{inv2 } l \ \text{inv2 } r$

shows $\text{entry-in-tree } x \ y \ (\text{balance-right } l \ k \ v \ r) = (\text{entry-in-tree } x \ y \ l \vee x = k \wedge y = v \vee \text{entry-in-tree } x \ y \ r)$
 $\langle \text{proof} \rangle$

fun

$\text{combine} :: ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

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)*

$\langle \text{proof} \rangle$

context *ord* **begin**

fun

rbt-del-from-left :: $'a \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **and**
rbt-del-from-right :: $'a \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **and**
rbt-del :: $'a \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$

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

$\llbracket \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; \text{inv1 } rt \rrbracket \implies$
 $\text{inv2 } (\text{rbt-del-from-left } x \text{ lt k v rt}) \wedge$
 $\text{bheight } (\text{rbt-del-from-left } x \text{ lt k v rt}) = \text{bheight } lt \wedge$
 $(\text{color-of } lt = B \wedge \text{color-of } rt = B \wedge \text{inv1 } (\text{rbt-del-from-left } x \text{ lt k v rt}) \vee$
 $(\text{color-of } lt \neq B \vee \text{color-of } rt \neq B) \wedge \text{inv1l } (\text{rbt-del-from-left } x \text{ lt k v rt}))$
and $\llbracket \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; \text{inv1 } rt \rrbracket \implies$
 $\text{inv2 } (\text{rbt-del-from-right } x \text{ lt k v rt}) \wedge$
 $\text{bheight } (\text{rbt-del-from-right } x \text{ lt k v rt}) = \text{bheight } lt \wedge$
 $(\text{color-of } lt = B \wedge \text{color-of } rt = B \wedge \text{inv1 } (\text{rbt-del-from-right } x \text{ lt k v rt}) \vee$
 $(\text{color-of } lt \neq B \vee \text{color-of } rt \neq B) \wedge \text{inv1l } (\text{rbt-del-from-right } x \text{ lt k v rt}))$
and *rbt-del-inv1-inv2*: $\text{inv2 } (\text{rbt-del } x \text{ lt}) \wedge (\text{color-of } lt = R \wedge \text{bheight } (\text{rbt-del } x$
 $lt) = \text{bheight } lt \wedge \text{inv1 } (\text{rbt-del } x \text{ lt})$
 $\vee \text{color-of } lt = B \wedge \text{bheight } (\text{rbt-del } x \text{ lt}) = \text{bheight } lt - 1 \wedge \text{inv1l } (\text{rbt-del } x \text{ lt}))$
 $\langle \text{proof} \rangle$

lemma

rbt-del-from-left-rbt-less: $\llbracket lt \mid\!\!\!< v; rt \mid\!\!\!< v; k < v \rrbracket \implies \text{rbt-del-from-left } x \text{ lt k y rt}$
 $\mid\!\!\!< v$
and *rbt-del-from-right-rbt-less*: $\llbracket lt \mid\!\!\!< v; rt \mid\!\!\!< v; k < v \rrbracket \implies \text{rbt-del-from-right } x$
 $lt k y rt \mid\!\!\!< v$
and *rbt-del-rbt-less*: $lt \mid\!\!\!< v \implies \text{rbt-del } x \text{ lt } \mid\!\!\!< v$
 $\langle \text{proof} \rangle$

lemma *rbt-del-from-left-rbt-greater*: $\llbracket v \ll lt; v \ll rt; k > v \rrbracket \implies v \ll \text{rbt-del-from-left } x \text{ } lt \text{ } k \text{ } y \text{ } rt$
and *rbt-del-from-right-rbt-greater*: $\llbracket v \ll lt; v \ll rt; k > v \rrbracket \implies v \ll \text{rbt-del-from-right } x \text{ } lt \text{ } k \text{ } y \text{ } rt$
and *rbt-del-rbt-greater*: $v \ll lt \implies v \ll \text{rbt-del } x \text{ } lt$
 $\langle \text{proof} \rangle$

lemma $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll k; k \mid rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-left } x \text{ } lt \text{ } k \text{ } y \text{ } rt)$
and $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll k; k \mid rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-right } x \text{ } lt \text{ } k \text{ } y \text{ } rt)$
and *rbt-del-rbt-sorted*: $\text{rbt-sorted } lt \implies \text{rbt-sorted } (\text{rbt-del } x \text{ } lt)$
 $\langle \text{proof} \rangle$

lemma $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll kt; kt \mid rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; x < kt \rrbracket \implies \text{entry-in-tree } k \text{ } v \text{ } (\text{rbt-del-from-left } x \text{ } lt \text{ } kt \text{ } y \text{ } rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } (\text{Branch } c \text{ } lt \text{ } kt \text{ } y \text{ } rt)))$
and $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll kt; kt \mid rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; x > kt \rrbracket \implies \text{entry-in-tree } k \text{ } v \text{ } (\text{rbt-del-from-right } x \text{ } lt \text{ } kt \text{ } y \text{ } rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } (\text{Branch } c \text{ } lt \text{ } kt \text{ } y \text{ } rt)))$
and *rbt-del-in-tree*: $\llbracket \text{rbt-sorted } t; \text{inv1 } t; \text{inv2 } t \rrbracket \implies \text{entry-in-tree } k \text{ } v \text{ } (\text{rbt-del } x \text{ } t) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } t))$
 $\langle \text{proof} \rangle$

definition (*in ord*) *rbt-delete* **where**
 $\text{rbt-delete } k \text{ } t = \text{paint } B \text{ } (\text{rbt-del } k \text{ } t)$

theorem *rbt-delete-is-rbt* [*simp*]: **assumes** *is-rbt* *t* **shows** *is-rbt* $(\text{rbt-delete } k \text{ } t)$
 $\langle \text{proof} \rangle$

lemma *rbt-delete-in-tree*:
assumes *is-rbt* *t*
shows $\text{entry-in-tree } k \text{ } v \text{ } (\text{rbt-delete } x \text{ } t) = (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } t)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-delete*:
assumes *is-rbt*: *is-rbt* *t*
shows $\text{rbt-lookup } (\text{rbt-delete } k \text{ } t) = (\text{rbt-lookup } t) \mid (-\{k\})$
 $\langle \text{proof} \rangle$

end

119.5 Modifying existing entries

context *ord* **begin**

primrec
 $\text{rbt-map-entry} :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

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>

```

119.6 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* ($\lambda(k, v). (k, f k v)$) (*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*

119.7 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
)

119.8 Bulkloading a tree

definition (**in** *ord*) *rbt-bulkload* :: (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* **where**
rbt-bulkload *xs* = *foldr* ($\lambda(k, v). \text{rbt-insert } k \ v$) *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

119.9 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* $\Rightarrow$ (*'a* $\times$ *'b*) *list* $\Rightarrow$ (*'a*, *'b*) *rbt* $\times$ (*'a* $\times$ *'b*) *list*
and *rbtreeify-g* :: *nat* $\Rightarrow$ (*'a* $\times$ *'b*) *list* $\Rightarrow$ (*'a*, *'b*) *rbt* $\times$ (*'a* $\times$ *'b*) *list*
where

rbtreeify-f *n* *kvs* =
 (if *n* = 0 then (*Empty*, *kvs*)
 else if *n* = 1 then
 case *kvs* of (*k*, *v*) # *kvs'* $\Rightarrow$ (*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'*) $\Rightarrow$
 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'*) $\Rightarrow$
 apfst (*Branch B t1 k v*) (*rbtreeify-f* (*n* div 2) *kvs'*))

| *rbtreeify-g* *n* *kvs* =
 (if *n* = 0 $\vee$ *n* = 1 then (*Empty*, *kvs*)
 else if *n* mod 2 = 0 then
 case *rbtreeify-g* (*n* div 2) *kvs* of (*t1*, (*k*, *v*) # *kvs'*) $\Rightarrow$
 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'*) $\Rightarrow$
 apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n* div 2) *kvs'*))

definition *rbtreeify* :: (*'a* $\times$ *'b*) *list* $\Rightarrow$ (*'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'* $\Rightarrow$
 (*Branch R Empty k v Empty*, *kvs'*)
 else let (*n'*, *r*) = *Divides.divmod-nat* *n* 2 in
 if *r* = 0 then
 case *rbtreeify-f* *n'* *kvs* of (*t1*, (*k*, *v*) # *kvs'*) $\Rightarrow$
 apfst (*Branch B t1 k v*) (*rbtreeify-g* *n'* *kvs'*)
 else case *rbtreeify-f* *n'* *kvs* of (*t1*, (*k*, *v*) # *kvs'*) $\Rightarrow$
 apfst (*Branch B t1 k v*) (*rbtreeify-f* *n'* *kvs'*))

$\langle \text{proof} \rangle$

lemma *rbtreeify-g-code* [code]:

```
rbtreeify-g n kvs =
  (if n = 0  $\vee$  n = 1 then (Empty, kvs)
   else let (n', r) = Divides.divmod-nat n 2 in
     if r = 0 then
       case rbtreeify-g n' kvs of (t1, (k, v) # kvs')  $\Rightarrow$ 
         apfst (Branch B t1 k v) (rbtreeify-g n' kvs')
     else case rbtreeify-f n' kvs of (t1, (k, v) # kvs')  $\Rightarrow$ 
       apfst (Branch B t1 k v) (rbtreeify-g n' kvs'))
```

$\langle \text{proof} \rangle$

lemma *Suc-double-half*: $\text{Suc } (2 * n) \text{ div } 2 = n$

$\langle \text{proof} \rangle$

lemma *div2-plus-div2*: $n \text{ div } 2 + n \text{ div } 2 = (n :: \text{nat}) - n \text{ mod } 2$

$\langle \text{proof} \rangle$

lemma *rbtreeify-f-rec-aux-lemma*:

```
 $\llbracket k - n \text{ div } 2 = \text{Suc } k'; n \leq k; n \text{ mod } 2 = \text{Suc } 0 \rrbracket$ 
 $\implies k' - n \text{ div } 2 = k - n$ 
```

$\langle \text{proof} \rangle$

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  $\implies$  rbtreeify-f (2 * n) kvs =
  (case rbtreeify-f n kvs of (t1, (k, v) # kvs')  $\Rightarrow$ 
    apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
0 < n  $\implies$  rbtreeify-f (Suc (2 * n)) kvs =
  (case rbtreeify-f n kvs of (t1, (k, v) # kvs')  $\Rightarrow$ 
    apfst (Branch B t1 k v) (rbtreeify-f n kvs'))
```

$\langle \text{proof} \rangle$

lemma *rbtreeify-g-simps*:

```
rbtreeify-g 0 kvs = (Empty, kvs)
rbtreeify-g (Suc 0) kvs = (Empty, kvs)
0 < n  $\implies$  rbtreeify-g (2 * n) kvs =
  (case rbtreeify-g n kvs of (t1, (k, v) # kvs')  $\Rightarrow$ 
    apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
0 < n  $\implies$  rbtreeify-g (Suc (2 * n)) kvs =
  (case rbtreeify-f n kvs of (t1, (k, v) # kvs')  $\Rightarrow$ 
    apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
```

$\langle \text{proof} \rangle$

declare *rbtreeify-f-simps*[simp] *rbtreeify-g-simps*[simp]

lemma *length-rbtreeify-f*: $n \leq \text{length } kvs$
 $\implies \text{length } (\text{snd } (\text{rbtreeify-f } n \ kvs)) = \text{length } kvs - n$
and *length-rbtreeify-g*: $\llbracket 0 < n; n \leq \text{Suc } (\text{length } kvs) \rrbracket$
 $\implies \text{length } (\text{snd } (\text{rbtreeify-g } n \ kvs)) = \text{Suc } (\text{length } kvs) - n$
 <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 $f0 == (\bigwedge kvs. P \ 0 \ kvs)$
and $f1 == (\bigwedge k \ v \ kvs. P \ (\text{Suc } 0) \ ((k, v) \# \ kvs))$
and $feven ==$
 $(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$
 $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$
 $\implies P \ (2 * n) \ kvs)$
and $fodd ==$
 $(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$
 $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{length } kvs'; P \ n \ kvs' \rrbracket$
 $\implies P \ (\text{Suc } (2 * n)) \ kvs)$
and $g0 == (\bigwedge kvs. Q \ 0 \ kvs)$
and $g1 == (\bigwedge kvs. Q \ (\text{Suc } 0) \ kvs)$
and $geven ==$
 $(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{Suc } (\text{length } kvs); Q \ n \ kvs;$
 $\text{rbtreeify-g } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$
 $\implies Q \ (2 * n) \ kvs)$
and $godd ==$
 $(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$
 $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$
 $\implies Q \ (\text{Suc } (2 * n)) \ kvs)$
shows $\llbracket n \leq \text{length } kvs;$
 $\text{PROP } f0; \text{PROP } f1; \text{PROP } feven; \text{PROP } fodd;$
 $\text{PROP } g0; \text{PROP } g1; \text{PROP } geven; \text{PROP } godd \rrbracket$
 $\implies P \ n \ kvs$
and $\llbracket n \leq \text{Suc } (\text{length } kvs);$
 $\text{PROP } f0; \text{PROP } f1; \text{PROP } feven; \text{PROP } fodd;$
 $\text{PROP } g0; \text{PROP } g1; \text{PROP } geven; \text{PROP } godd \rrbracket$
 $\implies Q \ n \ kvs$
 <proof>

lemma *inv1-rbtreeify-f*: $n \leq \text{length } kvs$
 $\implies \text{inv1 } (\text{fst } (\text{rbtreeify-f } n \ kvs))$
and *inv1-rbtreeify-g*: $n \leq \text{Suc } (\text{length } kvs)$
 $\implies \text{inv1 } (\text{fst } (\text{rbtreeify-g } n \ kvs))$
 <proof>

fun *plog2* :: $\text{nat} \Rightarrow \text{nat}$
where $\text{plog2 } n = (\text{if } n \leq 1 \text{ then } 0 \text{ else } \text{plog2 } (n \text{ div } 2) + 1)$

declare *plog2.simps* [simp del]

lemma *plog2-simps* [*simp*]:
 $plog2\ 0 = 0$ $plog2\ (Suc\ 0) = 0$
 $0 < n \implies plog2\ (2 * n) = 1 + plog2\ n$
 $0 < n \implies plog2\ (Suc\ (2 * n)) = 1 + plog2\ n$
 $\langle proof \rangle$

lemma *bheight-rbtreeify-f*: $n \leq length\ kvs$
 $\implies bheight\ (fst\ (rbtreeify-f\ n\ kvs)) = plog2\ n$
and *bheight-rbtreeify-g*: $n \leq Suc\ (length\ kvs)$
 $\implies bheight\ (fst\ (rbtreeify-g\ n\ kvs)) = plog2\ n$
 $\langle proof \rangle$

lemma *bheight-rbtreeify-f-eq-plog2I*:
 $\llbracket rbtreeify-f\ n\ kvs = (t, kvs');\ n \leq length\ kvs \rrbracket$
 $\implies bheight\ t = plog2\ n$
 $\langle proof \rangle$

lemma *bheight-rbtreeify-g-eq-plog2I*:
 $\llbracket rbtreeify-g\ n\ kvs = (t, kvs');\ n \leq Suc\ (length\ kvs) \rrbracket$
 $\implies bheight\ t = plog2\ n$
 $\langle proof \rangle$

hide-const (**open**) *plog2*

lemma *inv2-rbtreeify-f*: $n \leq length\ kvs$
 $\implies inv2\ (fst\ (rbtreeify-f\ n\ kvs))$
and *inv2-rbtreeify-g*: $n \leq Suc\ (length\ kvs)$
 $\implies inv2\ (fst\ (rbtreeify-g\ n\ kvs))$
 $\langle proof \rangle$

lemma $n \leq length\ kvs \implies True$
and *color-of-rbtreeify-g*:
 $\llbracket n \leq Suc\ (length\ kvs);\ 0 < n \rrbracket$
 $\implies color-of\ (fst\ (rbtreeify-g\ n\ kvs)) = B$
 $\langle proof \rangle$

lemma *entries-rbtreeify-f-append*:
 $n \leq length\ kvs$
 $\implies entries\ (fst\ (rbtreeify-f\ n\ kvs))\ @\ snd\ (rbtreeify-f\ n\ kvs) = kvs$
and *entries-rbtreeify-g-append*:
 $n \leq Suc\ (length\ kvs)$
 $\implies entries\ (fst\ (rbtreeify-g\ n\ kvs))\ @\ snd\ (rbtreeify-g\ n\ kvs) = kvs$
 $\langle proof \rangle$

lemma *length-entries-rbtreeify-f*:
 $n \leq length\ kvs \implies length\ (entries\ (fst\ (rbtreeify-f\ n\ kvs))) = n$
and *length-entries-rbtreeify-g*:
 $n \leq Suc\ (length\ kvs) \implies length\ (entries\ (fst\ (rbtreeify-g\ n\ kvs))) = n - 1$

$\langle \text{proof} \rangle$

lemma *rbtreeify-f-conv-drop*:

$$n \leq \text{length } kvs \implies \text{snd } (\text{rbtreeify-f } n \text{ } kvs) = \text{drop } n \text{ } kvs$$

$\langle \text{proof} \rangle$

lemma *rbtreeify-g-conv-drop*:

$$n \leq \text{Suc } (\text{length } kvs) \implies \text{snd } (\text{rbtreeify-g } n \text{ } kvs) = \text{drop } (n - 1) \text{ } kvs$$

$\langle \text{proof} \rangle$

lemma *entries-rbtreeify-f [simp]*:

$$n \leq \text{length } kvs \implies \text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) = \text{take } n \text{ } kvs$$

$\langle \text{proof} \rangle$

lemma *entries-rbtreeify-g [simp]*:

$$n \leq \text{Suc } (\text{length } kvs) \implies \text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = \text{take } (n - 1) \text{ } kvs$$

$\langle \text{proof} \rangle$

lemma *keys-rbtreeify-f [simp]*: $n \leq \text{length } kvs$

$$\implies \text{keys } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) = \text{take } n \text{ } (\text{map } \text{fst } kvs)$$

$\langle \text{proof} \rangle$

lemma *keys-rbtreeify-g [simp]*: $n \leq \text{Suc } (\text{length } kvs)$

$$\implies \text{keys } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = \text{take } (n - 1) \text{ } (\text{map } \text{fst } kvs)$$

$\langle \text{proof} \rangle$

lemma *rbtreeify-fD*:

$$\llbracket \text{rbtreeify-f } n \text{ } kvs = (t, kvs'); n \leq \text{length } kvs \rrbracket$$

$$\implies \text{entries } t = \text{take } n \text{ } kvs \wedge kvs' = \text{drop } n \text{ } kvs$$

$\langle \text{proof} \rangle$

lemma *rbtreeify-gD*:

$$\llbracket \text{rbtreeify-g } n \text{ } kvs = (t, kvs'); n \leq \text{Suc } (\text{length } kvs) \rrbracket$$

$$\implies \text{entries } t = \text{take } (n - 1) \text{ } kvs \wedge kvs' = \text{drop } (n - 1) \text{ } kvs$$

$\langle \text{proof} \rangle$

lemma *entries-rbtreeify [simp]*: $\text{entries } (\text{rbtreeify } kvs) = kvs$

$\langle \text{proof} \rangle$

context *linorder begin*

lemma *rbt-sorted-rbtreeify-f*:

$$\llbracket n \leq \text{length } kvs; \text{sorted } (\text{map } \text{fst } kvs); \text{distinct } (\text{map } \text{fst } kvs) \rrbracket$$

$$\implies \text{rbt-sorted } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs))$$

and *rbt-sorted-rbtreeify-g*:

$$\llbracket n \leq \text{Suc } (\text{length } kvs); \text{sorted } (\text{map } \text{fst } kvs); \text{distinct } (\text{map } \text{fst } kvs) \rrbracket$$

$$\implies \text{rbt-sorted } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs))$$

$\langle \text{proof} \rangle$

lemma *rbt-sorted-rbtreeify*:

$\llbracket \text{sorted } (\text{map fst kvs}); \text{distinct } (\text{map fst kvs}) \rrbracket \implies \text{rbt-sorted } (\text{rbtreeify kvs})$
 $\langle \text{proof} \rangle$

lemma *is-rbt-rbtreeify*:

$\llbracket \text{sorted } (\text{map fst kvs}); \text{distinct } (\text{map fst kvs}) \rrbracket$
 $\implies \text{is-rbt } (\text{rbtreeify kvs})$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbtreeify*:

$\llbracket \text{sorted } (\text{map fst kvs}); \text{distinct } (\text{map fst kvs}) \rrbracket \implies$
 $\text{rbt-lookup } (\text{rbtreeify kvs}) = \text{map-of kvs}$
 $\langle \text{proof} \rangle$

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-red (Branch color.R l k v r) = l
| *skip-red* t = t

definition *skip-black* :: ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt

where

skip-black t = (let t' = *skip-red* t in case t' of Branch color.B l k v r $\Rightarrow$ l | - $\Rightarrow$ t')

datatype *compare* = LT | GT | EQ

partial-function (tailrec) *compare-height* :: ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ *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' - - -) $\Rightarrow$
compare-height (*skip-black* sx') s' t' (*skip-black* tx')
| (-, rbt.Empty, -, Branch - - - -) $\Rightarrow$ LT
| (Branch - - - - -, -, rbt.Empty, -) $\Rightarrow$ GT
| (Branch - sx' - - -, Branch - s' - - -, Branch - t' - - -, rbt.Empty) $\Rightarrow$
compare-height (*skip-black* sx') s' t' rbt.Empty
| (rbt.Empty, Branch - s' - - -, Branch - t' - - -, Branch - tx' - - -) $\Rightarrow$
compare-height rbt.Empty s' t' (*skip-black* tx')
| - $\Rightarrow$ 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

119.10 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* ⇒ *b* ⇒ *b* ⇒ *b*) ⇒ (*a* × *b*) list ⇒ (*a* × *b*) list ⇒
(*a* × *b*) list

where

sinter-with f ((k, v) # as) ((k', v') # bs) =

(if k > k' then sinter-with f ((k, v) # as) bs

else if k < k' then sinter-with f as ((k', v') # bs)

else (k, f k v v') # sinter-with f as bs)

| *sinter-with f [] - = []*

| *sinter-with f - [] = []*

⟨*proof*⟩

termination ⟨*proof*⟩

end

declare *ord.sunion-with.simps* [*code*] *ord.sinter-with.simps*[*code*]

context *linorder* **begin**

lemma *set-fst-sunion-with*:

$set (map fst (sunion-with f xs ys)) = set (map fst xs) \cup set (map fst ys)$
 $\langle proof \rangle$

lemma *sorted-sunion-with* [simp]:
 $\llbracket sorted (map fst xs); sorted (map fst ys) \rrbracket$
 $\impl sorted (map fst (sunion-with f xs ys))$
 $\langle proof \rangle$

lemma *distinct-sunion-with* [simp]:
 $\llbracket distinct (map fst xs); distinct (map fst ys); sorted (map fst xs); sorted (map fst ys) \rrbracket$
 $\impl distinct (map fst (sunion-with f xs ys))$
 $\langle proof \rangle$

lemma *map-of-sunion-with*:
 $\llbracket sorted (map fst xs); sorted (map fst ys) \rrbracket$
 $\impl map-of (sunion-with f xs ys) k =$
 $(case map-of xs k of None \Rightarrow map-of ys k$
 $| Some v \Rightarrow case map-of ys k of None \Rightarrow Some v$
 $| Some w \Rightarrow Some (f k v w))$
 $\langle proof \rangle$

lemma *set-fst-sinter-with* [simp]:
 $\llbracket sorted (map fst xs); sorted (map fst ys) \rrbracket$
 $\impl set (map fst (sinter-with f xs ys)) = set (map fst xs) \cap set (map fst ys)$
 $\langle proof \rangle$

lemma *set-fst-sinter-with-subset1*:
 $set (map fst (sinter-with f xs ys)) \subseteq set (map fst xs)$
 $\langle proof \rangle$

lemma *set-fst-sinter-with-subset2*:
 $set (map fst (sinter-with f xs ys)) \subseteq set (map fst ys)$
 $\langle proof \rangle$

lemma *sorted-sinter-with* [simp]:
 $\llbracket sorted (map fst xs); sorted (map fst ys) \rrbracket$
 $\impl sorted (map fst (sinter-with f xs ys))$
 $\langle proof \rangle$

lemma *distinct-sinter-with* [simp]:
 $\llbracket distinct (map fst xs); distinct (map fst ys) \rrbracket$
 $\impl distinct (map fst (sinter-with f xs ys))$
 $\langle proof \rangle$

lemma *map-of-sinter-with*:
 $\llbracket sorted (map fst xs); sorted (map fst ys) \rrbracket$
 $\impl map-of (sinter-with f xs ys) k =$
 $(case map-of xs k of None \Rightarrow None | Some v \Rightarrow map-option (f k v) (map-of ys$

$k))$
 $\langle \text{proof} \rangle$

end

lemma *distinct-map-of-rev*: $\text{distinct} (\text{map fst } xs) \implies \text{map-of} (\text{rev } xs) = \text{map-of } xs$
 $\langle \text{proof} \rangle$

lemma *map-map-filter*:
 $\text{map } f (\text{List.map-filter } g \text{ } xs) = \text{List.map-filter} (\text{map-option } f \circ g) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *map-filter-map-option-const*:
 $\text{List.map-filter} (\lambda x. \text{map-option} (\lambda y. f \text{ } x) (g (f \text{ } x))) \text{ } xs = \text{filter} (\lambda x. g \text{ } x \neq \text{None})$
 $(\text{map } f \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *set-map-filter*: $\text{set} (\text{List.map-filter } P \text{ } xs) = \text{the } ' (P \text{ } ' \text{ set } xs - \{\text{None}\})$
 $\langle \text{proof} \rangle$

definition *is-rbt-empty* :: $('a, 'b) \text{ rbt} \Rightarrow \text{bool}$ **where**
 $\text{is-rbt-empty } t \longleftrightarrow (\text{case } t \text{ of } \text{RBT-Impl.Empty} \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

lemma *is-rbt-empty-prop[simp]*: $\text{is-rbt-empty } t \longleftrightarrow t = \text{RBT-Impl.Empty}$
 $\langle \text{proof} \rangle$

definition *small-rbt* :: $('a, 'b) \text{ rbt} \Rightarrow \text{bool}$ **where**
 $\text{small-rbt } t \longleftrightarrow \text{bheight } t < 4$

definition *flip-rbt* :: $('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow \text{bool}$ **where**
 $\text{flip-rbt } t1 \text{ } t2 \longleftrightarrow \text{bheight } t2 < \text{bheight } t1$

abbreviation *(input) MR* **where** $\text{MR } l \text{ } a \text{ } b \text{ } r \equiv \text{Branch } \text{RBT-Impl.R } l \text{ } a \text{ } b \text{ } r$

abbreviation *(input) MB* **where** $\text{MB } l \text{ } a \text{ } b \text{ } r \equiv \text{Branch } \text{RBT-Impl.B } l \text{ } a \text{ } b \text{ } r$

fun *rbt-baliL* :: $('a, 'b) \text{ rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **where**
 $\text{rbt-baliL } (MR (MR \text{ } t1 \text{ } a \text{ } b \text{ } t2) \text{ } a' \text{ } b' \text{ } t3) \text{ } a'' \text{ } b'' \text{ } t4 = MR (MB \text{ } t1 \text{ } a \text{ } b \text{ } t2) \text{ } a' \text{ } b' (MB$
 $t3 \text{ } a'' \text{ } b'' \text{ } t4)$
 $\mid \text{rbt-baliL } (MR \text{ } t1 \text{ } a \text{ } b (MR \text{ } t2 \text{ } a' \text{ } b' \text{ } t3)) \text{ } a'' \text{ } b'' \text{ } t4 = MR (MB \text{ } t1 \text{ } a \text{ } b \text{ } t2) \text{ } a' \text{ } b' (MB$
 $t3 \text{ } a'' \text{ } b'' \text{ } t4)$
 $\mid \text{rbt-baliL } t1 \text{ } a \text{ } b \text{ } t2 = MB \text{ } t1 \text{ } a \text{ } b \text{ } t2$

fun *rbt-baliR* :: $('a, 'b) \text{ rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **where**
 $\text{rbt-baliR } t1 \text{ } a \text{ } b (MR \text{ } t2 \text{ } a' \text{ } b' (MR \text{ } t3 \text{ } a'' \text{ } b'' \text{ } t4)) = MR (MB \text{ } t1 \text{ } a \text{ } b \text{ } t2) \text{ } a' \text{ } b' (MB$
 $t3 \text{ } a'' \text{ } b'' \text{ } t4)$
 $\mid \text{rbt-baliR } t1 \text{ } a \text{ } b (MR (MR \text{ } t2 \text{ } a' \text{ } b' \text{ } t3) \text{ } a'' \text{ } b'' \text{ } t4) = MR (MB \text{ } t1 \text{ } a \text{ } b \text{ } t2) \text{ } a' \text{ } b' (MB$
 $t3 \text{ } a'' \text{ } b'' \text{ } t4)$

| *rbt-baliR* *t1 a b t2* = *MB t1 a b t2*

fun *rbt-baldL* :: ('a, 'b) *rbt* $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ ('a, 'b) *rbt* $\Rightarrow$ ('a, 'b) *rbt* **where**
rbt-baldL (*MR t1 a b t2*) *a' b' t3* = *MR (MB t1 a b t2) a' b' t3*
| *rbt-baldL t1 a b (MB t2 a' b' t3)* = *rbt-baliR t1 a b (MR t2 a' b' t3)*
| *rbt-baldL t1 a b (MR (MB t2 a' b' t3) a'' b'' t4)* =
MR (MB t1 a b t2) a' b' (rbt-baliR t3 a'' b'' (paint RBT-Impl.R t4))
| *rbt-baldL t1 a b t2* = *MR t1 a b t2*

fun *rbt-baldR* :: ('a, 'b) *rbt* $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ ('a, 'b) *rbt* $\Rightarrow$ ('a, 'b) *rbt* **where**
rbt-baldR t1 a b (MR t2 a' b' t3) = *MR t1 a b (MB t2 a' b' t3)*
| *rbt-baldR (MB t1 a b t2) a' b' t3* = *rbt-baliL (MR t1 a b t2) a' b' t3*
| *rbt-baldR (MR t1 a b (MB t2 a' b' t3)) a'' b'' t4* =
MR (rbt-baliL (paint RBT-Impl.R t1) a b t2) a' b' (MB t3 a'' b'' t4)
| *rbt-baldR t1 a b t2* = *MR t1 a b t2*

fun *rbt-app* :: ('a, 'b) *rbt* $\Rightarrow$ ('a, 'b) *rbt* $\Rightarrow$ ('a, 'b) *rbt* **where**
rbt-app RBT-Impl.Empty t = *t*
| *rbt-app t RBT-Impl.Empty* = *t*
| *rbt-app (MR t1 a b t2) (MR t3 a'' b'' t4)* = (case *rbt-app t2 t3* of
MR u2 a' b' u3 $\Rightarrow$ (*MR (MR t1 a b u2) a' b' (MR u3 a'' b'' t4)*)
| *t23* $\Rightarrow$ *MR t1 a b (MR t23 a'' b'' t4)*)
| *rbt-app (MB t1 a b t2) (MB t3 a'' b'' t4)* = (case *rbt-app t2 t3* of
MR u2 a' b' u3 $\Rightarrow$ *MR (MB t1 a b u2) a' b' (MB u3 a'' b'' t4)*
| *t23* $\Rightarrow$ *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* $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ ('a, 'b) *rbt* $\Rightarrow$ ('a, 'b) *rbt* **where**
rbt-joinL l a b r = (if *bheight l* $\geq$ *bheight r* then *MR l a b r*
else case *r* of *MB l' a' b' r'* $\Rightarrow$ *rbt-baliL (rbt-joinL l a b l') a' b' r'*
| *MR l' a' b' r'* $\Rightarrow$ *MR (rbt-joinL l a b l') a' b' r'*)

declare *rbt-joinL.simps*[*simp del*]

fun *rbt-joinR* :: ('a, 'b) *rbt* $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ ('a, 'b) *rbt* $\Rightarrow$ ('a, 'b) *rbt* **where**
rbt-joinR l a b r = (if *bheight l* $\leq$ *bheight r* then *MR l a b r*
else case *l* of *MB l' a' b' r'* $\Rightarrow$ *rbt-baliR l' a' b' (rbt-joinR r' a b r)*
| *MR l' a' b' r'* $\Rightarrow$ *MR l' a' b' (rbt-joinR r' a b r)*)

declare *rbt-joinR.simps*[*simp del*]

definition *rbt-join* :: ('a, 'b) *rbt* $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ ('a, 'b) *rbt* $\Rightarrow$ ('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 $\longleftrightarrow$ inv1 t $\wedge$ inv2 t*

lemma *rbt-Node: inv-12 (RBT-Impl.Branch c l a b r) $\implies$ inv-12 l $\wedge$ inv-12 r*
<proof>

lemma *paint2: paint c2 (paint c1 t) = paint c2 t*
<proof>

lemma *inv1-rbt-baliL: inv1 l $\implies$ inv1 r $\implies$ inv1 (rbt-baliL l a b r)*
<proof>

lemma *inv1-rbt-baliR: inv1 l $\implies$ inv1 r $\implies$ inv1 (rbt-baliR l a b r)*
<proof>

lemma *rbt-bheight-rbt-baliL: bheight l = bheight r $\implies$ bheight (rbt-baliL l a b r)*
 $= \text{Suc } (\text{bheight } l)$
<proof>

lemma *rbt-bheight-rbt-baliR: bheight l = bheight r $\implies$ bheight (rbt-baliR l a b r)*
 $= \text{Suc } (\text{bheight } l)$

$\langle \text{proof} \rangle$

lemma *inv2-rbt-baliL*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r \implies \text{inv2 } (\text{rbt-baliL } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *inv2-rbt-baliR*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r \implies \text{inv2 } (\text{rbt-baliR } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *inv-rbt-baliR*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{inv1 } l \implies \text{inv1 } r \implies \text{bheight } l = \text{bheight } r \implies$
 $\text{inv1 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{inv2 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baliR } l \ a \ b \ r) =$
 $\text{Suc } (\text{bheight } l)$
 $\langle \text{proof} \rangle$

lemma *inv-rbt-baliL*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{inv1l } l \implies \text{inv1 } r \implies \text{bheight } l = \text{bheight } r \implies$
 $\text{inv1 } (\text{rbt-baliL } l \ a \ b \ r) \wedge \text{inv2 } (\text{rbt-baliL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baliL } l \ a \ b \ r) =$
 $\text{Suc } (\text{bheight } l)$
 $\langle \text{proof} \rangle$

lemma *inv2-rbt-baldL-inv1*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l + 1 = \text{bheight } r \implies \text{inv1 } r \implies$
 $\text{inv2 } (\text{rbt-baldL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldL } l \ a \ b \ r) = \text{bheight } r$
 $\langle \text{proof} \rangle$

lemma *inv2-rbt-baldL-B*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l + 1 = \text{bheight } r \implies \text{color-of } r = \text{RBT-Impl.B} \implies$
 $\text{inv2 } (\text{rbt-baldL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldL } l \ a \ b \ r) = \text{bheight } r$
 $\langle \text{proof} \rangle$

lemma *inv1-rbt-baldL*: $\text{inv1l } l \implies \text{inv1 } r \implies \text{color-of } r = \text{RBT-Impl.B} \implies \text{inv1 } (\text{rbt-baldL } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *inv1lI*: $\text{inv1 } t \implies \text{inv1l } t$
 $\langle \text{proof} \rangle$

lemma *neq-Black[simp]*: $(c \neq \text{RBT-Impl.B}) = (c = \text{RBT-Impl.R})$
 $\langle \text{proof} \rangle$

lemma *inv1l-rbt-baldL*: $\text{inv1l } l \implies \text{inv1 } r \implies \text{inv1l } (\text{rbt-baldL } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *inv2-rbt-baldR-inv1*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r + 1 \implies \text{inv1 } l \implies$
 $\text{inv2 } (\text{rbt-baldR } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldR } l \ a \ b \ r) = \text{bheight } l$
 $\langle \text{proof} \rangle$

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*: $inv2\ l \Longrightarrow inv2\ r \Longrightarrow bheight\ l = bheight\ r \Longrightarrow inv1\ l \Longrightarrow inv1\ r \Longrightarrow inv2\ (rbt-app\ l\ r) \wedge bheight\ (rbt-app\ l\ r) = bheight\ l \wedge inv1l\ (rbt-app\ l\ r) \wedge (color-of\ l = RBT-Impl.B \wedge color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-app\ l\ r))$
 ⟨proof⟩

lemma *inv1l-rbt-joinL*: $inv1\ l \Longrightarrow inv1\ r \Longrightarrow bheight\ l \leq bheight\ r \Longrightarrow inv1l\ (rbt-joinL\ l\ a\ b\ r) \wedge (bheight\ l \neq bheight\ r \wedge color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-joinL\ l\ a\ b\ r))$
 ⟨proof⟩

lemma *inv1l-rbt-joinR*: $inv1\ l \Longrightarrow inv2\ l \Longrightarrow inv1\ r \Longrightarrow inv2\ r \Longrightarrow bheight\ l \geq bheight\ r \Longrightarrow inv1l\ (rbt-joinR\ l\ a\ b\ r) \wedge (bheight\ l \neq bheight\ r \wedge color-of\ l = RBT-Impl.B \longrightarrow inv1\ (rbt-joinR\ l\ a\ b\ r))$
 ⟨proof⟩

lemma *bheight-rbt-joinL*: $inv2\ l \Longrightarrow inv2\ r \Longrightarrow bheight\ l \leq bheight\ r \Longrightarrow$

$bheight (rbt-joinL\ l\ a\ b\ r) = bheight\ r$
 $\langle proof \rangle$

lemma $inv2\text{-}rbt\text{-}joinL$: $inv2\ l \implies inv2\ r \implies bheight\ l \leq bheight\ r \implies inv2$
 $(rbt-joinL\ l\ a\ b\ r)$
 $\langle proof \rangle$

lemma $bheight\text{-}rbt\text{-}joinR$: $inv2\ l \implies inv2\ r \implies bheight\ l \geq bheight\ r \implies$
 $bheight (rbt-joinR\ l\ a\ b\ r) = bheight\ l$
 $\langle proof \rangle$

lemma $inv2\text{-}rbt\text{-}joinR$: $inv2\ l \implies inv2\ r \implies bheight\ l \geq bheight\ r \implies inv2$
 $(rbt-joinR\ l\ a\ b\ r)$
 $\langle proof \rangle$

lemma $keys\text{-}paint[simp]$: $RBT\text{-}Impl.keys (paint\ c\ t) = RBT\text{-}Impl.keys\ t$
 $\langle proof \rangle$

lemma $keys\text{-}rbt\text{-}baliL$: $RBT\text{-}Impl.keys (rbt-baliL\ l\ a\ b\ r) = RBT\text{-}Impl.keys\ l @ a$
 $\# RBT\text{-}Impl.keys\ r$
 $\langle proof \rangle$

lemma $keys\text{-}rbt\text{-}baliR$: $RBT\text{-}Impl.keys (rbt-baliR\ l\ a\ b\ r) = RBT\text{-}Impl.keys\ l @ a$
 $\# RBT\text{-}Impl.keys\ r$
 $\langle proof \rangle$

lemma $keys\text{-}rbt\text{-}baldL$: $RBT\text{-}Impl.keys (rbt-baldL\ l\ a\ b\ r) = RBT\text{-}Impl.keys\ l @ a$
 $\# RBT\text{-}Impl.keys\ r$
 $\langle proof \rangle$

lemma $keys\text{-}rbt\text{-}baldR$: $RBT\text{-}Impl.keys (rbt-baldR\ l\ a\ b\ r) = RBT\text{-}Impl.keys\ l @ a$
 $\# RBT\text{-}Impl.keys\ r$
 $\langle proof \rangle$

lemma $keys\text{-}rbt\text{-}app$: $RBT\text{-}Impl.keys (rbt-app\ l\ r) = RBT\text{-}Impl.keys\ l @ RBT\text{-}Impl.keys\ r$
 $\langle proof \rangle$

lemma $keys\text{-}rbt\text{-}joinL$: $bheight\ l \leq bheight\ r \implies$
 $RBT\text{-}Impl.keys (rbt-joinL\ l\ a\ b\ r) = RBT\text{-}Impl.keys\ l @ a \# RBT\text{-}Impl.keys\ r$
 $\langle proof \rangle$

lemma $keys\text{-}rbt\text{-}joinR$: $RBT\text{-}Impl.keys (rbt-joinR\ l\ a\ b\ r) = RBT\text{-}Impl.keys\ l @ a$
 $\# RBT\text{-}Impl.keys\ r$
 $\langle proof \rangle$

lemma $rbt\text{-}set\text{-}rbt\text{-}baliL$: $set (RBT\text{-}Impl.keys (rbt-baliL\ l\ a\ b\ r)) =$
 $set (RBT\text{-}Impl.keys\ l) \cup \{a\} \cup set (RBT\text{-}Impl.keys\ r)$
 $\langle proof \rangle$

lemma *set-rbt-joinL*: $\text{set } (\text{RBT-Impl.keys } (\text{rbt-joinL } l \ a \ b \ r)) =$
 $\text{set } (\text{RBT-Impl.keys } l) \cup \{a\} \cup \text{set } (\text{RBT-Impl.keys } r)$
 <proof>

lemma *rbt-set-rbt-baliR*: $\text{set } (\text{RBT-Impl.keys } (\text{rbt-baliR } l \ a \ b \ r)) =$
 $\text{set } (\text{RBT-Impl.keys } l) \cup \{a\} \cup \text{set } (\text{RBT-Impl.keys } r)$
 <proof>

lemma *set-rbt-joinR*: $\text{set } (\text{RBT-Impl.keys } (\text{rbt-joinR } l \ a \ b \ r)) =$
 $\text{set } (\text{RBT-Impl.keys } l) \cup \{a\} \cup \text{set } (\text{RBT-Impl.keys } r)$
 <proof>

lemma *set-keys-paint*: $\text{set } (\text{RBT-Impl.keys } (\text{paint } c \ t)) = \text{set } (\text{RBT-Impl.keys } t)$
 <proof>

lemma *set-rbt-join*: $\text{set } (\text{RBT-Impl.keys } (\text{rbt-join } l \ a \ b \ r)) =$
 $\text{set } (\text{RBT-Impl.keys } l) \cup \{a\} \cup \text{set } (\text{RBT-Impl.keys } r)$
 <proof>

lemma *inv-rbt-join*: $\text{inv-12 } l \implies \text{inv-12 } r \implies \text{inv-12 } (\text{rbt-join } l \ a \ b \ r)$
 <proof>

fun *rbt-recolor* :: $('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **where**
 $\text{rbt-recolor } (\text{Branch } \text{RBT-Impl.R } t1 \ k \ v \ t2) =$
 $(\text{if } \text{color-of } t1 = \text{RBT-Impl.B} \wedge \text{color-of } t2 = \text{RBT-Impl.B} \text{ then } \text{Branch}$
 $\text{RBT-Impl.B } t1 \ k \ v \ t2$
 $\text{else } \text{Branch } \text{RBT-Impl.R } t1 \ k \ v \ t2)$
 $| \text{rbt-recolor } t = t$

lemma *rbt-recolor*: $\text{inv-12 } t \implies \text{inv-12 } (\text{rbt-recolor } t)$
 <proof>

fun *rbt-split-min* :: $('a, 'b) \text{ rbt} \Rightarrow 'a \times 'b \times ('a, 'b) \text{ rbt}$ **where**
 $\text{rbt-split-min } \text{RBT-Impl.Empty} = \text{undefined}$
 $| \text{rbt-split-min } (\text{RBT-Impl.Branch } - \ l \ a \ b \ r) =$
 $(\text{if } \text{is-rbt-empty } l \text{ then } (a, b, r) \text{ else let } (a', b', l') = \text{rbt-split-min } l \text{ in } (a', b', \text{rbt-join}$
 $l' \ a \ b \ r))$

lemma *rbt-split-min-set*:
 $\text{rbt-split-min } t = (a, b, t') \implies t \neq \text{RBT-Impl.Empty} \implies$
 $a \in \text{set } (\text{RBT-Impl.keys } t) \wedge \text{set } (\text{RBT-Impl.keys } t) = \{a\} \cup \text{set } (\text{RBT-Impl.keys } t')$
 <proof>

lemma *rbt-split-min-inv*: $\text{rbt-split-min } t = (a, b, t') \implies \text{inv-12 } t \implies t \neq \text{RBT-Impl.Empty}$
 $\implies \text{inv-12 } t'$
 <proof>

definition *rbt-join2* :: ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ ('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) $\cup$ set (RBT-Impl.keys r)
 <proof>

lemma *inv-rbt-join2*: inv-12 l $\Longrightarrow$ inv-12 r $\Longrightarrow$ inv-12 (rbt-join2 l r)
 <proof>

context ord **begin**

fun *rbt-split* :: ('a, 'b) rbt $\Rightarrow$ 'a $\Rightarrow$ ('a, 'b) rbt $\times$ 'b option $\times$ ('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) $\Rightarrow$ (l1, β , rbt-join l2 a b r))
 else if a < x then (case rbt-split r x of (r1, β , r2) $\Rightarrow$ (rbt-join l a b r1, β , r2))
 else (l, Some b, r))

lemma *rbt-split*: rbt-split t x = (l, β ,r) $\Longrightarrow$ inv-12 t $\Longrightarrow$ inv-12 l $\wedge$ inv-12 r
 <proof>

lemma *rbt-split-size*: (l2, β ,r2) = rbt-split t2 a $\Longrightarrow$ size l2 + size r2 $\leq$ size t2
 <proof>

function *rbt-union-rec* :: ('a $\Rightarrow$ 'b $\Rightarrow$ 'b $\Rightarrow$ 'b) $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ ('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 $\Rightarrow$ t2
 | RBT-Impl.Branch - l1 a b r1 $\Rightarrow$
 case rbt-split t2 a of (l2, β , r2) $\Rightarrow$
 rbt-join (rbt-union-rec f l1 l2) a (case β of None $\Rightarrow$ b | Some b' $\Rightarrow$ 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 $\Rightarrow$ 'b $\Rightarrow$ 'b $\Rightarrow$ 'b) $\Rightarrow$ bool $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt $\Rightarrow$ ('a, 'b) rbt **where**
rbt-union-swap-rec f γ t1 t2 = (let (γ , t2, t1) =
 if flip-rbt t2 t1 then ($\neg\gamma$, 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 $\Rightarrow$ 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 (λk 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 (λk 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 (λk 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 (λk 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 (λk - 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*:
 $\llbracket (k, v) \in \text{set } (\text{entries } t); (k, v') \in \text{set } (\text{entries } t);$
rbt-sorted t $\rrbracket \implies v = v'$
 ⟨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 $\implies$ *inv-12* (rbt-delete x t)
 ⟨proof⟩

lemma *rbt-sorted-delete*: *rbt-sorted* t $\implies$ *rbt-sorted* (rbt-delete x t)
 ⟨proof⟩

lemma *rbt-fold-rbt-delete*:
assumes *inv-12* t2
shows *inv-12* (RBT-Impl.fold ($\lambda k - t.$ rbt-delete k t) t1 t2)
 ⟨proof⟩

lemma *rbt-minus-rec*: *inv-12* t1 $\implies$ *inv-12* t2 $\implies$ *inv-12* (rbt-minus-rec t1 t2)
 ⟨proof⟩

end

context *linorder* **begin**

lemma *rbt-sorted-rbt-baliL*: *rbt-sorted* l $\implies$ *rbt-sorted* r $\implies$ l |« a $\implies$ a «| r $\implies$
rbt-sorted (rbt-baliL l a b r)
 ⟨proof⟩

lemma *rbt-lookup-rbt-baliL*: *rbt-sorted* l $\implies$ *rbt-sorted* r $\implies$ l |« a $\implies$ a «| r $\implies$
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 $\implies$ *rbt-sorted* r $\implies$ l |« a $\implies$ a «| r $\implies$
rbt-sorted (rbt-baliR l a b r)
 ⟨proof⟩

lemma *rbt-lookup-rbt-baliR*: *rbt-sorted* l $\implies$ *rbt-sorted* r $\implies$ l |« a $\implies$ a «| r $\implies$
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) $\implies$ *bheight* l
 $\leq$ *bheight* r $\implies$
rbt-sorted (rbt-joinL l a b r)
 ⟨proof⟩

lemma *rbt-lookup-rbt-joinL*: *rbt-sorted* l $\implies$ *rbt-sorted* r $\implies$ l |« a $\implies$ a «| r $\implies$
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*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \mid \ll r \implies \text{rbt-sorted } (\text{rbt-joinR } l \ a \ b \ r)$
 ⟨proof⟩

lemma *rbt-lookup-rbt-joinR*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \mid \ll r \implies \text{rbt-lookup } (\text{rbt-joinR } l \ a \ b \ r) \ k =$
 (if $k < a$ then $\text{rbt-lookup } l \ k$ else if $k = a$ then $\text{Some } b$ else $\text{rbt-lookup } r \ k$)
 ⟨proof⟩

lemma *rbt-sorted-paint*: $\text{rbt-sorted } (\text{paint } c \ t) = \text{rbt-sorted } t$
 ⟨proof⟩

lemma *rbt-sorted-rbt-join*: $\text{rbt-sorted } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \implies \text{rbt-sorted } (\text{rbt-join } l \ a \ b \ r)$
 ⟨proof⟩

lemma *rbt-lookup-rbt-join*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \mid \ll r \implies \text{rbt-lookup } (\text{rbt-join } l \ a \ b \ r) \ k =$
 (if $k < a$ then $\text{rbt-lookup } l \ k$ else if $k = a$ then $\text{Some } b$ else $\text{rbt-lookup } r \ k$)
 ⟨proof⟩

lemma *rbt-split-min-rbt-sorted*: $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq \text{RBT-Impl.Empty} \implies \text{rbt-sorted } t' \wedge (\forall x \in \text{set } (\text{RBT-Impl.keys } t'). \ a < x)$
 ⟨proof⟩

lemma *rbt-split-min-rbt-lookup*: $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq \text{RBT-Impl.Empty} \implies \text{rbt-lookup } t \ k =$
 (if $k < a$ then None else if $k = a$ then $\text{Some } b$ else $\text{rbt-lookup } t' \ k$)
 ⟨proof⟩

lemma *rbt-sorted-rbt-join2*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies \forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies \text{rbt-sorted } (\text{rbt-join2 } l \ r)$
 ⟨proof⟩

lemma *rbt-lookup-rbt-join2*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies \forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies \text{rbt-lookup } (\text{rbt-join2 } l \ r) \ k =$
 (case $\text{rbt-lookup } l \ k$ of $\text{None} \Rightarrow \text{rbt-lookup } r \ k \mid \text{Some } v \Rightarrow \text{Some } v$)
 ⟨proof⟩

lemma *rbt-split-props*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies \text{set } (\text{RBT-Impl.keys } l) = \{a \in \text{set } (\text{RBT-Impl.keys } t). \ a < x\} \wedge \text{set } (\text{RBT-Impl.keys } r) = \{a \in \text{set } (\text{RBT-Impl.keys } t). \ x < a\} \wedge \text{rbt-sorted } l \wedge \text{rbt-sorted } r$
 ⟨proof⟩

lemma *rbt-split-lookup*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies$
 $\text{rbt-lookup } t \ k = (\text{if } k < x \text{ then } \text{rbt-lookup } l \ k \text{ else if } k = x \text{ then } \beta \text{ else } \text{rbt-lookup } r \ k)$
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-fold-insertwk*: $\text{rbt-sorted } t \implies$
 $\text{rbt-sorted } (\text{RBT-Impl.fold } (\text{rbt-insert-with-key } f) \ t' \ t)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-iff-keys*:
 $\text{rbt-sorted } t \implies \text{set } (\text{RBT-Impl.keys } t) = \{k. \exists v. \text{rbt-lookup } t \ k = \text{Some } v\}$
 $\text{rbt-sorted } t \implies \text{rbt-lookup } t \ k = \text{None} \longleftrightarrow k \notin \text{set } (\text{RBT-Impl.keys } t)$
 $\text{rbt-sorted } t \implies (\exists v. \text{rbt-lookup } t \ k = \text{Some } v) \longleftrightarrow k \in \text{set } (\text{RBT-Impl.keys } t)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-fold-rbt-insertwk*:
assumes $t1: \text{rbt-sorted } t1$ **and** $t2: \text{rbt-sorted } t2$
shows $\text{rbt-lookup } (\text{fold } (\text{rbt-insert-with-key } f) \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } t2 \ k$
 $\mid \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{Some } v$
 $\mid \text{Some } w \Rightarrow \text{Some } (f \ k \ w \ v))$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-union-rec*: $\text{rbt-sorted } t1 \implies \text{rbt-sorted } t2 \implies$
 $\text{rbt-sorted } (\text{rbt-union-rec } f \ t1 \ t2) \wedge \text{rbt-lookup } (\text{rbt-union-rec } f \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } t2 \ k$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{Some } v$
 $\mid \text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w)))$
 $\langle \text{proof} \rangle$

lemma *rbtreeify-map-filter-inter*:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$
assumes $\text{rbt-sorted } t2$
shows $\text{rbt-sorted } (\text{rbtreeify } (\text{map-filter-inter } f \ t1 \ t2))$
 $\text{rbt-lookup } (\text{rbtreeify } (\text{map-filter-inter } f \ t1 \ t2)) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{None}$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w)))$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-inter-rec*: $\text{rbt-sorted } t1 \implies \text{rbt-sorted } t2 \implies$
 $\text{rbt-sorted } (\text{rbt-inter-rec } f \ t1 \ t2) \wedge \text{rbt-lookup } (\text{rbt-inter-rec } f \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{None}$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w)))$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-delete*:
assumes $\text{inv-12 } t \ \text{rbt-sorted } t$

shows $\text{rbt-lookup } (\text{rbt-delete } x \ t) \ k = (\text{if } x = k \text{ then None else rbt-lookup } t \ k)$
 $\langle \text{proof} \rangle$

lemma *fold-rbt-delete*:

assumes $\text{inv-12 } t1 \ \text{rbt-sorted } t1 \ \text{rbt-sorted } t2$
shows $\text{inv-12 } (\text{RBT-Impl.fold } (\lambda k \ - \ t. \ \text{rbt-delete } k \ t) \ t2 \ t1) \wedge$
 $\text{rbt-sorted } (\text{RBT-Impl.fold } (\lambda k \ - \ t. \ \text{rbt-delete } k \ t) \ t2 \ t1) \wedge$
 $\text{rbt-lookup } (\text{RBT-Impl.fold } (\lambda k \ - \ t. \ \text{rbt-delete } k \ t) \ t2 \ t1) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \ \text{of } \text{None} \Rightarrow \text{None}$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \ \text{of } \text{None} \Rightarrow \text{Some } v \mid - \Rightarrow \text{None}))$
 $\langle \text{proof} \rangle$

lemma *rbtreeify-filter-minus*:

assumes $\text{rbt-sorted } t1$
shows $\text{rbt-sorted } (\text{rbtreeify } (\text{filter-minus } t1 \ t2)) \wedge$
 $\text{rbt-lookup } (\text{rbtreeify } (\text{filter-minus } t1 \ t2)) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \ \text{of } \text{None} \Rightarrow \text{None}$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \ \text{of } \text{None} \Rightarrow \text{Some } v \mid - \Rightarrow \text{None}))$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-minus-rec*: $\text{inv-12 } t1 \Rightarrow \text{rbt-sorted } t1 \Rightarrow \text{rbt-sorted } t2 \Rightarrow$
 $\text{rbt-sorted } (\text{rbt-minus-rec } t1 \ t2) \wedge \text{rbt-lookup } (\text{rbt-minus-rec } t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \ \text{of } \text{None} \Rightarrow \text{None}$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \ \text{of } \text{None} \Rightarrow \text{Some } v \mid - \Rightarrow \text{None}))$
 $\langle \text{proof} \rangle$

end

context *ord* **begin**

definition *rbt-union-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$
 $\Rightarrow ('a, 'b) \text{ rbt}$

where

$\text{rbt-union-with-key } f \ t1 \ t2 = \text{paint } B \ (\text{rbt-union-swap-rec } f \ \text{False } t1 \ t2)$

definition *rbt-union-with* **where**

$\text{rbt-union-with } f = \text{rbt-union-with-key } (\lambda \cdot. f)$

definition *rbt-union* **where**

$\text{rbt-union} = \text{rbt-union-with-key } (\% - \text{ rv. rv})$

definition *rbt-inter-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$
 $\Rightarrow ('a, 'b) \text{ rbt}$

where

$\text{rbt-inter-with-key } f \ t1 \ t2 = \text{paint } B \ (\text{rbt-inter-swap-rec } f \ \text{False } t1 \ t2)$

definition *rbt-inter-with* **where**

$\text{rbt-inter-with } f = \text{rbt-inter-with-key } (\lambda \cdot. f)$

definition *rbt-inter* **where**

rbt-inter = *rbt-inter-with-key* ($\lambda\text{-}$ - *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*]:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-union-with-key } f \text{ } t1 \text{ } t2)$
<proof>

lemma *rbt-lookup-rbt-unionwk*:

$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$
 $\implies \text{rbt-lookup } (\text{rbt-union-with-key } f \text{ } t1 \text{ } t2) \text{ } k =$
 $(\text{case } \text{rbt-lookup } t1 \text{ } k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } t2 \text{ } k$
 $\quad | \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \text{ } k \text{ of } \text{None} \Rightarrow \text{Some } v$
 $\quad \quad | \text{Some } w \Rightarrow \text{Some } (f \text{ } k \text{ } v \text{ } w))$
<proof>

lemma *rbt-unionw-is-rbt*: $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \implies \text{is-rbt } (\text{rbt-union-with } f \text{ } lt \text{ } rt)$

<proof>

lemma *rbt-union-is-rbt*: $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \implies \text{is-rbt } (\text{rbt-union } lt \text{ } rt)$

<proof>

lemma *rbt-lookup-rbt-union*:

$\llbracket \text{rbt-sorted } s; \text{rbt-sorted } t \rrbracket \implies$
 $\text{rbt-lookup } (\text{rbt-union } s \text{ } t) = \text{rbt-lookup } s \text{ } ++ \text{rbt-lookup } t$
<proof>

lemma *rbt-interwk-is-rbt* [*simp*]:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter-with-key } f \text{ } t1 \text{ } t2)$
<proof>

lemma *rbt-interw-is-rbt*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter-with } f \text{ } t1 \text{ } t2)$
<proof>

lemma *rbt-inter-is-rbt*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter } t1 \text{ } t2)$
<proof>

lemma *rbt-lookup-rbt-interwk*:

$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$
 $\implies \text{rbt-lookup } (\text{rbt-inter-with-key } f \text{ } t1 \text{ } t2) \text{ } k =$
 $(\text{case } \text{rbt-lookup } t1 \text{ } k \text{ of } \text{None} \Rightarrow \text{None}$

$$\begin{aligned} & | \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \text{ } k \text{ of } \text{None} \Rightarrow \text{None} \\ & | \text{Some } w \Rightarrow \text{Some } (f \text{ } k \text{ } v \text{ } w)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *rbt-lookup-rbt-inter*:

$$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket \implies \text{rbt-lookup } (\text{rbt-inter } t1 \text{ } t2) = \text{rbt-lookup } t2 \mid ' \text{dom } (\text{rbt-lookup } t1)$$

$$\langle \text{proof} \rangle$$

lemma *rbt-minus-is-rbt*:

$$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-minus } t1 \text{ } t2)$$

$$\langle \text{proof} \rangle$$

lemma *rbt-lookup-rbt-minus*:

$$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{rbt-lookup } (\text{rbt-minus } t1 \text{ } t2) = \text{rbt-lookup } t1 \mid ' (- \text{dom } (\text{rbt-lookup } t2))$$

$$\langle \text{proof} \rangle$$

end

119.11 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* ::

$((\alpha \times \beta) \times (\alpha, \beta) \text{ rbt}) \text{ list} \Rightarrow (\alpha, \beta) \text{ rbt} \Rightarrow (\alpha \times \beta) \text{ list}$

where

gen-entries kvs t = *entries t* @ *concat* (*map* ($\lambda(kv, t). kv \# \text{entries } t$) *kvs*)

lemma *gen-entries-simps* [*simp*, *code*]:

gen-entries [] *Empty* = []

gen-entries ((*kv*, *t*) # *kvs*) *Empty* = *kv* # *gen-entries kvs t*

gen-entries kvs (*Branch c l k v r*) = *gen-entries* (((*k*, *v*), *r*) # *kvs*) *l*

<proof>

lemma *entries-code* [*code*]:

entries = *gen-entries* []

<proof>

definition *gen-keys* :: $(\alpha \times (\alpha, \beta) \text{ rbt}) \text{ list} \Rightarrow (\alpha, \beta) \text{ rbt} \Rightarrow \alpha \text{ list}$

where *gen-keys kts t* = *RBT-Impl.keys t* @ *concat* (*List.map* ($\lambda(k, t). k \# \text{keys } t$) *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

120 Abstract type of RBT trees

theory *RBT*

imports *Main RBT-Impl*

begin

120.1 Type definition

typedef (overloaded) (*'a*, *'b*) *rbt* = {*t* :: (*'a*::*linorder*, *'b*) *RBT-Impl.rbt*. *is-rbt* *t*}
morphisms *impl-of RBT*
 $\langle \text{proof} \rangle$

lemma *rbt-eq-iff*:
 $t1 = t2 \iff \text{impl-of } t1 = \text{impl-of } t2$
 $\langle \text{proof} \rangle$

lemma *rbt-eqI*:
 $\text{impl-of } t1 = \text{impl-of } t2 \implies t1 = t2$
 $\langle \text{proof} \rangle$

lemma *is-rbt-impl-of* [*simp*, *intro*]:
 $\text{is-rbt } (\text{impl-of } t)$
 $\langle \text{proof} \rangle$

lemma *RBT-impl-of* [*simp*, *code abstype*]:
 $\text{RBT } (\text{impl-of } t) = t$
 $\langle \text{proof} \rangle$

120.2 Primitive operations

setup-lifting *type-definition-rbt*

lift-definition *lookup* :: (*'a*::*linorder*, *'b*) *rbt* $\Rightarrow$ *'a* $\rightarrow$ *'b* **is** *rbt-lookup* $\langle \text{proof} \rangle$

lift-definition *empty* :: (*'a*::*linorder*, *'b*) *rbt* **is** *RBT-Impl.Empty*
 $\langle \text{proof} \rangle$

lift-definition *insert* :: *'a*::*linorder* $\Rightarrow$ *'b* $\Rightarrow$ (*'a*, *'b*) *rbt* $\Rightarrow$ (*'a*, *'b*) *rbt* **is** *rbt-insert*
 $\langle \text{proof} \rangle$

lift-definition *delete* :: *'a*::*linorder* $\Rightarrow$ (*'a*, *'b*) *rbt* $\Rightarrow$ (*'a*, *'b*) *rbt* **is** *rbt-delete*
 $\langle \text{proof} \rangle$

lift-definition *entries* :: (*'a*::*linorder*, *'b*) *rbt* $\Rightarrow$ (*'a* $\times$ *'b*) *list* **is** *RBT-Impl.entries*
 $\langle \text{proof} \rangle$

lift-definition *keys* :: (*'a*::*linorder*, *'b*) *rbt* $\Rightarrow$ *'a* *list* **is** *RBT-Impl.keys* $\langle \text{proof} \rangle$

lift-definition *bulkload* :: (*'a*::*linorder* $\times$ *'b*) *list* $\Rightarrow$ (*'a*, *'b*) *rbt* **is** *rbt-bulkload*
 $\langle \text{proof} \rangle$

lift-definition *map-entry* :: *'a* $\Rightarrow$ (*'b* $\Rightarrow$ *'b*) $\Rightarrow$ (*'a*::*linorder*, *'b*) *rbt* $\Rightarrow$ (*'a*, *'b*) *rbt* **is** *rbt-map-entry*
 $\langle \text{proof} \rangle$

lift-definition $\text{map} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'c) \text{rbt}$ **is**
 RBT-Impl.map
 $\langle \text{proof} \rangle$

lift-definition $\text{fold} :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow 'c \Rightarrow 'c$ **is**
 RBT-Impl.fold $\langle \text{proof} \rangle$

lift-definition $\text{union} :: ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **is**
 rbt-union
 $\langle \text{proof} \rangle$

lift-definition $\text{foldi} :: ('c \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a :: \text{linorder}, 'b) \text{rbt} \Rightarrow 'c \Rightarrow 'c$
is RBT-Impl.foldi $\langle \text{proof} \rangle$

lift-definition $\text{combine-with-key} :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$
is $\text{RBT-Impl.rbt-union-with-key}$ $\langle \text{proof} \rangle$

lift-definition $\text{combine} :: ('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$
is $\text{RBT-Impl.rbt-union-with}$ $\langle \text{proof} \rangle$

120.3 Derived operations

definition $\text{is-empty} :: ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow \text{bool}$ **where**
 $[\text{code}]: \text{is-empty } t = (\text{case } \text{impl-of } t \text{ of } \text{RBT-Impl.Empty} \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

definition $\text{filter} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**
 $[\text{code}]: \text{filter } P \ t = \text{fold } (\lambda k \ v \ t. \text{if } P \ k \ v \text{ then } \text{insert } k \ v \ t \text{ else } t) \ t \ \text{empty}$

120.4 Abstract lookup properties

lemma lookup-RBT :
 $\text{is-rbt } t \Longrightarrow \text{lookup } (\text{RBT } t) = \text{rbt-lookup } t$
 $\langle \text{proof} \rangle$

lemma lookup-impl-of :
 $\text{rbt-lookup } (\text{impl-of } t) = \text{lookup } t$
 $\langle \text{proof} \rangle$

lemma entries-impl-of :
 $\text{RBT-Impl.entries } (\text{impl-of } t) = \text{entries } t$
 $\langle \text{proof} \rangle$

lemma keys-impl-of :
 $\text{RBT-Impl.keys } (\text{impl-of } t) = \text{keys } t$
 $\langle \text{proof} \rangle$

lemma *lookup-keys*:

$\text{dom } (\text{lookup } t) = \text{set } (\text{keys } t)$
 $\langle \text{proof} \rangle$

lemma *lookup-empty* [simp]:

$\text{lookup } \text{empty} = \text{Map.empty}$
 $\langle \text{proof} \rangle$

lemma *lookup-insert* [simp]:

$\text{lookup } (\text{insert } k \ v \ t) = (\text{lookup } t)(k \mapsto v)$
 $\langle \text{proof} \rangle$

lemma *lookup-delete* [simp]:

$\text{lookup } (\text{delete } k \ t) = (\text{lookup } t)(k := \text{None})$
 $\langle \text{proof} \rangle$

lemma *map-of-entries* [simp]:

$\text{map-of } (\text{entries } t) = \text{lookup } t$
 $\langle \text{proof} \rangle$

lemma *entries-lookup*:

$\text{entries } t1 = \text{entries } t2 \longleftrightarrow \text{lookup } t1 = \text{lookup } t2$
 $\langle \text{proof} \rangle$

lemma *lookup-bulkload* [simp]:

$\text{lookup } (\text{bulkload } xs) = \text{map-of } xs$
 $\langle \text{proof} \rangle$

lemma *lookup-map-entry* [simp]:

$\text{lookup } (\text{map-entry } k \ f \ t) = (\text{lookup } t)(k := \text{map-option } f \ (\text{lookup } t \ k))$
 $\langle \text{proof} \rangle$

lemma *lookup-map* [simp]:

$\text{lookup } (\text{map } f \ t) \ k = \text{map-option } (f \ k) \ (\text{lookup } t \ k)$
 $\langle \text{proof} \rangle$

lemma *lookup-combine-with-key* [simp]:

$\text{lookup } (\text{combine-with-key } f \ t1 \ t2) \ k = \text{combine-options } (f \ k) \ (\text{lookup } t1 \ k) \ (\text{lookup } t2 \ k)$
 $\langle \text{proof} \rangle$

lemma *combine-altdef*: $\text{combine } f \ t1 \ t2 = \text{combine-with-key } (\lambda \cdot. f) \ t1 \ t2$

$\langle \text{proof} \rangle$

lemma *lookup-combine* [simp]:

$\text{lookup } (\text{combine } f \ t1 \ t2) \ k = \text{combine-options } f \ (\text{lookup } t1 \ k) \ (\text{lookup } t2 \ k)$
 $\langle \text{proof} \rangle$

lemma *fold-fold*:

$\text{fold } f \ t = \text{List.fold } (\text{case-prod } f) \ (\text{entries } t)$
 $\langle \text{proof} \rangle$

lemma *impl-of-empty*:

$\text{impl-of empty} = \text{RBT-Impl.Empty}$
 $\langle \text{proof} \rangle$

lemma *is-empty-empty* [simp]:

$\text{is-empty } t \longleftrightarrow t = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *RBT-lookup-empty* [simp]:

$\text{rbt-lookup } t = \text{Map.empty} \longleftrightarrow t = \text{RBT-Impl.Empty}$
 $\langle \text{proof} \rangle$

lemma *lookup-empty-empty* [simp]:

$\text{lookup } t = \text{Map.empty} \longleftrightarrow t = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *sorted-keys* [iff]:

$\text{sorted } (\text{keys } t)$
 $\langle \text{proof} \rangle$

lemma *distinct-keys* [iff]:

$\text{distinct } (\text{keys } t)$
 $\langle \text{proof} \rangle$

lemma *finite-dom-lookup* [simp, intro!]: $\text{finite } (\text{dom } (\text{lookup } t))$

$\langle \text{proof} \rangle$

lemma *lookup-union*: $\text{lookup } (\text{union } s \ t) = \text{lookup } s \ ++ \ \text{lookup } t$

$\langle \text{proof} \rangle$

lemma *lookup-in-tree*: $(\text{lookup } t \ k = \text{Some } v) = ((k, v) \in \text{set } (\text{entries } t))$

$\langle \text{proof} \rangle$

lemma *keys-entries*: $(k \in \text{set } (\text{keys } t)) = (\exists v. (k, v) \in \text{set } (\text{entries } t))$

$\langle \text{proof} \rangle$

lemma *fold-def-alt*:

$\text{fold } f \ t = \text{List.fold } (\text{case-prod } f) \ (\text{entries } t)$
 $\langle \text{proof} \rangle$

lemma *distinct-entries*: $\text{distinct } (\text{List.map fst } (\text{entries } t))$

$\langle \text{proof} \rangle$

lemma *sorted-entries*: $\text{sorted } (\text{List.map fst } (\text{entries } t))$

$\langle \text{proof} \rangle$

[The following text is a placeholder for a proof, consisting of 15 instances of the word "proof" in italics.]

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *HOL–Library.RBT-Impl*.

121.1 Data type and invariant

The type $(\text{'}k, \text{'}v)$ *RBT-Impl.rbt* denotes red-black trees with keys of type $\text{'}k$ and values of type $\text{'}v$. To function properly, the key type must 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 $(\text{'}k, \text{'}v)$ *RBT.rbt* always obeys this invariant, and for this reason you should only use this in our application. Going back to $(\text{'}k, \text{'}v)$ *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:: $(\text{'}a, \text{'}b)$ *RBT.rbt* $\Rightarrow$ $\text{'}a \Rightarrow \text{'}b$ option

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 $O(\log n)$.

121.2 Operations

Currently, the following operations are supported:

RBT.empty:: $(\text{'}a, \text{'}b)$ *RBT.rbt*

Returns the empty tree. $O(1)$

RBT.insert:: $\text{'}a \Rightarrow \text{'}b \Rightarrow (\text{'}a, \text{'}b)$ *RBT.rbt* $\Rightarrow$ $(\text{'}a, \text{'}b)$ *RBT.rbt*

Updates the map at a given position. $O(\log n)$

RBT.delete:: $\text{'}a \Rightarrow (\text{'}a, \text{'}b)$ *RBT.rbt* $\Rightarrow$ $(\text{'}a, \text{'}b)$ *RBT.rbt*

Deletes a map entry at a given position. $O(\log n)$

RBT.entries:: $(\text{'}a, \text{'}b)$ *RBT.rbt* $\Rightarrow$ $(\text{'}a \times \text{'}b)$ list

Return a corresponding key-value list for a tree.

RBT.bulkload:: $(\text{'}a \times \text{'}b)$ list $\Rightarrow$ $(\text{'}a, \text{'}b)$ *RBT.rbt*

Builds a tree from a key-value list.

RBT.map-entry:: $\text{'}a \Rightarrow (\text{'}b \Rightarrow \text{'}b) \Rightarrow (\text{'}a, \text{'}b)$ *RBT.rbt* $\Rightarrow$ $(\text{'}a, \text{'}b)$ *RBT.rbt*

Maps a single entry in a tree.

$RBT.map::('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b) RBT.rbt \Rightarrow ('a, 'c) RBT.rbt$

Maps all values in a tree. $O(n)$

$RBT.fold::('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b) RBT.rbt \Rightarrow 'c \Rightarrow 'c$

Folds over all entries in a tree. $O(n)$

121.3 Invariant preservation

$is-rbt\ rbt.Empty$	$(Empty-is-rbt)$
$is-rbt\ ?t \Longrightarrow is-rbt\ (rbt-insert\ ?k\ ?v\ ?t)$	$(rbt-insert-is-rbt)$
$is-rbt\ ?t \Longrightarrow is-rbt\ (rbt-delete\ ?k\ ?t)$	$(delete-is-rbt)$
$is-rbt\ (rbt-bulkload\ ?xs)$	$(bulkload-is-rbt)$
$is-rbt\ (rbt-map-entry\ ?k\ ?f\ ?t) = is-rbt\ ?t$	$(map-entry-is-rbt)$
$is-rbt\ (RBT-Impl.map\ ?f\ ?t) = is-rbt\ ?t$	$(map-is-rbt)$
$\llbracket is-rbt\ ?lt; is-rbt\ ?rt \rrbracket \Longrightarrow is-rbt\ (rbt-union\ ?lt\ ?rt)$	$(union-is-rbt)$

121.4 Map Semantics

lookup-empty

$Mapping.lookup\ Mapping.empty\ ?k = None$

lookup-insert

$RBT.lookup\ (RBT.insert\ ?k\ ?v\ ?t) = RBT.lookup\ ?t(?k \mapsto ?v)$

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

122 Implementation of sets using RBT trees

theory *RBT-Set*

imports *RBT Product-Lexorder*

begin

123 Definition of code datatype constructors

definition *Set* :: (*'a*::*linorder*, *unit*) *rbt* $\Rightarrow$ *'a set*
 where *Set t* = {*x* . *RBT.lookup t x* = *Some* ()}

definition *Coset* :: (*'a*::*linorder*, *unit*) *rbt* $\Rightarrow$ *'a set*
 where [*simp*]: *Coset t* = − *Set t*

124 Deletion of already existing code equations

declare [[*code drop*: *Set.empty Set.is-empty uminus-set-inst.uminus-set*
Set.member Set.insert Set.remove UNIV Set.filter image
Set.subset-eq Ball Bex can-select Set.union minus-set-inst.minus-set Set.inter
card the-elem Pow sum prod Product-Type.product Id-on
Image trancl relcomp wf Min Inf-fin Max Sup-fin
(Inf :: 'a set set $\Rightarrow$ 'a set) (Sup :: 'a set set $\Rightarrow$ 'a set)
sorted-list-of-set List.map-project List.Bleast]]

125 Lemmas

125.1 Auxiliary lemmas

lemma [*simp*]: *x* $\neq$ *Some* () $\longleftrightarrow$ *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*⟩

125.2 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* $\Rightarrow$ *'b* $\Rightarrow$ *'b*) $\Rightarrow$ (*'a*, −) *rbt* $\Rightarrow$ *'b* $\Rightarrow$ *'b*
 where [*code-unfold*]: *fold-keys f t A* = *RBT.fold* ($\lambda k - t. f k t$) *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* ($\lambda k - A'. \text{if } P \text{ } k \text{ then } \text{Set.insert } k \text{ } A' \text{ else } A'$) *t* {}

lemma *Set-filter-rbt-filter*:
Set.filter *P* (*Set* *t*) = *rbt-filter* *P* *t*
 ⟨*proof*⟩

125.3 foldi and Ball

lemma *Ball-False*: *RBT-Impl.fold* ($\lambda k \text{ } v \text{ } s. s \wedge P \text{ } k$) *t* *False* = *False*
 ⟨*proof*⟩

lemma *rbt-foldi-fold-conj*:
RBT-Impl.foldi ($\lambda s. s = \text{True}$) ($\lambda k \text{ } v \text{ } s. s \wedge P \text{ } k$) *t* *val* = *RBT-Impl.fold* ($\lambda k \text{ } v \text{ } s. s \wedge P \text{ } k$) *t* *val*
 ⟨*proof*⟩

lemma *foldi-fold-conj*: *RBT.foldi* ($\lambda s. s = \text{True}$) ($\lambda k \text{ } v \text{ } s. s \wedge P \text{ } k$) *t* *val* = *fold-keys* ($\lambda k \text{ } s. s \wedge P \text{ } k$) *t* *val*
 ⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

125.4 foldi and Bex

lemma *Bex-True*: *RBT-Impl.fold* ($\lambda k \text{ } v \text{ } s. s \vee P \text{ } k$) *t* *True* = *True*
 ⟨*proof*⟩

lemma *rbt-foldi-fold-disj*:
RBT-Impl.foldi ($\lambda s. s = \text{False}$) ($\lambda k \text{ } v \text{ } s. s \vee P \text{ } k$) *t* *val* = *RBT-Impl.fold* ($\lambda k \text{ } v \text{ } s. s \vee P \text{ } k$) *t* *val*
 ⟨*proof*⟩

lemma *foldi-fold-disj*: *RBT.foldi* ($\lambda s. s = \text{False}$) ($\lambda k \text{ } v \text{ } s. s \vee P \text{ } k$) *t* *val* = *fold-keys* ($\lambda k \text{ } s. s \vee P \text{ } k$) *t* *val*
 ⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

125.5 folding over non empty trees and selecting the minimal and maximal element

125.5.1 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 $\text{rbt-fold1-keys } f \ t = \text{List.fold } f \ (\text{tl}(\text{RBT-Impl.keys } t)) \ (\text{hd}(\text{RBT-Impl.keys } t))$

minimum definition $\text{rbt-min} :: ('a :: \text{linorder}, \text{unit}) \text{RBT-Impl.rbt} \Rightarrow 'a$
where $\text{rbt-min } t = \text{rbt-fold1-keys } \text{min } t$

lemma $\text{key-le-right: rbt-sorted } (\text{Branch } c \ \text{lt } k \ v \ \text{rt}) \Longrightarrow (\bigwedge x. x \in \text{set } (\text{RBT-Impl.keys } \text{rt}) \Longrightarrow k \leq x)$
 $\langle \text{proof} \rangle$

lemma $\text{left-le-key: rbt-sorted } (\text{Branch } c \ \text{lt } k \ v \ \text{rt}) \Longrightarrow (\bigwedge x. x \in \text{set } (\text{RBT-Impl.keys } \text{lt}) \Longrightarrow x \leq k)$
 $\langle \text{proof} \rangle$

lemma fold-min-triv:
fixes $k :: - :: \text{linorder}$
shows $(\forall x \in \text{set } xs. k \leq x) \Longrightarrow \text{List.fold } \text{min } xs \ k = k$
 $\langle \text{proof} \rangle$

lemma rbt-min-simps:
 $\text{is-rbt } (\text{Branch } c \ \text{RBT-Impl.Empty } k \ v \ \text{rt}) \Longrightarrow \text{rbt-min } (\text{Branch } c \ \text{RBT-Impl.Empty } k \ v \ \text{rt}) = k$
 $\langle \text{proof} \rangle$

fun rbt-min-opt where
 $\text{rbt-min-opt } (\text{Branch } c \ \text{RBT-Impl.Empty } k \ v \ \text{rt}) = k \mid$
 $\text{rbt-min-opt } (\text{Branch } c \ (\text{Branch } lc \ \text{llc } lk \ lv \ \text{lrt}) \ k \ v \ \text{rt}) = \text{rbt-min-opt } (\text{Branch } lc \ \text{llc } lk \ lv \ \text{lrt})$

lemma $\text{rbt-min-opt-Branch:}$
 $t1 \neq \text{rbt.Empty} \Longrightarrow \text{rbt-min-opt } (\text{Branch } c \ t1 \ k \ () \ t2) = \text{rbt-min-opt } t1$
 $\langle \text{proof} \rangle$

lemma $\text{rbt-min-opt-induct [case-names empty left-empty left-non-empty]:}$
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{RBT-Impl.rbt}$
assumes $P \ \text{rbt.Empty}$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \Longrightarrow P \ t2 \Longrightarrow t1 = \text{rbt.Empty} \Longrightarrow P \ (\text{Branch } \text{color } t1 \ a \ b \ t2)$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \Longrightarrow P \ t2 \Longrightarrow t1 \neq \text{rbt.Empty} \Longrightarrow P \ (\text{Branch } \text{color } t1 \ a \ b \ t2)$
shows $P \ t$
 $\langle \text{proof} \rangle$

lemma $\text{rbt-min-opt-in-set:}$
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{RBT-Impl.rbt}$
assumes $t \neq \text{rbt.Empty}$
shows $\text{rbt-min-opt } t \in \text{set } (\text{RBT-Impl.keys } t)$
 $\langle \text{proof} \rangle$

lemma *rbt-min-opt-is-min*:
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes *rbt-sorted* t
assumes $t \neq \text{rbt.Empty}$
shows $\bigwedge y. y \in \text{set } (\text{RBT-Impl.keys } t) \implies y \geq \text{rbt-min-opt } t$
 $\langle \text{proof} \rangle$

lemma *rbt-min-eq-rbt-min-opt*:
assumes $t \neq \text{RBT-Impl.Empty}$
assumes *is-rbt* t
shows $\text{rbt-min } t = \text{rbt-min-opt } t$
 $\langle \text{proof} \rangle$

maximum definition *rbt-max* $:: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt} \Rightarrow 'a$
where $\text{rbt-max } t = \text{rbt-fold1-keys max } t$

lemma *fold-max-triv*:
fixes $k :: - :: \text{linorder}$
shows $(\forall x \in \text{set } xs. x \leq k) \implies \text{List.fold max } xs \ k = k$
 $\langle \text{proof} \rangle$

lemma *fold-max-rev-eq*:
fixes $xs :: ('a :: \text{linorder}) \text{ list}$
assumes $xs \neq []$
shows $\text{List.fold max } (\text{tl } xs) (\text{hd } xs) = \text{List.fold max } (\text{tl } (\text{rev } xs)) (\text{hd } (\text{rev } xs))$
 $\langle \text{proof} \rangle$

lemma *rbt-max-simps*:
assumes *is-rbt* $(\text{Branch } c \ \text{lt } k \ v \ \text{RBT-Impl.Empty})$
shows $\text{rbt-max } (\text{Branch } c \ \text{lt } k \ v \ \text{RBT-Impl.Empty}) = k$
 $\langle \text{proof} \rangle$

fun *rbt-max-opt* **where**
 $\text{rbt-max-opt } (\text{Branch } c \ \text{lt } k \ v \ \text{RBT-Impl.Empty}) = k \mid$
 $\text{rbt-max-opt } (\text{Branch } c \ \text{lt } k \ v \ (\text{Branch } rc \ rlc \ rk \ rv \ rrt)) = \text{rbt-max-opt } (\text{Branch } rc$
 $rlc \ rk \ rv \ rrt)$

lemma *rbt-max-opt-Branch*:
 $t2 \neq \text{rbt.Empty} \implies \text{rbt-max-opt } (\text{Branch } c \ t1 \ k \ () \ t2) = \text{rbt-max-opt } t2$
 $\langle \text{proof} \rangle$

lemma *rbt-max-opt-induct* [*case-names empty right-empty right-non-empty*]:
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes $P \ \text{rbt.Empty}$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t2 = \text{rbt.Empty} \implies P \ (\text{Branch}$
 $\text{color } t1 \ a \ b \ t2)$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t2 \neq \text{rbt.Empty} \implies P \ (\text{Branch}$
 $\text{color } t1 \ a \ b \ t2)$
shows $P \ t$

$\langle \text{proof} \rangle$

lemma *rbt-max-opt-in-set*:
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes $t \neq \text{rbt.Empty}$
shows $\text{rbt-max-opt } t \in \text{set } (\text{RBT-Impl.keys } t)$
 $\langle \text{proof} \rangle$

lemma *rbt-max-opt-is-max*:
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes $\text{rbt-sorted } t$
assumes $t \neq \text{rbt.Empty}$
shows $\bigwedge y. y \in \text{set } (\text{RBT-Impl.keys } t) \implies y \leq \text{rbt-max-opt } t$
 $\langle \text{proof} \rangle$

lemma *rbt-max-eq-rbt-max-opt*:
assumes $t \neq \text{RBT-Impl.Empty}$
assumes $\text{is-rbt } t$
shows $\text{rbt-max } t = \text{rbt-max-opt } t$
 $\langle \text{proof} \rangle$

125.5.2 abstract

context includes *rbt.lifting* **begin**

lift-definition *fold1-keys* $:: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a :: \text{linorder}, 'b) \text{ rbt} \Rightarrow 'a$
is $\text{rbt-fold1-keys } \langle \text{proof} \rangle$

lemma *fold1-keys-def-alt*:
 $\text{fold1-keys } f \ t = \text{List.fold } f \ (\text{tl } (\text{RBT.keys } t)) \ (\text{hd } (\text{RBT.keys } t))$
 $\langle \text{proof} \rangle$

lemma *finite-fold1-fold1-keys*:
assumes $\text{semilattice } f$
assumes $\neg \text{RBT.is-empty } t$
shows $\text{semilattice-set.F } f \ (\text{Set } t) = \text{fold1-keys } f \ t$
 $\langle \text{proof} \rangle$

minimum lift-definition *r-min* $:: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a$ **is** *rbt-min*
 $\langle \text{proof} \rangle$

lift-definition *r-min-opt* $:: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a$ **is** *rbt-min-opt* $\langle \text{proof} \rangle$

lemma *r-min-alt-def*: $\text{r-min } t = \text{fold1-keys } \text{min } t$
 $\langle \text{proof} \rangle$

lemma *r-min-eq-r-min-opt*:
assumes $\neg (\text{RBT.is-empty } t)$
shows $\text{r-min } t = \text{r-min-opt } t$
 $\langle \text{proof} \rangle$

lemma *fold-keys-min-top-eq*:

fixes $t :: ('a :: \{\text{linorder}, \text{bounded-lattice-top}\}, \text{unit}) \text{ rbt}$

assumes $\neg (\text{RBT.is-empty } t)$

shows $\text{fold-keys min } t \text{ top} = \text{fold1-keys min } t$

$\langle \text{proof} \rangle$

maximum lift-definition $r\text{-max} :: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a \text{ is } r\text{bt-max}$

$\langle \text{proof} \rangle$

lift-definition $r\text{-max-opt} :: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a \text{ is } r\text{bt-max-opt} \langle \text{proof} \rangle$

lemma $r\text{-max-alt-def}$: $r\text{-max } t = \text{fold1-keys max } t$

$\langle \text{proof} \rangle$

lemma $r\text{-max-eq-r-max-opt}$:

assumes $\neg (\text{RBT.is-empty } t)$

shows $r\text{-max } t = r\text{-max-opt } t$

$\langle \text{proof} \rangle$

lemma $\text{fold-keys-max-bot-eq}$:

fixes $t :: ('a :: \{\text{linorder}, \text{bounded-lattice-bot}\}, \text{unit}) \text{ rbt}$

assumes $\neg (\text{RBT.is-empty } t)$

shows $\text{fold-keys max } t \text{ bot} = \text{fold1-keys max } t$

$\langle \text{proof} \rangle$

end

126 Code equations

code-datatype *Set Coset*

declare $\text{list.set}[\text{code}]$

lemma $\text{empty-Set} [\text{code}]$:

$\text{Set.empty} = \text{Set RBT.empty}$

$\langle \text{proof} \rangle$

lemma $\text{UNIV-Coset} [\text{code}]$:

$\text{UNIV} = \text{Coset RBT.empty}$

$\langle \text{proof} \rangle$

lemma $\text{is-empty-Set} [\text{code}]$:

$\text{Set.is-empty } (\text{Set } t) = \text{RBT.is-empty } t$

$\langle \text{proof} \rangle$

lemma $\text{compl-code} [\text{code}]$:

– $\text{Set } xs = \text{Coset } xs$

– $\text{Coset } xs = \text{Set } xs$

$\langle proof \rangle$

lemma *member-code* [code]:

$x \in (Set\ t) = (RBT.lookup\ t\ x = Some\ ())$

$x \in (Coset\ t) = (RBT.lookup\ t\ x = None)$

$\langle proof \rangle$

lemma *insert-code* [code]:

$Set.insert\ x\ (Set\ t) = Set\ (RBT.insert\ x\ ()\ t)$

$Set.insert\ x\ (Coset\ t) = Coset\ (RBT.delete\ x\ t)$

$\langle proof \rangle$

lemma *remove-code* [code]:

$Set.remove\ x\ (Set\ t) = Set\ (RBT.delete\ x\ t)$

$Set.remove\ x\ (Coset\ t) = Coset\ (RBT.insert\ x\ ()\ t)$

$\langle proof \rangle$

lemma *union-Set* [code]:

$Set\ t \cup A = fold-keys\ Set.insert\ t\ A$

$\langle proof \rangle$

lemma *inter-Set* [code]:

$A \cap Set\ t = rbt-filter\ (\lambda k. k \in A)\ t$

$\langle proof \rangle$

lemma *minus-Set* [code]:

$A - Set\ t = fold-keys\ Set.remove\ t\ A$

$\langle proof \rangle$

lemma *union-Coset* [code]:

$Coset\ t \cup A = -\ rbt-filter\ (\lambda k. k \notin A)\ t$

$\langle proof \rangle$

lemma *union-Set-Set* [code]:

$Set\ t1 \cup Set\ t2 = Set\ (RBT.union\ t1\ t2)$

$\langle proof \rangle$

lemma *inter-Coset* [code]:

$A \cap Coset\ t = fold-keys\ Set.remove\ t\ A$

$\langle proof \rangle$

lemma *inter-Coset-Coset* [code]:

$Coset\ t1 \cap Coset\ t2 = Coset\ (RBT.union\ t1\ t2)$

$\langle proof \rangle$

lemma *minus-Coset* [code]:

$A - Coset\ t = rbt-filter\ (\lambda k. k \in A)\ t$

$\langle proof \rangle$

lemma *filter-Set* [code]:

$Set.filter\ P\ (Set\ t) = (rbt-filter\ P\ t)$

$\langle proof \rangle$

lemma *image-Set* [code]:

$image\ f\ (Set\ t) = fold-keys\ (\lambda k\ A.\ Set.insert\ (f\ k)\ A)\ t\ \{\}$

$\langle proof \rangle$

lemma *Ball-Set* [code]:

$Ball\ (Set\ t)\ P \longleftrightarrow RBT.foldi\ (\lambda s.\ s = True)\ (\lambda k\ v\ s.\ s \wedge P\ k)\ t\ True$

$\langle proof \rangle$

lemma *Bex-Set* [code]:

$Bex\ (Set\ t)\ P \longleftrightarrow RBT.foldi\ (\lambda s.\ s = False)\ (\lambda k\ v\ s.\ s \vee P\ k)\ t\ False$

$\langle proof \rangle$

lemma *subset-code* [code]:

$Set\ t \leq B \longleftrightarrow (\forall x \in Set\ t.\ x \in B)$

$A \leq Coset\ t \longleftrightarrow (\forall y \in Set\ t.\ y \notin A)$

$\langle proof \rangle$

lemma *subset-Coset-empty-Set-empty* [code]:

$Coset\ t1 \leq Set\ t2 \longleftrightarrow (case\ (RBT.impl-of\ t1,\ RBT.impl-of\ t2)\ of$

$(rbt.Empty,\ rbt.Empty) \Rightarrow False \mid$

$(-, -) \Rightarrow Code.abort\ (STR\ "non-empty-trees")\ (\lambda -. Coset\ t1 \leq Set\ t2))$

$\langle proof \rangle$

A frequent case – avoid intermediate sets

lemma [code-unfold]:

$Set\ t1 \subseteq Set\ t2 \longleftrightarrow RBT.foldi\ (\lambda s.\ s = True)\ (\lambda k\ v\ s.\ s \wedge k \in Set\ t2)\ t1\ True$

$\langle proof \rangle$

lemma *card-Set* [code]:

$card\ (Set\ t) = fold-keys\ (\lambda -. n.\ n + 1)\ t\ 0$

$\langle proof \rangle$

lemma *sum-Set* [code]:

$sum\ f\ (Set\ xs) = fold-keys\ (plus \circ f)\ xs\ 0$

$\langle proof \rangle$

lemma *the-elem-set* [code]:

fixes $t :: ('a :: linorder,\ unit)\ rbt$

shows $the-elem\ (Set\ t) = (case\ RBT.impl-of\ t\ of$

$(Branch\ RBT-Impl.B\ RBT-Impl.Empty\ x\ ())\ RBT-Impl.Empty) \Rightarrow x$

$\mid - \Rightarrow Code.abort\ (STR\ "not-a-singleton-tree")\ (\lambda -. the-elem\ (Set\ t))$

$\langle proof \rangle$

lemma *Pow-Set* [code]: $Pow\ (Set\ t) = fold-keys\ (\lambda x\ A.\ A \cup Set.insert\ x\ 'A)\ t$

$\{\{\}\}$

$\langle \text{proof} \rangle$

lemma *product-Set* [code]:

$\text{Product-Type.product } (\text{Set } t1) (\text{Set } t2) =$
 $\text{fold-keys } (\lambda x. A. \text{fold-keys } (\lambda y. \text{Set.insert } (x, y)) t2 A) t1 \{\}$
 $\langle \text{proof} \rangle$

lemma *Id-on-Set* [code]: $\text{Id-on } (\text{Set } t) = \text{fold-keys } (\lambda x. \text{Set.insert } (x, x)) t \{\}$
 $\langle \text{proof} \rangle$

lemma *Image-Set* [code]:

$(\text{Set } t) \text{ “} S = \text{fold-keys } (\lambda(x,y) A. \text{if } x \in S \text{ then } \text{Set.insert } y A \text{ else } A) t \{\}$
 $\langle \text{proof} \rangle$

lemma *trancl-set-ntrancl* [code]:

$\text{trancl } (\text{Set } t) = \text{ntrancl } (\text{card } (\text{Set } t) - 1) (\text{Set } t)$
 $\langle \text{proof} \rangle$

lemma *relcomp-Set*[code]:

$(\text{Set } t1) O (\text{Set } t2) = \text{fold-keys}$
 $(\lambda(x,y) A. \text{fold-keys } (\lambda(w,z) A'. \text{if } y = w \text{ then } \text{Set.insert } (x,z) A' \text{ else } A') t2 A)$
 $t1 \{\}$
 $\langle \text{proof} \rangle$

lemma *wf-set* [code]:

$\text{wf } (\text{Set } t) = \text{acyclic } (\text{Set } t)$
 $\langle \text{proof} \rangle$

lemma *Min-fin-set-fold* [code]:

$\text{Min } (\text{Set } t) =$
 $(\text{if } \text{RBT.is-empty } t$
 $\text{then } \text{Code.abort } (\text{STR "not-non-empty-tree"}) (\lambda-. \text{Min } (\text{Set } t))$
 $\text{else } \text{r-min-opt } t)$
 $\langle \text{proof} \rangle$

lemma *Inf-fin-set-fold* [code]:

$\text{Inf-fin } (\text{Set } t) = \text{Min } (\text{Set } t)$
 $\langle \text{proof} \rangle$

lemma *Inf-Set-fold*:

fixes $t :: ('a :: \{\text{linorder, complete-lattice}\}, \text{unit}) \text{rbt}$
shows $\text{Inf } (\text{Set } t) = (\text{if } \text{RBT.is-empty } t \text{ then } \text{top} \text{ else } \text{r-min-opt } t)$
 $\langle \text{proof} \rangle$

lemma *Max-fin-set-fold* [code]:

$\text{Max } (\text{Set } t) =$
 $(\text{if } \text{RBT.is-empty } t$
 $\text{then } \text{Code.abort } (\text{STR "not-non-empty-tree"}) (\lambda-. \text{Max } (\text{Set } t))$
 $\text{else } \text{r-max-opt } t)$

⟨proof⟩

lemma *Sup-fin-set-fold* [code]:
 $Sup\_fin (Set\ t) = Max (Set\ t)$
 ⟨proof⟩

lemma *Sup-Set-fold*:
fixes $t :: ('a :: \{linorder, complete-lattice\}, unit)\ rbt$
shows $Sup (Set\ t) = (if\ RBT.is-empty\ t\ then\ bot\ else\ r-max-opt\ t)$
 ⟨proof⟩

context
begin

declare [[code drop: $Gcd\_fin\ Lcm\_fin\ \langle Gcd :: - \Rightarrow nat \rangle\ \langle Gcd :: - \Rightarrow int \rangle\ \langle Lcm :: - \Rightarrow nat \rangle\ \langle Lcm :: - \Rightarrow int \rangle$]]

lemma [code]:
 $Gcd_{fin} (Set\ t) = fold-keys\ gcd\ t\ (0 :: 'a :: \{semiring-gcd, linorder\})$
 ⟨proof⟩

lemma [code]:
 $Gcd (Set\ t) = (Gcd_{fin} (Set\ t) :: nat)$
 ⟨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 drop: $Lcm :: - \Rightarrow nat, code$]:
 $Lcm (Set\ t) = (Lcm_{fin} (Set\ t) :: nat)$
 ⟨proof⟩

lemma [code drop: $Lcm :: - \Rightarrow int, code$]:
 $Lcm (Set\ t) = (Lcm_{fin} (Set\ t) :: int)$
 ⟨proof⟩ **definition** $Inf' :: 'a :: \{linorder, complete-lattice\}\ set \Rightarrow 'a$
where [code-abbrev]: $Inf' = Inf$

lemma *Inf'-Set-fold* [code]:
 $Inf' (Set\ t) = (if\ RBT.is-empty\ t\ then\ top\ else\ r-min-opt\ t)$
 ⟨proof⟩ **definition** $Sup' :: 'a :: \{linorder, complete-lattice\}\ set \Rightarrow 'a$
where [code-abbrev]: $Sup' = Sup$

lemma *Sup'-Set-fold* [code]:
 $Sup' (Set\ t) = (if\ RBT.is-empty\ t\ then\ bot\ else\ r-max-opt\ t)$

$\langle proof \rangle$

end

lemma *sorted-list-set*[*code*]: *sorted-list-of-set* (*Set t*) = *RBTree.keys t*
 $\langle proof \rangle$

lemma *Bleat-code* [*code*]:
Bleat (*Set t*) *P* =
 (case *List.filter P (RBTree.keys t)* of
 x # xs $\Rightarrow$ *x*
 | [] $\Rightarrow$ *abort-Bleat (Set t) P*)
 $\langle proof \rangle$

hide-const (**open**) *RBTree.Set RBTree.Coset*

end

$\langle ML \rangle$

theory *Predicate-Compile-Alternative-Defs*
imports *Main*
begin

127 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* $\Rightarrow$ *bool* $\Rightarrow$ *bool*) == ($\wedge$)
 $\langle proof \rangle$

lemma [*code-pred-inline*]:
 ((*A* :: *bool*) $\neq$ (*B* :: *bool*)) = ((*A* $\wedge$ $\neg$ *B*) $\vee$ (*B* $\wedge$ $\neg$ *A*))
 $\langle proof \rangle$

$\langle ML \rangle$

128 Pairs

$\langle ML \rangle$

129 Filters

$\langle ML \rangle$

130 Bounded quantifiers

declare *Ball-def*[code-pred-inline]
declare *Bex-def*[code-pred-inline]

131 Operations on Predicates

lemma *Diff*[code-pred-inline]:
 $(A - B) = (\%x. A\ x \wedge \neg B\ x)$
 $\langle proof \rangle$

lemma *subset-eq*[code-pred-inline]:
 $(P :: 'a \Rightarrow bool) < (Q :: 'a \Rightarrow bool) \equiv ((\exists x. Q\ x \wedge (\neg P\ x)) \wedge (\forall x. P\ x \longrightarrow Q\ x))$
 $\langle proof \rangle$

lemma *set-equality*[code-pred-inline]:
 $A = B \longleftrightarrow (\forall x. A\ x \longrightarrow B\ x) \wedge (\forall x. B\ x \longrightarrow A\ x)$
 $\langle proof \rangle$

132 Setup for Numerals

$\langle ML \rangle$

133 Arithmetic operations

133.1 Arithmetic on naturals and integers

definition *plus-eq-nat* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where
 $plus-eq-nat\ x\ y\ z = (x + y = z)$

definition *minus-eq-nat* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where
 $minus-eq-nat\ x\ y\ z = (x - y = z)$

definition *plus-eq-int* :: $int \Rightarrow int \Rightarrow int \Rightarrow bool$
where
 $plus-eq-int\ x\ y\ z = (x + y = z)$

definition *minus-eq-int* :: $int \Rightarrow int \Rightarrow int \Rightarrow bool$
where
 $minus-eq-int\ x\ y\ z = (x - y = z)$

definition *subtract*

where

[*code-unfold*]: $\text{subtract } x \ y = y - x$

$\langle ML \rangle$

133.2 Inductive definitions for ordering on naturals

inductive *less-nat*

where

$\text{less-nat } 0 \ (\text{Suc } y)$
 $| \text{less-nat } x \ y ==> \text{less-nat } (\text{Suc } x) \ (\text{Suc } y)$

lemma *less-nat*[*code-pred-inline*]:

$x < y = \text{less-nat } x \ y$

$\langle \text{proof} \rangle$

inductive *less-eq-nat*

where

$\text{less-eq-nat } 0 \ y$
 $| \text{less-eq-nat } x \ y ==> \text{less-eq-nat } (\text{Suc } x) \ (\text{Suc } y)$

lemma [*code-pred-inline*]:

$x \leq y = \text{less-eq-nat } x \ y$

$\langle \text{proof} \rangle$

134 Alternative list definitions

134.1 Alternative rules for *length*

definition *size-list'* :: 'a list => nat

where *size-list'* = *size*

lemma *size-list'-simps*:

$\text{size-list'} [] = 0$
 $\text{size-list'} (x \# xs) = \text{Suc } (\text{size-list'} xs)$
 $\langle \text{proof} \rangle$

declare *size-list'-simps*[*code-pred-def*]

declare *size-list'-def*[*symmetric*, *code-pred-inline*]

134.2 Alternative rules for *list-all2*

lemma *list-all2-NilI* [*code-pred-intro*]: $\text{list-all2 } P \ [] \ []$

$\langle \text{proof} \rangle$

lemma *list-all2-ConsI* [*code-pred-intro*]: $\text{list-all2 } P \ xs \ ys ==> P \ x \ y ==> \text{list-all2}$

$P \ (x \# xs) \ (y \# ys)$

$\langle \text{proof} \rangle$

```
code-pred [skip-proof] list-all2
  ⟨proof⟩
```

134.3 Alternative rules for membership in lists

```
declare in-set-member[code-pred-inline]
```

```
lemma member-intros [code-pred-intro]:
  List.member (x#xs) x
  List.member xs x  $\implies$  List.member (y#xs) x
  ⟨proof⟩
```

```
code-pred List.member
  ⟨proof⟩
```

```
code-identifier constant member-i-i
   $\rightarrow$  (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
   $\rightarrow$  (SML) List.member-i-o
  and (OCaml) List.member-i-o
  and (Haskell) List.member-i-o
  and (Scala) List.member-i-o
```

135 Setup for String.literal

⟨*ML*⟩

136 Simplification rules for optimisation

```
lemma [code-pred-simp]:  $\neg$  False == True
  ⟨proof⟩
```

```
lemma [code-pred-simp]:  $\neg$  True == False
  ⟨proof⟩
```

```
lemma less-nat-k-0 [code-pred-simp]: less-nat k 0 == False
  ⟨proof⟩
```

```
end
```

137 A Prototype of Quickcheck based on the Predicate Compiler

```
theory Predicate-Compile-Quickcheck
  imports Predicate-Compile-Alternative-Defs
begin
```

$\langle ML \rangle$

end

138 TFL: recursive function definitions

```
theory Old-Recdef
  imports Main
  keywords
    recdef :: thy-defn and
    permissive congs hints
begin
```

138.1 Lemmas for TFL

```
lemma tfl-wf-induct:  $\forall R. \text{wf } R \longrightarrow$   

 $(\forall P. (\forall x. (\forall y. (y,x) \in R \longrightarrow P y) \longrightarrow P x) \longrightarrow (\forall x. P x))$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-cut-def:  $\text{cut } f \ r \ x \equiv (\lambda y. \text{if } (y,x) \in r \text{ then } f \ y \text{ else undefined})$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-cut-apply:  $\forall f \ R. (x,a) \in R \longrightarrow (\text{cut } f \ R \ a)(x) = f(x)$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-wfrec:  

 $\forall M \ R \ f. (f = \text{wfrec } R \ M) \longrightarrow \text{wf } R \longrightarrow (\forall x. f \ x = M (\text{cut } f \ R \ x) \ x)$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-eq-True:  $(x = \text{True}) \longrightarrow x$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-rev-eq-mp:  $(x = y) \longrightarrow y \longrightarrow x$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-simp-thm:  $(x \longrightarrow y) \longrightarrow (x = x') \longrightarrow (x' \longrightarrow y)$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-P-imp-P-iff-True:  $P \Longrightarrow P = \text{True}$   

 $\langle \text{proof} \rangle$ 
```

```
lemma tfl-imp-trans:  $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$ 
```


$\langle proof \rangle$

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
 $\langle proof \rangle$

lemma *tfl-disjE*: $P \vee Q \implies P \longrightarrow R \implies Q \longrightarrow R \implies R$
 $\langle proof \rangle$

lemma *tfl-exE*: $\exists x. P\ x \implies \forall x. P\ x \longrightarrow Q \implies Q$
 $\langle proof \rangle$

$\langle ML \rangle$

138.2 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-empty

end

139 Program extraction from proofs involving datatypes and inductive predicates

theory *Realizers*
imports *Main*
begin

$\langle ML \rangle$

end

140 Refute

theory *Refute*

imports *Main*

keywords

refute :: *diag* **and**

refute-params :: *thy-decl*

begin

$\langle ML \rangle$

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] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.
- [2] A. Lochbihler and P. Stoop. Lazy algebraic types in Isabelle/HOL. In *Isabelle Workshop 2018*, 2018.
- [3] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.