

Contents

1 Library SLF.LibAxioms	2
1.1 Functional extensionality	2
1.2 Propositional extensionality	2
1.3 Indefinite description	3
2 Library SLF.LibTactics	4
2.1 Fixing Stdlib	5
2.2 Tools for programming with Ltac	5
2.2.1 Identity continuation	5
2.2.2 Untyped arguments for tactics	5
2.2.3 Optional arguments for tactics	5
2.2.4 Wildcard arguments for tactics	6
2.2.5 Position markers	6
2.2.6 List of arguments for tactics	7
2.2.7 Databases of lemmas	9
2.2.8 On-the-fly removal of hypotheses	10
2.2.9 Numbers as arguments	11
2.2.10 Testing tactics	12
2.2.11 Testing evars and non-evars	13
2.2.12 Check no evar in goal	13
2.2.13 Helper function for introducing evars	13
2.2.14 Tagging of hypotheses	14
2.2.15 Tagging of hypotheses	14
2.2.16 Deconstructing terms	14
2.2.17 Action at occurrence and action not at occurrence	15
2.2.18 An alias for <code>eq</code>	15
2.3 Common tactics for simplifying goals like <code>intuition</code>	16
2.4 Backward and forward chaining	16
2.4.1 Application	16
2.4.2 Assertions	19
2.4.3 Instantiation and forward-chaining	20
2.4.4 Experimental tactics for application	31
2.4.5 Adding assumptions	31

2.4.6	Application of tautologies	32
2.4.7	Application modulo equalities	32
2.4.8	Absurd goals	35
2.5	Introduction and generalization	37
2.5.1	Introduction using \Rightarrow	37
2.5.2	Introduction using \Rightarrow and \Rightarrow	39
2.5.3	Generalization	41
2.5.4	Naming	42
2.6	Rewriting	45
2.6.1	Renaming	47
2.6.2	Unfolding	48
2.6.3	Simplification	50
2.6.4	Evaluation	51
2.6.5	Substitution	51
2.6.6	Tactics to work with proof irrelevance	52
2.6.7	Proving equalities	53
2.7	Inversion	53
2.7.1	Basic inversion	53
2.7.2	Inversion with substitution	54
2.7.3	Injection with substitution	58
2.7.4	Inversion and injection with substitution –rough implementation	58
2.7.5	Case analysis	59
2.8	Induction	63
2.9	Coinduction	65
2.10	Decidable equality	66
2.11	Equivalence	66
2.12	N-ary Conjunctions and Disjunctions	66
2.13	Tactics to prove typeclass instances	72
2.14	Tactics to invoke automation	72
2.14.1	Definitions for parsing compatibility	72
2.14.2	<i>hint</i> to add hints local to a lemma	73
2.14.3	<i>jauto</i> , a new automation tactics	73
2.14.4	Definitions of automation tactics	74
2.14.5	Parsing for light automation	75
2.14.6	Parsing for strong automation	84
2.15	Tactics to sort out the proof context	92
2.15.1	Hiding hypotheses	92
2.15.2	Sorting hypotheses	95
2.15.3	Clearing hypotheses	95
2.16	Tactics for development purposes	97
2.16.1	Skipping subgoals	97
2.17	Compatibility with standard library	100

2.18	Additional notations for Coq	101
2.18.1	N-ary Existentials -TODO: DEPRECATED, Coq now supports it.	101
2.18.2	'let bindings (EXPERIMENTAL).	102
3	Library SLF.LibEqual	107
3.1	Definition of equality	107
3.1.1	Definition of Leibnitz' equality	107
3.1.2	Partial application of Leibnitz' equality	107
3.1.3	Typeclass to exploit extensionality	108
3.1.4	Tactic to exploit extensionality	108
3.2	Properties of equality	108
3.2.1	Equality as an equivalence relation	109
3.2.2	Properties of disequality	110
3.2.3	Symmetrized induction principles	110
3.3	Functional extensionality	110
3.3.1	Dependent functional extensionality	110
3.3.2	Non-dependent functional extensionality	112
3.3.3	Eta-conversion	113
3.4	Predicate extensionality	113
3.4.1	Dependend predicates	113
3.4.2	Non-dependend predicate extensionality	115
3.5	Equality of function and predicate applications	115
3.5.1	A same function applied to equal arguments yield equal result	115
3.5.2	Equal functions applied to same arguments return equal results	116
3.5.3	Equal predicates applied to same arguments return equivalent results	117
3.6	Proof Irrelevance	118
3.6.1	Proof of the proof-irrelevance result	118
3.6.2	Consequences of proof irrelevance	119
3.6.3	Injectivity of equality on dependent pairs	120
3.7	Dependent equality	122
3.8	John Major's equality	123
4	Library SLF.LibLogic	125
4.1	Strong existentials	125
4.2	Inhabited types	125
4.2.1	Typeclass for describing inhabited types	125
4.2.2	Tactics taking into account	126
4.2.3	Arbitrary values for inhabited types	126
4.2.4	Lemmas about inhabited types	126
4.3	Non-constructive conditionals	126
4.3.1	Excluded middle	126
4.3.2	Strong excluded middle	127
4.3.3	If-then-else on propositions	127

4.3.4	Consequences	128
4.3.5	Tactic for proving tautologies by bruteforce case-analysis	129
4.4	Properties of logical combinators	130
4.4.1	Simplification of conjunction and disjunction	130
4.4.2	Distribution of negation on basic operators	131
4.4.3	Double negation and contrapose	132
4.4.4	Distribution of negation on quantifiers	133
4.4.5	Propositions equal to True or False	134
4.4.6	Disequal propositions	135
4.4.7	Peirce rule and similar	135
4.4.8	Properties of logical equivalence	136
4.5	Tactics	138
4.5.1	Tactic for simplifying expressions	138
4.5.2	Tactic <i>tests</i> : P for classical disjunction on P	139
4.5.3	Tactic <i>absurds</i>	140
4.6	Predicate combinators, comparison and compatibility	141
4.6.1	Definition of predicate combinators	141
4.6.2	Properties of combinators	141
4.6.3	Order on predicates	141
4.7	Existentials	142
4.8	Properties of unique existentials	142
4.9	Conjunctions	143
4.9.1	Changing the order of branches	143
4.9.2	Parallel strengthening of a conjunction	144
4.9.3	Projections of lemmas concluding on a conjunction	144
5	Library SLF.LibOperation	148
5.1	Definitions	148
5.1.1	Commutativity, associativity	148
5.1.2	Distributivity	148
5.1.3	Neutral and absorbant	149
5.1.4	Inverses	150
5.1.5	Morphism and automorphism	150
5.1.6	Injectivity	150
5.2	Lemmas	150
6	Library SLF.LibBool	152
6.1	Boolean type	152
6.1.1	Definition	152
6.1.2	Inhabited	152
6.1.3	Extensionality	152
6.2	Boolean Operations	152
6.2.1	Comparison	152

6.3	Boolean Decision Procedure : tactic working by exponential case	154
6.3.1	Tactic <i>tautob</i>	154
6.4	Properties of boolean operators	154
6.4.1	Properties of <i>eqb</i>	154
6.4.2	Properties of <i>and</i>	155
6.4.3	Properties of <i>or</i>	155
6.4.4	Properties of <i>neg</i>	156
6.4.5	Properties of <i>if then else</i>	156
6.4.6	Properties of <i>impl</i> and <i>xor</i>	157
6.4.7	Opacity	157
6.5	Tactics	157
6.5.1	Tactic <i>rew_neg_neg</i>	157
6.5.2	Tactic <i>rew_bool</i>	158
7	Library SLF.LibReflect	160
7.1	Reflection between booleans and propositions	160
7.1.1	Translation from booleans into propositions	160
7.1.2	Translation from propositions into booleans	161
7.1.3	Extensionality for boolean equality, stated using <i>istrue</i>	162
7.1.4	Specification of boolean equality	162
7.2	Rewriting rules	162
7.2.1	Rewriting rules for distributing <i>istrue</i>	162
7.2.2	Rewriting rules for distributing <i>isTrue</i>	163
7.2.3	Lemmas for testing booleans	165
7.2.4	Lemmas for normalizing <i>b = true</i> and <i>b = false</i> terms	165
7.3	Tactics	166
7.3.1	Tactics <i>rew_istrue</i> to distribute <i>istrue</i>	166
7.3.2	Tactics <i>rew_isTrue</i> to distribute <i>isTrue</i>	166
7.3.3	Tactics useful for program verification, when reasoning about	167
7.3.4	Tactics extended for reflection	168
8	Library SLF.LibProd	169
8.1	Product type	169
8.1.1	Definition	169
8.1.2	Inhabited	169
8.2	Structure	169
8.3	Operations	170
8.3.1	Definition of projections	170
8.3.2	Notation for projections	170
8.3.3	Currying	171
8.3.4	Uncurrying	171
8.3.5	Uncurrying for relations	172
8.3.6	Inverse projections for relations	172

9 Library SLF.LibSum	175
9.1 Sum type	175
9.1.1 Definition	175
9.1.2 Inhabited	175
9.2 Operations	176
9.2.1 Testing the branch of the sum	176
9.2.2 Stripping of the branch tag	176
9.2.3 Lifting functions over sum types	177
10 Library SLF.LibRelation	178
10.1 Type of binary relations	178
10.1.1 Type of endorelation, i.e. homogeneous binary relations	178
10.1.2 Inhabited	178
10.1.3 Extensionality	178
10.2 Properties of relations	179
10.2.1 Reflexivity	179
10.2.2 Irreflexivity	179
10.2.3 Symmetry	180
10.2.4 Asymmetry	180
10.2.5 Antisymmetry	181
10.2.6 Antisymmetry with respect to an equivalence relation	181
10.2.7 Transitivity	181
10.2.8 Equivalence relation	183
10.2.9 Inclusion	183
10.2.10 Totality	183
10.2.11 Trichotomy	184
10.2.12 Definedness	185
10.2.13 Functionality	185
10.2.14 Criteria for equality	186
10.2.15 Properties of the equality relation	186
10.2.16 Properties of the equality relation	187
10.2.17 Properties of the equivalence relation	187
10.3 Basic constructions	187
10.3.1 The empty relation	187
10.3.2 Union of two relations	188
10.3.3 Intersection of two relations	189
10.3.4 Complement of a relation	189
10.3.5 Inverse of a relation	190
10.3.6 preimage	192
10.3.7 rel_seq	192
10.3.8 Turning a function into a relation	192
10.4 Products	192
10.4.1 Pointwise product	192

10.4.2 Lexicographical product	193
10.5 Closures	196
10.5.1 Reflexive closure	196
10.5.2 Symmetric closure	199
10.5.3 Reflexive-symmetric closure	201
10.5.4 Transitive closure (R^+), defined as $R \setminus \circ R \times$	203
10.5.5 Reflexive-transitive closure (R^*)	207
10.5.6 Symmetric-transitive closure	211
10.5.7 Reflexive-symmetric-transitive closure	213
10.5.8 Relationship between closures	215
10.5.9 Iterated closures	217
10.5.10 Other lemmas – TODO	219
10.5.11 Mixed transitivity between closures	220
10.5.12 Irreflexive restriction of a relation	221
10.5.13 Inclusion of a function in a relation	223
10.5.14 Inclusion of a relation in a function	224
11 Library <code>SLF.LibOrder</code>	225
11.1 Order relation upto an equivalence relation	227
11.2 Definition of order operators	233
11.2.1 Classes and notation for comparison operators	233
11.2.2 The operators <code>ge</code> , <code>lt</code> and <code>gt</code> are deduced from <code>le</code>	234
11.3 Classes for comparison properties	235
11.3.1 Definition of classes	235
11.4 Boolean comparison	246
11.5 Order on relations and on predicates	246
12 Library <code>SLF.LibNat</code>	248
12.1 Nat type	248
12.1.1 Definition	248
12.1.2 Inhabited	248
12.2 Order on natural numbers	248
12.2.1 Definition	248
12.2.2 Translating typeclass instances to Peano relations	249
12.2.3 The <code>nat_maths</code> database is used for registering automation	249
12.2.4 Total order instance	250
12.3 Induction	250
12.4 Simplification lemmas	251
12.4.1 Trivial monoid simplifications	251
12.4.2 Simplification tactic	252
12.5 Executable functions	252

13 Library <code>SLF.LibEpsilon</code>	254
13.1 Definition and specification of Hilbert's epsilon operator	254
13.1.1 Definition of epsilon	254
13.1.2 Lemmas about epsilon	255
13.1.3 (Private) tactic <i>epsilon_find</i>	255
13.1.4 Tactics <i>epsilon_name</i>	256
13.1.5 Tactics to work with <i>epsilon</i>	256
13.2 Construction of a function from a relation, using <i>epsilon</i>	257
14 Library <code>SLF.LibInt</code>	259
14.1 Parsing of integers and operations	259
14.1.1 Notation for type and operation	259
14.1.2 Inhabited type	259
14.1.3 Coercion from <i>nat</i>	259
14.1.4 Order relation	260
14.2 Conversion to natural numbers, for tactic programming	260
14.3 Decision procedure	260
14.3.1 Translation from typeclass order to ZArith, for using <i>lia</i>	261
14.3.2 Hypothesis selection	261
14.3.3 Normalization of arithmetic expressions	263
14.3.4 Setting up the goal for <i>lia</i>	263
14.3.5 Main driver for the set up process to goal <i>lia</i>	264
14.3.6 <i>math</i> tactic restricted to arithmetic goals	264
14.3.7 Elimination of multiplication, to call <i>lia</i>	264
14.3.8 Hint externs for calling <i>math</i> in the hint base <i>maths</i>	264
14.4 Rewriting on arithmetic expressions	266
14.4.1 Rewriting equalities provable by the <i>math</i> tactic	266
14.4.2 Addition and subtraction	266
14.4.3 Simplification tactic	267
14.5 Conversions of operations from nat to <i>int</i> and back	268
14.5.1 Lifting of comparisons from nat to <i>int</i>	268
14.5.2 Lifting of inequalities from nat to <i>int</i>	268
14.5.3 Lifting of operations from nat to <i>int</i>	269
14.5.4 Properties of comparison	269
14.5.5 Absolute function in nat	270
14.5.6 abs distribute on constants and operators	271
14.5.7 Tactic <i>rew_abs_nonneg</i> to normalize expressions involving abs	272
14.5.8 Positive part of an integer. Returns 0 on negative values.	272
14.5.9 to_nat distribute on constants and operators	274
14.5.10 Tactic <i>rew_to_nat_nonneg</i> to normalize expressions involving to_nat .	274

15 Library SLF.LibMonoid	275
15.1	275
15.2	275
15.3 Monoids	275
15.4 Structures	275
15.5 Examples	276
15.6 Properties	276
15.7 Derived Properties	276
16 Library SLF.LibContainer	278
16.1 Operators	278
16.1.1 Definitions	278
16.1.2 Notation	279
16.1.3 $\forall x \in E, P x$ notation	279
16.1.4 $\exists x \in E \text{ st } P x$ notation	280
16.1.5 Foreach	280
16.1.6 index for natural numbers	280
16.1.7 Derivable	281
16.1.8 Properties	281
16.1.9 Derived Properties	286
17 Library SLF.LibOption	297
17.1 Option type	297
17.1.1 Definition	297
17.1.2 Inhabited	297
17.2 Operations	297
17.2.1 is_some	297
17.2.2 unsome_default	298
17.2.3 unsome	298
17.2.4 map	298
17.2.5 map_on	299
17.2.6 apply	299
17.2.7 apply_on	299
18 Library SLF.LibWf	300
18.1 Compatibility	300
18.1.1 Tactics	300
18.1.2 Definition	300
18.1.3 Properties	301
18.1.4 Measure on pairs	301
18.2 Construction of well-founded relations	302
18.2.1 Empty relation	302
18.2.2 Inclusion	302

18.2.3	Peano.lt on nat	302
18.2.4	lt on nat	302
18.2.5	The relation “greater than” on the set of	303
18.2.6	The relation “less than” on the set of	303
18.2.7	The relation “greater than” on the set of	304
18.3	Inverse projections	304
18.4	Lexicographical product	306
18.5	Symmetric product	307
18.6	Well-foundedness of a function image	309
18.7	Transitive closure	309
19	Library SLF.LibList	310
19.1	Inhabited	310
19.2	Normalization tactics	310
19.2.1	<i>rew_list</i> for basic list properties	310
19.2.2	<i>rew_listx</i> for all other operations on lists	311
19.3	Properties of operations	312
19.3.1	Definitions of Fold-right and App	312
19.3.2	App	312
19.3.3	FoldRight	314
19.3.4	FoldLeft	314
19.3.5	Length	315
19.4	Inversion lemmas for structural composition	316
19.4.1	<i>nth_default</i> as a partial function with a default	327
19.4.2	<i>nth</i> as a partial function	329
19.4.3	Rev	332
19.4.4	Inversion for rev	335
19.5	Make	335
19.6	Update as a function	337
19.6.1		340
19.6.2	Concat	344
19.6.3	Filter	345
19.6.4	Remove	348
19.6.5	Combine	354
19.6.6	Split	355
19.6.7	Take	358
19.6.8	Drop	360
19.6.9	Take and drop decomposition of a list	362
19.6.10	TakeDropLast	363
19.7	Nat seq	380
19.8	Fold	380
20	Library SLF.LibListExec	386

21 Library SLF.LibListZ	389
21.1 List operations using indices in Z	389
21.2 Length, with result as <i>int</i>	389
21.3 Inversion lemmas for structural composition	391
21.4 <i>index</i> , with length as <i>int</i> , as typeclass	391
21.5 <i>read</i> , with length as <i>int</i> , as typeclass	392
21.6 Equality between two lists from equality of length and	394
21.7 <i>update</i> , with index as <i>int</i> , as typeclass	394
21.8 <i>make</i> , with length as <i>int</i>	397
21.9 <i>LibList.rev</i> interactions with <i>LibListZ</i> operations	399
21.10 <i>LibList.map</i> interactions with <i>LibListZ</i> operations	400
21.11 <i>LibList.filter</i> interactions with <i>LibListZ</i> operations	401
21.12 <i>LibList.remove</i> interactions with <i>LibListZ</i> operations	402
21.12.1 Take, with an <i>int</i> as the number of elements	402
21.12.2 Drop, with an <i>int</i> as the number of elements.	404
21.12.3 Take and drop decomposition of a list	406
21.12.4 <i>count</i> , returning an int	408
21.13 <i>card</i> , with result as nat , as typeclass	410
21.14 Normalization tactics	411
21.15 <i>binds</i> , with length as <i>int</i> , as typeclass	412
22 Library SLF.LibMin	413
23 Library SLF.LibSet	417
23.1 Construction of sets as predicates	417
23.1.1 Basic definitions	417
23.1.2 Notations to help the typechecker	418
23.1.3 Inhabited	419
23.1.4 Notation through typeclasses	419
23.1.5 Notations for building sets	420
23.2 Properties of sets	420
23.3 Tactics for proving set equalities and set inclusions	429
23.4 More properties	430
23.4.1 Structural properties	435
23.5 MORE	435
23.5.1 TEMPORARY Foreach	435
24 Library SLF.LibChoice	438
24.1 Functional choice	438
24.1.1 Functional choice	438
24.1.2 Dependent functional choice	438
24.1.3 Guarded functional choice	439
24.1.4 Omniscient functional choice	439

24.2	Functional unique choice	439
24.2.1	Functional unique choice	440
24.2.2	Dependent functional unique choice	440
24.2.3	Guarded functional unique choice	440
24.2.4	Omniscient functional unique choice	441
24.3	Relational choice	441
24.3.1	Relational choice	441
24.3.2	Guarded relational choice	441
24.3.3	Omniscient relation choice	442
25	Library SLF.LibUnit	443
25.1	Unit type	443
25.1.1	Definition	443
25.1.2	Inhabited	443
25.2	Properties	443
25.2.1	Uniqueness	443
26	Library SLF.LibFun	444
26.1	444
26.2	444
26.2.1	Identity function	444
26.2.2	Function update	446
26.2.3	Function image	446
26.2.4	Function preimage	448
26.2.5	Function iteration	448
27	Library SLF.LibString	449
27.1	Inhabited	449
28	Library SLF.LibMultiset	450
28.1	450
28.2	450
28.3	Construction of sets as predicates	450
28.3.1	Basic definitions	450
28.3.2	Notation through typeclasses	451
28.4	Properties of multisets	452
28.4.1	Structural properties	452
28.5	Additional predicates	453
28.5.1	Foreach	453
28.6	Tactics	455
28.6.1	Tactics to prove equalities on unions	455
28.6.2	Tactics to prove membership	460
28.6.3	Tactics to invert a membership hypothesis	462

29 Library SLF.LibCore	465
30 Library SLF.LibSepTLCbuffer	466
30.1 LibSepTLCbuffer: Appendix - Temporary Additions to TLC	466
31 Library SLF.LibSepFmap	467
31.1 LibSepFmap: Appendix - Finite Maps	467
31.2 Representation of Finite Maps	467
31.2.1 Representation of Potentially-Infinite Maps as Partial Functions	467
31.2.2 Representation of Finite Maps as Dependent Pairs	470
31.2.3 Predicates on Finite Maps	472
31.3 Properties of Operations on Finite Maps	472
31.3.1 Equality	472
31.3.2 Disjointness	473
31.3.3 Union	474
31.3.4 Compatibility	476
31.3.5 Domain	479
31.3.6 Disjoint and Domain	479
31.3.7 Read	480
31.3.8 Update	481
31.3.9 Removal	481
31.4 Tactics for Finite Maps	482
31.4.1 Tactic <i>disjoint</i> for proving disjointness results	482
31.4.2 Tactic <i>rew_map</i> for Normalizing Expressions	483
31.4.3 Tactic <i>fmap_eq</i> for Proving Equalities	483
31.5 Existence of Fresh Locations	486
31.5.1 Consecutive Locations	486
31.5.2 Existence of Fresh Locations	486
31.5.3 Existence of a Smallest Fresh Locations	488
31.5.4 Existence of Nonempty Heaps	489
32 Library SLF.LibSepVar	490
32.1 LibSepVar: Appendix - Program Variables	490
32.2 Representation of Program Variables	490
32.2.1 Representation of Variables	490
32.2.2 Tactic <i>case_var</i>	491
32.3 Representation of List of Variables	491
32.3.1 Definition of Distinct Variables	491
32.3.2 Definition of n Distinct Variables	492
32.3.3 Generation of n Distinct Variables	493
32.4 Notation for Program Variables	494
32.4.1 Program Variables Expressed Using Definitions	495
32.4.2 Program Variables Expressed Using Notation, with a Quote Symbol .	498

32.5 Optional Material	500
32.5.1 Bonuse: the Tactic <i>var_neq</i>	500
33 Library SLF.LibSepSimpl	502
33.1 LibSepSimpl: Appendix - Simplification of Entailments	502
33.2 A Functor for Separation Logic Entailment	502
33.3 Assumptions of the functor	503
33.3.1 Operators	503
33.3.2 Notation	503
33.3.3 Properties Assumed by the Functor	504
33.4 Body of the Functor	506
33.4.1 Properties of <i>himpl</i>	507
33.4.2 Properties of <i>qimpl</i>	507
33.4.3 Properties of <i>hstar</i>	507
33.4.4 Representation Predicates	508
33.4.5 <i>rew_heap</i> Tactic to Normalize Expressions with <i>hstar</i>	508
33.4.6 Auxiliary Tactics Used by <i>xpull</i> and <i>xsimpl</i>	509
33.4.7 Tactics <i>xsimpl</i> and <i>xpull</i> for Heap Entailments	510
33.5 Demos	534
34 Library SLF.LibSepMinimal	542
34.1 LibSepMinimal: Appendix - Minimalistic Soundness Proof	542
34.2 Source Language	542
34.2.1 Syntax	542
34.2.2 Semantics	544
34.2.3 Automation for Heap Equality and Heap Disjointness	545
34.3 Heap Predicates and Entailment	545
34.3.1 Extensionality Axioms	545
34.3.2 Core Heap Predicates	546
34.3.3 Entailment	547
34.3.4 Properties of <i>hstar</i>	548
34.3.5 Properties of <i>hpure</i>	549
34.3.6 Properties of <i>hexists</i>	550
34.3.7 Properties of <i>hforall</i>	550
34.3.8 Properties of <i>hsingle</i>	551
34.3.9 Basic Tactics for Simplifying Entailments	551
34.3.10 Reformulation of the Evaluation Rules for Primitive Operations.	552
34.4 Hoare Logic	553
34.4.1 Definition of Total Correctness Hoare Triples	553
34.4.2 Structural Rules for Hoare Triples	553
34.4.3 Reasoning Rules for Terms, for Hoare Triples	553
34.4.4 Specification of Primitive Operations, for Hoare Triples.	554
34.5 Separation Logic	556

34.5.1	Definition of Separation Logic triples	556
34.5.2	Structural Rules	556
34.5.3	Reasoning Rules for Terms	557
34.5.4	Specification of Primitive Operations	558
34.6	Bonus: Example Proof	559
35	Library SLF.LibSepReference	560
35.1	LibSepReference: Appendix - The Full Construction	560
35.2	Imports	560
35.2.1	Extensionality Axioms	560
35.2.2	Variables	561
35.2.3	Finite Maps	561
35.3	Source Language	561
35.3.1	Syntax	561
35.3.2	Coq Tweaks	563
35.3.3	Substitution	563
35.3.4	Semantics	564
35.4	Heap Predicates	566
35.4.1	Hprop and Entailment	566
35.4.2	Definition of Heap Predicates	567
35.4.3	Basic Properties of Separation Logic Operators	569
35.4.4	Properties of <code>haffine</code>	580
35.5	Reasoning Rules	581
35.5.1	Evaluation Rules for Primitives in Separation Style	581
35.5.2	Hoare Reasoning Rules	582
35.6	Definition of total correctness Hoare Triples.	582
35.6.1	Definition of Separation Logic Triples.	587
35.6.2	Structural Rules	588
35.6.3	Rules for Terms	590
35.6.4	Triple-Style Specification for Primitive Functions	592
35.6.5	Alternative Definition of <code>wp</code>	595
35.7	WP Generator	598
35.7.1	Definition of Context as List of Bindings	598
35.7.2	Multi-Substitution	600
35.7.3	Definition of <code>wpgen</code>	603
35.8	Practical Proofs	607
35.8.1	Lemmas for Tactics to Manipulate <code>wpgen</code> Formulae	608
35.8.2	Tactics to Manipulate <code>wpgen</code> Formulae	609
35.8.3	Notations for Triples and <code>wpgen</code>	612
35.8.4	Notation for Concrete Terms	615
35.8.5	Scopes, Coercions and Notations for Concrete Programs	618
35.9	Bonus	618
35.9.1	Disjunction: Definition and Properties of <code>hor</code>	618

35.9.2	Conjunction: Definition and Properties of hand	619
35.9.3	Treatment of Functions of 2 and 3 Arguments	620
35.9.4	Bonus: Triples for Applications to Several Arguments	625
35.9.5	Specification of Record Operations	625
35.10	Demo Programs	630
35.10.1	The Decrement Function	631
36	Library SLF.Preface	635
36.1	Preface: Introduction to the Course	635
36.2	Welcome	635
36.3	Separation Logic	636
36.4	Separation Logic in a proof assistant	636
36.5	Several Possible Depths of Reading	637
36.6	List of Chapters	638
36.7	Other Distributed Files	639
36.8	Practicalities	640
36.8.1	System Requirements	640
36.8.2	Note for CoqIDE Users	640
36.8.3	Feedback Welcome	640
36.8.4	Exercises	641
36.8.5	Recommended Citation Format	641
36.9	Thanks	641
37	Library SLF.Basic	642
37.1	Basic: Basic Proofs in Separation Logic	642
37.2	A First Taste	642
37.2.1	Parsing of Programs	642
37.2.2	Specification of the Increment Function	643
37.2.3	Verification of the Increment Function	644
37.2.4	A Function with a Return Value	645
37.3	Separation Logic Operators	647
37.3.1	Increment of Two References	647
37.3.2	Aliased Arguments	648
37.3.3	A Function that Takes Two References and Increments One	649
37.3.4	Transfer from one Reference to Another	650
37.3.5	Specification of Allocation	651
37.3.6	Allocation of a Reference with Greater Contents	652
37.3.7	Deallocation in Separation Logic	653
37.3.8	Combined Reading and Freeing of a Reference	655
37.4	Recursive Functions	655
37.4.1	Axiomatization of the Mathematical Factorial Function	655
37.4.2	A Partial Recursive Function, Without State	656
37.4.3	A Recursive Function with State	658

37.4.4	Trying to Prove Incorrect Specifications	659
37.4.5	A Recursive Function Involving two References	660
37.5	Summary	661
37.6	Historical Notes	662
38	Library SLF.Repr	663
38.1	Repr: Representation Predicates	663
38.2	First Pass	663
38.2.1	Formalization of the List Representation Predicate	664
38.2.2	Alternative Characterizations of <code>MList</code>	664
38.2.3	In-place Concatenation of Two Mutable Lists	666
38.2.4	Smart Constructors for Linked Lists	667
38.2.5	Copy Function for Lists	668
38.2.6	Length Function for Lists	669
38.2.7	Alternative Length Function for Lists	669
38.2.8	In-Place Reversal Function for Lists	670
38.3	More Details	671
38.3.1	Sized Stack	671
38.3.2	Formalization of the Tree Representation Predicate	674
38.3.3	Alternative Characterization of <code>MTree</code>	675
38.3.4	Additional Tooling for <code>MTree</code>	676
38.3.5	Tree Copy	677
38.3.6	Computing the Sum of the Items in a Tree	677
38.3.7	Verification of a Counter Function with Local State	678
38.3.8	Verification of a Counter Function with Local State, With Abstraction	680
38.4	Optional Material	681
38.4.1	Specification of a Higher-Order Repeat Operator	681
38.4.2	Specification of an Iterator on Mutable Lists	682
38.4.3	Computing the Length of a Mutable List using an Iterator	683
38.4.4	A Continuation-Passing-Style, In-Place Concatenation Function	684
38.4.5	Historical Notes	685
39	Library SLF.Hprop	686
39.1	Hprop: Heap Predicates	686
39.2	First Pass	686
39.2.1	Syntax and Semantics	688
39.2.2	Description of the State	689
39.2.3	Heap Predicates	689
39.2.4	Extensionality for Heap Predicates	691
39.2.5	Type and Syntax for Postconditions	691
39.2.6	Separation Logic Triples and the Frame Rule	692
39.2.7	Example of a Triple: the Increment Function	693
39.3	More Details	693

39.3.1	Example Applications of the Frame Rule	693
39.3.2	Power of the Frame Rule with Respect to Allocation	694
39.3.3	Notation for Heap Union	694
39.3.4	Introduction and Inversion Lemmas for Basic Operators	695
39.4	Optional Material	696
39.4.1	Alternative, Equivalent Definitions for Separation Logic Triples	696
39.4.2	Alternative Definitions for Heap Predicates	697
39.4.3	Additional Explanations for the Definition of \exists	698
39.4.4	Formulation of the Extensionality Axioms	698
39.4.5	Historical Notes	699
40	Library SLF.Himpl	700
40.1	Himpl: Heap Entailment	700
40.2	First Pass	700
40.2.1	Definition of Entailment	701
40.2.2	Entailment for Postconditions	702
40.2.3	Fundamental Properties of Separation Logic Operators	703
40.2.4	Introduction and Elimination Rules w.r.t. Entailments	704
40.2.5	Extracting Information from Heap Predicates	706
40.2.6	Consequence, Frame, and their Combination	706
40.2.7	The Extraction Rules for Triples	707
40.3	More Details	707
40.3.1	Identifying True and False Entailments	708
40.3.2	Proving Entailments by Hand	708
40.3.3	The <i>xsimp</i> Tactic	709
40.3.4	The <i>xchange</i> Tactic	713
40.4	Optional Material	714
40.4.1	Proofs for the Separation Algebra	714
40.4.2	Proof of the Consequence Rule	717
40.4.3	Proof of the Extraction Rules for Triples	718
40.4.4	Alternative Structural Rule for Existentials	720
40.4.5	Historical Notes	721
41	Library SLF.Rules	722
41.1	Rules: Reasoning Rules	722
41.2	First Pass	722
41.2.1	Semantic of Terms	723
41.2.2	Rules for Terms	729
41.2.3	Specification of Primitive Operations	732
41.2.4	Review of the Structural Rules	734
41.2.5	Verification Proof in Separation Logic	736
41.2.6	What's Next	739
41.3	More Details	739

41.3.1	Alternative Specification Style for Pure Preconditions	739
41.3.2	The Combined let-frame Rule Rule	740
41.3.3	Proofs for the Rules for Terms	741
41.3.4	Proofs for the Arithmetic Primitive Operations	743
41.3.5	Proofs for Primitive Operations Operating on the State	744
41.4	Optional Material	748
41.4.1	Reasoning Rules for Recursive Functions	748
41.4.2	Alternative Specification Style for Result Values.	756
41.4.3	Historical Notes	756
42	Library SLF.WPsem	758
42.1	WPsem: Semantics of Weakest Preconditions	758
42.2	First Pass	758
42.2.1	Notion of Weakest Precondition	759
42.2.2	Structural Rules in Weakest-Precondition Style	760
42.2.3	Reasoning Rules for Terms, in Weakest-Precondition Style	762
42.3	More Details	763
42.3.1	Other Reasoning Rules for Terms	763
42.4	Optional Material	765
42.4.1	A Concrete Definition for Weakest Precondition	765
42.4.2	Equivalence Between all Definitions Of <code>wp</code>	766
42.4.3	An Alternative Definition for Weakest Precondition	766
42.4.4	Extraction Rule for Existentials	767
42.4.5	Combined Structural Rule	767
42.4.6	Alternative Statement of the Rule for Conditionals	768
42.4.7	Definition of <code>wp</code> Directly from <code>hoare</code>	769
42.4.8	Historical Notes	771
43	Library SLF.WPgen	772
43.1	WPgen: Weakest Precondition Generator	772
43.2	First Pass	772
43.3	More Details	775
43.3.1	Definition of <code>wpgen</code> for Term Rules	775
43.3.2	Computing with <code>wpgen</code>	779
43.3.3	Optimizing the Readability of <code>wpgen</code> Output	784
43.3.4	Extension of <code>wpgen</code> to Handle Structural Rules	787
43.3.5	Lemmas for Handling <code>wpgen</code> Goals	791
43.3.6	An Example Proof	793
43.3.7	Making Proof Scripts More Concise	794
43.3.8	Further Improvements to the <code>xapp</code> Tactic.	795
43.3.9	Demo of a Practical Proof using x-Tactics.	796
43.4	Optional Material	797
43.4.1	Tactics <code>xconseq</code> and <code>xframe</code>	797

43.4.2	Soundness Proof for <code>wpgen</code>	798
43.4.3	Guessing the Definition of <code>mkstruct</code>	804
43.4.4	Proof of Properties of Iterated Substitution	805
43.4.5	Historical Notes	809
44	Library SLF.Wand	811
44.1	Wand: The Magic Wand Operator	811
44.2	First Pass	811
44.2.1	Intuition for the Magic Wand	812
44.2.2	Definition of the Magic Wand	812
44.2.3	Characteristic Property of the Magic Wand	813
44.2.4	Magic Wand for Postconditions	814
44.2.5	Frame Expressed with <code>hwand</code> : the Ramified Frame Rule	815
44.2.6	Ramified Frame Rule in Weakest-Precondition Style	816
44.2.7	Automation with <code>xsimpl</code> for <code>hwand</code> Expressions	817
44.2.8	Evaluation of <code>wpgen</code> Recursively in Locally Defined Functions	818
44.3	More Details	825
44.3.1	Benefits of the Ramified Rule over the Consequence-Frame Rule	825
44.3.2	Properties of <code>hwand</code>	827
44.4	Optional Material	830
44.4.1	Equivalence Between Alternative Definitions of the Magic Wand	830
44.4.2	Operator <code>hforall</code>	832
44.4.3	Alternative Definition of <code>qwand</code>	834
44.4.4	Equivalence between Alternative Definitions of the Magic Wand	835
44.4.5	Simplified Definition of <code>mkstruct</code>	837
44.4.6	Texan Triples	838
44.4.7	Direct Proof of <code>wp_ramified</code> Directly from Hoare Triples	843
44.4.8	Conjunction and Disjunction Operators on <code>hprop</code>	844
44.4.9	Summary of All Separation Logic Operators	847
44.4.10	Historical Notes	849
45	Library SLF.Affine	850
45.1	Affine: Affine Separation Logic	850
45.2	First Pass	850
45.2.1	Motivation for the Discard Rule	851
45.2.2	Statement of the Discard Rule	852
45.2.3	Fine-grained Control on Collectable Predicates	853
45.2.4	Definition of <code>heap_affine</code> and of <code>haffine</code>	854
45.2.5	Definition of the “Affine Top” Heap Predicates	855
45.2.6	Properties of the <code>\GC</code> Predicate	857
45.2.7	Instantiation of <code>heap_affine</code> for a Fully Affine Logic	858
45.2.8	Instantiation of <code>heap_affine</code> for a Fully Linear Logic	859
45.2.9	Refined Definition of Separation Logic Triples	860

45.2.10	Soundness of the Existing Rules	860
45.2.11	Soundness of the Discard Rules	862
45.2.12	Discard Rules in WP Style	863
45.2.13	Exploiting the Discard Rule in Proofs	864
45.2.14	Example Proof Involving Discarded Heap Predicates	865
45.3	More Details	866
45.3.1	Revised Definition of <code>mkstruct</code>	866
45.3.2	The Tactic <code>xaffine</code> , and Behavior of <code>xsimpl</code> on <code>\GC</code>	867
45.3.3	The Proof Tactics for Applying the Discard Rules	868
45.4	Optional Material	870
45.4.1	Alternative Statement for Distribution of <code>haffine</code> on Quantifiers	870
45.4.2	Pre and Post Rules	871
45.4.3	Low-level Definition of Refined Triples	871
45.4.4	Historical Notes	873
46	Library <code>SLF.Struct</code>	874
46.1	Struct: Arrays and Records	874
46.2	First Pass	874
46.2.1	Representation of a Set of Consecutive Cells	875
46.2.2	Representation of an Array with a Block Header	875
46.2.3	Specification of Allocation	876
46.2.4	Specification of the Deallocation	877
46.2.5	Specification of Array Operations	877
46.2.6	Representation of Individual Records Fields	878
46.2.7	Representation of Records	879
46.2.8	Example with Mutable Linked Lists	881
46.2.9	Reading in Record Fields	882
46.2.10	Writing in Record Fields	883
46.2.11	Allocation of Records	884
46.2.12	Deallocation of Records	885
46.2.13	Combined Record Allocation and Initialization	886
46.3	More Details	887
46.3.1	Extending <code>xapp</code> to Support Record Access Operations	887
46.3.2	Deallocation Function for Lists	888
46.4	Optional Material	889
46.4.1	Refined Source Language	889
46.4.2	Realization of <code>hheader</code>	890
46.4.3	Introduction and Elimination Lemmas for <code>hcells</code> and <code>harray</code>	891
46.4.4	Proving the Specification of Allocation and Deallocation	892
46.4.5	Splitting Lemmas for <code>hcells</code>	894
46.4.6	Specification of Pointer Arithmetic	896
46.4.7	Specification of the <code>length</code> Operation to Read the Header	896
46.4.8	Encoding of Array Operations using Pointer Arithmetic	897

46.4.9	Encoding of Record Operations using Pointer Arithmetic	899
46.4.10	Specification of Record Operations w.r.t. <code>hfields</code> and <code>hrecord</code>	900
46.4.11	Specification of Record Allocation and Deallocation	902
47 Library <code>SLF.Rich</code>		905
47.1	Rich: Assertions, Loops, N-ary Functions	905
47.2	First Pass	905
47.2.1	Reasoning Rule for Assertions	906
47.2.2	Semantics of Conditionals not in Administrative Normal Form	908
47.2.3	Semantics and Basic Evaluation Rules for While-Loops	909
47.2.4	Separation-Logic-friendly Reasoning Rule for While-Loops	910
47.3	More Details	913
47.3.1	Semantics and Basic Evaluation Rules for For-Loops	913
47.3.2	Treatment of Generalized Conditionals and Loops in <code>wpgen</code>	915
47.3.3	Notation and Tactics for Manipulating While-Loops	917
47.3.4	Example of the Application of Frame During Loop Iterations	918
47.3.5	Reasoning Rule for Loops in an Affine Logic	919
47.3.6	Curried Functions of Several Arguments	920
47.3.7	Primitive n-ary Functions	923
47.4	Optional Material	927
47.4.1	A Coercion for Parsing Primitive N-ary Applications	927
47.4.2	Historical Notes	929
48 Library <code>SLF.Nondet</code>		930
48.1	Nondet: Triples for Nondeterministic Languages	930
48.2	First Pass	930
48.2.1	Non-Deterministic Big-Step Semantics: the Predicate <code>evaln</code>	931
48.2.2	Interpretation of <code>evaln</code> w.r.t. <code>eval</code>	935
48.2.3	Triples for Non-Deterministic Big-Step Semantics	937
48.2.4	Triple-Style Reasoning Rules for a Non-Deterministic Semantics.	938
48.3	More Details	941
48.3.1	Weakest-Precondition Style Presentation.	941
48.3.2	Hoare Logic WP-Style Rules for a Non-Deterministic Semantics.	942
48.3.3	Separation Logic WP-Style Rules for a Non-Deterministic Semantics.	945
48.4	Optional Material	949
48.4.1	Interpretation of <code>evaln</code> w.r.t. <code>eval</code> and <code>terminates</code>	949
48.4.2	Small-Step Evaluation Relation	951
48.4.3	Small-Step Characterization of <code>evalns</code> : Attempts	952
48.4.4	Small-Step Characterization of <code>evaln</code> : A Solution	955
48.4.5	Triples for Small-Step Semantics	957
48.4.6	Reasoning Rules for <code>seval</code>	957
48.4.7	Reasoning Rules for <code>hoarens</code>	958
48.4.8	Equivalence Between Non-Deterministic Small-Step and Big-Step Sem.	962

48.4.9 Historical Notes	965
49 Library SLF.Partial	966
49.1 Partial: Triples for Partial Correctness	966
49.2 First Pass	966
49.2.1 Big-Step Characterization of Partial Correctness	967
49.2.2 From Total to Partial Correctness	969
49.2.3 Characterization of Divergence	969
49.2.4 Big-Step-Based Reasoning Rules	969
49.2.5 Big-Step-Based Reasoning Rules for Divergence	972
49.3 Optional Material	973
49.3.1 Interpretation of evalnp w.r.t. eval and safe	973
49.3.2 Small-Step Characterization of Partial Correctness	975
49.3.3 Reasoning Rules w.r.t. Small-Step Characterization	977
49.3.4 Small-Step Characterization of Divergence	978
49.3.5 Coinductive, Small-Step Characterization of Partial Correctness	979
49.3.6 Equivalence Between the Two Small-Step Charact. of Partial Correctness	981
49.3.7 Equivalence Between Small-Step and Big-Step Partial Correctness	982
49.3.8 Historical Notes	987
50 Library SLF.Postscript	988
50.1 Postscript: Conclusion and Perspectives	988
50.2 Beyond the Scope of This Course	988
50.3 Historical Notes	988
50.4 Tools Leveraging Separation Logic	989
50.5 Related Courses	990
50.6 Acknowledgments	990
51 Library SLF.Bib	991
51.1 Bib: Bibliography	991
51.2 Resources cited in this volume	991

Chapter 1

Library SLF.LibAxioms

Set Implicit Arguments.

This file is used to extend Coq with standard axioms from classical, non-constructive higher-order logic. (Such axioms are provided by default in other theorem provers like Isabelle/HOL or HOL4.)

Three axioms taken are: functional extensionality, and propositional extensionality and indefinite description.

All other common axioms are derivable, including: the excluded middle, the strong excluded middle, propositional degeneracy, proof irrelevance, injectivity of equality on dependent pairs, predicate extensionality, definite description, and all the versions of the axiom of choice.

1.1 Functional extensionality

Two functions that yield equal results on equal arguments are equal.

Axiom $\text{fun_ext_dep} : \forall (A : \text{Type}) (B : A \rightarrow \text{Type}) (f g : \forall x, B x),$
 $(\forall x, f x = g x) \rightarrow$
 $f = g.$

1.2 Propositional extensionality

Two propositions that are equivalent can be considered to be equal.

Axiom $\text{prop_ext} : \forall (P Q : \text{Prop}),$
 $(P \leftrightarrow Q) \rightarrow$
 $P = Q.$

1.3 Indefinite description

Proofs of existence can be reified from Prop to Type. The lemma below is a concise statement for $(\exists x, P x) \rightarrow \{ x : A \mid P x \}$.

Axiom *indefinite_description* : $\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}),$

ex $P \rightarrow$

sig $P.$

Chapter 2

Library `SLF.LibTactics`

This file contains a set of tactics that extends the set of builtin tactics provided with the standard distribution of Coq. It intends to overcome a number of limitations of the standard set of tactics, and thereby to help user to write shorter and more robust scripts.

Hopefully, Coq tactics will be improved as time goes by, and this file should ultimately be useless. In the meanwhile, serious Coq users will probably find it very useful.

The main features offered are:

- More convenient syntax for naming hypotheses, with tactics for introduction and inversion that take as input only the name of hypotheses of type `Prop`, rather than the name of all variables.
- Tactics providing true support for manipulating N-ary conjunctions, disjunctions and existentials, hiding the fact that the underlying implementation is based on binary propositions.
- Convenient support for automation: tactic followed with the symbol “`~`” or “`*`” will call automation on the generated subgoals. The symbol “`~`” stands for `auto` and “`*`” for `intuition eauto`. These bindings can be customized.
- Forward-chaining tactics are provided to instantiate lemmas either with variable or hypotheses or a mix of both.
- A more powerful implementation of `apply` is provided (it is based on `refine` and thus behaves better with respect to conversion).
- An improved inversion tactic which substitutes equalities on variables generated by the standard inversion mechanism. Moreover, it supports the elimination of dependently typed equalities (requires axiom K , which is a weak form of Proof Irrelevance).
- Tactics for saving time when writing proofs, with tactics to asserts hypotheses or subgoals, and improved tactics for clearing, renaming, and sorting hypotheses.

External credits:

- thanks to Xavier Leroy for providing the idea of tactic *forward*,
- thanks to Georges Gonthier for the implementation trick in *rapply*,

Set Implicit Arguments.

Require Import Coq.Lists.List.

Declare Scope ltac_scope.

2.1 Fixing Stdlib

Remove Hints Bool.trans_eq_bool : core.

2.2 Tools for programming with Ltac

2.2.1 Identity continuation

```
Ltac idcont tt :=  
  idtac.
```

2.2.2 Untyped arguments for tactics

Any Coq value can be boxed into the type **Boxer**. This is useful to use Coq computations for implementing tactics.

```
Inductive Boxer : Type :=  
  | boxer : ∀ (A:Type), A → Boxer.
```

2.2.3 Optional arguments for tactics

`ltac_no_arg` is a constant that can be used to simulate optional arguments in tactic definitions. Use *mytactic* `ltac_no_arg` on the tactic invocation, and use `match arg with ltac_no_arg ⇒ ..` or `match type of arg with ltac_no_arg ⇒ ..` to test whether an argument was provided.

```
Inductive Ltac_No_arg : Set :=  
  | ltac_no_arg : Ltac_No_arg.
```

2.2.4 Wildcard arguments for tactics

`Itac_wild` is a constant that can be used to simulate wildcard arguments in tactic definitions. Notation is `__`.

```
Inductive Itac_Wild : Set :=
| Itac_wild : Itac_Wild.
```

```
Notation "'__'" := Itac_wild : ltac_scope.
```

`Itac_wilds` is another constant that is typically used to simulate a sequence of N wildcards, with N chosen appropriately depending on the context. Notation is `___`.

```
Inductive Itac_Wilds : Set :=
| Itac_wilds : Itac_Wilds.
```

```
Notation "'___'" := Itac_wilds : ltac_scope.
```

```
Open Scope ltac_scope.
```

2.2.5 Position markers

`Itac_Mark` and `Itac_mark` are dummy definitions used as sentinel by tactics, to mark a certain position in the context or in the goal.

```
Inductive Itac_Mark : Type :=
| Itac_mark : Itac_Mark.
```

`gen_until_mark` repeats `generalize` on hypotheses from the context, starting from the bottom and stopping as soon as reaching an hypothesis of type `Mark`. If fails if `Mark` does not appear in the context.

```
Ltac gen_until_mark :=
match goal with H: ?T ⊢ _ ⇒
  match T with
  | Itac_Mark ⇒ clear H
  | _ ⇒ generalize H; clear H; gen_until_mark
  end end.
```

`gen_until_mark_with_processing F` is similar to `gen_until_mark` except that it calls `F` on each hypothesis immediately before generalizing it. This is useful for processing the hypotheses.

```
Ltac gen_until_mark_with_processing cont :=
match goal with H: ?T ⊢ _ ⇒
  match T with
  | Itac_Mark ⇒ clear H
  | _ ⇒ cont H; generalize H; clear H;
        gen_until_mark_with_processing cont
  end end.
```

`intro_until_mark` repeats `intro` until reaching an hypothesis of type *Mark*. It throws away the hypothesis *Mark*. It fails if *Mark* does not appear as an hypothesis in the goal.

```
Ltac intro_until_mark :=
  match goal with
  | ⊢ (ltac_mark → _) ⇒ intros _
  | _ ⇒ intro; intro_until_mark
  end.
```

2.2.6 List of arguments for tactics

A datatype of type `list Boxer` is used to manipulate list of Coq values in ltac. Notation is `» v1 v2 ... vN` for building a list containing the values *v1* through *vN*.

Notation "»" :=

```
(@nil Boxer)
(at level 0)
: ltac_scope.
```

Notation "»' v1" :=

```
((boxer v1) :: nil)
(at level 0, v1 at level 0)
: ltac_scope.
```

Notation "»' v1 v2" :=

```
((boxer v1) :: (boxer v2) :: nil)
(at level 0, v1 at level 0, v2 at level 0)
: ltac_scope.
```

Notation "»' v1 v2 v3" :=

```
((boxer v1) :: (boxer v2) :: (boxer v3) :: nil)
(at level 0, v1 at level 0, v2 at level 0, v3 at level 0)
: ltac_scope.
```

Notation "»' v1 v2 v3 v4" :=

```
((boxer v1) :: (boxer v2) :: (boxer v3) :: (boxer v4) :: nil)
(at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
v4 at level 0)
: ltac_scope.
```

Notation "»' v1 v2 v3 v4 v5" :=

```
((boxer v1) :: (boxer v2) :: (boxer v3) :: (boxer v4) :: (boxer v5) :: nil)
(at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
v4 at level 0, v5 at level 0)
: ltac_scope.
```

Notation "»' v1 v2 v3 v4 v5 v6" :=

```
((boxer v1) :: (boxer v2) :: (boxer v3) :: (boxer v4) :: (boxer v5)
  :: (boxer v6) :: nil)
(at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
```

```

 $v_4$  at level 0,  $v_5$  at level 0,  $v_6$  at level 0)
: ltac_scope.

Notation "'>' v1 v2 v3 v4 v5 v6 v7" :=
((boxer  $v_1$ ) :: (boxer  $v_2$ ) :: (boxer  $v_3$ ) :: (boxer  $v_4$ ) :: (boxer  $v_5$ )
 :: (boxer  $v_6$ ) :: (boxer  $v_7$ ) :: nil)
(at level 0,  $v_1$  at level 0,  $v_2$  at level 0,  $v_3$  at level 0,
 $v_4$  at level 0,  $v_5$  at level 0,  $v_6$  at level 0,  $v_7$  at level 0)
: ltac_scope.

Notation "'>' v1 v2 v3 v4 v5 v6 v7 v8" :=
((boxer  $v_1$ ) :: (boxer  $v_2$ ) :: (boxer  $v_3$ ) :: (boxer  $v_4$ ) :: (boxer  $v_5$ )
 :: (boxer  $v_6$ ) :: (boxer  $v_7$ ) :: (boxer  $v_8$ ) :: nil)
(at level 0,  $v_1$  at level 0,  $v_2$  at level 0,  $v_3$  at level 0,
 $v_4$  at level 0,  $v_5$  at level 0,  $v_6$  at level 0,  $v_7$  at level 0,
 $v_8$  at level 0)
: ltac_scope.

Notation "'>' v1 v2 v3 v4 v5 v6 v7 v8 v9" :=
((boxer  $v_1$ ) :: (boxer  $v_2$ ) :: (boxer  $v_3$ ) :: (boxer  $v_4$ ) :: (boxer  $v_5$ )
 :: (boxer  $v_6$ ) :: (boxer  $v_7$ ) :: (boxer  $v_8$ ) :: (boxer  $v_9$ ) :: nil)
(at level 0,  $v_1$  at level 0,  $v_2$  at level 0,  $v_3$  at level 0,
 $v_4$  at level 0,  $v_5$  at level 0,  $v_6$  at level 0,  $v_7$  at level 0,
 $v_8$  at level 0,  $v_9$  at level 0)
: ltac_scope.

Notation "'>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10" :=
((boxer  $v_1$ ) :: (boxer  $v_2$ ) :: (boxer  $v_3$ ) :: (boxer  $v_4$ ) :: (boxer  $v_5$ )
 :: (boxer  $v_6$ ) :: (boxer  $v_7$ ) :: (boxer  $v_8$ ) :: (boxer  $v_9$ ) :: (boxer  $v_{10}$ ) :: nil)
(at level 0,  $v_1$  at level 0,  $v_2$  at level 0,  $v_3$  at level 0,
 $v_4$  at level 0,  $v_5$  at level 0,  $v_6$  at level 0,  $v_7$  at level 0,
 $v_8$  at level 0,  $v_9$  at level 0,  $v_{10}$  at level 0)
: ltac_scope.

Notation "'>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11" :=
((boxer  $v_1$ ) :: (boxer  $v_2$ ) :: (boxer  $v_3$ ) :: (boxer  $v_4$ ) :: (boxer  $v_5$ )
 :: (boxer  $v_6$ ) :: (boxer  $v_7$ ) :: (boxer  $v_8$ ) :: (boxer  $v_9$ ) :: (boxer  $v_{10}$ )
 :: (boxer  $v_{11}$ ) :: nil)
(at level 0,  $v_1$  at level 0,  $v_2$  at level 0,  $v_3$  at level 0,
 $v_4$  at level 0,  $v_5$  at level 0,  $v_6$  at level 0,  $v_7$  at level 0,
 $v_8$  at level 0,  $v_9$  at level 0,  $v_{10}$  at level 0,  $v_{11}$  at level 0)
: ltac_scope.

Notation "'>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12" :=
((boxer  $v_1$ ) :: (boxer  $v_2$ ) :: (boxer  $v_3$ ) :: (boxer  $v_4$ ) :: (boxer  $v_5$ )
 :: (boxer  $v_6$ ) :: (boxer  $v_7$ ) :: (boxer  $v_8$ ) :: (boxer  $v_9$ ) :: (boxer  $v_{10}$ )
 :: (boxer  $v_{11}$ ) :: (boxer  $v_{12}$ ) :: nil)
(at level 0,  $v_1$  at level 0,  $v_2$  at level 0,  $v_3$  at level 0,
```

```

 $v_4$  at level 0,  $v_5$  at level 0,  $v_6$  at level 0,  $v_7$  at level 0,  

 $v_8$  at level 0,  $v_9$  at level 0,  $v_{10}$  at level 0,  $v_{11}$  at level 0,  

 $v_{12}$  at level 0)
: ltac_scope.

Notation "'>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13" :=
((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
 ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
 ::(boxer v11)::(boxer v12)::(boxer v13)::nil)
(at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
v8 at level 0, v9 at level 0, v10 at level 0, v11 at level 0,
v12 at level 0, v13 at level 0)
: ltac_scope.

```

The tactic *list_boxer_of* inputs a term E and returns a term of type “list boxer”, according to the following rules:

- if E is already of type “list Boxer”, then it returns E ;
- otherwise, it returns the list (boxer E)::nil.

```

Ltac list_boxer_of E :=
match type of E with
| List.list Boxer  $\Rightarrow$  constr:(E)
| _  $\Rightarrow$  constr:(boxer E)::nil
end.

```

2.2.7 Databases of lemmas

Use the hint facility to implement a database mapping terms to terms. To declare a new database, use a definition: **Definition** *mydatabase* := **True**.

Then, to map *mykey* to *myvalue*, write the hint: **Hint Extern 1 (Register mydatabase mykey) \Rightarrow Provide myvalue.**

Finally, to query the value associated with a key, run the tactic *ltac_database_get mydatabase mykey*. This will leave at the head of the goal the term *myvalue*. It can then be named and exploited using *intro*.

Inductive **Ltac_database_token** : Prop := ltac_database_token.

Definition ltac_database ($D:\text{Boxer}$) ($T:\text{Boxer}$) ($A:\text{Boxer}$) := Ltac_database_token.

```

Notation "'Register' D T" := (ltac_database (boxer D) (boxer T) _)
(at level 69, D at level 0, T at level 0).

```

```

Lemma ltac_database_provide :  $\forall (A:\text{Boxer}) (D:\text{Boxer}) (T:\text{Boxer}),$ 
ltac_database D T A.

```

Proof using. split. Qed.

```

Ltac Provide  $T :=$  apply (@ltac_database_provide (boxer  $T$ )).

Ltac ltac_database_get  $D\ T :=$ 
  let  $A :=$  fresh "TEMP" in evar ( $A:\text{Boxer}$ );
  let  $H :=$  fresh "TEMP" in
  assert ( $H : \text{ltac\_database}(\text{boxer } D) (\text{boxer } T) A$ );
  [ subst  $A$ ; auto
  | subst  $A$ ; match type of  $H$  with ltac_database _ _ (boxer ? $L$ )  $\Rightarrow$ 
    generalize  $L$  end; clear  $H$  ].
```

2.2.8 On-the-fly removal of hypotheses

In a list of arguments $\gg H_1\ H_2\ \dots\ H_N$ passed to a tactic such as *lets* or *applys* or *forwards* or *specializes*, the term **rm**, an identity function, can be placed in front of the name of an hypothesis to be deleted.

Definition $\text{rm } (A:\text{Type})\ (X:A) := X$.

$\text{rm_term } E$ removes one hypothesis that admits the same type as E .

```

Ltac rm_term  $E :=$ 
  let  $T :=$  type of  $E$  in
  match goal with  $H: T \vdash \_ \Rightarrow$  try clear  $H$  end.
```

$\text{rm_inside } E$ calls $\text{rm_term } E_i$ for any subterm of the form **rm** E_i found in E

```

Ltac rm_inside  $E :=$ 
  let  $go\ E := \text{rm\_inside } E$  in
  match  $E$  with
  | rm ? $X \Rightarrow$   $\text{rm\_term } X$ 
  | ? $X_1\ ?X_2 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ 
  | ? $X_1\ ?X_2\ ?X_3 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ ; go  $X_3$ 
  | ? $X_1\ ?X_2\ ?X_3\ ?X_4 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ ; go  $X_3$ ; go  $X_4$ 
  | ? $X_1\ ?X_2\ ?X_3\ ?X_4\ ?X_5 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ ; go  $X_3$ ; go  $X_4$ ; go  $X_5$ 
  | ? $X_1\ ?X_2\ ?X_3\ ?X_4\ ?X_5\ ?X_6 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ ; go  $X_3$ ; go  $X_4$ ; go  $X_5$ ; go  $X_6$ 
  | ? $X_1\ ?X_2\ ?X_3\ ?X_4\ ?X_5\ ?X_6\ ?X_7 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ ; go  $X_3$ ; go  $X_4$ ; go  $X_5$ ; go  $X_6$ ; go  $X_7$ 
  | ? $X_1\ ?X_2\ ?X_3\ ?X_4\ ?X_5\ ?X_6\ ?X_7\ ?X_8 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ ; go  $X_3$ ; go  $X_4$ ; go  $X_5$ ; go  $X_6$ ; go  $X_7$ ; go  $X_8$ 
  | ? $X_1\ ?X_2\ ?X_3\ ?X_4\ ?X_5\ ?X_6\ ?X_7\ ?X_8\ ?X_9 \Rightarrow$ 
    go  $X_1$ ; go  $X_2$ ; go  $X_3$ ; go  $X_4$ ; go  $X_5$ ; go  $X_6$ ; go  $X_7$ ; go  $X_8$ ; go  $X_9$ 
  | ? $X_1\ ?X_2\ ?X_3\ ?X_4\ ?X_5\ ?X_6\ ?X_7\ ?X_8\ ?X_9\ ?X_{10} \Rightarrow$ 
```

```

    go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8; go X9; go X10
| _ ⇒ idtac
end.

```

For faster performance, one may deactivate *rm_inside* by replacing the body of this definition with *idtac*.

```
Ltac fast_rm_inside E :=
  rm_inside E.
```

2.2.9 Numbers as arguments

When tactic takes a natural number as argument, it may be parsed either as a natural number or as a relative number. In order for tactics to convert their arguments into natural numbers, we provide a conversion tactic.

Note: the tactic *number_to_nat* is extended in *LibInt* to take into account the *Z* type.

```
Require Coq.Numbers.BinNums Coq.ZArith.BinInt.
```

```
Definition ltac_int_to_nat (x:BinInt.Z) : nat :=
  match x with
  | BinInt.Z0 ⇒ 0%nat
  | BinInt.Zpos p ⇒ BinPos.nat_of_P p
  | BinInt.Zneg p ⇒ 0%nat
  end.
```

```
Ltac number_to_nat N :=
  match type of N with
  | nat ⇒ constr:(N)
  | BinInt.Z ⇒ let N' := constr:(ltac_int_to_nat N) in eval compute in N'
  end.
```

ltac_pattern E at K is the same as pattern *E at K* except that *K* is a Coq number (nat or Z) rather than a Ltac integer. Syntax *ltac_pattern E as K in H* is also available.

```
Tactic Notation "ltac_pattern" constr(E) "at" constr(K) :=
  match number_to_nat K with
  | 1 ⇒ pattern E at 1
  | 2 ⇒ pattern E at 2
  | 3 ⇒ pattern E at 3
  | 4 ⇒ pattern E at 4
  | 5 ⇒ pattern E at 5
  | 6 ⇒ pattern E at 6
  | 7 ⇒ pattern E at 7
  | 8 ⇒ pattern E at 8
  | _ ⇒ fail "ltac_pattern: arity not supported"
  end.
```

```

Tactic Notation "ltac_pattern" constr(E) "at" constr(K) "in" hyp(H) :=
  match number_to_nat K with
  | 1 => pattern E at 1 in H
  | 2 => pattern E at 2 in H
  | 3 => pattern E at 3 in H
  | 4 => pattern E at 4 in H
  | 5 => pattern E at 5 in H
  | 6 => pattern E at 6 in H
  | 7 => pattern E at 7 in H
  | 8 => pattern E at 8 in H
  | _ => fail "ltac_pattern: arity not supported"
  end.

```

ltac_set ($x := E$) at K is the same as *set* ($x := E$) at K except that K is a Coq number (nat or Z) rather than a Ltac integer.

```

Tactic Notation "ltac_set" "(" ident(X) ":"= constr(E) ")" "at" constr(K) :=
  match number_to_nat K with
  | 1%nat => set (X := E) at 1
  | 2%nat => set (X := E) at 2
  | 3%nat => set (X := E) at 3
  | 4%nat => set (X := E) at 4
  | 5%nat => set (X := E) at 5
  | 6%nat => set (X := E) at 6
  | 7%nat => set (X := E) at 7
  | 8%nat => set (X := E) at 8
  | 9%nat => set (X := E) at 9
  | 10%nat => set (X := E) at 10
  | 11%nat => set (X := E) at 11
  | 12%nat => set (X := E) at 12
  | 13%nat => set (X := E) at 13
  | _ => fail "ltac_set: arity not supported"
  end.

```

2.2.10 Testing tactics

show tac executes a tactic *tac* that produces a result, and then display its result.

```

Tactic Notation "show" tactic(tac) :=
  let R := tac in pose R.

```

dup N produces N copies of the current goal. It is useful for building examples on which to illustrate behaviour of tactics. *dup* is short for *dup 2*.

Lemma dup_lemma : $\forall P, P \rightarrow P \rightarrow P$.

Proof using. auto. Qed.

```

Ltac dup_tactic N :=
  match number_to_nat N with
  | 0 => idtac
  | S 0 => idtac
  | S ?N' => apply dup_lemma; [ | dup_tactic N' ]
  end.

```

```

Tactic Notation "dup" constr(N) :=
  dup_tactic N.

```

```

Tactic Notation "dup" :=
  dup 2.

```

2.2.11 Testing evars and non-evars

is_not_evar E succeeds only if *E* is not an evar; it fails otherwise. It thus implements the negation of *is_evar*

```

Ltac is_not_evar E :=
  first [ is_evar E; fail 1
    | idtac ].
```

is_evar_as_bool E evaluates to **true** if *E* is an evar and to **false** otherwise.

```

Ltac is_evar_as_bool E :=
  constr:(ltac:(first
    [ is_evar E; exact true
    | exact false ])).
```

has_no_evar E succeeds if *E* contains no evars.

```

Ltac has_no_evar E :=
  first [ has_evar E; fail 1 | idtac ].
```

2.2.12 Check no evar in goal

```

Ltac check_noevar M :=
  first [ has_evar M; fail 2 | idtac ].
```

```

Ltac check_noevar_hyp H :=
  let T := type of H in check_noevar T.
```

```

Ltac check_noevar_goal :=
  match goal with ⊢ ?G ⇒ check_noevar G end.
```

2.2.13 Helper function for introducing evars

with_evar T (fun M ⇒ tac) creates a new evar that can be used in the tactic *tac* under the name *M*.

```

Ltac with_evar_base T cont :=
  let x := fresh "TEMP" in evar (x:T); cont x; subst x.

Tactic Notation "with_evar" constr(T) tactic(cont) :=
  with_evar_base T cont.

```

2.2.14 Tagging of hypotheses

get_last_hyp tt is a function that returns the last hypothesis at the bottom of the context. It is useful to obtain the default name associated with the hypothesis, e.g. `intro; let H := get_last_hyp tt in let H' := fresh "P" H in ...`

```

Ltac get_last_hyp tt :=
  match goal with H: _ ⊢ _ ⇒ constr:(H) end.

```

2.2.15 Tagging of hypotheses

`ltac_tag_subst` is a specific marker for hypotheses which is used to tag hypotheses that are equalities to be substituted.

Definition `ltac_tag_subst (A:Type) (x:A) := x.`

`ltac_to_generalize` is a specific marker for hypotheses to be generalized.

Definition `ltac_to_generalize (A:Type) (x:A) := x.`

```

Ltac gen_to_generalize :=
  repeat match goal with
    H: ltac_to_generalize _ ⊢ _ ⇒ generalize H; clear H end.

```

```

Ltac mark_to_generalize H :=
  let T := type of H in
  change T with (ltac_to_generalize T) in H.

```

2.2.16 Deconstructing terms

`E` is a tactic that returns the head constant of the term `E`, ie, when applied to a term of the form `P x1 ... xN` it returns `P`. If `E` is not an application, it returns `E`. Warning: the tactic seems to loop in some cases when the goal is a product and one uses the result of this function.

```

Ltac get_head E :=
  match E with
  | ?E' ?x ⇒ get_head E'
  | _ ⇒ constr:(E)
  end.

```

`get_fun_arg E` is a tactic that decomposes an application term `E`, ie, when applied to a term of the form `X1 ... XN` it returns a pair made of `X1 .. X(N-1)` and `XN`.

```

Ltac get_fun_arg E :=
  match E with
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X => constr:(X1 X2 X3 X4 X5 X6 X7, X)
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X => constr:(X1 X2 X3 X4 X5 X6, X)
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X => constr:(X1 X2 X3 X4 X5, X)
  | ?X1 ?X2 ?X3 ?X4 ?X => constr:(X1 X2 X3 X4, X)
  | ?X1 ?X2 ?X3 ?X => constr:(X1 X2 X3, X)
  | ?X1 ?X2 ?X => constr:(X1 X2, X)
  | ?X1 ?X => constr:(X1, X)
  end.

```

2.2.17 Action at occurence and action not at occurence

ltac_action_at K of E do Tac isolates the *K*-th occurence of *E* in the goal, setting it in the form *P E* for some named pattern *P*, then calls tactic *Tac*, and finally unfolds *P*. Syntax *ltac_action_at K of E in H do Tac* is also available.

```

Tactic Notation "ltac_action_at" constr(K) "of" constr(E) "do" tactic(Tac) :=
  let p := fresh "TEMP" in ltac_pattern E at K;
  match goal with ⊢ ?P _ => set (p:=P) end;
  Tac; unfold p; clear p.

```

```

Tactic Notation "ltac_action_at" constr(K) "of" constr(E) "in" hyp(H) "do" tactic(Tac) :=
  let p := fresh "TEMP" in ltac_pattern E at K in H;
  match type of H with ?P _ => set (p:=P) in H end;
  Tac; unfold p in H; clear p.

```

protects E do Tac temporarily assigns a name to the expression *E* so that the execution of tactic *Tac* will not modify *E*. This is useful for instance to restrict the action of `simpl`.

```

Tactic Notation "protects" constr(E) "do" tactic(Tac) :=

```

```

  let x := fresh "TEMP" in let H := fresh "TEMP" in
  set (X := E) in *; assert (H : X = E) by reflexivity;
  clearbody X; Tac; subst x.

```

```

Tactic Notation "protects" constr(E) "do" tactic(Tac) "/" :=
  protects E do Tac.

```

2.2.18 An alias for `eq`

`eq'` is an alias for `eq` to be used for equalities in inductive definitions, so that they don't get mixed with equalities generated by `inversion`.

```

Definition eq' := @eq.

```

```

Hint Unfold eq' : core.

```

```
Notation "x '==' y" := (@eq' _ x y)
  (at level 70, y at next level).
```

2.3 Common tactics for simplifying goals like intuition

```
Ltac jauto_set_hyps :=
  repeat match goal with H: ?T ⊢ _ ⇒
    match T with
    | _ ∧ _ ⇒ destruct H
    | ∃ a, _ ⇒ destruct H
    | _ ⇒ generalize H; clear H
    end
  end.
Ltac jauto_set_goal :=
  repeat match goal with
  | ⊢ ∃ a, _ ⇒ esplit
  | ⊢ _ ∧ _ ⇒ split
  end.
Ltac jauto_set :=
  intros; jauto_set_hyps;
  intros; jauto_set_goal;
  unfold not in *.
```

2.4 Backward and forward chaining

2.4.1 Application

```
Ltac old_refine f :=
  refine f.
```

rapply is a tactic similar to *eapply* except that it is based on the *refine* tactics, and thus is strictly more powerful (at least in theory :). In short, it is able to perform on-the-fly conversions when required for arguments to match, and it is able to instantiate existentials when required.

```
Tactic Notation "rapply" constr(t) :=
  first
  [ eexact (@t)
  | old_refine (@t)
  | old_refine (@t _)
  | old_refine (@t _ _)
  | old_refine (@t _ _ _)]
```

```

| old_refine (@t _ _ _ _)
| old_refine (@t _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
| old_refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
].

```

No-typeclass refine apply, TEMPORARY for Coq < 8.11.

Ltac *napply H* :=

```

first
[ notypeclasses refine (H)
| notypeclasses refine (H _)
| notypeclasses refine (H _ _)
| notypeclasses refine (H _ _ _)
| notypeclasses refine (H _ _ _ _)
| notypeclasses refine (H _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _ _ _ _ _ _ _).
]
```

The tactics *applys-N T*, where *N* is a natural number, provides a more efficient way of using *applys T*. It avoids trying out all possible arities, by specifying explicitly the arity of function *T*.

```
Tactic Notation "rapply_0" constr(t) :=
  old_refine (@t).
```

```
Tactic Notation "rapply_1" constr(t) :=
  old_refine (@t _).
```

```
Tactic Notation "rapply_2" constr(t) :=
  old_refine (@t _ _).
```

```
Tactic Notation "rapply_3" constr(t) :=
```

```

old_refine (@t _ _ _).
Tactic Notation "rapply_4" constr(t) :=
  old_refine (@t _ _ _ _).
Tactic Notation "rapply_5" constr(t) :=
  old_refine (@t _ _ _ _ _).
Tactic Notation "rapply_6" constr(t) :=
  old_refine (@t _ _ _ _ _ _).
Tactic Notation "rapply_7" constr(t) :=
  old_refine (@t _ _ _ _ _ _ _).
Tactic Notation "rapply_8" constr(t) :=
  old_refine (@t _ _ _ _ _ _ _ _).
Tactic Notation "rapply_9" constr(t) :=
  old_refine (@t _ _ _ _ _ _ _ _ _).
Tactic Notation "rapply_10" constr(t) :=
  old_refine (@t _ _ _ _ _ _ _ _ _ _).

```

lets_base H E adds an hypothesis $H : T$ to the context, where T is the type of term E . If H is an introduction pattern, it will destruct H according to the pattern.

Ltac *lets_base I E* := generalize E ; intros I .

applys_to H E transform the type of hypothesis H by replacing it by the result of the application of the term E to H . Intuitively, it is equivalent to *lets H: (E H)*.

```

Tactic Notation "applys_to" hyp(H) constr(E) :=
  let  $H'$  := fresh "TEMP" in rename  $H$  into  $H'$ ;
  (first [ lets_base  $H$  ( $E H'$ )
    | lets_base  $H$  ( $E - H'$ )
    | lets_base  $H$  ( $E - - H'$ )
    | lets_base  $H$  ( $E - - - H'$ )
    | lets_base  $H$  ( $E - - - - H'$ )
    | lets_base  $H$  ( $E - - - - - H'$ )
    | lets_base  $H$  ( $E - - - - - - H'$ )
    | lets_base  $H$  ( $E - - - - - - - H'$ )
    | lets_base  $H$  ( $E - - - - - - - - H'$ ) ])
  ); clear  $H'$ .

```

applys_to H1,...,HN E applies E to several hypotheses

```

Tactic Notation "applys_to" hyp(H1) "," hyp(H2) constr(E) :=
  applics_to  $H1 E$ ; applics_to  $H2 E$ .
Tactic Notation "applys_to" hyp(H1) "," hyp(H2) "," hyp(H3) constr(E) :=
  applics_to  $H1 E$ ; applics_to  $H2 E$ ; applics_to  $H3 E$ .
Tactic Notation "applys_to" hyp(H1) "," hyp(H2) "," hyp(H3) "," hyp(H4) constr(E)
  :=
  applics_to  $H1 E$ ; applics_to  $H2 E$ ; applics_to  $H3 E$ ; applics_to  $H4 E$ .

```

constructors calls `constructor` or `econstructor`.

```
Tactic Notation "constructors" :=
  first [ constructor | econstructor ]; unfold eq'.
```

2.4.2 Assertions

asserts H: T is another syntax for `assert (H : T)`, which also works with introduction patterns. For instance, one can write: *asserts* $\backslash[x P\backslash] (\exists n, n = 3)$, or *asserts* $\backslash[H|H\backslash] (n = 0 \vee n = 1)$.

```
Tactic Notation "asserts" simple_intropattern(I) ":" constr(T) :=
  let H := fresh "TEMP" in assert (H : T);
  [| generalize H; clear H; intros I ].
```

asserts H1 .. HN: T is a shorthand for *asserts* $\backslash[H1 \backslash[H2 \backslash[\dots HN\backslash]\backslash]\backslash]$: T .

```
Tactic Notation "asserts" simple_intropattern(I1)
simple_intropattern(I2) ":" constr(T) :=
  asserts [I1 I2]: T.
```

```
Tactic Notation "asserts" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  asserts [I1 [I2 I3]]: T.
```

```
Tactic Notation "asserts" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4) ":" constr(T) :=
  asserts [I1 [I2 [I3 I4]]]: T.
```

```
Tactic Notation "asserts" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
  asserts [I1 [I2 [I3 [I4 I5]]]]: T.
```

```
Tactic Notation "asserts" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) ":" constr(T) :=
  asserts [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.
```

asserts: T is *asserts H*: T with H being chosen automatically.

```
Tactic Notation "asserts" ":" constr(T) :=
  let H := fresh "TEMP" in asserts H : T.
```

cuts H: T is the same as *asserts H*: T except that the two subgoals generated are swapped: the subgoal T comes second. Note that contrary to `cut`, it introduces the hypothesis.

```
Tactic Notation "cuts" simple_intropattern(I) ":" constr(T) :=
  cut (T); [ intros I | idtac ].
```

cuts: T is *cuts* H : T with H being chosen automatically.

```

Tactic Notation "cuts" ":" constr(T) :=
  let H := fresh "TEMP" in cuts H: T.

  cuts H1 .. HN: T is a shorthand for cuts \[H1 \[H2 \[.. HN\]\]\]: T.

Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(T) :=
  cuts [I1 I2]: T.

Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  cuts [I1 [I2 I3]]: T.

Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) ":" constr(T) :=
  cuts [I1 [I2 [I3 I4]]]: T.

Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
  cuts [I1 [I2 [I3 [I4 I5]]]]: T.

Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) ":" constr(T) :=
  cuts [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.

```

2.4.3 Instantiation and forward-chaining

The instantiation tactics are used to instantiate a lemma E (whose type is a product) on some arguments. The type of E is made of implications and universal quantifications, e.g. $\forall x, P x \rightarrow \forall y z, Q x y z \rightarrow R z$.

The first possibility is to provide arguments in order: first x , then a proof of $P x$, then y etc... In this mode, called “Args”, all the arguments are to be provided. If a wildcard is provided (written $_$), then an existential variable will be introduced in place of the argument.

It is very convenient to give some arguments the lemma should be instantiated on, and let the tactic find out automatically where underscores should be inserted. Underscore arguments $_$ are interpreted as follows: an underscore means that we want to skip the argument that has the same type as the next real argument provided (real means not an underscore). If there is no real argument after underscore, then the underscore is used for the first possible argument.

The general syntax is $tactic (\> E1 .. EN)$ where *tactic* is the name of the tactic (possibly with some arguments) and Ei are the arguments. Moreover, some tactics accept the syntax *tactic E1 .. EN* as short for *tactic (\> E1 .. EN)* for values of N up to 5.

Finally, if the argument EN given is a triple-underscore $---$, then it is equivalent to providing a list of wildcards, with the appropriate number of wildcards. This means that all the remaining arguments of the lemma will be instantiated. Definitions in the conclusion are not unfolded in this case.

```

Ltac app_assert t P cont :=
  let H := fresh "TEMP" in
  assert (H : P); [ | cont(t H); clear H ].

Ltac app_evar t A cont :=
  let x := fresh "TEMP" in
  evar (x:A);
  let t' := constr:(t x) in
  let t'' := (eval unfold x in t') in
  subst x; cont t''.

Ltac app_arg t P v cont :=
  let H := fresh "TEMP" in
  assert (H : P); [ apply v | cont(t H); try clear H ].

Ltac build_app_alls t final :=
  let rec go t :=
    match type of t with
    | ?P → ?Q ⇒ app_assert t P go
    | ∀ _:?A, _ ⇒ app_evar t A go
    | _ ⇒ final t
    end in
  go t.

Ltac boxerlist_next_type vs :=
  match vs with
  | nil ⇒ constr:(ltac_wild)
  | (boxer ltac_wild) :: ?vs' ⇒ boxerlist_next_type vs'
  | (boxer ltac_wilds) :: _ ⇒ constr:(ltac_wild)
  | (@boxer ?T _) :: _ ⇒ constr:(T)
  end.

Ltac build_app_hnts t vs final :=
  let rec go t vs :=
    match vs with
    | nil ⇒ first [ final t | fail 1 ]
    | (boxer ltac_wilds) :: _ ⇒ first [ build_app_alls t final | fail 1 ]
    | (boxer ?v) :: ?vs' ⇒
        let cont t' := go t' vs in
        let cont' t' := go t' vs' in
        let T := type of t in

```

```

let T := eval hnf in T in
match v with
| ltac_wild =>
  first [ let U := boxerlist_next_type vs' in
    match U with
    | ltac_wild =>
      match T with
      | ?P → ?Q => first [ app_assert t P cont' | fail 3 ]
      | ∀ _:?A, _ => first [ app_evar t A cont' | fail 3 ]
      end
    | _ =>
      match T with
      | U → ?Q => first [ app_assert t U cont' | fail 3 ]
      | ∀ _:U, _ => first [ app_evar t U cont' | fail 3 ]
      | ?P → ?Q => first [ app_assert t P cont | fail 3 ]
      | ∀ _:?A, _ => first [ app_evar t A cont | fail 3 ]
      end
    end
  | fail 2 ]
| _ =>
  match T with
  | ?P → ?Q => first [ app_arg t P v cont'
    | app_assert t P cont
    | fail 3 ]
  | ∀ _:Type, _ =>
    match type of v with
    | Type => first [ cont' (t v)
      | app_evar t Type cont
      | fail 3 ]
    | _ => first [ app_evar t Type cont
      | fail 3 ]
    end
  | ∀ _:?A, _ =>
    let V := type of v in
    match type of V with
    | Prop => first [ app_evar t A cont
      | fail 3 ]
    | _ => first [ cont' (t v)
      | app_evar t A cont
      | fail 3 ]
    end
  end
end

```

```

    end
  end in
go t vs.
```

newer version : support for typeclasses

```
Ltac app_typeclass t cont :=
  let t' := constr:(t _) in
  cont t'.
```

```
Ltac build_app_all t final ::=

let rec go t :=
  match type of t with
  | ?P → ?Q ⇒ app_assert t P go
  | ∀ _:?A, _ ⇒
    first [ app_evar t A go
           | app_typeclass t go
           | fail 3 ]
  | _ ⇒ final t
end in
go t.
```

```
Ltac build_app_hnts t vs final ::=

let rec go t vs :=
  match vs with
  | nil ⇒ first [ final t | fail 1 ]
  | (boxer ltac_wilds) :: _ ⇒ first [ build_app_all t final | fail 1 ]
  | (boxer ?v) :: ?vs' ⇒
    let cont t' := go t' vs in
    let cont' t' := go t' vs' in
    let T := type of t in
    let T := eval hnf in T in
    match v with
    | ltac_wild ⇒
      first [ let U := boxerlist_next_type vs' in
              match U with
              | ltac_wild ⇒
                match T with
                | ?P → ?Q ⇒ first [ app_assert t P cont' | fail 3 ]
                | ∀ _:?A, _ ⇒ first [ app_typeclass t cont'
                                         | app_evar t A cont'
                                         | fail 3 ]
              end
    | _ ⇒
      match T with
```

```

|  $U \rightarrow ?Q \Rightarrow \text{first} [ \text{app\_assert } t U \text{ cont'} | \text{fail } 3 ]$ 
|  $\forall \_ : U, \_ \Rightarrow \text{first}$ 
  [  $\text{app\_typeclass } t \text{ cont'}$ 
  |  $\text{app\_evar } t U \text{ cont'}$ 
  |  $\text{fail } 3$  ]
|  $?P \rightarrow ?Q \Rightarrow \text{first} [ \text{app\_assert } t P \text{ cont} | \text{fail } 3 ]$ 
|  $\forall \_ : ?A, \_ \Rightarrow \text{first}$ 
  [  $\text{app\_typeclass } t \text{ cont}$ 
  |  $\text{app\_evar } t A \text{ cont}$ 
  |  $\text{fail } 3$  ]
  end
end
|  $\text{fail } 2$  ]
|  $\_ \Rightarrow$ 
  match  $T$  with
  |  $?P \rightarrow ?Q \Rightarrow \text{first} [ \text{app\_arg } t P v \text{ cont'}$ 
    |  $\text{app\_assert } t P \text{ cont}$ 
    |  $\text{fail } 3$  ]
  |  $\forall \_ : \text{Type}, \_ \Rightarrow$ 
    match type of  $v$  with
    |  $\text{Type} \Rightarrow \text{first} [ \text{cont'} (t v)$ 
      |  $\text{app\_evar } t \text{ Type cont}$ 
      |  $\text{fail } 3$  ]
    |  $\_ \Rightarrow \text{first} [ \text{app\_evar } t \text{ Type cont}$ 
      |  $\text{fail } 3$  ]
    end
  |  $\forall \_ : ?A, \_ \Rightarrow$ 
    let  $V := \text{type of } v$  in
    match type of  $V$  with
    |  $\text{Prop} \Rightarrow \text{first} [ \text{app\_typeclass } t \text{ cont}$ 
      |  $\text{app\_evar } t A \text{ cont}$ 
      |  $\text{fail } 3$  ]
    |  $\_ \Rightarrow \text{first} [ \text{cont'} (t v)$ 
      |  $\text{app\_typeclass } t \text{ cont}$ 
      |  $\text{app\_evar } t A \text{ cont}$ 
      |  $\text{fail } 3$  ]
    end
  end
end
end in
go  $t$  vs.

```

Ltac *build_app args final* :=

```

first [
  match args with (@boxer ?T ?t) :: ?vs =>
    let t := constr:(t:T) in
    build_app_hnts t vs final;
    fast_rm_inside args
  end
| fail 1 "Instantiation fails for:" args].
Ltac unfold_head_until_product T :=
  eval hnf in T.

Ltac args_unfold_head_if_not_product args :=
  match args with (@boxer ?T ?t) :: ?vs =>
    let T' := unfold_head_until_product T in
    constr:((@boxer T' t) :: vs)
  end.

Ltac args_unfold_head_if_not_product_but_params args :=
  match args with
  | (@boxer ?t) :: (@boxer ?v) :: ?vs =>
    args_unfold_head_if_not_product args
  | _ => constr:(args)
  end.

```

lets H: (» E0 E1 .. EN) will instantiate lemma *E0* on the arguments *Ei* (which may be wildcards *_*), and name *H* the resulting term. *H* may be an introduction pattern, or a sequence of introduction patterns *I1 I2 IN*, or empty. Syntax *lets H: E0 E1 .. EN* is also available. If the last argument *EN* is *---* (triple-underscore), then all arguments of *H* will be instantiated.

```

Ltac lets_build I Ei :=
  let args := list_boxer_of Ei in
  let args := args_unfold_head_if_not_product_but_params args in

  build_app args ltac:(fun R => lets_base I R).

```

Tactic Notation "lets" simple_intropattern(I) ":" constr(E) :=

lets_build I E.

Tactic Notation "lets" ":" constr(E) :=

let H := fresh in lets H: E.

Tactic Notation "lets" ":" constr(E0)

constr(A1) :=

lets: (» E0 A1).

Tactic Notation "lets" ":" constr(E0)

constr(A1) constr(A2) :=

lets: (» E0 A1 A2).

Tactic Notation "lets" ":" constr(E0)

```

constr(A1) constr(A2) constr(A3) :=
  lets: (» E0 A1 A2 A3).
Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: (» E0 A1 A2 A3 A4).
Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: (» E0 A1 A2 A3 A4 A5).

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  ":" constr(E) :=
  lets [I1 I2]: E.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) ":" constr(E) :=
  lets [I1 [I2 I3]]: E.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) ":" constr(E) :=
  lets [I1 [I2 [I3 I4]]]: E.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  ":" constr(E) :=
  lets [I1 [I2 [I3 [I4 I5]]]]: E.

Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  lets I: (» E0 A1).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
  lets I: (» E0 A1 A2).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets I: (» E0 A1 A2 A3).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets I: (» E0 A1 A2 A3 A4).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets I: (» E0 A1 A2 A3 A4 A5).

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
  constr(A1) :=
  lets [I1 I2]: E0 A1.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
  constr(A1) constr(A2) :=
  lets [I1 I2]: E0 A1 A2.

```

```

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets [I1 I2]: E0 A1 A2 A3.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets [I1 I2]: E0 A1 A2 A3 A4.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets [I1 I2]: E0 A1 A2 A3 A4 A5.

```

forwards H: (» E0 E1 .. EN) is short for *forwards H: (» E0 E1 .. EN ___)*. The arguments E_i can be wildcards $__$ (except E_0). H may be an introduction pattern, or a sequence of introduction pattern, or empty. Syntax *forwards H: E0 E1 .. EN* is also available.

```

Ltac forwards_build_app_arg Ei :=
  let args := list_boxer_of Ei in
  let args := (eval simpl in (args ++ ((boxer \_\_)::nil))) in
  let args := args_unfold_head_if_not_product args in
  args.

```

```

Ltac forwards_then Ei cont :=
  let args := forwards_build_app_arg Ei in
  let args := args_unfold_head_if_not_product_but_params args in
  build_app args cont.

```

```

Tactic Notation "forwards" simple_intropattern(I) ":" constr(Ei) :=
  let args := forwards_build_app_arg Ei in
  lets I: args.

```

```

Tactic Notation "forwards" ":" constr(E) :=
  let H := fresh in forwards H: E.

```

```

Tactic Notation "forwards" ":" constr(E0)
  constr(A1) :=
  forwards: (» E0 A1).

```

```

Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) :=
  forwards: (» E0 A1 A2).

```

```

Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  forwards: (» E0 A1 A2 A3).

```

```

Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: (» E0 A1 A2 A3 A4).

```

```

Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=

```

forwards: ($\gg E_0 A_1 A_2 A_3 A_4 A_5$).

Tactic Notation "forwards" *simple_intropattern*(I_1) *simple_intropattern*(I_2)
"::" constr(E) :=
forwards [$I_1 I_2$]: E .

Tactic Notation "forwards" *simple_intropattern*(I_1) *simple_intropattern*(I_2)
simple_intropattern(I_3) ":" constr(E) :=
forwards [$I_1 [I_2 I_3]$]: E .

Tactic Notation "forwards" *simple_intropattern*(I_1) *simple_intropattern*(I_2)
simple_intropattern(I_3) *simple_intropattern*(I_4) ":" constr(E) :=
forwards [$I_1 [I_2 [I_3 I_4]]$]: E .

Tactic Notation "forwards" *simple_intropattern*(I_1) *simple_intropattern*(I_2)
simple_intropattern(I_3) *simple_intropattern*(I_4) *simple_intropattern*(I_5)
 ":" constr(E) :=
forwards [$I_1 [I_2 [I_3 [I_4 I_5]]]$]: E .

Tactic Notation "forwards" *simple_intropattern*(I) ":" constr(E_0)
constr(A_1) :=
forwards I : ($\gg E_0 A_1$).

Tactic Notation "forwards" *simple_intropattern*(I) ":" constr(E_0)
constr(A_1) constr(A_2) :=
forwards I : ($\gg E_0 A_1 A_2$).

Tactic Notation "forwards" *simple_intropattern*(I) ":" constr(E_0)
constr(A_1) constr(A_2) constr(A_3) :=
forwards I : ($\gg E_0 A_1 A_2 A_3$).

Tactic Notation "forwards" *simple_intropattern*(I) ":" constr(E_0)
constr(A_1) constr(A_2) constr(A_3) constr(A_4) :=
forwards I : ($\gg E_0 A_1 A_2 A_3 A_4$).

Tactic Notation "forwards" *simple_intropattern*(I) ":" constr(E_0)
constr(A_1) constr(A_2) constr(A_3) constr(A_4) constr(A_5) :=
forwards I : ($\gg E_0 A_1 A_2 A_3 A_4 A_5$).

forwards_nounfold I : E is like *forwards* I : E but does not unfold the head constant of E if there is no visible quantification or hypothesis in E . It is meant to be used mainly by tactics.

Tactic Notation "forwards_nounfold" *simple_intropattern*(I) ":" constr(E_i) :=
let args := list_boxer_of E_i in
let args := (eval simpl in (args ++ ((boxer ___) :: nil))) in
build_app args ltac:(fun $R \Rightarrow$ lets_base $I R$).

forwards_nounfold_then E ltac:(fun $K \Rightarrow ..$) is like *forwards*: E but it provides the resulting term to a continuation, under the name K .

Ltac *forwards_nounfold_then* E_i cont :=
let args := list_boxer_of E_i in
let args := (eval simpl in (args ++ ((boxer ___) :: nil))) in

build_app args cont.

applys ($\gg E0 E1 \dots EN$) instantiates lemma $E0$ on the arguments Ei (which may be wildcards $_$), and apply the resulting term to the current goal, using the tactic *applys* defined earlier on. *applys* $E0 E1 E2 \dots EN$ is also available.

```
Ltac applys_build Ei :=
  let args := list_boxer_of Ei in
  let args := args_unfold_head_if_not_product_but_params args in
  build_app args ltac:(fun R =>
    first [ apply R | eapply R | rapply R ]).
```

```
Ltac applys_base E :=
  match type of E with
  | list Boxer => applys_build E
  | _ => first [ rapply E | applys_build E ]
  end; fast_rm_inside E.
```

```
Tactic Notation "applys" constr(E) :=
  applys_base E.
```

```
Tactic Notation "applys" constr(E0) constr(A1) :=
  applys ( $\gg E0 A1$ ).
```

```
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) :=
  applys ( $\gg E0 A1 A2$ ).
```

```
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applys ( $\gg E0 A1 A2 A3$ ).
```

```
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4) :=
  applys ( $\gg E0 A1 A2 A3 A4$ ).
```

```
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  applys ( $\gg E0 A1 A2 A3 A4 A5$ ).
```

fapplys ($\gg E0 E1 \dots EN$) instantiates lemma $E0$ on the arguments Ei and on the argument $__$ meaning that all evars should be explicitly instantiated, and apply the resulting term to the current goal. *fapplys* $E0 E1 E2 \dots EN$ is also available.

```
Ltac fapplys_build Ei :=
  let args := list_boxer_of Ei in
  let args := (eval simpl in (args ++ ((boxer __) :: nil))) in
  let args := args_unfold_head_if_not_product_but_params args in
  build_app args ltac:(fun R => apply R).
```

```
Tactic Notation "fapplys" constr(E0) :=
  match type of E0 with
  | list Boxer => fapplys_build E0
  | _ => fapplys_build ( $\gg E0$ )
  end.
```

```

Tactic Notation "fapplys" constr(E0) constr(A1) :=
  fapplys (» E0 A1).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) :=
  fapplys (» E0 A1 A2).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) constr(A3) :=
  fapplys (» E0 A1 A2 A3).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4) :=
  fapplys (» E0 A1 A2 A3 A4).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  fapplys (» E0 A1 A2 A3 A4 A5).

```

specializes H (» E1 E2 .. EN) will instantiate hypothesis *H* on the arguments *Ei* (which may be wildcards *_*). If the last argument *EN* is *___* (triple-underscore), then all arguments of *H* get instantiated.

```

Ltac specializes_build H Ei :=
  let H' := fresh "TEMP" in rename H into H';
  let args := list_boxer_of Ei in
  let args := constr:(boxer H')::args in
  let args := args_unfold_head_if_not_product args in
  build_app args 1tac:(fun R => lets H: R);
  clear H'.

Ltac specializes_base H Ei :=
  specializes_build H Ei; fast_rm_inside Ei.

Tactic Notation "specializes" hyp(H) :=
  specializes_base H (___).

Tactic Notation "specializes" hyp(H) constr(A) :=
  specializes_base H A.

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H (» A1 A2).

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H (» A1 A2 A3).

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4) :=
  specializes H (» A1 A2 A3 A4).

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  specializes H (» A1 A2 A3 A4 A5).

```

specializes_vars H is equivalent to *specializes H ___ .. ___* with as many double underscore as the number of dependent arguments visible from the type of *H*. Note that no unfolding is currently being performed (this behavior might change in the future). The current imple-

mentation is restricted to the case where H is an existing hypothesis – TODO: generalize.

```
Ltac specializes_var_base H :=
```

```
  match type of H with
  | ?P → ?Q ⇒ fail 1
  | ∀ _, _ ⇒ specializes H --
  end.
```

```
Ltac specializes_vars_base H :=
```

```
  repeat (specializes_var_base H).
```

```
Tactic Notation "specializes_var" hyp(H) :=
```

```
  specializes_var_base H.
```

```
Tactic Notation "specializes_vars" hyp(H) :=
```

```
  specializes_vars_base H.
```

2.4.4 Experimental tactics for application

fapply is a version of *apply* based on *forwards*.

```
Tactic Notation "fapply" constr(E) :=
```

```
  let H := fresh "TEMP" in forwards H: E;
  first [ apply H | eapply H | rapply H | hnf; apply H
    | hnf; eapply H | applics H ].
```

sapply stands for “super apply”. It tries *apply*, *eapply*, *applys* and *fapply*, and also tries to head-normalize the goal first.

```
Tactic Notation "sapply" constr(H) :=
```

```
  first [ apply H | eapply H | rapply H | applics H
    | hnf; apply H | hnf; eapply H | hnf; applics H
    | fapply H ].
```

2.4.5 Adding assumptions

lets_simpl H: E is the same as *lets H: E* excepts that it calls *simpl* on the hypothesis H . *lets_simpl: E* is also provided.

```
Tactic Notation "lets_simpl" ident(H) ":" constr(E) :=
```

```
  lets H: E; try simpl in H.
```

```
Tactic Notation "lets_simpl" ":" constr(T) :=
```

```
  let H := fresh "TEMP" in lets_simpl H: T.
```

lets_hnf H: E is the same as *lets H: E* excepts that it calls *hnf* to set the definition in head normal form. *lets_hnf: E* is also provided.

```
Tactic Notation "lets_hnf" ident(H) ":" constr(E) :=
```

```
  lets H: E; hnf in H.
```

```

Tactic Notation "lets_hnf" ":" constr(T) :=
  let H := fresh "TEMP" in lets_hnf H: T.

  puts X: E is a synonymous for pose (X := E). Alternative syntax is puts: E.

Tactic Notation "puts" ident(X) ":" constr(E) :=
  pose (X := E).

Tactic Notation "puts" ":" constr(E) :=
  let X := fresh "X" in pose (X := E).

```

2.4.6 Application of tautologies

logic E, where *E* is a fact, is equivalent to assert *H:E*; [tauto | eapply *H*; clear *H*]. It is useful for instance to prove a conjunction $[A \wedge B]$ by showing first $[A]$ and then $[A \rightarrow B]$, through the command [logic (foral *A B*, $A \rightarrow (A \rightarrow B) \rightarrow A \wedge B$)]

```

Ltac logic_base E cont :=
  assert (H:E); [ cont tt | eapply H; clear H ].

Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ => tauto).

```

2.4.7 Application modulo equalities

The tactic *equates* replaces a goal of the form $P \times y z$ with a goal of the form $P \times ?a z$ and a subgoal $?a = y$. The introduction of the evar $?a$ makes it possible to apply lemmas that would not apply to the original goal, for example a lemma of the form $\forall n m, P n n m$, because *x* and *y* might be equal but not convertible.

Usage is *equates i1 ... ik*, where the indices are the positions of the arguments to be replaced by evars, counting from the right-hand side. If 0 is given as argument, then the entire goal is replaced by an evar.

```

Section equatesLemma.

Variables (A0 A1 : Type).
Variables (A2 :  $\forall (x1 : A1)$ , Type).
Variables (A3 :  $\forall (x1 : A1) (x2 : A2 x1)$ , Type).
Variables (A4 :  $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2)$ , Type).
Variables (A5 :  $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3)$ , Type).
Variables (A6 :  $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4)$ , Type).

Lemma equates_0 :  $\forall (P Q:\text{Prop})$ ,
  P → P = Q → Q.
Proof using. intros. subst. auto. Qed.

Lemma equates_1 :
   $\forall (P:A0 \rightarrow \text{Prop}) x1 y1$ ,
  P y1 → x1 = y1 → P x1.
Proof using. intros. subst. auto. Qed.

```

Lemma equates_2 :

$$\begin{aligned} \forall y1 \ (P:A0 \rightarrow \forall(x1:A1),\text{Prop}) \ x1 \ x2, \\ P \ y1 \ x2 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2. \end{aligned}$$

Proof using. intros. subst. auto. Qed.

Lemma equates_3 :

$$\begin{aligned} \forall y1 \ (P:A0 \rightarrow \forall(x1:A1)(x2:A2 \ x1),\text{Prop}) \ x1 \ x2 \ x3, \\ P \ y1 \ x2 \ x3 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2 \ x3. \end{aligned}$$

Proof using. intros. subst. auto. Qed.

Lemma equates_4 :

$$\begin{aligned} \forall y1 \ (P:A0 \rightarrow \forall(x1:A1)(x2:A2 \ x1)(x3:A3 \ x2),\text{Prop}) \ x1 \ x2 \ x3 \ x4, \\ P \ y1 \ x2 \ x3 \ x4 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2 \ x3 \ x4. \end{aligned}$$

Proof using. intros. subst. auto. Qed.

Lemma equates_5 :

$$\begin{aligned} \forall y1 \ (P:A0 \rightarrow \forall(x1:A1)(x2:A2 \ x1)(x3:A3 \ x2)(x4:A4 \ x3),\text{Prop}) \ x1 \ x2 \ x3 \ x4 \ x5, \\ P \ y1 \ x2 \ x3 \ x4 \ x5 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2 \ x3 \ x4 \ x5. \end{aligned}$$

Proof using. intros. subst. auto. Qed.

Lemma equates_6 :

$$\begin{aligned} \forall y1 \ (P:A0 \rightarrow \forall(x1:A1)(x2:A2 \ x1)(x3:A3 \ x2)(x4:A4 \ x3)(x5:A5 \ x4),\text{Prop}) \\ x1 \ x2 \ x3 \ x4 \ x5 \ x6, \\ P \ y1 \ x2 \ x3 \ x4 \ x5 \ x6 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2 \ x3 \ x4 \ x5 \ x6. \end{aligned}$$

Proof using. intros. subst. auto. Qed.

End equatesLemma.

Ltac equates_lemma n :=
 match number_to_nat n with
 | 0 ⇒ constr:(equates_0)
 | 1 ⇒ constr:(equates_1)
 | 2 ⇒ constr:(equates_2)
 | 3 ⇒ constr:(equates_3)
 | 4 ⇒ constr:(equates_4)
 | 5 ⇒ constr:(equates_5)
 | 6 ⇒ constr:(equates_6)
 | _ ⇒ fail 100 "equates tactic only support up to arity 6"
 end.

Ltac equates_one n :=
 let L := equates_lemma n in
 eapply L.

Ltac equates_several E cont :=
 let all_pos := match type of E with
 | List.list Boxer ⇒ constr:(E)
 | _ ⇒ constr:(boxer E)::nil)

```

    end in
let rec go pos :=
  match pos with
  | nil => cont tt
  | (boxer ?n) :: ?pos' => equates_one n; [ instantiate; go pos' | ]
  end in
go all_pos.

```

Tactic Notation "equates" constr(E) :=
 $\text{equates_several } E \text{ ltac:(fun } _\text{ } \Rightarrow \text{idtac)}.$

Tactic Notation "equates" constr(n_1) constr(n_2) :=
 $\text{equates } (\gg n_1 n_2).$

Tactic Notation "equates" constr(n_1) constr(n_2) constr(n_3) :=
 $\text{equates } (\gg n_1 n_2 n_3).$

Tactic Notation "equates" constr(n_1) constr(n_2) constr(n_3) constr(n_4) :=
 $\text{equates } (\gg n_1 n_2 n_3 n_4).$

applys_eq H i1 .. iK is the same as *equates i1 .. iK* followed by *applys H* on the first subgoal.

DEPRECATED: use *applys_eq H* instead.

Tactic Notation "applys_eq" constr(H) constr(E) :=
 $\text{equates_several } E \text{ ltac:(fun } _\text{ } \Rightarrow \text{sapply } H).$

Tactic Notation "applys_eq" constr(H) constr(n_1) constr(n_2) :=
 $\text{applys_eq } H (\gg n_1 n_2).$

Tactic Notation "applys_eq" constr(H) constr(n_1) constr(n_2) constr(n_3) :=
 $\text{applys_eq } H (\gg n_1 n_2 n_3).$

Tactic Notation "applys_eq" constr(H) constr(n_1) constr(n_2) constr(n_3) constr(n_4) :=
 $\text{applys_eq } H (\gg n_1 n_2 n_3 n_4).$

applys_eq H helps proving a goal of the form $P x_1 .. x_N$ from an hypothesis H that concludes $P y_1 .. y_N$, where the arguments x_i and y_i may or may not be convertible. Equalities are produced for all arguments that don't unify.

The tactic invokes *equates* on all arguments, then calls *applys K*, and attempts **reflexivity** on the side equalities.

Lemma applys_eq_init : $\forall (P Q:\text{Prop}),$

$$\begin{aligned} P &= Q \rightarrow \\ Q &\rightarrow \\ P. \end{aligned}$$

Proof using. intros. subst. auto. Qed.

Lemma applys_eq_step_dep : $\forall B (P Q: (\forall A, A \rightarrow B)) (T:\text{Type}),$

$$\begin{aligned} P &= Q \rightarrow \\ P T &= Q T. \end{aligned}$$

Proof using. intros. subst. auto. Qed.

```

Lemma applics_eq_step : ∀ A B (P Q:A→B) x y,
  P = Q →
  x = y →
  P x = Q y.

```

Proof using. intros. subst. auto. Qed.

```

Ltac applics_eq_loop tt :=
  match goal with
  | ⊢ ?P ?x ⇒
    first [ eapply applics_eq_step; [ applics_eq_loop tt | ]
            | eapply applics_eq_step_dep; applics_eq_loop tt ]
  | _ ⇒ reflexivity
  end.

```

```

Ltac applics_eq_core H :=
  eapply applics_eq_init;
  [ applics_eq_loop tt | applics H ];
  try reflexivity.

```

```

Tactic Notation "applics_eq" constr(H) :=
  applics_eq_core H.

```

2.4.8 Absurd goals

false_goal replaces any goal by the goal **False**. Contrary to the tactic **false** (below), it does not try to do anything else

```

Tactic Notation "false_goal" :=
  elimtype False.

```

false_post is the underlying tactic used to prove goals of the form **False**. In the default implementation, it proves the goal if the context contains **False** or an hypothesis of the form $C \times 1 .. xN = D \times 1 .. yM$, or if the **congruence** tactic finds a proof of $x \neq x$ for some x .

```

Ltac false_post :=
  solve [ assumption | discriminate | congruence ].

```

false replaces any goal by the goal **False**, and calls *false_post*

```

Tactic Notation "false" :=
  false_goal; try false_post.

```

tryfalse tries to solve a goal by contradiction, and leaves the goal unchanged if it cannot solve it. It is equivalent to **try solve \[false \]**.

```

Tactic Notation "tryfalse" :=
  try solve [ false ].

```

false E tries to exploit lemma *E* to prove the goal **false**. **false E1 .. EN** is equivalent to **false (» E1 .. EN)**, which tries to apply *applics* ($\gg E1 .. EN$) and if it does not work then tries *forwards H*: ($\gg E1 .. EN$) followed with **false**

```

Ltac false_then E cont :=
  false_goal; first
  [ applys E; instantiate
  | forwards_then E ltac:(fun M =>
    pose M; jauto_set_hyps; intros; false) ];
  cont tt.

Tactic Notation "false" constr(E) :=
  false_then E ltac:(fun _ => idtac).

Tactic Notation "false" constr(E) constr(E1) :=
  false (» E E1).

Tactic Notation "false" constr(E) constr(E1) constr(E2) :=
  false (» E E1 E2).

Tactic Notation "false" constr(E) constr(E1) constr(E2) constr(E3) :=
  false (» E E1 E2 E3).

Tactic Notation "false" constr(E) constr(E1) constr(E2) constr(E3) constr(E4) :=
  false (» E E1 E2 E3 E4).

false_invert H proves a goal if it absurd after calling inversion H and false

Ltac false_invert_for H :=
  let M := fresh "TEMP" in pose (M := H); inversion H; false.

Tactic Notation "false_invert" constr(H) :=
  try solve [ false_invert_for H | false ].

false_invert proves any goal provided there is at least one hypothesis H in the context
(or as a universally quantified hypothesis visible at the head of the goal) that can be proved
absurd by calling inversion H.

Ltac false_invert_iter :=
  match goal with H:_ ⊢ _ =>
  solve [ inversion H; false
    | clear H; false_invert_iter
    | fail 2 ] end.

Tactic Notation "false_invert" :=
  intros; solve [ false_invert_iter | false ].

tryfalse_invert H and tryfalse_invert are like the above but leave the goal unchanged if
they don't solve it.

Tactic Notation "tryfalse_invert" constr(H) :=
  try (false_invert H).

Tactic Notation "tryfalse_invert" :=
  try false_invert.

false_neq_self_hyp proves any goal if the context contains an hypothesis of the form  $E \neq E$ . It is a restricted and optimized version of false. It is intended to be used by other tactics
only.

```

```
Ltac false_neq_self_hyp :=
  match goal with H: ?x ≠ ?x ⊢ _ ⇒
    false_goal; apply H; reflexivity end.
```

2.5 Introduction and generalization

2.5.1 Introduction using \Rightarrow

intros is used to name only non-dependent hypothesis.

- If *intros* is called on a goal of the form $\forall x, H$, it should introduce all the variables quantified with a \forall at the head of the goal, but it does not introduce hypotheses that precede an arrow constructor, like in $P \rightarrow Q$.
- If *intros* is called on a goal that is not of the form $\forall x, H$ nor $P \rightarrow Q$, the tactic unfolds definitions until the goal takes the form $\forall x, H$ or $P \rightarrow Q$. If unfolding definitions does not produce a goal of this form, then the tactic *intros* does nothing at all.

```
Ltac intros_rec :=
  match goal with
  | ⊢ ?P → ?Q ⇒ idtac
  | ⊢ ∀ _, _ ⇒ intro; intros_rec
  | ⊢ _ ⇒ idtac
  end.
```

```
Ltac intros_noarg :=
  match goal with
  | ⊢ ?P → ?Q ⇒ idtac
  | ⊢ ∀ _, _ ⇒ intros_rec
  | ⊢ ?G ⇒ hnf;
    match goal with
    | ⊢ ?P → ?Q ⇒ idtac
    | ⊢ ∀ _, _ ⇒ intros_rec
    end
  | ⊢ _ ⇒ idtac
  end.
```

```
Ltac intros_noarg_not_optimized :=
  intro; match goal with H: _ ⊢ _ ⇒ revert H end; intros_rec.
```

```
Ltac intros_arg H :=
  hnf; match goal with
  | ⊢ ?P → ?Q ⇒ intros H
  | ⊢ ∀ _, _ ⇒ intro; intros_arg H
```

```
end.
```

```
Tactic Notation "introv" :=
  introv_noarg.
Tactic Notation "introv" simple_intropattern(I1) :=
  introv_arg I1.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2) :=
  introv I1; introv I2.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) :=
  introv I1; introv I2 I3.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) :=
  introv I1; introv I2 I3 I4.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5) :=
  introv I1; introv I2 I3 I4 I5.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) :=
  introv I1; introv I2 I3 I4 I5 I6.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) :=
  introv I1; introv I2 I3 I4 I5 I6 I7.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8) :=
  introv I1; introv I2 I3 I4 I5 I6 I7 I8.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) :=
  introv I1; introv I2 I3 I4 I5 I6 I7 I8 I9.
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) simple_intropattern(I10) :=
  introv I1; introv I2 I3 I4 I5 I6 I7 I8 I9 I10.
```

`intros_all` repeats `intro` as long as possible. Contrary to `intros`, it unfolds any definition on the way. Remark that it also unfolds the definition of negation, so applying `intros_all` to a goal of the form $\forall x, P x \rightarrow \neg Q$ will introduce x and $P x$ and Q , and will leave `False` in the goal.

```

Tactic Notation "intros_all" :=
  repeat intro.

  intros_hnf introduces an hypothesis and sets in head normal form

Tactic Notation "intro_hnf" :=
  intro; match goal with H: _ ⊢ _ ⇒ hnf in H end.



### 2.5.2 Introduction using $\Rightarrow$ and $\Rightarrow\!\Rightarrow$



Ltac ltac_intros_post := idtac.

Tactic Notation " $\Rightarrow$ " :=
  intros.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) :=
  intros I1.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2) :=
  intros I1 I2.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) :=
  intros I1 I2 I3.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) :=
  intros I1 I2 I3 I4.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5) :=
  intros I1 I2 I3 I4 I5.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) :=
  intros I1 I2 I3 I4 I5 I6.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) :=
  intros I1 I2 I3 I4 I5 I6 I7.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8) :=
  intros I1 I2 I3 I4 I5 I6 I7 I8.

Tactic Notation " $\Rightarrow$ " simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) :=
  intros I1 I2 I3 I4 I5 I6 I7 I8 I9.

```

```

Tactic Notation " $=>$ " simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
simple_intropattern(I9) simple_intropattern(I10) :=
intros I1 I2 I3 I4 I5 I6 I7 I8 I9 I10.

Ltac intro_nondeps_aux_special_intro G :=
fail.

Ltac intro_nondeps_aux is_already_hnf :=
match goal with
|  $\vdash (?P \rightarrow ?Q) \Rightarrow$  idtac
|  $\vdash ?G \rightarrow _- \Rightarrow$  intro_nondeps_aux_special_intro G;
  intro; intro_nondeps_aux true
|  $\vdash (\forall _,_ ) \Rightarrow$  intros ?; intro_nondeps_aux true
|  $\vdash _- \Rightarrow$ 
  match is_already_hnf with
  | true  $\Rightarrow$  idtac
  | false  $\Rightarrow$  hnf; intro_nondeps_aux true
  end
end.

Ltac intro_nondeps tt := intro_nondeps_aux false.

Tactic Notation " $=\gg$ " :=
intro_nondeps tt.

Tactic Notation " $=\gg$ " simple_intropattern(I1) :=
 $=\gg$ ; intros I1.

Tactic Notation " $=\gg$ " simple_intropattern(I1) simple_intropattern(I2) :=
 $=\gg$ ; intros I1 I2.

Tactic Notation " $=\gg$ " simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) :=
 $=\gg$ ; intros I1 I2 I3.

Tactic Notation " $=\gg$ " simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) :=
 $=\gg$ ; intros I1 I2 I3 I4.

Tactic Notation " $=\gg$ " simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5) :=
 $=\gg$ ; intros I1 I2 I3 I4 I5.

Tactic Notation " $=\gg$ " simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) :=
 $=\gg$ ; intros I1 I2 I3 I4 I5 I6.

Tactic Notation " $=\gg$ " simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)

```

```

simple_intropattern(I6) simple_intropattern(I7) :=
  =>; intros I1 I2 I3 I4 I5 I6 I7.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8) :=
  =>; intros I1 I2 I3 I4 I5 I6 I7 I8.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) :=
  =>; intros I1 I2 I3 I4 I5 I6 I7 I8 I9.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) simple_intropattern(I10) :=
  =>; intros I1 I2 I3 I4 I5 I6 I7 I8 I9 I10.

```

2.5.3 Generalization

`gen X1 .. XN` is a shorthand for calling `generalize dependent` successively on variables $XN \dots X1$. Note that the variables are generalized in reverse order, following the convention of the `generalize` tactic: it means that $X1$ will be the first quantified variable in the resulting goal.

```

Tactic Notation "gen" ident(X1) :=
  generalize dependent X1.
Tactic Notation "gen" ident(X1) ident(X2) :=
  gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) :=
  gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) :=
  gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) ident(X5) :=
  gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) ident(X5)
  ident(X6) :=
  gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) ident(X5)
  ident(X6) ident(X7) :=
  gen X7; gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) ident(X5)
  ident(X6) ident(X7) ident(X8) :=
  gen X8; gen X7; gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.

```

```

Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) ident(X5)
  ident(X6) ident(X7) ident(X8) ident(X9) :=
  gen X9; gen X8; gen X7; gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) ident(X5)
  ident(X6) ident(X7) ident(X8) ident(X9) ident(X10) :=
  gen X10; gen X9; gen X8; gen X7; gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.

generalizes X is a shorthand for calling generalize X; clear X. It is weaker than tactic gen X since it does not support dependencies. It is mainly intended for writing tactics.

Tactic Notation "generalizes" hyp(X) :=
  generalize X; clear X.
Tactic Notation "generalizes" hyp(X1) hyp(X2) :=
  generalizes X1; generalizes X2.
Tactic Notation "generalizes" hyp(X1) hyp(X2) hyp(X3) :=
  generalizes X1 X2; generalizes X3.
Tactic Notation "generalizes" hyp(X1) hyp(X2) hyp(X3) hyp(X4) :=
  generalizes X1 X2 X3; generalizes X4.

```

2.5.4 Naming

`sets X: E` is the same as `set (X := E) in *`, that is, it replaces all occurrences of *E* by a fresh meta-variable *X* whose definition is *E*.

```

Tactic Notation "sets" ident(X) ":" constr(E) :=
  set (X := E) in *.

```

`def_to_eq E X H` applies when *X* := *E* is a local definition. It adds an assumption *H*: *X* = *E* and then clears the definition of *X*. `def_to_eq_sym` is similar except that it generates the equality *H*: *E* = *X*.

```

Ltac def_to_eq X HX E :=
  assert (HX : X = E) by reflexivity; clearbody X.
Ltac def_to_eq_sym X HX E :=
  assert (HX : E = X) by reflexivity; clearbody X.

```

`set_eq X H: E` generates the equality *H*: *X* = *E*, for a fresh name *X*, and replaces *E* by *X* in the current goal. Syntaxes `set_eq X: E` and `set_eq: E` are also available. Similarly, `set_eq ← X H: E` generates the equality *H*: *E* = *X*.

`sets_eq X HX: E` does the same but replaces *E* by *X* everywhere in the goal. `sets_eq X HX: E in H` replaces in *H*. `set_eq X HX: E in ⊢` performs no substitution at all.

```

Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) :=
  set (X := E); def_to_eq X HX E.
Tactic Notation "set_eq" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in set_eq X HX: E.
Tactic Notation "set_eq" ":" constr(E) :=
  let X := fresh "X" in set_eq X: E.

```

```

Tactic Notation "set_eq" "<-" ident(X) ident(HX) ":" constr(E) :=
  set (X := E); def_to_eq_sym X HX E.
Tactic Notation "set_eq" "<-" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in set_eq ← X HX: E.
Tactic Notation "set_eq" "<-" ":" constr(E) :=
  let X := fresh "X" in set_eq ← X: E.
Tactic Notation "sets_eq" ident(X) ident(HX) ":" constr(E) :=
  set (X := E) in *; def_to_eq X HX E.
Tactic Notation "sets_eq" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in sets_eq X HX: E.
Tactic Notation "sets_eq" ":" constr(E) :=
  let X := fresh "X" in sets_eq X: E.
Tactic Notation "sets_eq" "<-" ident(X) ident(HX) ":" constr(E) :=
  set (X := E) in *; def_to_eq_sym X HX E.
Tactic Notation "sets_eq" "<-" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in sets_eq ← X HX: E.
Tactic Notation "sets_eq" "<-" ":" constr(E) :=
  let X := fresh "X" in sets_eq ← X: E.
Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) "in" hyp(H) :=
  set (X := E) in H; def_to_eq X HX E.
Tactic Notation "set_eq" ident(X) ":" constr(E) "in" hyp(H) :=
  let HX := fresh "EQ" X in set_eq X HX: E in H.
Tactic Notation "set_eq" ":" constr(E) "in" hyp(H) :=
  let X := fresh "X" in set_eq X: E in H.
Tactic Notation "set_eq" "<-" ident(X) ident(HX) ":" constr(E) "in" hyp(H) :=
  set (X := E) in H; def_to_eq_sym X HX E.
Tactic Notation "set_eq" "<-" ident(X) ":" constr(E) "in" hyp(H) :=
  let HX := fresh "EQ" X in set_eq ← X HX: E in H.
Tactic Notation "set_eq" "<-" ":" constr(E) "in" hyp(H) :=
  let X := fresh "X" in set_eq ← X: E in H.
Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) "in" "|-:" :=
  set (X := E) in |-; def_to_eq X HX E.
Tactic Notation "set_eq" ident(X) ":" constr(E) "in" "|-:" :=
  let HX := fresh "EQ" X in set_eq X HX: E in |-.
Tactic Notation "set_eq" ":" constr(E) "in" "|-:" :=
  let X := fresh "X" in set_eq X: E in |-.
Tactic Notation "set_eq" "<-" ident(X) ident(HX) ":" constr(E) "in" "|-:" :=
  set (X := E) in |-; def_to_eq_sym X HX E.
Tactic Notation "set_eq" "<-" ident(X) ":" constr(E) "in" "|-:" :=
  let HX := fresh "EQ" X in set_eq ← X HX: E in |-.
Tactic Notation "set_eq" "<-" ":" constr(E) "in" "|-:" :=

```

```
let X := fresh "X" in set_eq  $\leftarrow$  X: E in  $\vdash$ .
```

gen_eq X: E is a tactic whose purpose is to introduce equalities so as to work around the limitation of the `induction` tactic which typically loses information. *gen_eq E as X* replaces all occurrences of term *E* with a fresh variable *X* and the equality $X = E$ as extra hypothesis to the current conclusion. In other words a conclusion *C* will be turned into $(X = E) \rightarrow C$. *gen_eq: E* and *gen_eq: E as X* are also accepted.

```
Tactic Notation "gen_eq" ident(X) ":" constr(E) :=
  let EQ := fresh "EQ" X in sets_eq X EQ: E; revert EQ.
Tactic Notation "gen_eq" ":" constr(E) :=
  let X := fresh "X" in gen_eq X: E.
Tactic Notation "gen_eq" ":" constr(E) "as" ident(X) :=
  gen_eq X: E.
Tactic Notation "gen_eq" ident(X1) ":" constr(E1) ","
  ident(X2) ":" constr(E2) :=
  gen_eq X2: E2; gen_eq X1: E1.
Tactic Notation "gen_eq" ident(X1) ":" constr(E1) ","
  ident(X2) ":" constr(E2) "," ident(X3) ":" constr(E3) :=
  gen_eq X3: E3; gen_eq X2: E2; gen_eq X1: E1.
```

sets_let X finds the first let-expression in the goal and names its body *X*. *sets_eq_let X* is similar, except that it generates an explicit equality. Tactics *sets_let X in H* and *sets_eq_let X in H* allow specifying a particular hypothesis (by default, the first one that contains a `let` is considered).

Known limitation: it does not seem possible to support naming of multiple let-in constructs inside a term, from ltac.

```
Ltac sets_let_base tac :=
  match goal with
  |  $\vdash$  context[let _ := ?E in _]  $\Rightarrow$  tac E; cbv zeta
  | H: context[let _ := ?E in _]  $\vdash$  _  $\Rightarrow$  tac E; cbv zeta in H
  end.

Ltac sets_let_in_base H tac :=
  match type of H with context[let _ := ?E in _]  $\Rightarrow$ 
    tac E; cbv zeta in H end.

Tactic Notation "sets_let" ident(X) :=
  sets_let_base ltac:(fun E  $\Rightarrow$  sets X: E).
Tactic Notation "sets_let" ident(X) "in" hyp(H) :=
  sets_let_in_base H ltac:(fun E  $\Rightarrow$  sets X: E).
Tactic Notation "sets_eq_let" ident(X) :=
  sets_let_base ltac:(fun E  $\Rightarrow$  sets_eq X: E).
Tactic Notation "sets_eq_let" ident(X) "in" hyp(H) :=
  sets_let_in_base H ltac:(fun E  $\Rightarrow$  sets_eq X: E).
```

2.6 Rewriting

`rewrites E` is similar to `rewrite` except that it supports the `rm` directives to clear hypotheses on the fly, and that it supports a list of arguments in the form `rewrites (» E1 E2 E3)` to indicate that *forwards* should be invoked first before `rewrites` is called.

```
Ltac rewrites_base E cont :=
```

```
  match type of E with
  | List.list Boxer ⇒ forwards_then E cont
  | _ ⇒ cont E; fast_rm_inside E
  end.
```

```
Tactic Notation "rewrites" constr(E) :=
```

```
  rewrites_base E ltac:(fun M ⇒ rewrite M ).
```

```
Tactic Notation "rewrites" constr(E) "in" hyp(H) :=
```

```
  rewrites_base E ltac:(fun M ⇒ rewrite M in H).
```

```
Tactic Notation "rewrites" constr(E) "in" "*" :=
```

```
  rewrites_base E ltac:(fun M ⇒ rewrite M in *).
```

```
Tactic Notation "rewrites" "<->" constr(E) :=
```

```
  rewrites_base E ltac:(fun M ⇒ rewrite ← M ).
```

```
Tactic Notation "rewrites" "<->" constr(E) "in" hyp(H) :=
```

```
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in H).
```

```
Tactic Notation "rewrites" "<->" constr(E) "in" "*" :=
```

```
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in *).
```

`rewrite_all E` iterates version of `rewrite E` as long as possible. Warning: this tactic can easily get into an infinite loop. Syntax for rewriting from right to left and/or into an hypothesis is similar to the one of `rewrite`.

```
Tactic Notation "rewrite_all" constr(E) :=
```

```
  repeat rewrite E.
```

```
Tactic Notation "rewrite_all" "<->" constr(E) :=
```

```
  repeat rewrite ← E.
```

```
Tactic Notation "rewrite_all" constr(E) "in" ident(H) :=
```

```
  repeat rewrite E in H.
```

```
Tactic Notation "rewrite_all" "<->" constr(E) "in" ident(H) :=
```

```
  repeat rewrite ← E in H.
```

```
Tactic Notation "rewrite_all" constr(E) "in" "*" :=
```

```
  repeat rewrite E in *.
```

```
Tactic Notation "rewrite_all" "<->" constr(E) "in" "*" :=
```

```
  repeat rewrite ← E in *.
```

`asserts_rewrite E` asserts that an equality `E` holds (generating a corresponding subgoal) and rewrite it straight away in the current goal. It avoids giving a name to the equality and later clearing it. Syntax for rewriting from right to left and/or into an hypothesis is similar to the one of `rewrite`. Note: the tactic `replaces` plays a similar role.

```

Ltac asserts_rewrite_tactic E action :=
  let EQ := fresh "TEMP" in (assert (EQ : E);
  [ idtac | action EQ; clear EQ ]).

Tactic Notation "asserts_rewrite" constr(E) :=
  asserts_rewrite_tactic E ltac:(fun EQ => rewrite EQ).

Tactic Notation "asserts_rewrite" "<->" constr(E) :=
  asserts_rewrite_tactic E ltac:(fun EQ => rewrite <-> EQ).

Tactic Notation "asserts_rewrite" constr(E) "in" hyp(H) :=
  asserts_rewrite_tactic E ltac:(fun EQ => rewrite EQ in H).

Tactic Notation "asserts_rewrite" "<->" constr(E) "in" hyp(H) :=
  asserts_rewrite_tactic E ltac:(fun EQ => rewrite <-> EQ in H).

Tactic Notation "asserts_rewrite" constr(E) "in" "*" :=
  asserts_rewrite_tactic E ltac:(fun EQ => rewrite EQ in *).

Tactic Notation "asserts_rewrite" "<->" constr(E) "in" "*" :=
  asserts_rewrite_tactic E ltac:(fun EQ => rewrite <-> EQ in *).

cuts_rewrite E is the same as asserts_rewrite E except that subgoals are permuted.

```

```

Ltac cuts_rewrite_tactic E action :=
  let EQ := fresh "TEMP" in (cuts EQ: E;
  [ action EQ; clear EQ | idtac ]).

Tactic Notation "cuts_rewrite" constr(E) :=
  cuts_rewrite_tactic E ltac:(fun EQ => rewrite EQ).

Tactic Notation "cuts_rewrite" "<->" constr(E) :=
  cuts_rewrite_tactic E ltac:(fun EQ => rewrite <-> EQ).

Tactic Notation "cuts_rewrite" constr(E) "in" hyp(H) :=
  cuts_rewrite_tactic E ltac:(fun EQ => rewrite EQ in H).

Tactic Notation "cuts_rewrite" "<->" constr(E) "in" hyp(H) :=
  cuts_rewrite_tactic E ltac:(fun EQ => rewrite <-> EQ in H).

```

rewrite_except H EQ rewrites equality *EQ* everywhere but in hypothesis *H*. Mainly useful for other tactics.

```

Ltac rewrite_except H EQ :=
  let K := fresh "TEMP" in let T := type of H in
  set (K := T) in H;
  rewrite EQ in *; unfold K in H; clear K.

```

rewrites E at K applies when *E* is of the form $T_1 = T_2$ rewrites the equality *E* at the *K*-th occurrence of *T₁* in the current goal. Syntaxes *rewrites ← E at K* and *rewrites E at K in H* are also available.

```

Tactic Notation "rewrites" constr(E) "at" constr(K) :=
  match type of E with ?T1 = ?T2 =>
    ltac_action_at K of T1 do (rewrites E) end.

Tactic Notation "rewrites" "<->" constr(E) "at" constr(K) :=

```

```

match type of E with ?T1 = ?T2 =>
  ltac_action_at K of T2 do (rewrites ← E) end.
Tactic Notation "rewrites" constr(E) "at" constr(K) "in" hyp(H) :=
  match type of E with ?T1 = ?T2 =>
    ltac_action_at K of T1 in H do (rewrites E in H) end.
Tactic Notation "rewrites" "<->" constr(E) "at" constr(K) "in" hyp(H) :=
  match type of E with ?T1 = ?T2 =>
    ltac_action_at K of T2 in H do (rewrites ← E in H) end.

```

replaces E with F is the same as *replace* E with F except that the equality $E = F$ is generated as first subgoal. Syntax *replaces* E with F in H is also available. Note that contrary to *replace*, *replaces* does not try to solve the equality by *assumption*. Note: *replaces* E with F is similar to *asserts_rewrite* ($E = F$).

```

Tactic Notation "replaces" constr(E) "with" constr(F) :=
  let T := fresh "TEMP" in assert (T: E = F); [| replace E with F; clear T ].
Tactic Notation "replaces" constr(E) "with" constr(F) "in" hyp(H) :=
  let T := fresh "TEMP" in assert (T: E = F); [| replace E with F in H; clear T ].
  replaces E at K with F replaces the K-th occurrence of E with F in the current goal.
  Syntax replaces E at K with F in H is also available.

```

```

Tactic Notation "replaces" constr(E) "at" constr(K) "with" constr(F) :=
  let T := fresh "TEMP" in assert (T: E = F); [| rewrites T at K; clear T ].
Tactic Notation "replaces" constr(E) "at" constr(K) "with" constr(F) "in" hyp(H) :=
  let T := fresh "TEMP" in assert (T: E = F); [| rewrites T at K in H; clear T ].

```

changes is like *change* except that it does not silently fail to perform its task. (Note that, *changes* is implemented using *rewrite*, meaning that it might perform additional beta-reductions compared with the original *change* tactic.

```

Tactic Notation "changes" constr(E1) "with" constr(E2) "in" hyp(H) :=
  asserts_rewrite (E1 = E2) in H; [ reflexivity | ].
Tactic Notation "changes" constr(E1) "with" constr(E2) :=
  asserts_rewrite (E1 = E2); [ reflexivity | ].
Tactic Notation "changes" constr(E1) "with" constr(E2) "in" "*" :=
  asserts_rewrite (E1 = E2) in *; [ reflexivity | ].

```

2.6.1 Renaming

renames X_1 to Y_1 , ..., X_N to Y_N is a shorthand for a sequence of renaming operations *rename* X_i into Y_i .

```

Tactic Notation "renames" ident(X1) "to" ident(Y1) :=
  rename X1 into Y1.

```

```

Tactic Notation "renames" ident(X1) "to" ident(Y1) ","  

ident(X2) "to" ident(Y2) :=  

  renames X1 to Y1; renames X2 to Y2.  

Tactic Notation "renames" ident(X1) "to" ident(Y1) ","  

ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) :=  

  renames X1 to Y1; renames X2 to Y2, X3 to Y3.  

Tactic Notation "renames" ident(X1) "to" ident(Y1) ","  

ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) ","  

ident(X4) "to" ident(Y4) :=  

  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4.  

Tactic Notation "renames" ident(X1) "to" ident(Y1) ","  

ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) ","  

ident(X4) "to" ident(Y4) "," ident(X5) "to" ident(Y5) :=  

  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4, X5 to Y5.  

Tactic Notation "renames" ident(X1) "to" ident(Y1) ","  

ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) ","  

ident(X4) "to" ident(Y4) "," ident(X5) "to" ident(Y5) ","  

ident(X6) "to" ident(Y6) :=  

  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4, X5 to Y5, X6 to Y6.

```

2.6.2 Unfolding

unfolds unfolds the head definition in the goal, i.e. if the goal has form $P \times_1 \dots x_N$ then it calls `unfold P`. If the goal is an equality, it tries to unfold the head constant on the left-hand side, and otherwise tries on the right-hand side. If the goal is a product, it calls `intros` first. warning: this tactic is overridden in LibReflect.

```

Ltac apply_to_head_of E cont :=  

  let go E :=  

    let P := get_head E in cont P in  

    match E with  

    | ∀ _,_ ⇒ intros; apply_to_head_of E cont  

    | ?A = ?B ⇒ first [ go A | go B ]  

    | ?A ⇒ go A  

  end.  

  

Ltac unfolds_base :=  

  match goal with ⊢ ?G ⇒  

    apply_to_head_of G 1tac:(fun P ⇒ unfold P) end.

```

```

Tactic Notation "unfolds" :=  

  unfolds_base.

```

unfolds in H unfolds the head definition of hypothesis H , i.e. if H has type $P \times_1 \dots x_N$ then it calls `unfold P in H`.

```

Ltac unfolds_in_base H :=
  match type of H with ?G ⇒
    apply_to_head_of G ltac:(fun P ⇒ unfold P in H) end.

```

```

Tactic Notation "unfolds" "in" hyp(H) :=
  unfolds_in_base H.

```

unfolds in H₁,H₂,..,H_N allows unfolding the head constant in several hypotheses at once.

```

Tactic Notation "unfolds" "in" hyp(H1) hyp(H2) :=
  unfolds in H1; unfolds in H2.

```

```

Tactic Notation "unfolds" "in" hyp(H1) hyp(H2) hyp(H3) :=
  unfolds in H1; unfolds in H2 H3.

```

```

Tactic Notation "unfolds" "in" hyp(H1) hyp(H2) hyp(H3) hyp(H4) :=
  unfolds in H1; unfolds in H2 H3 H4.

```

```

Tactic Notation "unfolds" "in" hyp(H1) hyp(H2) hyp(H3) hyp(H4) hyp(H5) :=
  unfolds in H1; unfolds in H2 H3 H4 H5.

```

unfold P₁,..,P_N is a shortcut for *unfold P₁,..,P_N in **.

```

Tactic Notation "unfolds" constr(F1) :=
  unfold F1 in *.

```

```

Tactic Notation "unfolds" constr(F1) "," constr(F2) :=
  unfold F1,F2 in *.

```

```

Tactic Notation "unfolds" constr(F1) "," constr(F2)
  "," constr(F3) :=
  unfold F1,F2,F3 in *.

```

```

Tactic Notation "unfolds" constr(F1) "," constr(F2)
  "," constr(F3) "," constr(F4) :=
  unfold F1,F2,F3,F4 in *.

```

```

Tactic Notation "unfolds" constr(F1) "," constr(F2)
  "," constr(F3) "," constr(F4) "," constr(F5) :=
  unfold F1,F2,F3,F4,F5 in *.

```

```

Tactic Notation "unfolds" constr(F1) "," constr(F2)
  "," constr(F3) "," constr(F4) "," constr(F5) ","
  constr(F6) :=
  unfold F1,F2,F3,F4,F5,F6 in *.

```

```

Tactic Notation "unfolds" constr(F1) "," constr(F2)
  "," constr(F3) "," constr(F4) "," constr(F5)
  "," constr(F6) "," constr(F7) :=
  unfold F1,F2,F3,F4,F5,F6,F7 in *.

```

```

Tactic Notation "unfolds" constr(F1) "," constr(F2)
  "," constr(F3) "," constr(F4) "," constr(F5)
  "," constr(F6) "," constr(F7) "," constr(F8) :=
  unfold F1,F2,F3,F4,F5,F6,F7,F8 in *.

```

fold P₁,..,P_N is a shortcut for *fold P₁ in *; ..; fold P_N in **.

```

Tactic Notation "folds" constr(H) :=

```

```

fold H in *.
Tactic Notation "folds" constr(H1) "," constr(H2) :=
  folds H1; folds H2.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3) :=
  folds H1; folds H2; folds H3.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3)
  "," constr(H4) :=
  folds H1; folds H2; folds H3; folds H4.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3)
  "," constr(H4) "," constr(H5) :=
  folds H1; folds H2; folds H3; folds H4; folds H5.

```

2.6.3 Simplification

simpls is a shortcut for `simpl in *`.

```

Tactic Notation "simpls" :=
  simpl in *.

```

simpls P1,..,PN is a shortcut for `simpl P1 in *; ..; simpl PN in *`.

```

Tactic Notation "simpls" constr(F1) :=
  simpl F1 in *.

```

```

Tactic Notation "simpls" constr(F1) "," constr(F2) :=
  simpls F1; simpls F2.

```

```

Tactic Notation "simpls" constr(F1) "," constr(F2)
  "," constr(F3) :=
  simpls F1; simpls F2; simpls F3.

```

```

Tactic Notation "simpls" constr(F1) "," constr(F2)
  "," constr(F3) "," constr(F4) :=
  simpls F1; simpls F2; simpls F3; simpls F4.

```

unsimpl E replaces all occurrence of *X* by *E*, where *X* is the result which the tactic `simpl` would give when applied to *E*. It is useful to undo what `simpl` has simplified too far.

```

Tactic Notation "unsimpl" constr(E) :=
  let F := (eval simpl in E) in change F with E.

```

unsimpl E in H is similar to *unsimpl E* but it applies inside a particular hypothesis *H*.

```

Tactic Notation "unsimpl" constr(E) "in" hyp(H) :=
  let F := (eval simpl in E) in change F with E in H.

```

*unsimpl E in ** applies *unsimpl E* everywhere possible. *unsimpls E* is a synonymous.

```

Tactic Notation "unsimpl" constr(E) "in" "*" :=
  let F := (eval simpl in E) in change F with E in *.

```

```

Tactic Notation "unsimpls" constr(E) :=
  unsimpl E in *.

```

`nosimpl t` protects the Coq term `t` against some forms of simplification. See Gonthier's work for details on this trick.

```
Notation "'nosimpl' t" := (match tt with tt => t end)
  (at level 10).
```

2.6.4 Evaluation

```
Tactic Notation "hnfs" := hnf in *.
```

2.6.5 Substitution

`substs` does the same as `subst`, except that it does not fail when there are circular equalities in the context.

```
Tactic Notation "substs" :=
repeat (match goal with H: ?x = ?y ⊢ _ =>
  first [ subst x | subst y ] end).
```

Implementation of `substs below`, which allows to call `subst` on all the hypotheses that lie beyond a given position in the proof context.

```
Ltac substs_below limit :=
  match goal with H: ?T ⊢ _ =>
    match T with
    | limit => idtac
    | ?x = ?y =>
      first [ subst x; substs_below limit
        | subst y; substs_below limit
        | generalizes H; substs_below limit; intro ]
    end end.
```

`substs below body E` applies `subst` on all equalities that appear in the context below the first hypothesis whose body is `E`. If there is no such hypothesis in the context, it is equivalent to `subst`. For instance, if `H` is an hypothesis, then `substs below H` will substitute equalities below hypothesis `H`.

```
Tactic Notation "substs" "below" "body" constr(M) :=
  substs_below M.
```

`substs below H` applies `subst` on all equalities that appear in the context below the hypothesis named `H`. Note that the current implementation is technically incorrect since it will confuse different hypotheses with the same body.

```
Tactic Notation "substs" "below" hyp(H) :=
  match type of H with ?M => substs below body M end.
```

`subst_hyp H` substitutes the equality contained in the first hypothesis from the context.

```

Ltac intro_subst_hyp := fail.

subst_hyp  $H$  substitutes the equality contained in  $H$ .

Ltac subst_hyp_base  $H$  :=
  match type of  $H$  with
  |  $(\_, \_, \_, \_, \_, \_) = (\_, \_, \_, \_, \_, \_)$   $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 4 intro_subst_hyp
  |  $(\_, \_, \_, \_, \_) = (\_, \_, \_, \_, \_)$   $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 4 intro_subst_hyp
  |  $(\_, \_, \_, \_) = (\_, \_, \_, \_)$   $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 3 intro_subst_hyp
  |  $(\_, \_, \_) = (\_, \_, \_)$   $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 2 intro_subst_hyp
  |  $?x = ?x \Rightarrow$  clear  $H$ 
  |  $?x = ?y \Rightarrow$  first [ subst  $x$  | subst  $y$  ]
  end.

```

Tactic Notation "subst_hyp" $hyp(H) := subst_hyp_base H$.

```

Ltac intro_subst_hyp ::=

let  $H$  := fresh "TEMP" in intros  $H$ ; subst_hyp  $H$ .

```

intro_subst is a shorthand for *intro* H ; *subst_hyp* H : it introduces and substitutes the equality at the head of the current goal.

```

Tactic Notation "intro_subst" ::=
  let  $H$  := fresh "TEMP" in intros  $H$ ; subst_hyp  $H$ .

  subst_local substitutes all local definition from the context

```

```

Ltac subst_local :=
  repeat match goal with  $H:=_=\vdash \_ \Rightarrow$  subst  $H$  end.

  subst_eq  $E$  takes an equality  $x = t$  and replace  $x$  with  $t$  everywhere in the goal

```

```

Ltac subst_eq_base  $E$  :=
  let  $H$  := fresh "TEMP" in lets  $H: E$ ; subst_hyp  $H$ .

Tactic Notation "subst_eq" constr( $E$ ) :=
  subst_eq_base  $E$ .

```

2.6.6 Tactics to work with proof irrelevance

Require Import Coq.Logic.ProofIrrelevance.

pi_rewrite E replaces E of type Prop with a fresh unification variable, and is thus a practical way to exploit proof irrelevance, without writing explicitly *rewrite* (*proof_irrelevance* $E' E$). Particularly useful when E' is a big expression.

```

Ltac pi_rewrite_base  $E$  rewrite_tac :=
  let  $E' :=$  fresh "TEMP" in let  $T :=$  type of  $E$  in evar ( $E':T$ );
  rewrite_tac (@proof_irrelevance  $_ E E'$ ); subst  $E'$ .

```

```

Tactic Notation "pi_rewrite" constr( $E$ ) :=
  pi_rewrite_base  $E$  ltac:(fun  $X \Rightarrow$  rewrite  $X$ ).

```

```
Tactic Notation "pi_rewrite" constr(E) "in" hyp(H) :=
  pi_rewrite_base E ltac:(fun X => rewrite X in H).
```

2.6.7 Proving equalities

Note: current implementation only supports up to arity 5

f_equal is a variation on *f_equal* which has a better behaviour on equalities between n-ary tuples.

```
Ltac f_equal_support_for_exist tt :=
  fail.
```

```
Ltac f_equal_base :=
  let go := f_equal; [ f_equal_base | ] in
  first
  [ f_equal_support_for_exist tt
  | match goal with
    | ⊢ ( _, _, _ ) = ( _, _, _ ) ⇒ go
    | ⊢ ( _, _, _, _ ) = ( _, _, _, _ ) ⇒ go
    | ⊢ ( _, _, _, _, _ ) = ( _, _, _, _, _ ) ⇒ go
    | ⊢ ( _, _, _, _, _, _ ) = ( _, _, _, _, _, _ ) ⇒ go
    | ⊢ _ ⇒ f_equal
  end ].
```

```
Tactic Notation "f_equal" :=
  f_equal_base.
```

*f_equal*s is the same as *f_equal* except that it tries and solve all trivial subgoals, using *reflexivity* and *congruence* (as well as the proof-irrelevance principle). *f_equal*s applies to goals of the form $f\ x_1 \dots x_N = f\ y_1 \dots y_N$ and produces some subgoals of the form $x_i = y_i$.

```
Ltac f_equal_post :=
  first [ reflexivity | congruence | apply proof_irrelevance | idtac ].
```

```
Tactic Notation "f_equal" :=
  f_equal; f_equal_post.
```

f_equal_rec calls *f_equal*s recursively. It is equivalent to *repeat (progress f_equal)*.

```
Tactic Notation "f_equal_rec" :=
  repeat (progress f_equal).
```

2.7 Inversion

2.7.1 Basic inversion

invert keep H is same to *inversion H* except that it puts all the facts obtained in the goal. The keyword *keep* means that the hypothesis *H* should not be removed.

```
Tactic Notation "invert" "keep" hyp(H) :=
  pose ltac_mark; inversion H; gen_until_mark.
```

invert keep H as X1 .. XN is the same as *inversion H as ...* except that only hypotheses which are not variable need to be named explicitly, in a similar fashion as *intros* is used to name only hypotheses.

```
Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I1) :=
  invert keep H; intros I1.
```

```
Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) :=
  invert keep H; intros I1 I2.
```

```
Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) :=
  invert keep H; intros I1 I2 I3.
```

invert H is same to *inversion H* except that it puts all the facts obtained in the goal and clears hypothesis *H*. In other words, it is equivalent to *invert keep H; clear H*.

```
Tactic Notation "invert" hyp(H) :=
  invert keep H; clear H.
```

invert H as X1 .. XN is the same as *invert keep H as X1 .. XN* but it also clears hypothesis *H*.

```
Tactic Notation "invert_tactic" hyp(H) tactic(tac) :=
  let H' := fresh "TEMP" in rename H into H'; tac H'; clear H'.
```

```
Tactic Notation "invert" hyp(H) "as" simple_intropattern(I1) :=
  invert_tactic H (fun H => invert keep H as I1).
```

```
Tactic Notation "invert" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) :=
  invert_tactic H (fun H => invert keep H as I1 I2).
```

```
Tactic Notation "invert" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) :=
  invert_tactic H (fun H => invert keep H as I1 I2 I3).
```

2.7.2 Inversion with substitution

Our inversion tactics is able to get rid of dependent equalities generated by *inversion*, using proof irrelevance.

```
Axiom inj_pair2 :
   $\forall (U : \text{Type}) (P : U \rightarrow \text{Type}) (p : U) (x y : P p),$ 
   $\text{existT } P p x = \text{existT } P p y \rightarrow x = y.$ 
```

```
Ltac inverts_tactic H i1 i2 i3 i4 i5 i6 :=
  let rec go i1 i2 i3 i4 i5 i6 :=
    match goal with
```

```

| ⊢ (ltac_mark → _) ⇒ intros _
| ⊢ (?x = ?y → _) ⇒ let H := fresh "TEMP" in intro H;
    first [ subst x | subst y ];
    go i1 i2 i3 i4 i5 i6
| ⊢ (existT ?P ?p ?x = existT ?P ?p ?y → _) ⇒
    let H := fresh "TEMP" in intro H;
    generalize (@inj_pair2 _ P p x y H);
    clear H; go i1 i2 i3 i4 i5 i6
| ⊢ (?P → ?Q) ⇒ i1; go i2 i3 i4 i5 i6 ltac:(intro)
| ⊢ (∀ _, _) ⇒ intro; go i1 i2 i3 i4 i5 i6
end in
generalize ltac_mark; invert keep H; go i1 i2 i3 i4 i5 i6;
unfold eq' in *.

```

inverts keep H is same to *invert keep H* except that it applies `subst` to all the equalities generated by the inversion.

```

Tactic Notation "inverts" "keep" hyp(H) :=
  inverts_tactic H ltac:(intro) ltac:(intro) ltac:(intro)
    ltac:(intro) ltac:(intro) ltac:(intro).

```

inverts keep H as X1 .. XN is the same as *invert keep H as X1 .. XN* except that it applies `subst` to all the equalities generated by the inversion

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1) :=
  inverts_tactic H ltac:(intros I1)
    ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) := 
  inverts_tactic H ltac:(intros I1) ltac:(intros I2)
    ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) := 
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)
    ltac:(intro) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4) := 
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)
    ltac:(intros I4) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
simple_intropattern(I5) := 
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)
    ltac:(intros I4) ltac:(intros I5) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)

```

```

simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
simple_intropattern(I5) simple_intropattern(I6) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)
  ltac:(intros I4) ltac:(intros I5) ltac:(intros I6).

```

inverts H is same to *inverts keep H* except that it clears hypothesis *H*.

```

Tactic Notation "inverts" hyp(H) :=
  inverts keep H; try clear H.

```

inverts H as X1 .. XN is the same as *inverts keep H as X1 .. XN* but it also clears the hypothesis *H*.

```

Tactic Notation "inverts_tactic" hyp(H) tactic(tac) :=
  let H' := fresh "TEMP" in rename H into H'; tac H'; clear H'.

```

```

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1) :=
  invert_tactic H (fun H => inverts keep H as I1).

```

```

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) :=
  invert_tactic H (fun H => inverts keep H as I1 I2).

```

```

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3).

```

```

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3 I4).

```

```

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
simple_intropattern(I5) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3 I4 I5).

```

```

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
simple_intropattern(I5) simple_intropattern(I6) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3 I4 I5 I6).

```

inverts H as performs an inversion on hypothesis *H*, substitutes generated equalities, and put in the goal the other freshly created hypotheses, for the user to name explicitly. *inverts keep H as* is the same except that it does not clear *H*. TODO: reimplement *inverts* above using this one

```

Ltac inverts_as_tactic H :=
  let rec go tt :=
    match goal with
    | ⊢ (ltac_mark → _) ⇒ intros _
    | ⊢ (?x = ?y → _) ⇒ let H := fresh "TEMP" in intro H;
      first [ subst x | subst y ];
      go tt

```

```

| ⊢ (existT ?P ?p ?x = existT ?P ?p ?y → _) ⇒
  let H := fresh "TEMP" in intro H;
  generalize (@inj_pair2 _ P p x y H);
  clear H; go tt
| ⊢ (∀ _, _) ⇒
  intro; let H := get_last_hyp tt in mark_to_generalize H; go tt
end in
pose ltac_mark; inversion H;
generalize ltac_mark; gen_until_mark;
go tt; gen_to_generalize; unfolds ltac_to_generalize;
unfold eq' in *.

Tactic Notation "inverts" "keep" hyp(H) "as" :=
  inverts_as_tactic H.

Tactic Notation "inverts" hyp(H) "as" :=
  inverts_as_tactic H; clear H.

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
simple_intropattern(I5) simple_intropattern(I6) simple_intropattern(I7) :=
  inverts H as; introv I1 I2 I3 I4 I5 I6 I7.

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
simple_intropattern(I5) simple_intropattern(I6) simple_intropattern(I7)
simple_intropattern(I8) :=
  inverts H as; introv I1 I2 I3 I4 I5 I6 I7 I8.

lets_inverts E as I1 .. IN is intuitively equivalent to inverts E, with the difference that it applies to any expression and not just to the name of an hypothesis.

Ltac lets_inverts_base E cont :=
  let H := fresh "TEMP" in lets H: E; try cont H.

Tactic Notation "lets_inverts" constr(E) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1)
simple_intropattern(I2) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2 I3).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1)
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2 I3 I4).

```

2.7.3 Injection with substitution

Underlying implementation of *injects*

```
Ltac injects_tactic H :=
let rec go _ :=
  match goal with
  | ⊢ (ltac_mark → _) ⇒ intros _
  | ⊢ (?x = ?y → _) ⇒ let H := fresh "TEMP" in intro H;
    first [ subst x | subst y | idtac ];
    go tt
  end in
generalize ltac_mark; injection H; go tt.
```

injects keep H takes an hypothesis *H* of the form $C \ a1 \dots aN = C \ b1 \dots bN$ and substitute all equalities $ai = bi$ that have been generated.

```
Tactic Notation "injects" "keep" hyp(H) :=
injects_tactic H.
```

injects H is similar to *injects keep H* but clears the hypothesis *H*.

```
Tactic Notation "injects" hyp(H) :=
injects_tactic H; clear H.
```

inject H as X1 .. XN is the same as *injection* followed by *intros X1 .. XN*

```
Tactic Notation "inject" hyp(H) :=
injection H.
```

```
Tactic Notation "inject" hyp(H) "as" ident(X1) :=
injection H; intros X1.
```

```
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) :=
injection H; intros X1 X2.
```

```
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) ident(X3) :=
injection H; intros X1 X2 X3.
```

```
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) ident(X3)
ident(X4) :=
injection H; intros X1 X2 X3 X4.
```

```
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) ident(X3)
ident(X4) ident(X5) :=
injection H; intros X1 X2 X3 X4 X5.
```

2.7.4 Inversion and injection with substitution –rough implementation

The tactics *inversions* and *injections* provided in this section are similar to *inverts* and *injects* except that they perform substitution on all equalities from the context and not only the ones freshly generated. The counterpart is that they have simpler implementations.

DEPRECATED: these tactics should no longer be used.

inversions keep H is the same as *inversions* H but it does not clear hypothesis H .

Tactic Notation "inversions" "keep" $hyp(H) :=$
 $\text{inversion } H; \text{subst}.$

inversions H is a shortcut for *inversion* H followed by *subst* and *clear* H . It is a rough implementation of *inverts keep* H which behave badly when the proof context already contains equalities. It is provided in case the better implementation turns out to be too slow.

Tactic Notation "inversions" $hyp(H) :=$
 $\text{inversion } H; \text{subst; try clear } H.$

injections keep H is the same as *injection* H followed by *intros* and *subst*. It is a rough implementation of *injects keep* H which behave badly when the proof context already contains equalities, or when the goal starts with a forall or an implication.

Tactic Notation "injections" "keep" $hyp(H) :=$
 $\text{injection } H; \text{intros; subst}.$

injections H is the same as *injection* H followed by *clear* H and *intros* and *subst*. It is a rough implementation of *injects keep* H which behave badly when the proof context already contains equalities, or when the goal starts with a forall or an implication.

Tactic Notation "injections" "keep" $hyp(H) :=$
 $\text{injection } H; \text{clear } H; \text{intros; subst}.$

2.7.5 Case analysis

cases is similar to *case_eq* E except that it generates the equality in the context and not in the goal, and generates the equality the other way round. The syntax *cases* E as H allows specifying the name H of that hypothesis.

Tactic Notation "cases" constr(E) "as" ident(H) :=
 $\text{let } X := \text{fresh "TEMP"} \text{ in}$
 $\text{set } (X := E) \text{ in } *; \text{def_to_eq_sym } X H E;$
 $\text{destruct } X.$

Tactic Notation "cases" constr(E) :=
 $\text{let } H := \text{fresh "Eq"} \text{ in cases } E \text{ as } H.$

case_if_post H is to be defined later as a tactic to clean up hypothesis H and the goal. By defaults, it looks for obvious contradictions. Currently, this tactic is extended in LibReflect to clean up boolean propositions.

Ltac *case_if_post* $H :=$
 $\text{tryfalse}.$

case_if looks for a pattern of the form `if ?B then ?E1 else ?E2` in the goal, and perform a case analysis on B by calling *destruct* B . Subgoals containing a contradiction

are discarded. *case_if* looks in the goal first, and otherwise in the first hypothesis that contains an *if* statement. *case_if in H* can be used to specify which hypothesis to consider. Syntaxes *case_if as Eq* and *case_if in H as Eq* allows to name the hypothesis coming from the case analysis.

```
Ltac case_if_on_tactic_core E Eq :=
  match type of E with
  | { }+{ } => destruct E as [Eq | Eq]
  | _ => let X := fresh "TEMP" in
    sets_eq ← X Eq: E;
    destruct X
  end.
```

```
Ltac case_if_on_tactic E Eq :=
  case_if_on_tactic_core E Eq; case_if_post Eq.
```

```
Tactic Notation "case_if_on" constr(E) "as" simple_intropattern(Eq) :=
  case_if_on_tactic E Eq.
```

```
Tactic Notation "case_if" "as" simple_intropattern(Eq) :=
  match goal with
  | ⊢ context [if ?B then _ else _] => case_if_on B as Eq
  | K: context [if ?B then _ else _] ⊢ _ => case_if_on B as Eq
  end.
```

```
Tactic Notation "case_if" "in" hyp(H) "as" simple_intropattern(Eq) :=
  match type of H with context [if ?B then _ else _] =>
  case_if_on B as Eq end.
```

```
Tactic Notation "case_if" :=
  let Eq := fresh "C" in case_if as Eq.
```

```
Tactic Notation "case_if" "in" hyp(H) :=
  let Eq := fresh "C" in case_if in H as Eq.
```

cases_if is similar to *case_if* with two main differences: if it creates an equality of the form $x = y$ and then substitutes it in the goal

```
Ltac cases_if_on_tactic_core E Eq :=
  match type of E with
  | { }+{ } => destruct E as [Eq|Eq]; try subst_hyp Eq
  | _ => let X := fresh "TEMP" in
    sets_eq ← X Eq: E;
    destruct X
  end.
```

```
Ltac cases_if_on_tactic E Eq :=
  cases_if_on_tactic_core E Eq; tryfalse; case_if_post Eq.
```

```
Tactic Notation "cases_if_on" constr(E) "as" simple_intropattern(Eq) :=
```

cases_if_on_tactic E Eq .

Tactic Notation "cases_if" "as" *simple_intropattern*(Eq) :=
 match goal with
 | \vdash context [if ? B then _ else_] \Rightarrow cases_if_on B as Eq
 | K : context [if ? B then _ else_] \vdash _ \Rightarrow cases_if_on B as Eq
 end.

Tactic Notation "cases_if" "in" *hyp*(H) "as" *simple_intropattern*(Eq) :=
 match type of H with context [if ? B then _ else_] \Rightarrow
 cases_if_on B as Eq end.

Tactic Notation "cases_if" :=
 let $Eq :=$ fresh "C" in cases_if as Eq .

Tactic Notation "cases_if" "in" *hyp*(H) :=
 let $Eq :=$ fresh "C" in cases_if in H as Eq .
 case_ifs is like repeat case_if

Ltac case_ifs_core :=
 repeat case_if.

Tactic Notation "case_ifs" :=
 case_ifs_core.

destruct_if looks for a pattern of the form if ? B then ? $E1$ else ? $E2$ in the goal, and perform a case analysis on B by calling *destruct* B . It looks in the goal first, and otherwise in the first hypothesis that contains an if statement.

Ltac destruct_if_post := tryfalse.

Tactic Notation "destruct_if"
 "as" *simple_intropattern*($Eq1$) *simple_intropattern*($Eq2$) :=
 match goal with
 | \vdash context [if ? B then _ else_] \Rightarrow destruct B as [$Eq1 | Eq2$]
 | K : context [if ? B then _ else_] \vdash _ \Rightarrow destruct B as [$Eq1 | Eq2$]
 end;
 destruct_if_post.

Tactic Notation "destruct_if" "in" *hyp*(H)
 "as" *simple_intropattern*($Eq1$) *simple_intropattern*($Eq2$) :=
 match type of H with context [if ? B then _ else_] \Rightarrow
 destruct B as [$Eq1 | Eq2$] end;
 destruct_if_post.

Tactic Notation "destruct_if" "as" *simple_intropattern*(Eq) :=
 destruct_if as Eq Eq .

Tactic Notation "destruct_if" "in" *hyp*(H) "as" *simple_intropattern*(Eq) :=
 destruct_if in H as Eq Eq .

Tactic Notation "destruct_if" :=

```

let Eq := fresh "C" in destruct_if as Eq Eq.
Tactic Notation "destruct_if" "in" hyp(H) :=
  let Eq := fresh "C" in destruct_if in H as Eq Eq.

```

BROKEN since v8.5beta2. TODO: cleanup.

destruct_head_match performs a case analysis on the argument of the head pattern matching when the goal has the form `match ?E with ...` or `match ?E with ... = _ or _ = match ?E with` Due to the limits of Ltac, this tactic will not fail if a match does not occur. Instead, it might perform a case analysis on an unspecified subterm from the goal. Warning: experimental.

```

Ltac find_head_match T :=
  match T with context [?E] =>
    match T with
      | E => fail 1
      | _ => constr:(E)
    end
  end.

```

```

Ltac destruct_head_match_core cont :=
  match goal with | ⊢ ?T1 = ?T2 => first [
    let E := find_head_match T1 in cont E
    | let E := find_head_match T2 in cont E
    | ⊢ ?T1 => let E := find_head_match T1 in cont E
  end;
  destruct_if_post.

```

```

Tactic Notation "destruct_head_match" "as" simple_intropattern(I) :=
  destruct_head_match_core ltac:(fun E => destruct E as I).

```

```

Tactic Notation "destruct_head_match" :=
  destruct_head_match_core ltac:(fun E => destruct E).

```

cases' E is similar to *case_eq* E except that it generates the equality in the context and not in the goal. The syntax *cases'* E as H allows specifying the name H of that hypothesis.

```

Tactic Notation "cases'" constr(E) "as" ident(H) :=
  let X := fresh "TEMP" in
  set (X := E) in *; def_to_eq X H E;
  destruct X.

```

```

Tactic Notation "cases'" constr(E) :=
  let x := fresh "Eq" in cases' E as H.

```

cases_if' is similar to *cases_if* except that it generates the symmetric equality.

```

Ltac cases_if_on' E Eq :=
  match type of E with
  | { }+{ } => destruct E as [Eq|Eq]; try subst_hyp Eq
  | _ => let X := fresh "TEMP" in

```

```

sets_eq X Eq: E;
destruct X
end; case_if_post Eq.

Tactic Notation "cases_if" "as" simple_intropattern(Eq) :=
  match goal with
  | ⊢ context [if ?B then _ else _] ⇒ cases_if_on' B Eq
  | K: context [if ?B then _ else _] ⊢ _ ⇒ cases_if_on' B Eq
  end.

Tactic Notation "cases_if" :=
  let Eq := fresh "C" in cases_if' as Eq.

```

2.8 Induction

inductions E is a shorthand for *dependent induction E*. *inductions E gen X1 .. XN* is a shorthand for *dependent induction E generalizing X1 .. XN*.

Require Import Coq.Program.Equality.

```
Ltac inductions_post :=
  unfold eq' in *.
```

```

Tactic Notation "inductions" ident(E) :=
  dependent induction E; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) :=
  dependent induction E generalizing X1; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2) :=
  dependent induction E generalizing X1 X2; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) :=
  dependent induction E generalizing X1 X2 X3; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) :=
  dependent induction E generalizing X1 X2 X3 X4; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) :=
  dependent induction E generalizing X1 X2 X3 X4 X5; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) ident(X6) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) ident(X6) ident(X7) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6 X7; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) ident(X6) ident(X7) ident(X8) :=
```

`dependent induction E generalizing X1 X2 X3 X4 X5 X6 X7 X8; inductions_post.`

`induction_wf IH: E X` is used to apply the well-founded induction principle, for a given well-founded relation. It applies to a goal PX where PX is a proposition on X . First, it sets up the goal in the form $(\text{fun } a \Rightarrow P a) X$, using pattern X , and then it applies the well-founded induction principle instantiated on E .

Here E may be either:

- a proof of `wf R` for R of type $A \rightarrow A \rightarrow \text{Prop}$
- a binary relation of type $A \rightarrow A \rightarrow \text{Prop}$
- a measure of type $A \rightarrow \text{nat}$ // only when LibWf is used

```
Ltac induction_wf_process_wf_hyp tt :=
  idtac.
```

```
Ltac induction_wf_process_measure E :=
  fail.
```

```
Ltac induction_wf_core_then IH E X cont :=
  let clearX tt :=
    first [ clear X | fail 3 "the variable on which the induction is done appears in the
hypotheses" ] in
    pattern X;
  first [ eapply (@well_founded_ind _ E)
    | eapply (@well_founded_ind _ (E _))
    | eapply (@well_founded_ind _ (E _ _))
    | eapply (@well_founded_ind _ (E _ _ _))
    | induction_wf_process_measure E
    | applys well_founded_ind E ];
  clearX tt;
  first [ induction_wf_process_wf_hyp tt
    | intros X IH; cont tt ].
```

```
Ltac induction_wf_core IH E X :=
  induction_wf_core_then IH E X ltac:(fun _ => idtac).
```

```
Tactic Notation "induction_wf" ident(IH) ":" constr(E) ident(X) :=
  induction_wf_core IH E X.
```

Induction on the height of a derivation: the helper tactic `induct_height` helps proving the equivalence of the auxiliary judgment that includes a counter for the maximal height (see LibTacticsDemos for an example)

```
Require Import Coq.Arith.Compare_dec.
```

```
Require Import Coq.micromega.Lia.
```

```
Lemma induct_height_max2 : ∀ n1 n2 : nat,
```

$\exists n, n1 < n \wedge n2 < n.$

Proof using.

```
intros. destruct (lt_dec n1 n2).
 $\exists (\text{S } n2).$  lia.
 $\exists (\text{S } n1).$  lia.
```

Qed.

```
Ltac induct_height_step x :=
match goal with
| H:  $\exists \_, \_ \vdash \_ \Rightarrow$ 
  let n := fresh "n" in let y := fresh "x" in
  destruct H as [n ?];
  forwards (y&?&?): induct_height_max2 n x;
  induct_height_step y
| _  $\Rightarrow \exists (\text{S } x); \text{eauto}$ 
end.
```

```
Ltac induct_height := induct_height_step O.
```

2.9 Coinduction

Tactic `cofixs IH` is like `cofix IH` except that the coinduction hypothesis is tagged in the form `IH: COIND P` instead of being just `IH: P`. This helps other tactics clearing the coinduction hypothesis using `clear_coind`

Definition `COIND (P:Prop) := P.`

```
Tactic Notation "cofixs" ident(IH) :=
  cofix IH;
  match type of IH with ?P  $\Rightarrow$  change P with (COIND P) in IH end.
```

Tactic `clear_coind` clears all the coinduction hypotheses, assuming that they have been tagged

```
Ltac clear_coind :=
repeat match goal with H: COIND _  $\vdash \_ \Rightarrow$  clear H end.
```

Tactic `abstracts tac` is like `abstract tac` except that it clears the coinduction hypotheses so that the productivity check will be happy. For example, one can use `abstracts lia` to obtain the same behavior as `lia` but with an auxiliary lemma being generated.

```
Tactic Notation "abstracts" tactic(tac) :=
  clear_coind; tac.
```

2.10 Decidable equality

decides_equality is the same as *decide equality* excepts that it is able to unfold definitions at head of the current goal.

```
Ltac decides_equality_tactic :=  
  first [ decide equality | progress(unfolds); decides_equality_tactic ].
```

```
Tactic Notation "decides_equality" :=  
  decides_equality_tactic.
```

2.11 Equivalence

iff H can be used to prove an equivalence $P \leftrightarrow Q$ and name H the hypothesis obtained in each case. The syntaxes *iff* and *iff H1 H2* are also available to specify zero or two names. The tactic *iff* $\leftarrow H$ swaps the two subgoals, i.e. produces $(Q \rightarrow P)$ as first subgoal.

```
Lemma iff_intro_swap : ∀ (P Q : Prop),  
  (Q → P) → (P → Q) → (P ↔ Q).
```

Proof using. intuition. Qed.

```
Tactic Notation "iff" simple_intropattern(H1) simple_intropattern(H2) :=  
  split; [ intros H1 | intros H2 ].
```

```
Tactic Notation "iff" simple_intropattern(H) :=  
  iff H H.
```

```
Tactic Notation "iff" :=  
  let H := fresh "H" in iff H.
```

```
Tactic Notation "iff" "<->" simple_intropattern(H1) simple_intropattern(H2) :=  
  apply iff_intro_swap; [ intros H1 | intros H2 ].
```

```
Tactic Notation "iff" "<->" simple_intropattern(H) :=  
  iff ← H H.
```

```
Tactic Notation "iff" "<->" :=  
  let H := fresh "H" in iff ← H.
```

2.12 N-ary Conjunctions and Disjunctions

Underlying implementation of *splits*.

```
Ltac splits_tactic N :=  
  match N with  
  | O ⇒ fail  
  | S O ⇒ idtac  
  | S ?N' ⇒ split; [] splits_tactic N'  
  end.
```

```
Ltac unfold_goal_until_conjunction :=
  match goal with
  | ⊢ _ ∧ _ ⇒ idtac
  | _ ⇒ progress(unfolds); unfold_goal_until_conjunction
  end.
```

```
Ltac get_term_conjunction_arity T :=
  match T with
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(8)
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(7)
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(6)
  | _ ∧ _ ∧ _ ∧ _ ⇒ constr:(5)
  | _ ∧ _ ∧ _ ⇒ constr:(4)
  | _ ∧ _ ⇒ constr:(3)
  | _ ⇒ constr:(2)
  | _ → ?T' ⇒ get_term_conjunction_arity T'
  | _ ⇒ let P := get_head T in
    let T' := eval unfold P in T in
    match T' with
    | T ⇒ fail 1
    | _ ⇒ get_term_conjunction_arity T'
  end

end.
```

```
Ltac get_goal_conjunction_arity :=
  match goal with ⊢ ?T ⇒ get_term_conjunction_arity T end.
```

splits applies to a goal of the form $(T_1 \wedge \dots \wedge T_N)$ and destruct it into N subgoals $T_1 \dots T_N$. If the goal is not a conjunction, then it unfolds the head definition.

```
Tactic Notation "splits" :=
  unfold_goal_until_conjunction;
  let N := get_goal_conjunction_arity in
  splits_tactic N.
```

splits N is similar to *splits*, except that it will unfold as many definitions as necessary to obtain an N -ary conjunction.

```
Tactic Notation "splits" constr(N) :=
  let N := number_to_nat N in
  splits_tactic N.
```

Underlying implementation of *destructs*.

```
Ltac destructs_conjunction_tactic N T :=
  match N with
  | 2 ⇒ destruct T as [? ?]
```

```

| 3 => destruct T as [? [? ?]]
| 4 => destruct T as [? [? [? ?]]]
| 5 => destruct T as [? [? [? [? ?]]]]
| 6 => destruct T as [? [? [? [? [? ?]]]]]
| 7 => destruct T as [? [? [? [? [? [? ?]]]]]]
end.

```

destructs T allows destructing a term T which is a N-ary conjunction. It is equivalent to `destruct T as (H1 .. HN)`, except that it does not require to manually specify N different names.

```

Tactic Notation "destructs" constr(T) :=
let TT := type of T in
let N := get_term_conjunction_arity TT in
destructs_conjunction_tactic N T.

```

destructs N T is equivalent to `destruct T as (H1 .. HN)`, except that it does not require to manually specify N different names. Remark that it is not restricted to N-ary conjunctions.

```

Tactic Notation "destructs" constr(N) constr(T) :=
let N := number_to_nat N in
destructs_conjunction_tactic N T.

```

Underlying implementation of *branch*.

```

Ltac branch_tactic K N :=
match constr:(K,N) with
| (_,0) => fail 1
| (0,_) => fail 1
| (1,1) => idtac
| (1,_) => left
| (S ?K', S ?N') => right; branch_tactic K' N'
end.

```

```

Ltac unfold_goal_until_disjunction :=
match goal with
| ⊢ _ ∨ _ => idtac
| _ => progress(unfolds); unfold_goal_until_disjunction
end.

```

```

Ltac get_term_disjunction_arity T :=
match T with
| _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ => constr:(8)
| _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ => constr:(7)
| _ ∨ _ ∨ _ ∨ _ ∨ _ => constr:(6)
| _ ∨ _ ∨ _ ∨ _ => constr:(5)
| _ ∨ _ ∨ _ ∨ _ => constr:(4)

```

```

| _  $\vee$  _  $\vee$  _  $\Rightarrow$  constr:(3)
| _  $\vee$  _  $\Rightarrow$  constr:(2)
| _  $\rightarrow$  ?T'  $\Rightarrow$  get_term_disjunction_arity T'
| _  $\Rightarrow$  let P := get_head T in
    let T' := eval unfold P in T in
    match T' with
    | T  $\Rightarrow$  fail 1
    | _  $\Rightarrow$  get_term_disjunction_arity T'
    end
end.

```

```

Ltac get_goal_disjunction_arity :=
  match goal with  $\vdash$  ?T  $\Rightarrow$  get_term_disjunction_arity T end.

```

branch N applies to a goal of the form $P_1 \vee \dots \vee P_K \vee \dots \vee P_N$ and leaves the goal P_K . It only able to unfold the head definition (if there is one), but for more complex unfolding one should use the tactic *branch K of N*.

```

Tactic Notation "branch" constr(K) :=
  let K := number_to_nat K in
  unfold_goal_until_disjunction;
  let N := get_goal_disjunction_arity in
  branch_tactic K N.

```

branch K of N is similar to *branch K* except that the arity of the disjunction N is given manually, and so this version of the tactic is able to unfold definitions. In other words, applies to a goal of the form $P_1 \vee \dots \vee P_K \vee \dots \vee P_N$ and leaves the goal P_K .

```

Tactic Notation "branch" constr(K) "of" constr(N) :=
  let N := number_to_nat N in
  let K := number_to_nat K in
  branch_tactic K N.

```

Underlying implementation of *branches*.

```

Ltac destructs_disjunction_tactic N T :=
  match N with
  | 2  $\Rightarrow$  destruct T as [? | ?]
  | 3  $\Rightarrow$  destruct T as [? | [? | ?]]
  | 4  $\Rightarrow$  destruct T as [? | [? | [? | ?]]]
  | 5  $\Rightarrow$  destruct T as [? | [? | [? | [? | ?]]]]
  end.

```

branches T allows destructing a term T which is a N -ary disjunction. It is equivalent to *destruct T as [H₁ | .. | H_N]*, and produces N subgoals corresponding to the N possible cases.

```

Tactic Notation "branches" constr(T) :=
  let TT := type of T in

```

```
let N := get_term_disjunction_arity TT in
destructs_disjunction_tactic N T.
```

branches N T is the same as *branches T* except that the arity is forced to *N*. This version is useful to unfold definitions on the fly.

```
Tactic Notation "branches" constr(N) constr(T) :=
  let N := number_to_nat N in
  destructs_disjunction_tactic N T.
```

branches automatically finds a hypothesis *h* that is a disjunction and destructs it.

```
Tactic Notation "branches" :=
  match goal with h: _ ∨ _ ⊢ _ ⇒ branches h end.
```

$\exists T_1 \dots T_N$ is a shorthand for $\exists T_1; \dots; \exists T_N$. It is intended to prove goals of the form *exist X1 .. XN, P*. If an argument provided is *__* (double underscore), then an evar is introduced. $\exists T_1 \dots T_N __$ is equivalent to $\exists T_1 \dots T_N __ \dots __$ with as many *__* as possible.

```
Tactic Notation "exists_original" constr(T1) :=
  ∃ T1.
```

```
Tactic Notation "exists" constr(T1) :=
  match T1 with
  | ltac_wild ⇒ esplit
  | ltac_wilds ⇒ repeat esplit
  | _ ⇒ ∃ T1
  end.
```

```
Tactic Notation "exists" constr(T1) constr(T2) :=
  ∃ T1; ∃ T2.
```

```
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) :=
  ∃ T1; ∃ T2; ∃ T3.
```

```
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4.
```

```
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
  constr(T5) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5.
```

```
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
  constr(T5) constr(T6) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5; ∃ T6.
```

For compatibility with Coq syntax, $\exists T_1, \dots, T_N$ is also provided.

```
Tactic Notation "exists" constr(T1) "," constr(T2) :=
  ∃ T1 T2.
```

```
Tactic Notation "exists" constr(T1) "," constr(T2) "," constr(T3) :=
  ∃ T1 T2 T3.
```

```
Tactic Notation "exists" constr(T1) "," constr(T2) "," constr(T3) "," constr(T4) :=
```

```

 $\exists T_1 T_2 T_3 T_4.$ 
Tactic Notation "exists" constr( $T_1$ ) "," constr( $T_2$ ) "," constr( $T_3$ ) "," constr( $T_4$ ) ","  

constr( $T_5$ ) :=  

 $\exists T_1 T_2 T_3 T_4 T_5.$ 
Tactic Notation "exists" constr( $T_1$ ) "," constr( $T_2$ ) "," constr( $T_3$ ) "," constr( $T_4$ ) ","  

constr( $T_5$ ) "," constr( $T_6$ ) :=  

 $\exists T_1 T_2 T_3 T_4 T_5 T_6.$ 

```

The tactic \exists , without arguments, repeats *esplit* as many times as there are visible existentials at the head of the goal. If there are zero existentials visible, the tactic `hnf` is called once. If there are still zero existentials visible, the tactic fails.

To obtain a tactic that supports arbitrary number of existentials including the possibility of having zero existentials, use `try` \exists .

```

Ltac exists_noarg_positive tt :=  

  match goal with  

  |  $\vdash \exists \_, \_ \Rightarrow esplit$ ; try exists_noarg_positive tt  

  end.  
  

Ltac exists_noarg tt :=  

  match goal with  

  |  $\vdash \exists \_, \_ \Rightarrow exists\_noarg\_positive tt$   

  |  $\_ \Rightarrow hnf$ ; exists_noarg tt  

  end.  
  

Tactic Notation "exists" :=  

  exists_noarg tt.

```

```

Definition def_with_exists (n:nat) : Prop :=  

   $\exists x, x = n.$ 

```

The tactic *exists_nounfold* is similar to \exists , except that it does nothing if there are no existentials visible in the goal. This tactic may be useful for programming other tactics in a robust way.

```

Tactic Notation "exists_nounfold" :=  

  try exists_noarg_positive tt.

```

unpack or *unpack H* destructs conjunctions and existentials in all or one hypothesis.

```

Ltac unpack_core :=  

  repeat match goal with  

  |  $H: \_ \wedge \_ \vdash \_ \Rightarrow destruct H$   

  |  $H: \exists (varname: \_), \_ \vdash \_ \Rightarrow$   

    let name := fresh varname in  

    destruct H as [name ?]  

  end.

```

```

Ltac unpack_hypothesis H :=
  try match type of H with
  | _ ∧ _ ⇒
    let h1 := fresh "TEMP" in
    let h2 := fresh "TEMP" in
    destruct H as [ h1 h2 ];
    unpack_hypothesis h1;
    unpack_hypothesis h2
  | ∃ (varname: _), _ ⇒
    let name := fresh varname in
    let body := fresh "TEMP" in
    destruct H as [name body];
    unpack_hypothesis body
  end.

Tactic Notation "unpack" :=
  unpack_core.

Tactic Notation "unpack" constr(H) :=
  unpack_hypothesis H.

```

2.13 Tactics to prove typeclass instances

typeclass is an automation tactic specialized for finding typeclass instances.

```

Tactic Notation "typeclass" :=
  let go _ := eauto with typeclass_instances in
  solve [ go tt | constructor; go tt ].

```

solve_typeclass is a simpler version of *typeclass*, to use in hint tactics for resolving instances

```

Tactic Notation "solve_typeclass" :=
  solve [ eauto with typeclass_instances ].

```

2.14 Tactics to invoke automation

2.14.1 Definitions for parsing compatibility

```

Tactic Notation "f_equal" :=
  f_equal.

Tactic Notation "constructor" :=
  constructor.

Tactic Notation "simple" :=

```

```
simpl.
```

```
Tactic Notation "split" :=
  split.
```

```
Tactic Notation "right" :=
  right.
```

```
Tactic Notation "left" :=
  left.
```

2.14.2 *hint* to add hints local to a lemma

hint E adds *E* as an hypothesis so that automation can use it. Syntax *hint E₁,..,E_N* is available

```
Tactic Notation "hint" constr(E) :=
  let H := fresh "Hint" in let H: E.
```

```
Tactic Notation "hint" constr(E1) "," constr(E2) :=
  hint E1; hint E2.
```

```
Tactic Notation "hint" constr(E1) "," constr(E2) "," constr(E3) :=
  hint E1; hint E2; hint(E3).
```

```
Tactic Notation "hint" constr(E1) "," constr(E2) "," constr(E3) "," constr(E4) :=
  hint E1; hint E2; hint(E3); hint(E4).
```

2.14.3 *jauto*, a new automation tactics

jauto is better at *intuition eauto* because it can open existentials from the context. In the same time, *jauto* can be faster than *intuition eauto* because it does not destruct disjunctions from the context. The strategy of *jauto* can be summarized as follows:

- open all the existentials and conjunctions from the context
- call *esplit* and *split* on the existentials and conjunctions in the goal
- call *eauto*.

```
Tactic Notation "jauto" :=
  try solve [ jauto_set; eauto ].
```

```
Tactic Notation "jauto_fast" :=
  try solve [ auto | eauto | jauto ].
```

iauto is a shorthand for *intuition eauto*

```
Tactic Notation "iauto" := try solve [intuition eauto].
```

2.14.4 Definitions of automation tactics

The two following tactics defined the default behaviour of “light automation” and “strong automation”. These tactics may be redefined at any time using the syntax `Ltac .. ::= ...`

auto_tilde is the tactic which will be called each time a symbol \neg is used after a tactic.

```
Ltac auto_tilde_default := auto.
```

```
Ltac auto_tilde := auto_tilde_default.
```

auto_star is the tactic which will be called each time a symbol \times is used after a tactic.

```
Ltac auto_star_default := try solve [ auto | eauto | intuition eauto ].
```

```
Ltac auto_star := auto_star_default.
```

autos \neg is a notation for tactic *auto_tilde*. It may be followed by lemmas (or proofs terms) which auto will be able to use for solving the goal.

autos is an alias for *autos \neg*

```
Tactic Notation "autos" :=
```

```
  auto_tilde.
```

```
Tactic Notation "autos" " $\sim$ " :=
```

```
  auto_tilde.
```

```
Tactic Notation "autos" " $\sim$ " constr(E1) :=
```

```
  lets: E1; auto_tilde.
```

```
Tactic Notation "autos" " $\sim$ " constr(E1) constr(E2) :=
```

```
  lets: E1; lets: E2; auto_tilde.
```

```
Tactic Notation "autos" " $\sim$ " constr(E1) constr(E2) constr(E3) :=
```

```
  lets: E1; lets: E2; lets: E3; auto_tilde.
```

autos \times is a notation for tactic *auto_star*. It may be followed by lemmas (or proofs terms) which auto will be able to use for solving the goal.

```
Tactic Notation "autos" "*" :=
```

```
  auto_star.
```

```
Tactic Notation "autos" "*" constr(E1) :=
```

```
  lets: E1; auto_star.
```

```
Tactic Notation "autos" "*" constr(E1) constr(E2) :=
```

```
  lets: E1; lets: E2; auto_star.
```

```
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) :=
```

```
  lets: E1; lets: E2; lets: E3; auto_star.
```

auto_false is a version of auto able to spot some contradictions. There is an ad-hoc support for goals in \leftrightarrow : split is called first. *auto_false \neg* and *auto_false \times* are also available.

```
Ltac auto_false_base cont :=
```

```
  try solve [
```

```
    intros_all; try match goal with  $\vdash \_ \leftrightarrow \_ \Rightarrow$  split end;
```

```
    solve [ cont tt | intros_all; false; cont tt ] ].
```

```
Tactic Notation "auto_false" :=
```

```

auto_false_base ltac:(fun tt => auto).
Tactic Notation "auto_false" "˜" :=
  auto_false_base ltac:(fun tt => auto_tilde).
Tactic Notation "auto_false" "*" :=
  auto_false_base ltac:(fun tt => auto_star).
Tactic Notation "dauto" :=
  dintuition eauto.

```

2.14.5 Parsing for light automation

Any tactic followed by the symbol \neg will have *auto_tilde* called on all of its subgoals. Three exceptions:

- *cuts* and *asserts* only call `auto` on their first subgoal,
- `apply` \neg relies on *sapply* rather than *apply*,
- `tryfalse` \neg is defined as `tryfalse` by *auto_tilde*.

Some builtin tactics are not defined using tactic notations and thus cannot be extended, e.g. `simpl` and `unfold`. For these, notation such as `simpl` \neg will not be available.

```

Tactic Notation "equates" "˜" constr(E) :=
  equates E; auto_tilde.
Tactic Notation "equates" "˜" constr(n1) constr(n2) :=
  equates n1 n2; auto_tilde.
Tactic Notation "equates" "˜" constr(n1) constr(n2) constr(n3) :=
  equates n1 n2 n3; auto_tilde.
Tactic Notation "equates" "˜" constr(n1) constr(n2) constr(n3) constr(n4) :=
  equates n1 n2 n3 n4; auto_tilde.
Tactic Notation "applys_eq" "˜" constr(H) :=
  applys_eq H; auto_tilde.
Tactic Notation "applys_eq" "˜" constr(H) constr(E) :=
  applys_eq H E; auto_tilde.
Tactic Notation "applys_eq" "˜" constr(H) constr(n1) constr(n2) :=
  applys_eq H n1 n2; auto_tilde.
Tactic Notation "applys_eq" "˜" constr(H) constr(n1) constr(n2) constr(n3) :=
  applys_eq H n1 n2 n3; auto_tilde.
Tactic Notation "applys_eq" "˜" constr(H) constr(n1) constr(n2) constr(n3) constr(n4) :=
  applys_eq H n1 n2 n3 n4; auto_tilde.
Tactic Notation "apply" "˜" constr(H) :=
  sapply H; auto_tilde.

```

```

Tactic Notation "destruct" "~~" constr(H) :=
  destruct H; auto_tilde.
Tactic Notation "destruct" "~~" constr(H) "as" simple_intropattern(I) :=
  destruct H as I; auto_tilde.
Tactic Notation "f_equal" "~~" :=
  f_equal; auto_tilde.
Tactic Notation "induction" "~~" constr(H) :=
  induction H; auto_tilde.
Tactic Notation "inversion" "~~" constr(H) :=
  inversion H; auto_tilde.
Tactic Notation "split" "~~" :=
  split; auto_tilde.
Tactic Notation "subst" "~~" :=
  subst; auto_tilde.
Tactic Notation "right" "~~" :=
  right; auto_tilde.
Tactic Notation "left" "~~" :=
  left; auto_tilde.
Tactic Notation "constructor" "~~" :=
  constructor; auto_tilde.
Tactic Notation "constructors" "~~" :=
  constructors; auto_tilde.
Tactic Notation "false" "~~" :=
  false; auto_tilde.
Tactic Notation "false" "~~" constr(E) :=
  false_then E ltac:(fun _ => auto_tilde).
Tactic Notation "false" "~~" constr(E0) constr(E1) :=
  false¬ (» E0 E1).
Tactic Notation "false" "~~" constr(E0) constr(E1) constr(E2) :=
  false¬ (» E0 E1 E2).
Tactic Notation "false" "~~" constr(E0) constr(E1) constr(E2) constr(E3) :=
  false¬ (» E0 E1 E2 E3).
Tactic Notation "false" "~~" constr(E0) constr(E1) constr(E2) constr(E3) constr(E4) :=
  false¬ (» E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "~~" :=
  try solve [ false¬ ].
Tactic Notation "asserts" "~~" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_tilde | idtac ].
Tactic Notation "asserts" "~~" ":" constr(E) :=
  let H := fresh "H" in asserts¬ H: E.
Tactic Notation "cuts" "~~" simple_intropattern(H) ":" constr(E) :=

```

```

cuts H: E; [ auto_tilde | idtac ].
Tactic Notation "cuts" "~~" ":" constr(E) :=
  cuts: E; [ auto_tilde | idtac ].

Tactic Notation "lets" "~~" simple_intropattern(I) ":" constr(E) :=
  lets I: E; auto_tilde.

Tactic Notation "lets" "~~" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  lets I: E0 A1; auto_tilde.

Tactic Notation "lets" "~~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
  lets I: E0 A1 A2; auto_tilde.

Tactic Notation "lets" "~~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets I: E0 A1 A2 A3; auto_tilde.

Tactic Notation "lets" "~~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets I: E0 A1 A2 A3 A4; auto_tilde.

Tactic Notation "lets" "~~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets I: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "lets" "~~" ":" constr(E) :=
  lets: E; auto_tilde.

Tactic Notation "lets" "~~" ":" constr(E0)
  constr(A1) :=
  lets: E0 A1; auto_tilde.

Tactic Notation "lets" "~~" ":" constr(E0)
  constr(A1) constr(A2) :=
  lets: E0 A1 A2; auto_tilde.

Tactic Notation "lets" "~~" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets: E0 A1 A2 A3; auto_tilde.

Tactic Notation "lets" "~~" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: E0 A1 A2 A3 A4; auto_tilde.

Tactic Notation "lets" "~~" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "forwards" "~~" simple_intropattern(I) ":" constr(E) :=
  forwards I: E; auto_tilde.

Tactic Notation "forwards" "~~" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  forwards I: E0 A1; auto_tilde.

```

```

Tactic Notation "forwards" " $\sim$ " simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
    forwards I: E0 A1 A2; auto_tilde.
Tactic Notation "forwards" " $\sim$ " simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
    forwards I: E0 A1 A2 A3; auto_tilde.
Tactic Notation "forwards" " $\sim$ " simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
    forwards I: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "forwards" " $\sim$ " ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    forwards I: E0 A1 A2 A3 A4 A5; auto_tilde.
Tactic Notation "forwards" " $\sim$ " ":" constr(E) :=
  forwards: E; auto_tilde.
Tactic Notation "forwards" " $\sim$ " ":" constr(E0)
  constr(A1) :=
    forwards: E0 A1; auto_tilde.
Tactic Notation "forwards" " $\sim$ " ":" constr(E0)
  constr(A1) constr(A2) :=
    forwards: E0 A1 A2; auto_tilde.
Tactic Notation "forwards" " $\sim$ " ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
    forwards: E0 A1 A2 A3; auto_tilde.
Tactic Notation "forwards" " $\sim$ " ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
    forwards: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "forwards" " $\sim$ " ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    forwards: E0 A1 A2 A3 A4 A5; auto_tilde.
Tactic Notation "applys" " $\sim$ " constr(H) :=
  applys H; auto_tilde. Tactic Notation "applys" " $\sim$ " constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" " $\sim$ " constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" " $\sim$ " constr(E0) constr(A1) constr(A2) :=
  applys E0 A1 A2; auto_tilde.
Tactic Notation "applys" " $\sim$ " constr(E0) constr(A1) constr(A2) constr(A3) :=
  applys E0 A1 A2 A3; auto_tilde.
Tactic Notation "applys" " $\sim$ " constr(E0) constr(A1) constr(A2) constr(A3) constr(A4) :=
  applys E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "applys" " $\sim$ " constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
  :=
  applys E0 A1 A2 A3 A4; auto_tilde.

```

```

constr(A5) :=
  applys E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "specializes" " $\sim$ " hyp(H) :=
  specializes H; auto_tilde.

Tactic Notation "specializes" " $\sim$ " hyp(H) constr(A1) :=
  specializes H A1; auto_tilde.

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_tilde.

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H A1 A2 A3; auto_tilde.

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4) :=
  specializes H A1 A2 A3 A4; auto_tilde.

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "fapply" " $\sim$ " constr(E) :=
  fapply E; auto_tilde.

Tactic Notation "sapply" " $\sim$ " constr(E) :=
  sapply E; auto_tilde.

Tactic Notation "logic" " $\sim$ " constr(E) :=
  logic_base E ltac:(fun _ => auto_tilde).

Tactic Notation "intros_all" " $\sim$ " :=
  intros_all; auto_tilde.

Tactic Notation "unfolds" " $\sim$ " :=
  unfolds; auto_tilde.

Tactic Notation "unfolds" " $\sim$ " constr(F1) :=
  unfolds F1; auto_tilde.

Tactic Notation "unfolds" " $\sim$ " constr(F1) "," constr(F2) :=
  unfolds F1, F2; auto_tilde.

Tactic Notation "unfolds" " $\sim$ " constr(F1) "," constr(F2) "," constr(F3) :=
  unfolds F1, F2, F3; auto_tilde.

Tactic Notation "unfolds" " $\sim$ " constr(F1) "," constr(F2) "," constr(F3) "," constr(F4) :=
  unfolds F1, F2, F3, F4; auto_tilde.

Tactic Notation "simple" " $\sim$ " :=
  simpl; auto_tilde.

Tactic Notation "simple" " $\sim$ " "in" hyp(H) :=
  simpl in H; auto_tilde.

Tactic Notation "simples" " $\sim$ " :=
  simpls; auto_tilde.

```

```

Tactic Notation "hnfs" "~~" :=
  hnfs; auto_tilde.
Tactic Notation "hnfs" "~~" "in" hyp(H) :=
  hnf in H; auto_tilde.
Tactic Notation "substs" "~~" :=
  substs; auto_tilde.
Tactic Notation "intro_hyp" "~~" hyp(H) :=
  subst_hyp H; auto_tilde.
Tactic Notation "intro_subst" "~~" :=
  intro_subst; auto_tilde.
Tactic Notation "subst_eq" "~~" constr(E) :=
  subst_eq E; auto_tilde.
Tactic Notation "rewrite" "~~" constr(E) :=
  rewrite E; auto_tilde.
Tactic Notation "rewrite" "~~" "<->" constr(E) :=
  rewrite ← E; auto_tilde.
Tactic Notation "rewrite" "~~" "in" hyp(H) :=
  rewrite E in H; auto_tilde.
Tactic Notation "rewrite" "~~" "<->" "in" hyp(H) :=
  rewrite ← E in H; auto_tilde.
Tactic Notation "rewrites" "~~" constr(E) :=
  rewrites E; auto_tilde.
Tactic Notation "rewrites" "~~" "in" hyp(H) :=
  rewrites E in H; auto_tilde.
Tactic Notation "rewrites" "~~" "in" "*" :=
  rewrites E in *; auto_tilde.
Tactic Notation "rewrites" "~~" "<->" constr(E) :=
  rewrites ← E; auto_tilde.
Tactic Notation "rewrites" "~~" "<->" "in" hyp(H) :=
  rewrites ← E in H; auto_tilde.
Tactic Notation "rewrites" "~~" "<->" "in" "*" :=
  rewrites ← E in *; auto_tilde.
Tactic Notation "rewrite_all" "~~" constr(E) :=
  rewrite_all E; auto_tilde.
Tactic Notation "rewrite_all" "~~" "<->" constr(E) :=
  rewrite_all ← E; auto_tilde.
Tactic Notation "rewrite_all" "~~" "in" ident(H) :=
  rewrite_all E in H; auto_tilde.
Tactic Notation "rewrite_all" "~~" "<->" constr(E) "in" ident(H) :=
  rewrite_all ← E in H; auto_tilde.
Tactic Notation "rewrite_all" "~~" "in" "*" :=
  rewrite_all E in *; auto_tilde.

```

```

Tactic Notation "rewrite_all" "~~" "<-" constr(E) "in" "*" :=
  rewrite_all ← E in *; auto_tilde.

Tactic Notation "asserts_rewrite" "~~" constr(E) :=
  asserts_rewrite E; auto_tilde.

Tactic Notation "asserts_rewrite" "~~" "<-" constr(E) :=
  asserts_rewrite ← E; auto_tilde.

Tactic Notation "asserts_rewrite" "~~" constr(E) "in" hyp(H) :=
  asserts_rewrite E in H; auto_tilde.

Tactic Notation "asserts_rewrite" "~~" "<-" constr(E) "in" hyp(H) :=
  asserts_rewrite ← E in H; auto_tilde.

Tactic Notation "asserts_rewrite" "~~" "*" constr(E) "in" "*" :=
  asserts_rewrite E in *; auto_tilde.

Tactic Notation "asserts_rewrite" "~~" "<-" constr(E) "in" "*" :=
  asserts_rewrite ← E in *; auto_tilde.

Tactic Notation "cuts_rewrite" "~~" constr(E) :=
  cuts_rewrite E; auto_tilde.

Tactic Notation "cuts_rewrite" "~~" "<-" constr(E) :=
  cuts_rewrite ← E; auto_tilde.

Tactic Notation "cuts_rewrite" "~~" constr(E) "in" hyp(H) :=
  cuts_rewrite E in H; auto_tilde.

Tactic Notation "cuts_rewrite" "~~" "<-" constr(E) "in" hyp(H) :=
  cuts_rewrite ← E in H; auto_tilde.

Tactic Notation "erewrite" "~~" constr(E) :=
  erewrite E; auto_tilde.

Tactic Notation "fequal" "~~" :=
  fequal; auto_tilde.

Tactic Notation "fequals" "~~" :=
  fequals; auto_tilde.

Tactic Notation "pi_rewrite" "~~" constr(E) :=
  pi_rewrite E; auto_tilde.

Tactic Notation "pi_rewrite" "~~" constr(E) "in" hyp(H) :=
  pi_rewrite E in H; auto_tilde.

Tactic Notation "invert" "~~" hyp(H) :=
  invert H; auto_tilde.

Tactic Notation "inverts" "~~" hyp(H) :=
  inverts H; auto_tilde.

Tactic Notation "inverts" "~~" hyp(E) "as" :=
  inverts E as; auto_tilde.

Tactic Notation "injects" "~~" hyp(H) :=
  injects H; auto_tilde.

Tactic Notation "inversions" "~~" hyp(H) :=

```

inversions H; auto_tilde.

Tactic Notation "cases" " \sim " constr(E) "as" ident(H) :=
cases E as H; auto_tilde.

Tactic Notation "cases" " \sim " constr(E) :=
cases E; auto_tilde.

Tactic Notation "case_if" " \sim " :=
case_if; auto_tilde.

Tactic Notation "case_ifs" " \sim " :=
case_ifs; auto_tilde.

Tactic Notation "case_if" " \sim " "in" hyp(H) :=
case_if in H; auto_tilde.

Tactic Notation "cases_if" " \sim " :=
cases_if; auto_tilde.

Tactic Notation "cases_if" " \sim " "in" hyp(H) :=
cases_if in H; auto_tilde.

Tactic Notation "destruct_if" " \sim " :=
destruct_if; auto_tilde.

Tactic Notation "destruct_if" " \sim " "in" hyp(H) :=
destruct_if in H; auto_tilde.

Tactic Notation "destruct_head_match" " \sim " :=
destruct_head_match; auto_tilde.

Tactic Notation "cases'" " \sim " constr(E) "as" ident(H) :=
cases' E as H; auto_tilde.

Tactic Notation "cases'" " \sim " constr(E) :=
cases' E; auto_tilde.

Tactic Notation "cases_if'" " \sim " "as" ident(H) :=
cases_if' as H; auto_tilde.

Tactic Notation "cases_if'" " \sim " :=
cases_if'; auto_tilde.

Tactic Notation "decides_equality" " \sim " :=
decides_equality; auto_tilde.

Tactic Notation "iff" " \sim " :=
iff; auto_tilde.

Tactic Notation "iff" " \sim " simple_intropattern(I) :=
iff I; auto_tilde.

Tactic Notation "splits" " \sim " :=
splits; auto_tilde.

Tactic Notation "splits" " \sim " constr(N) :=
splits N; auto_tilde.

Tactic Notation "destructs" " \sim " constr(T) :=
destructs T; auto_tilde.

```

Tactic Notation "destructs" "~~" constr(N) constr(T) :=
  destructs N T; auto_tilde.

Tactic Notation "branch" "~~" constr(N) :=
  branch N; auto_tilde.

Tactic Notation "branch" "~~" constr(K) "of" constr(N) :=
  branch K of N; auto_tilde.

Tactic Notation "branches" "~~" :=
  branches; auto_tilde.

Tactic Notation "branches" "~~" constr(T) :=
  branches T; auto_tilde.

Tactic Notation "branches" "~~" constr(N) constr(T) :=
  branches N T; auto_tilde.

Tactic Notation "exists" "~~" :=
  ∃; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) :=
  ∃ T1; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) constr(T2) :=
  ∃ T1 T2; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) constr(T2) constr(T3) :=
  ∃ T1 T2 T3; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) constr(T2) constr(T3) constr(T4) :=
  ∃ T1 T2 T3 T4; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) constr(T2) constr(T3) constr(T4)
  constr(T5) :=
  ∃ T1 T2 T3 T4 T5; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) constr(T2) constr(T3) constr(T4)
  constr(T5) constr(T6) :=
  ∃ T1 T2 T3 T4 T5 T6; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) "," constr(T2) :=
  ∃ T1 T2; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) "," constr(T2) "," constr(T3) :=
  ∃ T1 T2 T3; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) "," constr(T2) "," constr(T3) ","
  constr(T4) :=
  ∃ T1 T2 T3 T4; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) "," constr(T2) "," constr(T3) ","
  constr(T4) "," constr(T5) :=
  ∃ T1 T2 T3 T4 T5; auto_tilde.

Tactic Notation "exists" "~~" constr(T1) "," constr(T2) "," constr(T3) ","
  constr(T4) "," constr(T5) "," constr(T6) :=
  ∃ T1 T2 T3 T4 T5 T6; auto_tilde.

```

2.14.6 Parsing for strong automation

Any tactic followed by the symbol \times will have `auto \times` called on all of its subgoals. The exceptions to these rules are the same as for light automation.

Exception: use `subs \times` instead of `subst \times` if you import the library *Coq.Classes.Equivalence*.

```
Tactic Notation "equates" "*" constr(E) :=
  equates E; auto_star.

Tactic Notation "equates" "*" constr(n1) constr(n2) :=
  equates n1 n2; auto_star.

Tactic Notation "equates" "*" constr(n1) constr(n2) constr(n3) :=
  equates n1 n2 n3; auto_star.

Tactic Notation "equates" "*" constr(n1) constr(n2) constr(n3) constr(n4) :=
  equates n1 n2 n3 n4; auto_star.

Tactic Notation "applys_eq" "*" constr(H) :=
  applys_eq H; auto_star.

Tactic Notation "applys_eq" "*" constr(H) constr(E) :=
  applys_eq H E; auto_star.

Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2) :=
  applys_eq H n1 n2; auto_star.

Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2) constr(n3) :=
  applys_eq H n1 n2 n3; auto_star.

Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2) constr(n3) constr(n4) :=
  applys_eq H n1 n2 n3 n4; auto_star.

Tactic Notation "apply" "*" constr(H) :=
  sapply H; auto_star.

Tactic Notation "destruct" "*" constr(H) :=
  destruct H; auto_star.

Tactic Notation "destruct" "*" constr(H) "as" simple_intropattern(I) :=
  destruct H as I; auto_star.

Tactic Notation "f_equal" "*" :=
  f_equal; auto_star.

Tactic Notation "induction" "*" constr(H) :=
  induction H; auto_star.

Tactic Notation "inversion" "*" constr(H) :=
  inversion H; auto_star.

Tactic Notation "split" "*" :=
  split; auto_star.

Tactic Notation "subs" "*" :=
  subst; auto_star.

Tactic Notation "subst" "*" :=
```

```

subst; auto_star.

Tactic Notation "right" "*" :=
  right; auto_star.

Tactic Notation "left" "*" :=
  left; auto_star.

Tactic Notation "constructor" "*" :=
  constructor; auto_star.

Tactic Notation "constructors" "*" :=
  constructors; auto_star.

Tactic Notation "false" "*" :=
  false; auto_star.

Tactic Notation "false" "*" constr(E) :=
  false_then E ltac:(fun _ => auto_star).

Tactic Notation "false" "*" constr(E0) constr(E1) :=
  false× (» E0 E1).

Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) :=
  false× (» E0 E1 E2).

Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) constr(E3) :=
  false× (» E0 E1 E2 E3).

Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) constr(E3) constr(E4) :=
  false× (» E0 E1 E2 E3 E4).

Tactic Notation "tryfalse" "*" :=
  try solve [ false× ].

Tactic Notation "asserts" "*" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_star | idtac ].

Tactic Notation "asserts" "*" ":" constr(E) :=
  let H := fresh "H" in asserts× H: E.

Tactic Notation "cuts" "*" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_star | idtac ].

Tactic Notation "cuts" "*" ":" constr(E) :=
  cuts: E; [ auto_star | idtac ].

Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E) :=
  lets I: E; auto_star.

Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  lets I: E0 A1; auto_star.

Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
  lets I: E0 A1 A2; auto_star.

Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=

```

```

lets I: E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets I: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "lets" "*" ":" constr(E) :=
  lets: E; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
  constr(A1) :=
  lets: E0 A1; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
  constr(A1) constr(A2) :=
  lets: E0 A1 A2; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets: E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E) :=
  forwards I: E; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  forwards I: E0 A1; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
  forwards I: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  forwards I: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards I: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "forwards" "*" ":" constr(E) :=

```

```

forwards: E; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) :=
  forwards: E0 A1; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) :=
  forwards: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) constr(A3) :=
  forwards: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "applys" "*" constr(H) :=
  sapply H; auto_star. Tactic Notation "applys" "*" constr(E0) constr(A1) :=
  applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) :=
  applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) :=
  applys E0 A1 A2; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applys E0 A1 A2 A3; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4) :=
  applys E0 A1 A2 A3 A4; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  applys E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "specializes" "*" hyp(H) :=
  specializes H; auto_star.
Tactic Notation "specializes" "~" hyp(H) constr(A1) :=
  specializes H A1; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H A1 A2 A3; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4) :=
  specializes H A1 A2 A3 A4; auto_star.

```

```

Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_star.

Tactic Notation "fapply" "*" constr(E) :=
  fapply E; auto_star.

Tactic Notation "sapply" "*" constr(E) :=
  sapply E; auto_star.

Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ => auto_star).

Tactic Notation "intros_all" "*" :=
  intros_all; auto_star.

Tactic Notation "unfolds" "*" :=
  unfolds; auto_star.

Tactic Notation "unfolds" "*" constr(F1) :=
  unfolds F1; auto_star.

Tactic Notation "unfolds" "*" constr(F1) ,," constr(F2) :=
  unfolds F1, F2; auto_star.

Tactic Notation "unfolds" "*" constr(F1) ,," constr(F2) ,," constr(F3) :=
  unfolds F1, F2, F3; auto_star.

Tactic Notation "unfolds" "*" constr(F1) ,," constr(F2) ,," constr(F3) ,,"
constr(F4) :=
  unfolds F1, F2, F3, F4; auto_star.

Tactic Notation "simple" "*" :=
  simpl; auto_star.

Tactic Notation "simple" "*" "in" hyp(H) :=
  simpl in H; auto_star.

Tactic Notation "simpls" "*" :=
  simpls; auto_star.

Tactic Notation "hnfs" "*" :=
  hnfs; auto_star.

Tactic Notation "hnfs" "*" "in" hyp(H) :=
  hnf in H; auto_star.

Tactic Notation "substs" "*" :=
  substs; auto_star.

Tactic Notation "intro_hyp" "*" hyp(H) :=
  subst_hyp H; auto_star.

Tactic Notation "intro_subst" "*" :=
  intro_subst; auto_star.

Tactic Notation "subst_eq" "*" constr(E) :=
  subst_eq E; auto_star.

Tactic Notation "rewrite" "*" constr(E) :=

```

```

rewrite E; auto_star.

Tactic Notation "rewrite" "*" "<-" constr(E) :=
  rewrite ← E; auto_star.

Tactic Notation "rewrite" "*" constr(E) "in" hyp(H) :=
  rewrite E in H; auto_star.

Tactic Notation "rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  rewrite ← E in H; auto_star.

Tactic Notation "rewrites" "*" constr(E) :=
  rewrites E; auto_star.

Tactic Notation "rewrites" "*" constr(E) "in" hyp(H) :=
  rewrites E in H; auto_star.

Tactic Notation "rewrites" "*" constr(E) "in" "*":=
  rewrites E in *; auto_star.

Tactic Notation "rewrites" "*" "<-" constr(E) :=
  rewrites ← E; auto_star.

Tactic Notation "rewrites" "*" "<-" constr(E) "in" hyp(H) :=
  rewrites ← E in H; auto_star.

Tactic Notation "rewrites" "*" "<-" constr(E) "in" "*":=
  rewrites ← E in *; auto_star.

Tactic Notation "rewrite_all" "*" constr(E) :=
  rewrite_all E; auto_star.

Tactic Notation "rewrite_all" "*" "<-" constr(E) :=
  rewrite_all ← E; auto_star.

Tactic Notation "rewrite_all" "*" constr(E) "in" ident(H) :=
  rewrite_all E in H; auto_star.

Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" ident(H) :=
  rewrite_all ← E in H; auto_star.

Tactic Notation "rewrite_all" "*" constr(E) "in" "*":=
  rewrite_all E in *; auto_star.

Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" "*":=
  rewrite_all ← E in *; auto_star.

Tactic Notation "asserts_rewrite" "*" constr(E) :=
  asserts_rewrite E; auto_star.

Tactic Notation "asserts_rewrite" "*" "<-" constr(E) :=
  asserts_rewrite ← E; auto_star.

Tactic Notation "asserts_rewrite" "*" constr(E) "in" hyp(H) :=
  asserts_rewrite E; auto_star.

Tactic Notation "asserts_rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  asserts_rewrite ← E; auto_star.

Tactic Notation "asserts_rewrite" "*" constr(E) "in" "*":=
  asserts_rewrite E in *; auto_tilde.

Tactic Notation "asserts_rewrite" "*" "<-" constr(E) "in" "*":=

```

```

asserts_rewrite ← E in *; auto_tilde.

Tactic Notation "cuts_rewrite" "*" constr(E) :=
  cuts_rewrite E; auto_star.

Tactic Notation "cuts_rewrite" "*" "<-" constr(E) :=
  cuts_rewrite ← E; auto_star.

Tactic Notation "cuts_rewrite" "*" "in" hyp(H) :=
  cuts_rewrite E in H; auto_star.

Tactic Notation "cuts_rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  cuts_rewrite ← E in H; auto_star.

Tactic Notation "erewrite" "*" constr(E) :=
  erewrite E; auto_star.

Tactic Notation "fequal" "*" :=
  fequal; auto_star.

Tactic Notation "fequals" "*" :=
  fequals; auto_star.

Tactic Notation "pi_rewrite" "*" constr(E) :=
  pi_rewrite E; auto_star.

Tactic Notation "pi_rewrite" "*" constr(E) "in" hyp(H) :=
  pi_rewrite E in H; auto_star.

Tactic Notation "invert" "*" hyp(H) :=
  invert H; auto_star.

Tactic Notation "inverts" "*" hyp(H) :=
  inverts H; auto_star.

Tactic Notation "inverts" "*" hyp(E) "as" :=
  inverts E as; auto_star.

Tactic Notation "injects" "*" hyp(H) :=
  injects H; auto_star.

Tactic Notation "inversions" "*" hyp(H) :=
  inversions H; auto_star.

Tactic Notation "cases" "*" constr(E) "as" ident(H) :=
  cases E as H; auto_star.

Tactic Notation "cases" "*" constr(E) :=
  cases E; auto_star.

Tactic Notation "case_if" "*" :=
  case_if; auto_star.

Tactic Notation "case_ifs" "*" :=
  case_ifs; auto_star.

Tactic Notation "case_if" "*" "in" hyp(H) :=
  case_if in H; auto_star.

Tactic Notation "cases_if" "*" :=
  cases_if; auto_star.

```

```

Tactic Notation "cases_if" "*" "in" hyp(H) :=
  cases_if in H; auto_star.
Tactic Notation "destruct_if" "*" :=
  destruct_if; auto_star.
Tactic Notation "destruct_if" "*" "in" hyp(H) :=
  destruct_if in H; auto_star.
Tactic Notation "destruct_head_match" "*" :=
  destruct_head_match; auto_star.
Tactic Notation "cases'" "*" constr(E) "as" ident(H) :=
  cases' E as H; auto_star.
Tactic Notation "cases'" "*" constr(E) :=
  cases' E; auto_star.
Tactic Notation "cases_if'" "*" "as" ident(H) :=
  cases_if' as H; auto_star.
Tactic Notation "cases_if'" "*" :=
  cases_if'; auto_star.
Tactic Notation "decides_equality" "*" :=
  decides_equality; auto_star.
Tactic Notation "iff" "*" :=
  iff; auto_star.
Tactic Notation "iff" "*" simple_intropattern(I) :=
  iff I; auto_star.
Tactic Notation "splits" "*" :=
  splits; auto_star.
Tactic Notation "splits" "*" constr(N) :=
  splits N; auto_star.
Tactic Notation "destructs" "*" constr(T) :=
  destructs T; auto_star.
Tactic Notation "destructs" "*" constr(N) constr(T) :=
  destructs N T; auto_star.
Tactic Notation "branch" "*" constr(N) :=
  branch N; auto_star.
Tactic Notation "branch" "*" constr(K) "of" constr(N) :=
  branch K of N; auto_star.
Tactic Notation "branches" "*" constr(T) :=
  branches T; auto_star.
Tactic Notation "branches" "*" constr(N) constr(T) :=
  branches N T; auto_star.
Tactic Notation "exists" "*" :=
   $\exists$ ; auto_star.
Tactic Notation "exists" "*" constr(T1) :=

```

```

 $\exists T_1; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) constr( $T_2$ ) :=
   $\exists T_1 T_2; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) constr( $T_2$ ) constr( $T_3$ ) :=
   $\exists T_1 T_2 T_3; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) constr( $T_2$ ) constr( $T_3$ ) constr( $T_4$ ) :=
   $\exists T_1 T_2 T_3 T_4; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) constr( $T_2$ ) constr( $T_3$ ) constr( $T_4$ )
  constr( $T_5$ ) :=
   $\exists T_1 T_2 T_3 T_4 T_5; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) constr( $T_2$ ) constr( $T_3$ ) constr( $T_4$ )
  constr( $T_5$ ) constr( $T_6$ ) :=
   $\exists T_1 T_2 T_3 T_4 T_5 T_6; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) "," constr( $T_2$ ) :=
   $\exists T_1 T_2; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) "," constr( $T_2$ ) "," constr( $T_3$ ) :=
   $\exists T_1 T_2 T_3; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) "," constr( $T_2$ ) "," constr( $T_3$ ) ","
  constr( $T_4$ ) :=
   $\exists T_1 T_2 T_3 T_4; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) "," constr( $T_2$ ) "," constr( $T_3$ ) ","
  constr( $T_4$ ) :=
   $\exists T_1 T_2 T_3 T_4 T_5; \text{auto\_star}.$ 
Tactic Notation "exists" "*" constr( $T_1$ ) "," constr( $T_2$ ) "," constr( $T_3$ ) ","
  constr( $T_4$ ) "," constr( $T_5$ ) :=
   $\exists T_1 T_2 T_3 T_4 T_5 T_6; \text{auto\_star}.$ 

```

2.15 Tactics to sort out the proof context

2.15.1 Hiding hypotheses

Definition `ltac_something` ($P:\text{Type}$) ($e:P$) := e .

Notation "'Something'" :=
 $(@\text{ltac_something} _ _).$

Lemma `ltac_something_eq` : $\forall (e:\text{Type}),$
 $e = (@\text{ltac_something} _ e).$

Proof using. auto. Qed.

Lemma `ltac_something_hide` : $\forall (e:\text{Type}),$
 $e \rightarrow (@\text{ltac_something} _ e).$

Proof using. auto. Qed.

```
Lemma ltac_something_show : ∀ (e:Type),
  (@ltac_something _ e) → e.
```

Proof using. auto. Qed.

hide_def × and *show_def* × can be used to hide/show the body of the definition ×.

```
Tactic Notation "hide_def" hyp(x) :=
  let x' := constr:(x) in
  let T := eval unfold x in x' in
  change T with (@ltac_something _ T) in x.
```

```
Tactic Notation "show_def" hyp(x) :=
  let x' := constr:(x) in
  let U := eval unfold x in x' in
  match U with @ltac_something _ ?T ⇒
    change U with T in x end.
```

show_def unfolds *Something* in the goal

```
Tactic Notation "show_def" :=
  unfold ltac_something.
```

```
Tactic Notation "show_def" "in" hyp(H) :=
  unfold ltac_something in H.
```

```
Tactic Notation "show_def" "in" "*" :=
  unfold ltac_something in *.
```

hide_defs and *show_defs* applies to all definitions

```
Tactic Notation "hide_defs" :=
  repeat match goal with H := ?T ⊢ _ ⇒
    match T with
    | @ltac_something _ _ ⇒ fail 1
    | _ ⇒ change T with (@ltac_something _ T) in H
    end
  end.
```

```
Tactic Notation "show_defs" :=
  repeat match goal with H := (@ltac_something _ ?T) ⊢ _ ⇒
    change (@ltac_something _ T) with T in H end.
```

hide_hyp H replaces the type of *H* with the notation *Something* and *show_hyp H* reveals the type of the hypothesis. Note that the hidden type of *H* remains convertible the real type of *H*.

```
Tactic Notation "show_hyp" hyp(H) :=
  apply ltac_something_show in H.
```

```
Tactic Notation "hide_hyp" hyp(H) :=
  apply ltac_something_hide in H.
```

hide_hyps and *show_hyps* can be used to hide/show all hypotheses of type Prop.

```

Tactic Notation "show_hyps" :=
  repeat match goal with
    H: @ltac_something _ _ ⊢ _ ⇒ show_hyp H end.

```

```

Tactic Notation "hide_hyps" :=
  repeat match goal with H: ?T ⊢ _ ⇒
    match type of T with
    | Prop ⇒
      match T with
      | @ltac_something _ _ ⇒ fail 2
      | _ ⇒ hide_hyp H
      end
    | _ ⇒ fail 1
    end
  end.

```

hide H and *show H* automatically select between *hide_hyp* or *hide_def*, and *show_hyp* or *show_def*. Similarly *hide_all* and *show_all* apply to all.

```

Tactic Notation "hide" hyp(H) :=
  first [hide_def H | hide_hyp H].

```

```

Tactic Notation "show" hyp(H) :=
  first [show_def H | show_hyp H].

```

```

Tactic Notation "hide_all" :=
  hide_hyps; hide_defs.

```

```

Tactic Notation "show_all" :=
  unfold ltac_something in *.

```

hide_term E can be used to hide a term from the goal. *show_term* or *show_term E* can be used to reveal it. *hide_term E in H* can be used to specify an hypothesis.

```

Tactic Notation "hide_term" constr(E) :=
  change E with (@ltac_something _ E).

```

```

Tactic Notation "show_term" constr(E) :=
  change (@ltac_something _ E) with E.

```

```

Tactic Notation "show_term" :=
  unfold ltac_something.

```

```

Tactic Notation "hide_term" constr(E) "in" hyp(H) :=
  change E with (@ltac_something _ E) in H.

```

```

Tactic Notation "show_term" constr(E) "in" hyp(H) :=
  change (@ltac_something _ E) with E in H.

```

```

Tactic Notation "show_term" "in" hyp(H) :=
  unfold ltac_something in H.

```

show_unfold R unfolds the definition of *R* and reveals the hidden definition of *R*. – todo:test, and implement using *unfold simply*

```

Tactic Notation "show_unfold" constr(R1) :=
  unfold R1; show_def.
Tactic Notation "show_unfold" constr(R1) "," constr(R2) :=
  unfold R1, R2; show_def.

```

2.15.2 Sorting hypotheses

sort sorts out hypotheses from the context by moving all the propositions (hypotheses of type Prop) to the bottom of the context.

```

Ltac sort_tactic :=
  try match goal with H: ?T ⊢ _ ⇒
    match type of T with Prop ⇒
      generalizes H; (try sort_tactic); intro
    end end.

```

```

Tactic Notation "sort" :=
  sort_tactic.

```

2.15.3 Clearing hypotheses

clears X1 ... XN is a variation on *clear* which clears the variables *X1..XN* as well as all the hypotheses which depend on them. Contrary to *clear*, it never fails.

```

Tactic Notation "clears" ident(X1) :=
  let rec doit _ :=
    match goal with
    | H:context[X1] ⊢ _ ⇒ clear H; try (doit tt)
    | _ ⇒ clear X1
  end in doit tt.

Tactic Notation "clears" ident(X1) ident(X2) :=
  clears X1; clears X2.

Tactic Notation "clears" ident(X1) ident(X2) ident(X3) :=
  clears X1; clears X2; clears X3.

Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4) :=
  clears X1; clears X2; clears X3; clears X4.

Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
  ident(X5) :=
  clears X1; clears X2; clears X3; clears X4; clears X5.

Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
  ident(X5) ident(X6) :=
  clears X1; clears X2; clears X3; clears X4; clears X5; clears X6.

```

clears (without any argument) clears all the unused variables from the context. In other words, it removes any variable which is not a proposition (i.e. not of type Prop) and which does not appear in another hypothesis nor in the goal.

```

Ltac clears_tactic :=
  match goal with H: ?T ⊢ _ ⇒
    match type of T with
    | Prop ⇒ generalizes H; (try clears_tactic); intro
    | ?TT ⇒ clear H; (try clears_tactic)
    | ?TT ⇒ generalizes H; (try clears_tactic); intro
  end end.

Tactic Notation "clears" :=
  clears_tactic.

clears_all clears all the hypotheses from the context that can be cleared. It leaves only
the hypotheses that are mentioned in the goal.

Ltac clears_or_generalizes_all_core :=
  repeat match goal with H: _ ⊢ _ ⇒
    first [ clear H | generalizes H] end.

Tactic Notation "clears_all" :=
  generalize ltac_mark;
  clears_or_generalizes_all_core;
  intro_until_mark.

clears_but  $H_1 H_2 \dots H_N$  clears all hypotheses except the one that are mentioned and
those that cannot be cleared.

Ltac clears_but_core cont :=
  generalize ltac_mark;
  cont tt;
  clears_or_generalizes_all_core;
  intro_until_mark.

Tactic Notation "clears_but" :=
  clears_but_core ltac:(fun _ ⇒ idtac).

Tactic Notation "clears_but" ident(H1) :=
  clears_but_core ltac:(fun _ ⇒ gen H1).

Tactic Notation "clears_but" ident(H1) ident(H2) :=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2).

Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) :=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2 H3).

Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) ident(H4) :=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2 H3 H4).

Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) ident(H4) ident(H5) :=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2 H3 H4 H5).

Lemma demo_clears_all_and_clears_but :
   $\forall x y:\text{nat}, y < 2 \rightarrow x = x \rightarrow x \geq 2 \rightarrow x < 3 \rightarrow \text{True}.$ 

Proof using.

```

```

introv M1 M2 M3. dup 6.
clears_all. auto.
clears_but M3. auto.
clears_but y. auto.
clears_but x. auto.
clears_but M2 M3. auto.
clears_but x y. auto.

```

Qed.

clears_last clears the last hypothesis in the context. *clears_last N* clears the last *N* hypotheses in the context.

```

Tactic Notation "clears_last" :=
  match goal with H: ?T ⊢ _ ⇒ clear H end.

```

```

Ltac clears_last_base N :=
  match number_to_nat N with
  | 0 ⇒ idtac
  | S ?p ⇒ clears_last; clears_last_base p
  end.

```

```

Tactic Notation "clears_last" constr(N) :=
  clears_last_base N.

```

2.16 Tactics for development purposes

2.16.1 Skipping subgoals

The *skip* tactic can be used at any time to admit the current goal. Unlike *admit*, it does not require ending the proof with *Admitted* instead of *Qed*. It thus saves the pain of renaming *Qed* into *Admitted* and vice-versa all the time.

The implementation of *skip* relies on an axiom **False**. To obtain a safe development, it suffices to replace **False** with **True** in the statement of that axiom.

Note that it is still necessary to instantiate all the existential variables introduced by other tactics in order for *Qed* to be accepted.

To obtain a safe development, change to *skip_axiom : True Axiom skip_axiom : False*.

```

Ltac skip_with_axiom :=
  elimtype False; apply skip_axiom.

```

```

Tactic Notation "skip" :=
  skip_with_axiom.

```

To use traditional *admit* instead of *skip* in the tactics defined below, uncomment the following definition, to bind *skip* to *admit*.

demo is like *admit* but it documents the fact that *admit* is intended

```
Tactic Notation "demo" :=
```

```
  skip.
```

admits H: T adds an assumption named *H* of type *T* to the current context, blindly assuming that it is true. *admit: T* is another possible syntax. Note that *H* may be an intro pattern.

```
Tactic Notation "admits" simple_intropattern(I) ":" constr(T) :=  
  asserts I: T; [ skip | ].
```

```
Tactic Notation "admits" ":" constr(T) :=  
  let H := fresh "TEMP" in admits H: T.
```

```
Tactic Notation "admits" "~" ":" constr(T) :=  
  admits: T; auto_tilde.
```

```
Tactic Notation "admits" "*" ":" constr(T) :=  
  admits: T; auto_star.
```

admit_cuts T simply replaces the current goal with *T*.

```
Tactic Notation "admit_cuts" constr(T) :=  
  cuts: T; [ skip | ].
```

admit_goal H applies to any goal. It simply assumes the current goal to be true. The assumption is named “H”. It is useful to set up proof by induction or coinduction. Syntax *admit_goal* is also accepted.

```
Tactic Notation "admit_goal" ident(H) :=  
  match goal with ⊢ ?G ⇒ admits H: G end.
```

```
Tactic Notation "admit_goal" :=  
  let IH := fresh "IH" in admit_goal IH.
```

admit_rewrite T can be applied when *T* is an equality. It blindly assumes this equality to be true, and rewrite it in the goal.

```
Tactic Notation "admit_rewrite" constr(T) :=  
  let M := fresh "TEMP" in admits M: T; rewrite M; clear M.
```

admit_rewrite T in H is similar as *admit_rewrite*, except that it rewrites in hypothesis *H*.

```
Tactic Notation "admit_rewrite" constr(T) "in" hyp(H) :=  
  let M := fresh "TEMP" in admits M: T; rewrite M in H; clear M.
```

admit_rewrites_all T is similar as *admit_rewrite*, except that it rewrites everywhere (goal and all hypotheses).

```
Tactic Notation "admit_rewrite_all" constr(T) :=  
  let M := fresh "TEMP" in admits M: T; rewrite_all M; clear M.
```

forwards_nounfold_admit_sides_then E ltac:(fun K ⇒ ..) is like *forwards: E* but it provides the resulting term to a continuation, under the name *K*, and it admits any side-condition produced by the instantiation of *E*, using the *skip* tactic.

```
Inductive ltac_goal_to_discard := ltac_goal_to_discard_intro.
```

```
Ltac forwards_nounfold_admit_sides_then S cont :=
  let MARK := fresh "TEMP" in
  generalize ltac_goal_to_discard_intro;
  intro MARK;
  forwards_nounfold_then S ltac:(fun K =>
    clear MARK;
    cont K);
  match goal with
  | MARK : ltac_goal_to_discard ⊢ _ => skip
  | _ => idtac
  end.
```

DEPRECATED – FOR BACKWARD COMPATIBILITY

```
Tactic Notation "skip" simple_intropattern(I) ":" constr(T) :=
  admits I: T.
```

```
Tactic Notation "skip" ":" constr(T) :=
  admits: T.
```

```
Tactic Notation "skip" "~~" ":" constr(T) :=
  admits~:T.
```

```
Tactic Notation "skip" "*" ":" constr(T) :=
  admits*:T.
```

```
Tactic Notation "skip" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(T) :=
  skip [I1 I2]: T.
```

```
Tactic Notation "skip" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  skip [I1 [I2 I3]]: T.
```

```
Tactic Notation "skip" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) ":" constr(T) :=
  skip [I1 [I2 [I3 I4]]]: T.
```

```
Tactic Notation "skip" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
  skip [I1 [I2 [I3 [I4 I5]]]]: T.
```

```
Tactic Notation "skip" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) ":" constr(T) :=
  skip [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.
```

```
Tactic Notation "skip_asserts" simple_intropattern(I) ":" constr(T) :=
```

```

admits I: T.
Tactic Notation "skip_asserts" ":" constr(T) :=
  admits: T.
Tactic Notation "skip_cuts" constr(T) :=
  admit_cuts T.
Tactic Notation "skip_goal" ident(H) :=
  admit_goal H.
Tactic Notation "skip_goal" :=
  admit_goal.
Tactic Notation "skip_rewrite" constr(T) :=
  admit_rewrite T.
Tactic Notation "skip_rewrite" constr(T) "in" hyp(H) :=
  admit_rewrite T in H.
Tactic Notation "skip_rewrite_all" constr(T) :=
  admit_rewrite_all T.
Ltac forwards_nounfold_skip_sides_then S cont :=
  forwards_nounfold_admit_sides_then S cont.
Tactic Notation "skip_induction" constr(E) :=
  let IH := fresh "IH" in admit_goal IH; destruct E.
Tactic Notation "skip_induction" constr(E) "as" simple_intropattern(I) :=
  let IH := fresh "IH" in admit_goal IH; destruct E as I.

```

2.17 Compatibility with standard library

The module *Program* contains definitions that conflict with the current module. If you import *Program*, either directly or indirectly (e.g. through *Setoid* or *ZArith*), you will need to import the compatibility definitions through the top-level command: `Import LIBTACTICSCOMPATIBILITY.`

```

Module LIBTACTICSCOMPATIBILITY.
  Tactic Notation "apply" "*" constr(H) :=
    sapply H; auto_star.
  Tactic Notation "subst" "*" :=
    subst; auto_star.
End LIBTACTICSCOMPATIBILITY.

Open Scope nat_scope.

```

2.18 Additional notations for Coq

2.18.1 N-ary Existentials –TODO: DEPRECATED, Coq now supports it.

$\exists T_1 \dots T_N, P$ is a shorthand for $\exists T_1, \dots, \exists T_N, P$. Note that *Coq.Program.Syntax* already defines exists for arity up to 4.

Notation "'exists' x1 ',' P" :=

($\exists x_1, P$)
(at level 200, x_1 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 ',' P" :=

($\exists x_1, \exists x_2, P$)
(at level 200, x_1 ident, x_2 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 ',' P" :=

($\exists x_1, \exists x_2, \exists x_3, P$)
(at level 200, x_1 ident, x_2 ident, x_3 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 x4 ',' P" :=

($\exists x_1, \exists x_2, \exists x_3, \exists x_4, P$)
(at level 200, x_1 ident, x_2 ident, x_3 ident, x_4 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 x4 x5 ',' P" :=

($\exists x_1, \exists x_2, \exists x_3, \exists x_4, \exists x_5, P$)
(at level 200, x_1 ident, x_2 ident, x_3 ident, x_4 ident, x_5 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 x4 x5 x6 ',' P" :=

($\exists x_1, \exists x_2, \exists x_3, \exists x_4, \exists x_5, \exists x_6, P$)
(at level 200, x_1 ident, x_2 ident, x_3 ident, x_4 ident, x_5 ident,
 x_6 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 x4 x5 x6 x7 ',' P" :=

($\exists x_1, \exists x_2, \exists x_3, \exists x_4, \exists x_5, \exists x_6,$
 $\exists x_7, P$)
(at level 200, x_1 ident, x_2 ident, x_3 ident, x_4 ident, x_5 ident,
 x_6 ident, x_7 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 x4 x5 x6 x7 x8 ',' P" :=

($\exists x_1, \exists x_2, \exists x_3, \exists x_4, \exists x_5, \exists x_6,$
 $\exists x_7, \exists x_8, P$)
(at level 200, x_1 ident, x_2 ident, x_3 ident, x_4 ident, x_5 ident,

```

x6 ident, x7 ident, x8 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 x4 x5 x6 x7 x8 x9 , P" :=
(∃ x1, ∃ x2, ∃ x3, ∃ x4, ∃ x5, ∃ x6,
 ∃ x7, ∃ x8, ∃ x9, P)
(at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
 x6 ident, x7 ident, x8 ident, x9 ident,
right associativity) : type_scope.

Notation "'exists' x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 , P" :=
(∃ x1, ∃ x2, ∃ x3, ∃ x4, ∃ x5, ∃ x6,
 ∃ x7, ∃ x8, ∃ x9, ∃ x10, P)
(at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
 x6 ident, x7 ident, x8 ident, x9 ident, x10 ident,
right associativity) : type_scope.

```

2.18.2 'let bindings (EXPERIMENTAL).

The syntax `'let x := v in e` has the same meaning as `let x := v in e` except that the binding is implemented using a beta-redex that is not reduced automatically by `simpl`. The `'let` construct therefore makes it possible to simplify or push to the context let-bindings one by one.

Definition of `'let`

Declare Scope let_scope.

Definition `let_binding (A B:Type) (v:A) (K:A → B) := K v.`

Notation `"'let' x ':=' v 'in' e" := (let_binding v (fun x ⇒ e))`
`(at level 69, x ident, right associativity,`
`format "[v '[' "let' x ':=' v 'in' ']' '/ '[' e ']']'"")`
`: let_scope.`

Notation `"'let' x ':' A ':=' v 'in' e" := (let_binding (v:A) (fun x:A ⇒ e))`
`(at level 69, x ident, right associativity,`
`format "[v '[' "let' x ':' A ':=' v 'in' ']' '/ '[' e ']']'"")`
`: let_scope.`

Global Open Scope `let_scope.`

Lemma `let_binding_unfold : ∀ (A B:Type) (v:A) (K:A → B),`
`let_binding v K = K v.`

Proof using. `reflexivity.` `Qed.`

Ltac `let_get_fresh_binding_name K :=`
`match K with (fun x ⇒ _) ⇒ let y := fresh x in y end.`

`let_simpl` finds the first occurrence of a `'let` binding and substitutes it.

Tactic Notation `"let_simpl" "in" hyp(H) :=`

```

match type of H with context [ let_binding ?v ?K ] =>
  changes (let_binding v K) with (K v) in H
end.

```

```

Tactic Notation "let_simpl" :=
  match goal with
  | ⊢ context [ let_binding ?v ?K ] =>
    changes (let_binding v K) with (K v)
  | H: context [ let_binding ?v ?K ] ⊢ _ =>
    let_simpl in H
  end.

```

```

Tactic Notation "let_simpl" constr(v) "in" hyp(H) :=
  repeat match type of H with context [ let_binding v ?K ] =>
    changes (let_binding v K) with (K v) in H
  end.

```

```

Tactic Notation "let_simpl" constr(v) :=
  repeat match goal with
  | ⊢ context [ let_binding v ?K ] =>
    changes (let_binding v K) with (K v)
  | H: context [ let_binding v ?K ] ⊢ _ =>
    let_simpl v in H
  end.

```

let_name finds the first occurrence of a 'let binding and moves this binding to the proof context.

```

Tactic Notation "let_name" "in" hyp(H) :=
  match type of H with context [ let_binding ?v ?K ] =>
    let x := let_get_fresh_binding_name K in
    set_eq x: v in H;
    let_simpl in H
  end.

```

```

Tactic Notation "let_name" "in" hyp(H) "as" ident(x) :=
  match type of H with context [ let_binding ?v ?K ] =>
    set_eq x: v in H;
    let_simpl in H
  end.

```

```

Tactic Notation "let_name" :=
  match goal with
  | ⊢ context [ let_binding ?v ?K ] =>
    let x := let_get_fresh_binding_name K in
    set_eq x: v;
    let_simpl
  | H: context [ let_binding ?v ?K ] ⊢ _ =>

```

```

let_name in H
end.

Tactic Notation "let_name" "as" ident(x) :=
  match goal with
  | ⊢ context [ let_binding ?v ?K ] ⇒
    set_eq x: v;
    let_simpl
  | H: context [ let_binding ?v ?K ] ⊢ _ ⇒
    let_name in H as x
  end.

```

let_name_all finds the first occurrence of a 'let binding, moves this binding to the proof context, and further simplify all the other 'let bindings that are binding the same value. (See LibFixDemos for a practical motivation.)

```

Tactic Notation "let_name_all" "in" hyp(H) :=
  match type of H with context [ let_binding ?v ?K ] ⇒
    let x := let_get_fresh_binding_name K in
    set_eq x: v in H;
    let_simpl x in H
  end.

```

```

Tactic Notation "let_name_all" "in" hyp(H) "as" ident(x) :=
  match type of H with context [ let_binding ?v ?K ] ⇒
    set_eq x: v in H;
    let_simpl x in H
  end.

```

```

Tactic Notation "let_name_all" :=
  match goal with
  | ⊢ context [ let_binding ?v ?K ] ⇒
    let x := let_get_fresh_binding_name K in
    set_eq x: v;
    let_simpl x
  | H: context [ let_binding ?v ?K ] ⊢ _ ⇒
    let_name_all in H
  end.

```

```

Tactic Notation "let_name_all" "as" ident(x) :=
  match goal with
  | ⊢ context [ let_binding ?v ?K ] ⇒
    set_eq x: v;
    let_simpl x
  | H: context [ let_binding ?v ?K ] ⊢ _ ⇒
    let_name_all in H as x
  end.

```

Section FuncEq.

Variables ($A_1 A_2 A_3 A_4 A_5 B$: Type).

Lemma args_eq_1 : $\forall (f:A_1 \rightarrow B) x_1 y_1,$
 $x_1 = y_1 \rightarrow$
 $f x_1 = f y_1.$

Proof using. intros. subst \neg . Qed.

Lemma args_eq_2 : $\forall (f:A_1 \rightarrow A_2 \rightarrow B) x_1 y_1 x_2 y_2,$
 $x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow$
 $f x_1 x_2 = f y_1 y_2.$

Proof using. intros. subst \neg . Qed.

Lemma args_eq_3 : $\forall (f:A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow B) x_1 y_1 x_2 y_2 x_3 y_3,$
 $x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow x_3 = y_3 \rightarrow$
 $f x_1 x_2 x_3 = f y_1 y_2 y_3.$

Proof using. intros. subst \neg . Qed.

Lemma args_eq_4 : $\forall (f:A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4 \rightarrow B) x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4,$
 $x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow x_3 = y_3 \rightarrow x_4 = y_4 \rightarrow$
 $f x_1 x_2 x_3 x_4 = f y_1 y_2 y_3 y_4.$

Proof using. intros. subst \neg . Qed.

Lemma args_eq_5 : $\forall (f:A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4 \rightarrow A_5 \rightarrow B) x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4 x_5 y_5,$
 $x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow x_3 = y_3 \rightarrow x_4 = y_4 \rightarrow x_5 = y_5 \rightarrow$
 $f x_1 x_2 x_3 x_4 x_5 = f y_1 y_2 y_3 y_4 y_5.$

Proof using. intros. subst \neg . Qed.

End FuncEq.

Ltac f_equal_fixed :=
try (
 first
 [apply args_eq_1
 | apply args_eq_2
 | apply args_eq_3
 | apply args_eq_4
 | apply args_eq_5];
 try reflexivity).

Ltac $fequal_base$:=
let $go := f_equal_fixed$; [$fequal_base$ |] in
match goal with
| $\vdash (_, _, _) = (_, _, _) \Rightarrow go$
| $\vdash (_, _, _, _) = (_, _, _, _) \Rightarrow go$
| $\vdash (_, _, _, _, _) = (_, _, _, _, _) \Rightarrow go$
| $\vdash (_, _, _, _, _, _) = (_, _, _, _, _, _) \Rightarrow go$
| $\vdash _ \Rightarrow f_equal_fixed$

```
end.
```

Generalize all propositions into the goal. Naive implementation:

```
Ltac generalize_all_prop := repeat match goal with H: ?T |- _ => match type of T with Prop => generalizes H end end.
```

The real implementation is careful to not generalized **ltac_mark**, even though it is of type `Prop`. TODO: investigate whether it would be sufficient to put **ltac_mark** in `Type` to obtain the desired behavior.

```
Ltac generalize_all_prop :=
repeat match goal with H: ?T ⊢ _ ⇒
try match T with ltac_mark ⇒ fail 2 end;
match type of T with Prop ⇒
generalizes H
end end.
```

Work around for inefficiency bug of `autorewrite in *`. Usage, e.g.: Tactic Notation "`rew_list`" "`in`" "*" := `autorewrite_in_star_patch` ltac:(fun tt ⇒ `autorewrite with rew_list`).

```
Ltac autorewrite_in_star_patch cont :=
generalize ltac_mark;
generalize_all_prop;
cont tt;
intro_until_mark.
```

Chapter 3

Library `SLF.LibEqual`

```
Set Implicit Arguments.  
From SLF Require Import LibTactics LibAxioms.  
Generalizable Variables  $A$ .
```

3.1 Definition of equality

3.1.1 Definition of Leibnitz' equality

Arguments `eq` $\{A\}$.

3.1.2 Partial application of Leibnitz' equality

$= x$ is a unary predicate which holds of values equal to x . It simply denotes the partial application of equality. $= x :> A$ allows to specify the type.

Notation "' $='$ $x :> A$ " := (fun $y \Rightarrow y = x :> A$)
(at level 71, x at next level).

Notation "' $='$ x " := (fun $y \Rightarrow y = x$)
(at level 71).

$\neq x$ is a unary predicate which holds of values disequal to x . It simply denotes the partial application of disequality. $\neq x :> A$ allows to specify the type.

Notation "' $<>$ ' $x :> A$ " := (fun $y \Rightarrow y \neq x :> A$)
(at level 71, x at next level).

Notation "' $<>$ ' x " := (fun $y \Rightarrow y \neq x$)
(at level 71).

3.1.3 Typeclass to exploit extensionality

The property EXTENSIONALITY A captures the fact that the type A features an extensional equality, in the sense that to prove the equality between two values of type A it suffices to prove that those two values are related by some binary relation.

```
Class Extensionality (A:Type) := Extensionality_make {  
  extensionality_hyp : A → A → Prop;  
  extensionality : ∀ (x y : A), extensionality_hyp x y → x = y }.
```

Arguments extensionality [A].

Arguments Extensionality_make [A] [extensionality_hyp].

Instance for propositional extensionality

```
Global Instance extensionality_prop : Extensionality Prop.
```

Proof using. intros. apply (Extensionality_make prop_ext). Defined.

3.1.4 Tactic to exploit extensionality

```
Ltac extens_reveal_eq tt :=  
  match goal with  
  | ⊢ _ = _ ⇒ idtac  
  | _ ⇒ first [ intro; extens_reveal_eq tt  
    | fail 2 "extens needs hnf to reveal an equality" ]  
  end.
```

```
Ltac extens_core tt :=  
  extens_reveal_eq tt;  
  applys extensionality;  
  simpl extensionality_hyp.
```

```
Tactic Notation "extens" :=  
  extens_core tt.
```

```
Tactic Notation "extens" "~~" :=  
  extens; auto_tilde.
```

```
Tactic Notation "extens" "***" :=  
  extens; auto_star.
```

3.2 Properties of equality

This section contains a reformulation of the lemmas provided by the standard library concerning equality.

3.2.1 Equality as an equivalence relation

See also sectin Eq from LibRelation for reformulation of theses results using high-level definitions.

Section EqualityProp.

Variables ($A : \text{Type}$).

Implicit Types $x y z : A$.

Reflexivity is captured by the constructor eq_refl .

Symmetry

Lemma $\text{eq_sym} : \forall x y,$

$x = y \rightarrow$

$y = x.$

Proof using. *introv H. destruct* $\neg H$. Qed.

Transitivity

Lemma $\text{eq_trans_ll} : \forall y x z,$

$x = y \rightarrow$

$y = z \rightarrow$

$x = z.$

Proof using. *introv H1 H2. destruct* $\neg H2$. Qed.

Definition $\text{eq_trans} := \text{eq_trans_ll}.$

Lemma $\text{eq_trans_lr} : \forall y x z,$

$x = y \rightarrow$

$z = y \rightarrow$

$x = z.$

Proof using. *introv H1 H2. destruct* $\neg H2$. Qed.

Lemma $\text{eq_trans_rl} : \forall y x z,$

$y = x \rightarrow$

$y = z \rightarrow$

$x = z.$

Proof using. *introv H1 H2. destruct* $\neg H2$. Qed.

Lemma $\text{eq_trans_rr} : \forall y x z,$

$y = x \rightarrow$

$z = y \rightarrow$

$x = z.$

Proof using. *introv H1 H2. destruct* $\neg H2$. Qed.

End EqualityProp.

Arguments $\text{eq_trans_ll} [A].$

Arguments $\text{eq_trans_lr} [A].$

Arguments $\text{eq_trans_rl} [A].$

Arguments $\text{eq_trans_rr} [A].$

Arguments eq_trans [A].

3.2.2 Properties of disequality

Section DisequalityProp.

Variables (A : Type).

Implicit Types x y : A.

Symmetry

Lemma neq_sym : $\forall x y,$

$x \neq y \rightarrow$

$y \neq x.$

Proof using. introv H K. destruct \neg K. Qed.

End DisequalityProp.

3.2.3 Symmetrized induction principles

Section EqInductionSym.

Variables (A : Type) (x : A).

Definition eq_ind_r : $\forall (P:A \rightarrow \text{Prop}),$

$P x \rightarrow \forall y, y = x \rightarrow P y.$

Proof using. intros. subst \times . Qed.

Definition eq_rec_r : $\forall (P:A \rightarrow \text{Set}),$

$P x \rightarrow \forall y, y = x \rightarrow P y.$

Proof using. intros. subst \times . Qed.

Definition eq_rect_r : $\forall (P:A \rightarrow \text{Type}),$

$P x \rightarrow \forall y, y = x \rightarrow P y.$

Proof using. intros. subst \times . Qed.

End EqInductionSym.

3.3 Functional extensionality

3.3.1 Dependent functional extensionality

Section FuncExtDep.

Variables (A1 : Type).

Variables (A2 : $\forall (x1 : A1), \text{Type}.$)

Variables (A3 : $\forall (x1 : A1) (x2 : A2 x1), \text{Type}.$)

Variables (A4 : $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), \text{Type}.$)

Variables (A5 : $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), \text{Type}.$)

Variables ($A6 : \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4)$, Type).

Lemma fun_ext_1 : $\forall (f g : \forall (x1:A1), A2 x1)$,

$$(\forall x1, f x1 = g x1) \rightarrow$$

$$f = g.$$

Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.

Lemma fun_ext_2 : $\forall (f g : \forall (x1:A1) (x2:A2 x1), A3 x2)$,

$$(\forall x1 x2, f x1 x2 = g x1 x2) \rightarrow$$

$$f = g.$$

Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.

Lemma fun_ext_3 : $\forall (f g : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2), A4 x3)$,

$$(\forall x1 x2 x3, f x1 x2 x3 = g x1 x2 x3) \rightarrow$$

$$f = g.$$

Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.

Lemma fun_ext_4 : $\forall (f g : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$

$$(x4:A4 x3), A5 x4)$$
,

$$(\forall x1 x2 x3 x4, f x1 x2 x3 x4 = g x1 x2 x3 x4) \rightarrow$$

$$f = g.$$

Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.

Lemma fun_ext_5 : $\forall (f g : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$

$$(x4:A4 x3) (x5:A5 x4), A6 x5)$$
,

$$(\forall x1 x2 x3 x4 x5, f x1 x2 x3 x4 x5 = g x1 x2 x3 x4 x5) \rightarrow$$

$$f = g.$$

Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.

Global Instance Extensionality_fun_1 :

Extensionality ($\forall (x1:A1), A2 x1$).

Proof using. intros. apply (Extensionality_make fun_ext_1). Defined.

Global Instance Extensionality_fun_2 :

Extensionality ($\forall (x1:A1) (x2:A2 x1), A3 x2$).

Proof using. intros. apply (Extensionality_make fun_ext_2). Defined.

Global Instance Extensionality_fun_3 :

Extensionality ($\forall (x1:A1) (x2:A2 x1) (x3:A3 x2), A4 x3$).

Proof using. intros. apply (Extensionality_make fun_ext_3). Defined.

Global Instance Extensionality_fun_4 :

Extensionality ($\forall (x1:A1) (x2:A2 x1) (x3:A3 x2) (x4:A4 x3), A5 x4$).

Proof using. intros. apply (Extensionality_make fun_ext_4). Defined.

Global Instance Extensionality_fun_5 :

Extensionality ($\forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$

$$(x4:A4 x3) (x5:A5 x4), A6 x5)$$
.

Proof using. intros. apply (Extensionality_make fun_ext_5). Defined.

Lemma fun_eta_dep_1 : $\forall (f : \forall (x1:A1), A2 x1)$,

```

(fun x1 ⇒ f x1) = f.
Proof using. intros. apply¬ fun_ext_1. Qed.

Lemma fun_eta_dep_2 : ∀ (f : ∀ (x1:A1) (x2:A2 x1), A3 x2),
  (fun x1 x2 ⇒ f x1 x2) = f.
Proof using. intros. apply¬ fun_ext_2. Qed.

Lemma fun_eta_dep_3 : ∀ (f : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2), A4 x3),
  (fun x1 x2 x3 ⇒ f x1 x2 x3) = f.
Proof using. intros. apply¬ fun_ext_3. Qed.

Lemma fun_eta_dep_4 : ∀ (f : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2) (x4:A4 x3), A5 x4),
  (fun x1 x2 x3 x4 ⇒ f x1 x2 x3 x4) = f.
Proof using. intros. apply¬ fun_ext_4. Qed.

Lemma fun_eta_dep_5 : ∀ (f : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2) (x4:A4 x3) (x5:A5 x4), A6 x5),
  (fun x1 x2 x3 x4 x5 ⇒ f x1 x2 x3 x4 x5) = f.
Proof using. intros. apply¬ fun_ext_5. Qed.

End FuncExtDep.

```

3.3.2 Non-dependent functional extensionality

```

Lemma fun_ext_nondep_1 : ∀ A1 B (f g : A1 → B),
  (forall x1, f x1 = g x1) →
  f = g.
Proof using. intros. apply¬ fun_ext_1. Qed.

Lemma fun_ext_nondep_2 : ∀ A1 A2 B (f g : A1 → A2 → B),
  (forall x1 x2, f x1 x2 = g x1 x2) →
  f = g.
Proof using. intros. apply¬ fun_ext_2. Qed.

Lemma fun_ext_nondep_3 : ∀ A1 A2 A3 B (f g : A1 → A2 → A3 → B),
  (forall x1 x2 x3, f x1 x2 x3 = g x1 x2 x3) →
  f = g.
Proof using. intros. apply¬ fun_ext_3. Qed.

Lemma fun_ext_nondep_4 : ∀ A1 A2 A3 A4 B (f g : A1 → A2 → A3 → A4 → B),
  (forall x1 x2 x3 x4, f x1 x2 x3 x4 = g x1 x2 x3 x4) →
  f = g.
Proof using. intros. apply¬ fun_ext_4. Qed.

Lemma fun_ext_nondep_5 : ∀ A1 A2 A3 A4 A5 B (f g : A1 → A2 → A3 → A4 → A5 → B),
  (forall x1 x2 x3 x4 x5, f x1 x2 x3 x4 x5 = g x1 x2 x3 x4 x5) →
  f = g.
Proof using. intros. apply¬ fun_ext_5. Qed.

```

3.3.3 Eta-conversion

Lemma fun_eta_1 : $\forall A1 B (f : A1 \rightarrow B),$
 $(\text{fun } x1 \Rightarrow f x1) = f.$

Proof using. intros. apply \neg fun_ext_1. Qed.

Lemma fun_eta_2 : $\forall A1 A2 B (f : A1 \rightarrow A2 \rightarrow B),$
 $(\text{fun } x1 x2 \Rightarrow f x1 x2) = f.$

Proof using. intros. apply \neg fun_ext_2. Qed.

Lemma fun_eta_3 : $\forall A1 A2 A3 B (f : A1 \rightarrow A2 \rightarrow A3 \rightarrow B),$
 $(\text{fun } x1 x2 x3 \Rightarrow f x1 x2 x3) = f.$

Proof using. intros. apply \neg fun_ext_3. Qed.

Lemma fun_eta_4 : $\forall A1 A2 A3 A4 B (f : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow B),$
 $(\text{fun } x1 x2 x3 x4 \Rightarrow f x1 x2 x3 x4) = f.$

Proof using. intros. apply \neg fun_ext_4. Qed.

Lemma fun_eta_5 : $\forall A1 A2 A3 A4 A5 B (f : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow B),$
 $(\text{fun } x1 x2 x3 x4 x5 \Rightarrow f x1 x2 x3 x4 x5) = f.$

Proof using. intros. apply \neg fun_ext_4. Qed.

Hint Rewrite fun_eta_1 fun_eta_2 fun_eta_3 fun_eta_4 fun_eta_5 : rew_eta.

3.4 Predicate extensionality

3.4.1 Dependend predicates

Section PropExt.

Variables ($A1 : \text{Type}$).

Variables ($A2 : \forall (x1 : A1), \text{Type}$).

Variables ($A3 : \forall (x1 : A1) (x2 : A2 x1), \text{Type}$).

Variables ($A4 : \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), \text{Type}$).

Variables ($A5 : \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), \text{Type}$).

Variables ($A6 : \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), \text{Type}$).

Lemma pred_ext_1 : $\forall (P Q : \forall (x1 : A1), \text{Prop}),$
 $(\forall x1, P x1 \leftrightarrow Q x1) \rightarrow$
 $P = Q.$

Proof using. repeat (intros; apply fun_ext_dep). intros. apply \neg prop_ext. Qed.

Lemma pred_ext_2 : $\forall (P Q : \forall (x1 : A1) (x2 : A2 x1), \text{Prop}),$
 $(\forall x1 x2, P x1 x2 \leftrightarrow Q x1 x2) \rightarrow$
 $P = Q.$

Proof using. repeat (intros; apply fun_ext_dep). intros. apply \neg prop_ext. Qed.

Lemma pred_ext_3 : $\forall (P Q : \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), \text{Prop}),$

$$(\forall x_1 x_2 x_3, P x_1 x_2 x_3 \leftrightarrow Q x_1 x_2 x_3) \rightarrow \\ P = Q.$$

Proof using. repeat (intros; apply *fun_ext_dep*). intros. apply \neg *prop_ext*. Qed.

$$\text{Lemma pred_ext_4 : } \forall (P Q : \forall (x_1:A_1) (x_2:A_2 x_1) (x_3:A_3 x_2) \\ (x_4:A_4 x_3), \text{Prop}), \\ (\forall x_1 x_2 x_3 x_4, P x_1 x_2 x_3 x_4 \leftrightarrow Q x_1 x_2 x_3 x_4) \rightarrow \\ P = Q.$$

Proof using. repeat (intros; apply *fun_ext_dep*). intros. apply \neg *prop_ext*. Qed.

$$\text{Lemma pred_ext_5 : } \forall (P Q : \forall (x_1:A_1) (x_2:A_2 x_1) (x_3:A_3 x_2) \\ (x_4:A_4 x_3) (x_5:A_5 x_4), \text{Prop}), \\ (\forall x_1 x_2 x_3 x_4 x_5, P x_1 x_2 x_3 x_4 x_5 \leftrightarrow Q x_1 x_2 x_3 x_4 x_5) \rightarrow \\ P = Q.$$

Proof using. repeat (intros; apply *fun_ext_dep*). intros. apply \neg *prop_ext*. Qed.

$$\text{Lemma pred_ext_6 : } \forall (P Q : \forall (x_1:A_1) (x_2:A_2 x_1) (x_3:A_3 x_2) \\ (x_4:A_4 x_3) (x_5:A_5 x_4) (x_6:A_6 x_5), \text{Prop}), \\ (\forall x_1 x_2 x_3 x_4 x_5 x_6, P x_1 x_2 x_3 x_4 x_5 x_6 \leftrightarrow Q x_1 x_2 x_3 x_4 x_5 x_6) \rightarrow \\ P = Q.$$

Proof using. repeat (intros; apply *fun_ext_dep*). intros. apply \neg *prop_ext*. Qed.

Global Instance Extensionality_pred_1 :

$$\text{Extensionality } (\forall (x_1:A_1), \text{Prop}).$$

Proof using. intros. apply (*Extensionality_make pred_ext_1*). Defined.

Global Instance Extensionality_pred_2 :

$$\text{Extensionality } (\forall (x_1:A_1) (x_2:A_2 x_1), \text{Prop}).$$

Proof using. intros. apply (*Extensionality_make pred_ext_2*). Defined.

Global Instance Extensionality_pred_3 :

$$\text{Extensionality } (\forall (x_1:A_1) (x_2:A_2 x_1) (x_3:A_3 x_2), \text{Prop}).$$

Proof using. intros. apply (*Extensionality_make pred_ext_3*). Defined.

Global Instance Extensionality_pred_4 :

$$\text{Extensionality } (\forall (x_1:A_1) (x_2:A_2 x_1) (x_3:A_3 x_2) (x_4:A_4 x_3), \text{Prop}).$$

Proof using. intros. apply (*Extensionality_make pred_ext_4*). Defined.

Global Instance Extensionality_pred_5 :

$$\text{Extensionality } (\forall (x_1:A_1) (x_2:A_2 x_1) (x_3:A_3 x_2) \\ (x_4:A_4 x_3) (x_5:A_5 x_4), \text{Prop}).$$

Proof using. intros. apply (*Extensionality_make pred_ext_5*). Defined.

Global Instance Extensionality_pred_6 :

$$\text{Extensionality } (\forall (x_1:A_1) (x_2:A_2 x_1) (x_3:A_3 x_2) \\ (x_4:A_4 x_3) (x_5:A_5 x_4) (x_6:A_6 x_5), \text{Prop}).$$

Proof using. intros. apply (*Extensionality_make pred_ext_6*). Defined.

End PropExt.

3.4.2 Non-dependend predicate extensionality

Lemma pred_ext_nondep_1 :

$$\begin{aligned} \forall A1 \ (P \ Q : A1 \rightarrow \text{Prop}), \\ (\forall x1, P \ x1 \leftrightarrow Q \ x1) \rightarrow \\ P = Q. \end{aligned}$$

Proof using. intros. apply \neg pred_ext_1. Qed.

Lemma pred_ext_nondep_2 :

$$\begin{aligned} \forall A1 \ A2 \ (P \ Q : A1 \rightarrow A2 \rightarrow \text{Prop}), \\ (\forall x1 \ x2, P \ x1 \ x2 \leftrightarrow Q \ x1 \ x2) \rightarrow \\ P = Q. \end{aligned}$$

Proof using. intros. apply \neg pred_ext_2. Qed.

Lemma pred_ext_nondep_3 :

$$\begin{aligned} \forall A1 \ A2 \ A3 \ (P \ Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow \text{Prop}), \\ (\forall x1 \ x2 \ x3, P \ x1 \ x2 \ x3 \leftrightarrow Q \ x1 \ x2 \ x3) \rightarrow \\ P = Q. \end{aligned}$$

Proof using. intros. apply \neg pred_ext_3. Qed.

Lemma pred_ext_nondep_4 :

$$\begin{aligned} \forall A1 \ A2 \ A3 \ A4 \ (P \ Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow \text{Prop}), \\ (\forall x1 \ x2 \ x3 \ x4, P \ x1 \ x2 \ x3 \ x4 \leftrightarrow Q \ x1 \ x2 \ x3 \ x4) \rightarrow \\ P = Q. \end{aligned}$$

Proof using. intros. apply \neg pred_ext_4. Qed.

Lemma pred_ext_nondep_5 :

$$\begin{aligned} \forall A1 \ A2 \ A3 \ A4 \ A5 \ (P \ Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow \text{Prop}), \\ (\forall x1 \ x2 \ x3 \ x4 \ x5, P \ x1 \ x2 \ x3 \ x4 \ x5 \leftrightarrow Q \ x1 \ x2 \ x3 \ x4 \ x5) \rightarrow \\ P = Q. \end{aligned}$$

Proof using. intros. apply \neg pred_ext_5. Qed.

Lemma pred_ext_nondep_6 :

$$\begin{aligned} \forall A1 \ A2 \ A3 \ A4 \ A5 \ A6 \ (P \ Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow A6 \rightarrow \text{Prop}), \\ (\forall x1 \ x2 \ x3 \ x4 \ x5 \ x6, P \ x1 \ x2 \ x3 \ x4 \ x5 \ x6 \leftrightarrow Q \ x1 \ x2 \ x3 \ x4 \ x5 \ x6) \rightarrow \\ P = Q. \end{aligned}$$

Proof using. intros. apply \neg pred_ext_6. Qed.

3.5 Equality of function and predicate applications

3.5.1 A same function applied to equal arguments yield equal result

Section ArgsEq.

Variables (A1 A2 A3 A4 A5 B : Type).

```

Lemma args_eq_1 : ∀ (f:A1 → B) x1 y1,
  x1 = y1 →
  f x1 = f y1.

```

Proof using. intros. subst \neg . Qed.

```

Lemma args_eq_2 : ∀ (f:A1 → A2 → B) x1 y1 x2 y2,
  x1 = y1 →
  x2 = y2 →
  f x1 x2 = f y1 y2.

```

Proof using. intros. subst \neg . Qed.

```

Lemma args_eq_3 : ∀ (f:A1 → A2 → A3 → B) x1 y1 x2 y2 x3 y3,
  x1 = y1 →
  x2 = y2 →
  x3 = y3 →
  f x1 x2 x3 = f y1 y2 y3.

```

Proof using. intros. subst \neg . Qed.

```

Lemma args_eq_4 : ∀ (f:A1 → A2 → A3 → A4 → B) x1 y1 x2 y2 x3 y3 x4 y4,
  x1 = y1 →
  x2 = y2 →
  x3 = y3 →
  x4 = y4 →
  f x1 x2 x3 x4 = f y1 y2 y3 y4.

```

Proof using. intros. subst \neg . Qed.

```

Lemma args_eq_5 : ∀ (f:A1 → A2 → A3 → A4 → A5 → B) x1 y1 x2 y2 x3 y3 x4 y4 x5 y5,
  x1 = y1 →
  x2 = y2 →
  x3 = y3 →
  x4 = y4 →
  x5 = y5 →
  f x1 x2 x3 x4 x5 = f y1 y2 y3 y4 y5.

```

Proof using. intros. subst \neg . Qed.

End ArgsEq.

3.5.2 Equal functions applied to same arguments return equal results

These results are exploited by tactic *f>equals* (see LibTactics); however the lemmas remain useful for forward-reasoning.

Section FuncEq.

Variables (A1 A2 A3 A4 A5 B:Type).

Variables (x1:A1) (x2:A2) (x3:A3) (x4:A4) (x5:A5).

```

Lemma fun_eq_1 : ∀ f g,
  f = g →
  f x1 = g x1 :> B.
Proof using. intros. subst¬. Qed.

Lemma fun_eq_2 : ∀ f g,
  f = g →
  f x1 x2 = g x1 x2 :> B.
Proof using. intros. subst¬. Qed.

Lemma fun_eq_3 : ∀ f g,
  f = g →
  f x1 x2 x3 = g x1 x2 x3 :> B.
Proof using. intros. subst¬. Qed.

Lemma fun_eq_4 : ∀ f g,
  f = g →
  f x1 x2 x3 x4 = g x1 x2 x3 x4 :> B.
Proof using. intros. subst¬. Qed.

Lemma fun_eq_5 : ∀ f g,
  f = g →
  f x1 x2 x3 x4 x5 = g x1 x2 x3 x4 x5 :> B.
Proof using. intros. subst¬. Qed.

End FuncEq.

```

3.5.3 Equal predicates applied to same arguments return equivalent results

These results are exploited by tactic *fequals* (see LibTactics); however the lemmas remain useful for forward-reasoning.

```

Section PredEq.
Variables (A1 A2 A3 A4 A5 B:Type).
Variables (x1:A1) (x2:A2) (x3:A3) (x4:A4) (x5:A5).

Lemma pred_eq_1 : ∀ P Q,
  P = Q →
  P x1 ↔ Q x1.
Proof using. intros. subst×. Qed.

Lemma pred_eq_2 : ∀ P Q,
  P = Q →
  P x1 x2 ↔ Q x1 x2.
Proof using. intros. subst×. Qed.

Lemma pred_eq_3 : ∀ P Q,
  P = Q →

```

```

P x1 x2 x3 ↔ Q x1 x2 x3.
Proof using. intros. subst×. Qed.

Lemma pred_eq_4 : ∀ P Q,
  P = Q →
  P x1 x2 x3 x4 ↔ Q x1 x2 x3 x4.
Proof using. intros. subst×. Qed.

Lemma pred_eq_5 : ∀ P Q,
  P = Q →
  P x1 x2 x3 x4 x5 ↔ Q x1 x2 x3 x4 x5.
Proof using. intros. subst×. Qed.

End PredEq.

```

3.6 Proof Irrelevance

The proof irrelevance lemma states that two proofs of a same proposition are always equal.
 $\forall (P : \text{Prop}) (\mathbf{p} \mathbf{q} : P), \mathbf{p} = \mathbf{q}$ This result is a consequence of propositional extensionality.

3.6.1 Proof of the proof-irrelevance result

Module PIFROMEXT.

Implicit Types $P : \text{Prop}$.

First, we prove that on an inhabited proposition type, there exists a fixpoint combinator.

```

Lemma prop_eq_self_impl_when_true : ∀ P,
  P →
  P = (P → P).
Proof using. intros. apply× prop_ext. Qed.

Record has_fixpoint (P:Prop) : Prop := has_fixpoint_make
  { has_fixpoint_F : (P → P) → P;
    has_fixpoint_fix : ∀ f, has_fixpoint_F f = f (has_fixpoint_F f) }.

```

```

Lemma prop_has_fixpoint_when_true : ∀ P,
  P →
  has_fixpoint P.
Proof using.
  intros P a. set (P' := P).
  set (g1 := id : P' → P). set (g2 := id : P → P').
  asserts¬ Fix: (∀ x, g1 (g2 x) = x).
  clearbody g1 g2. gen g1 g2.
  rewrite (prop_eq_self_impl_when_true a).
  subst P'. intros.

```

```

set (Y := fun f => (fun x => f (g1 x x)) (g2 (fun x => f (g1 x x)))).  

applys (has_fixpoint_make Y). { intros f. unfold Y at 1. rewrite¬ Fix. }  

Qed.

```

We exploit the fixpoint combinator on the negation function, applied to the following special proposition type (isomorphic to booleans, but living in Prop).

```

Inductive boolP : Prop :=
| trueP : boolP
| falseP : boolP.

```

Lemma trueP_eq_falseP : trueP = falseP.

Proof using.

```

lets (Y & Yfix): (@prop_has_fixpoint_when_true boolP trueP).
set (neg := fun b => match b with| trueP => falseP | falseP => trueP end).
lets F: ((rm Yfix) neg).
set (b := Y neg).
asserts¬ E: (b = Y neg).
destruct b.
{ change (trueP = neg trueP) in ⊢ ×. rewrite E. rewrite¬ ← F. }
{ change (neg falseP = falseP) in ⊢ ×. rewrite E. rewrite¬ ← F. }

```

Qed.

We now have two distinct constructors trueP and falseP, which we can distinguish in the logic using `match` for any goal concluding on a proposition; and, at the same time, these two constructors are provably equal. We can exploit these properties to prove that two proofs of a same theorem are equal.

Lemma proof_irrelevance :

```
   $\forall (P : \text{Prop}) (p q : P), p = q.$ 
```

Proof using.

```

  intros P p q.
  set (f := fun b => match b with| trueP => p | falseP => q end).
  change p with (f trueP).
  change q with (f falseP).
  rewrite¬ trueP_eq_falseP.

```

Qed.

End PIfromExt.

Lemma proof_irrelevance : $\forall (P : \text{Prop}) (p q : P), p = q.$

Proof using. exact PIfromExt.proof_irrelevance. Qed.

3.6.2 Consequences of proof irrelevance

Uniqueness of identity proofs

Lemma identity_proofs_unique :

```

 $\forall (A : \text{Type}) (x y : A) (p q : x = y),$ 
 $p = q.$ 

```

Proof using. intros. apply proof_irrelevance. Qed.

Uniqueness of reflexive identity proofs (special case)

Lemma reflexive_identity_proofs_unique :

```

 $\forall (A : \text{Type}) (x : A) (p : x = x),$ 
 $p = \text{refl_equal } x.$ 

```

Proof using. intros. applys identity_proofs_unique. Qed.

Invariance by substitution of reflexive equality proofs

Lemma eq_rect_refl_eq :

```

 $\forall (A : \text{Type}) (p : A) (Q : A \rightarrow \text{Type}) (x : Q p) (h : p = p),$ 
 $\text{eq_rect } p Q x p h = x.$ 

```

Proof using. intros. rewrite¬ (reflexive_identity_proofs_unique h). Qed.

Streicher's axiom K

Lemma streicher_K :

```

 $\forall (A : \text{Type}) (x : A) (P : x = x \rightarrow \text{Prop}),$ 
 $P (\text{refl_equal } x) \rightarrow$ 
 $\forall (p : x = x), P p.$ 

```

Proof using. intros. rewrite¬ (reflexive_identity_proofs_unique p). Qed.

3.6.3 Injectivity of equality on dependent pairs

This section establishes that `existT P p x = existT P p y` implies that `x` is equal to `y`. It indirectly results from the proof irrelevance property.

Definition of dependent equality, with non-dependent return type

Inductive **eq_dep_nd** ($A : \text{Type}$) ($P : A \rightarrow \text{Type}$)
 $(p : A) (x : P p) (q : A) (y : P q) : \text{Prop} :=$
 $| \text{eq_dep_nd_intro} : \forall (h : q = p),$
 $x = \text{eq_rect } q P y p h \rightarrow \text{eq_dep_nd } P p x q y.$

Arguments **eq_dep_nd** [A] [P] [p] [x] [q] [y].

Arguments **eq_dep_nd_intro** [A] [P] [p] [x] [q] [y].

Reflexivity of **eq_dep_nd**

Lemma **eq_dep_nd_refl** : $\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x : P p),$
 $\text{eq_dep_nd } x x.$

Proof using. intros. apply (eq_dep_nd_intro (refl_equal p)). auto. Qed.

Injectivity of **eq_dep_nd**

Lemma **eq_dep_nd_same_inv** :

```

 $\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x y : P p),$ 
 $\text{eq_dep_nd } x y \rightarrow$ 

```

$x = y$.

Proof using. introv H . inversions H . rewrite \neg eq_rect_refl_eq. Qed.

Equality on dependent pairs implies **eq_dep_nd**

Lemma eq_existT_inv :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p q : A) (x : P p) (y : P q),$
 $\text{existT } P p x = \text{existT } P q y \rightarrow$
eq_dep_nd $x y$.

Proof using. introv E . dependent rewrite E . simpl. apply eq_dep_nd_refl. Qed.

Injectivity of equality on dependent pairs

Lemma eq_existT_same_inv :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x y : P p),$
 $\text{existT } P p x = \text{existT } P p y \rightarrow$
 $x = y$.

Proof using. intros. apply eq_dep_nd_same_inv. apply \neg eq_existT_inv. Qed.

Reformulated as an equality

Lemma eq_existT_same_eq :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x y : P p),$
 $(\text{existT } P p x = \text{existT } P p y) = (x = y)$.

Proof using.

extens. iff M .

{ apply eq_dep_nd_same_inv. apply \neg eq_existT_inv. }
{ subst \times . }

Qed.

This is another consequence of proof irrelevance

Scheme eq_indd := Induction for **eq** Sort Prop.

Lemma exist_eq_exist : $\forall (A:\text{Type}) (P : A \rightarrow \text{Prop}) (x y : A) (p : P x) (q : P y),$
 $x = y \rightarrow$
 $\text{exist } P x p = \text{exist } P y q$.

Proof using.

intros. rewrite (proof_irrelevance q (eq_rect $x P p y H$)). subst \times .

Qed.

Lemma existT_eq_existT : $\forall (A:\text{Type}) (P : A \rightarrow \text{Prop}) (x y : A) (p : P x) (q : P y),$
 $x = y \rightarrow$
 $\text{existT } P x p = \text{existT } P y q$.

Proof using.

intros. rewrite (proof_irrelevance q (eq_rect $x P p y H$)). subst \times .

Qed.

Ltac fequal_support_for_exist tt ::=

first [apply exist_eq_exist | apply existT_eq_existT].

3.7 Dependent equality

In this section, we prove that **eq_dep** $x y$ implies $x = y$.

Definition of **eq_dep** (copied from the Prelude)

Inductive **eq_dep** ($A : \text{Type}$) ($P : A \rightarrow \text{Type}$) ($p : A$) ($x : P p$)
 $\quad : \forall q, P q \rightarrow \text{Prop} :=$
 $\quad | \text{eq_dep_refl} : \text{eq_dep } P p x p x.$

Arguments **eq_dep** [A] [P] [p] x [q].

Symmetry of **eq_dep**

Lemma **eq_dep_sym** : $\forall (A : \text{Type}) (P : A \rightarrow \text{Type})$
 $(p q : A) (x : P p) (y : P q),$
 $\text{eq_dep } x y \rightarrow$
 $\text{eq_dep } y x.$

Proof using. *introv E. destruct E. constructor. Qed.*

Transitivity of **eq_dep**

Lemma **eq_dep_trans** : $\forall (A : \text{Type}) (P : A \rightarrow \text{Type})$
 $(p q r : A) (y : P q) (x : P p) (z : P r),$
 $\text{eq_dep } x y \rightarrow$
 $\text{eq_dep } y z \rightarrow$
 $\text{eq_dep } x z.$

Proof using. *introv E F. destruct E. Qed.*

Proof of equivalence between **eq_dep_nd** and **eq_dep**

Scheme **eq_induction** := Induction for **eq** Sort Prop.

Lemma **eq_dep_of_eq_dep_nd** :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p q : A) (x : P p) (y : P q),$
 $\text{eq_dep_nd } x y \rightarrow$
 $\text{eq_dep } x y.$

Proof using.

introv E. destruct E as (h,H).
destruct h using eq_induction. subst¬. constructor.

Qed.

Lemma **eq_dep_nd_of_eq_dep** :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p q : A) (x : P p) (y : P q),$
 $\text{eq_dep } x y \rightarrow$
 $\text{eq_dep_nd } x y.$

Proof using. *introv H. destruct H. apply (eq_dep_nd_intro (refl_equal p)); auto. Qed.*

Injectivity of dependent equality

Lemma **eq_dep_same_inv** :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x y : P p),$

```
eq_dep x y →
```

```
x = y.
```

Proof using.

```
introv R. inversion R. apply eq_dep_nd_same_inv. apply¬ eq_dep_nd_of_eq_dep.
```

Qed.

Equality on dependent pairs implies dependent equality

Lemma eq_dep_of_eq_existT :

```
forall (A : Type) (P : A → Type) (p q : A) (x : P p) (y : P q),
```

```
existT P p x = existT P q y →
```

```
eq_dep x y.
```

Proof using. introv E. dependent rewrite E. simple¬. constructor. Qed.

3.8 John Major's equality

Require Import Coq.Logic.JMeq.

The module above defines John Major's equality:

Inductive JMeq (A : Type) (x : A) : forall B : Type, B -> Prop := | JMeq_refl : JMeq x x.

In this section, we prove that $\text{JMeq } x \ y$ implies $x = y$ when x and y have the same type.

Symmetry, transitivity of JMeq

Lemma JMeq_sym : $\forall (A B : \text{Type}) (x : A) (y : B)$,

```
JMeq x y →
```

```
JMeq y x.
```

Proof using. introv E. destruct¬ E. Qed.

Lemma JMeq_trans : $\forall (A B C : \text{Type}) (y : B) (x : A) (z : C)$,

```
JMeq x y →
```

```
JMeq y z →
```

```
JMeq x z.
```

Proof using. introv E F. destruct¬ E. Qed.

Local Hint Immediate JMeq_sym.

Relation between JMeq and eq_dep

Lemma eq_dep_of_JMeq : $\forall (A B : \text{Type}) (x : A) (y : B)$,

```
JMeq x y →
```

```
@eq_dep Type (fun T ⇒ T) A x B y.
```

Proof using. introv E. destruct E. constructor. Qed.

Lemma JMeq_of_eq_dep : $\forall (A B : \text{Type}) (x : A) (y : B)$,

```
@eq_dep Type (fun T ⇒ T) A x B y →
```

```
JMeq x y.
```

Proof using. introv E. destruct¬ E. Qed.

Injectivity of `JMeq`

Lemma `JMeq_same_inv` : $\forall (A : \text{Type}) (x y : A),$

JMeq $x y \rightarrow$

$x = y.$

Proof using.

introv E. apply (@eq_dep_same_inv Type (fun T => T)).

apply¬ eq_dep_of_JMeq.

Qed.

Chapter 4

Library `SLF.LibLogic`

```
Set Implicit Arguments.  
From SLF Require Import LibTactics.  
From SLF Require Export LibAxioms LibEqual.  
Generalizable Variables A B P.
```

4.1 Strong existentials

Type `sig` is defined in `LibLogicCore`
Projections

```
Definition sig_val (A : Type) (P : A → Prop) (e : sig P) : A :=  
  match e with exist _ a _ ⇒ a end.
```

```
Definition sig_proof (A : Type) (P : A → Prop) (e : sig P) : P (sig_val e) :=  
  match e with exist _ _ b ⇒ b end.
```

4.2 Inhabited types

4.2.1 Typeclass for describing inhabited types

The proposition `Inhab` A captures the fact that the type A is inhabited (i.e., there exists at least one value of type A).

```
Class Inhab (A:Type) : Prop :=  
  { Inhab_intro : (exists (x:A), True) }.
```

Hint Mode `Inhab` + : *typeclass_instances*.

4.2.2 Tactics taking into account

Extension to LibTactics' fast introduction tactic => to handle specifically the case of the Inhabited typeclass.

```
Ltac intro_nondeps_aux_special_intro G ::=  
  match G with  
  | (Inhab _) ⇒ idtac  
  end.
```

4.2.3 Arbitrary values for inhabited types

The value arbitrary can be used as a dummy value at any place where a value of an inhabited type is expected.

```
Definition arbitrary '{Inhab A} : A :=  
  sig_val (@indefinite_description A _ Inhab_intro).
```

Extraction of arbitrary constants as a runtime error.

4.2.4 Lemmas about inhabited types

Proving a type to be inhabited

```
Lemma Inhab_of_val : ∀ (A:Type),  
  A →  
  Inhab A.
```

Proof using. intros A x. constructor. ∃ x. auto. Qed.

Arrows are inhabited if their codomain is inhabited.

```
Instance Inhab_impl : ∀ A B {I:Inhab B},  
  Inhab (A → B).
```

Proof using. intros. apply (Inhab_of_val (fun _ ⇒ arbitrary)). Qed.

4.3 Non-constructive conditionals

4.3.1 Excluded middle

Every proposition is either **True** or **False**.

```
Lemma classic : ∀ (P : Prop),  
  P ∨ ¬ P.
```

Proof using.

```
  intros.  
  set (B1 := fun b ⇒ b = true ∨ P).  
  set (B2 := fun b ⇒ b = false ∨ P).
```

```

asserts H1: (ex B1).  $\exists$  true. left $\neg$ .
asserts H2: (ex B2).  $\exists$  false. left $\neg$ .
sets i1: (indefinite_description H1).
sets i2: (indefinite_description H2).
destruct (sig_proof i1) as [HA]; [|auto].
destruct (sig_proof i2) as [HB]; [|auto].
right. intros HP. asserts EB: (B1 = B2).
    apply pred_ext_1. intros b. split; intros _; right; auto.
    subst i1 i2. destruct EB.
    rewrite (proof_irrelevance H2 H1) in HB. congruence.
Qed.

```

Definition prop_inv := classic.

4.3.2 Strong excluded middle

From classical logic and indefinite description, we can prove the strong (or informative) excluded middle, which allows Coq definitions to make a case analysis on the truth value of any proposition.

Lemma classicT : $\forall (P : \text{Prop}), \{P\} + \{\neg P\}$.

Proof using

```

intros. pose (select := fun (b:bool)  $\Rightarrow$  if b then P else  $\neg$ P).
cuts (M,HP): { b:bool | select b }.
    destruct M. left $\neg$ . right $\neg$ .
    apply indefinite_description.
    destruct (prop_inv P).  $\exists\neg$  true.  $\exists\neg$  false.

```

Qed.

Simplification lemmas

Lemma classicT_l : $\forall (P : \text{Prop}) (H:P), \text{classicT } P = \text{left } _H$.

Proof using. intros. destruct (classicT P). f_equal. false \neg . Qed.

Lemma classicT_r : $\forall (P : \text{Prop}) (H:\neg P), \text{classicT } P = \text{right } _H$.

Proof using. intros. destruct (classicT P). false \neg . f_equal. Qed.

4.3.3 If-then-else on propositions

The expression *If* P *then* v1 *else* v2 can be used to build a conditional that depends on a proposition P.

Notation "'If' P 'then' v1 'else' v2" :=

(if (classicT P) then v1 else v2)

(at level 200, right associativity) : type_scope.

Section Ifthenelse.

Variables A : Type.

Implicit Types P : Prop.

Implicit Types x y : A.

Lemmas to simplify If-then-else statement

Lemma If_l : $\forall P x y,$

$P \rightarrow$

$(\text{If } P \text{ then } x \text{ else } y) = x.$

Proof using. intros. case_if \times . Qed.

Lemma If_r : $\forall P x y,$

$\neg P \rightarrow$

$(\text{If } P \text{ then } x \text{ else } y) = y.$

Proof using. intros. case_if \times . Qed.

A lemma to prove an equality between two If-then-else

Lemma If_eq : $\forall P P' x x' y y',$

$(P \leftrightarrow P') \rightarrow$

$(P \rightarrow x = x') \rightarrow$

$(\neg P \rightarrow y = y') \rightarrow$

$(\text{If } P \text{ then } x \text{ else } y) = (\text{If } P' \text{ then } x' \text{ else } y').$

Proof using. intros. do 2 case_if; autos \times . Qed.

A simpler version of the above lemma

Lemma If_eq_simple : $\forall P P' x x' y y',$

$(P \leftrightarrow P') \rightarrow$

$(x = x') \rightarrow$

$(y = y') \rightarrow$

$(\text{If } P \text{ then } x \text{ else } y) = (\text{If } P' \text{ then } x' \text{ else } y').$

Proof using. intros. subst. asserts_rewrite (P = P'). apply \neg prop_ext. auto. Qed.

End Ifthenelse.

4.3.4 Consequences

Propositional extensionality stated using itself

Lemma eq_prop_eq_iff : $\forall P Q,$

$(P = Q) = (P \leftrightarrow Q).$

Proof using. intros. extens. iff. subst \times . apply \neg prop_ext. Qed.

Propositional completeness (degeneracy)

Lemma prop_degeneracy : $\forall (P : \text{Prop}),$

$P = \text{True} \vee P = \text{False}.$

Proof using.

```
intros. destruct (prop_inv P).
  left. apply× prop_ext.
  right. apply× prop_ext.
```

Qed.

Independence of general premises

Lemma indep_general_premises :

```
∀ A ‘{Inhab A} (P : A → Prop) (Q : Prop),
  (Q → ∃ x, P x) →
  (∃ x, Q → P x).
```

Proof using.

```
introv I M. destruct (prop_inv Q).
destruct× (M H).
∃ (arbitrary (A:=A)). auto_false.
```

Qed.

Small drinker's paradox

Lemma small_drinker_paradox : ∀ ‘{Inhab A} (P : A → Prop),
 ∃ x, (exists x, P x) → P x.

Proof using.

```
intros A I P. destruct (prop_inv (exists x, P x)).
destruct H. ∃ x. auto.
∃ (arbitrary (A:=A)). auto_false.
```

Qed.

4.3.5 Tactic for proving tautologies by bruteforce case-analysis

The tactic *tautop* $P_1 \dots PN$ helps performing case analysis on the propositions/booleans that are given to it as parameters.

```
Ltac tautop_case P :=
  match type of P with
  | Prop ⇒ destruct (prop_degeneracy P)
  | bool ⇒ destruct P
  end.
```

```
Ltac tautop_pre tt :=
  intros;
  repeat rewrite eq_prop_eq_iff in *.
```

```
Ltac tautop_post tt :=
  subst;
  try solve [ intuition auto_false ].
```

```
Ltac tautop_core args :=
```

```

tautop_pre tt;
let rec aux Ps :=
  match Ps with
  | nil => idtac
  | cons (boxer ?P) ?Ps' => tautop_case P; aux Ps'
  end in
aux args;
tautop_post tt.

Ltac tautop_noargs tt :=
let rec aux Ps :=
  match goal with
  | ⊢ ∀ (_:Prop), _ =>
    let P := fresh "P" in intro P; aux (cons (boxer P) Ps)
  | _ => tautop_core Ps
  end
  in
aux constr:(@nil Boxer).

Tactic Notation "tautop" :=
  tautop_noargs tt.

Tactic Notation "tautop" constr(P1) :=
  tautop_core (» P1).

Tactic Notation "tautop" constr(P1) constr(P2) :=
  tautop_core (» P1 P2).

Tactic Notation "tautop" constr(P1) constr(P2) constr(P3) :=
  tautop_core (» P1 P2 P3).

```

4.4 Properties of logical combinators

4.4.1 Simplification of conjunction and disjunction

Section SimplConjDisj.
Implicit Types $P Q : \text{Prop}$.
Lemma and_True_l_eq : $\forall P,$
 $(\text{True} \wedge P) = P.$
Proof using. tautop. Qed.
Lemma and_True_r_eq : $\forall P,$
 $(P \wedge \text{True}) = P.$
Proof using. tautop. Qed.
Lemma and_False_l_eq : $\forall P,$
 $(\text{False} \wedge P) = \text{False}.$

```

Proof using. tautop. Qed.

Lemma and_False_r_eq :  $\forall P,$   

 $(P \wedge \text{False}) = \text{False}.$ 

Proof using. tautop. Qed.

Lemma or_True_l_eq :  $\forall P,$   

 $(\text{True} \vee P) = \text{True}.$ 

Proof using. tautop. Qed.

Lemma or_True_r_eq :  $\forall P,$   

 $(P \vee \text{True}) = \text{True}.$ 

Proof using. tautop. Qed.

Lemma or_False_l_eq :  $\forall P,$   

 $(\text{False} \vee P) = P.$ 

Proof using. tautop. Qed.

Lemma or_False_r_eq :  $\forall P,$   

 $(P \vee \text{False}) = P.$ 

Proof using. tautop. Qed.

End SimplConjDisj.

```

4.4.2 Distribution of negation on basic operators

```

Section SimplNot.

Implicit Types  $P Q$  : Prop.

Lemma not_True_eq :  

 $(\neg \text{True}) = \text{False}.$ 

Proof using. tautop. Qed.

Lemma not_False_eq :  

 $(\neg \text{False}) = \text{True}.$ 

Proof using. tautop. Qed.

Lemma not_not_eq :  $\forall P,$   

 $(\neg (\neg P)) = P.$ 

Proof using. tautop. Qed.

Lemma not_and_eq :  $\forall P Q,$   

 $(\neg (P \wedge Q)) = (\neg P \vee \neg Q).$ 

Proof using. tautop. Qed.

Lemma not_or_eq :  $\forall P Q,$   

 $(\neg (P \vee Q)) = (\neg P \wedge \neg Q).$ 

Proof using. tautop. Qed.

Lemma not_impl_eq :  $\forall P Q,$   

 $(\neg (P \rightarrow Q)) = (P \wedge \neg Q).$ 

```

Proof using. *tautop*. Qed.

Lemma not_or_nots_eq : $\forall P Q, (\neg(\sim P \vee \neg Q)) = (P \wedge Q)$.

Proof using. *tautop*. Qed.

Lemma not_and_nots_eq : $\forall P Q, (\neg(\sim P \wedge \neg Q)) = (P \vee Q)$.

Proof using. *tautop*. Qed.

End SimplNot.

4.4.3 Double negation and contrapose

Section DoubleNeg.

Implicit Types $P Q : \text{Prop}$.

Double negation

Lemma not_not_inv : $\forall P, \neg\neg P \rightarrow P$.

Proof using. *tautop*. Qed.

Lemma not_not : $\forall P, P \rightarrow \neg\neg P$.

Proof using. *tautop*. Qed.

Contrapose

Lemma contrapose_eq : $\forall P Q, (\neg P \rightarrow \neg Q) = (Q \rightarrow P)$.

Proof using. *tautop*. Qed.

Lemma contrapose_inv : $\forall P Q, (\neg Q \rightarrow \neg P) \rightarrow (P \rightarrow Q)$.

Proof using. *tautop*. Qed.

Lemma contrapose : $\forall P Q, (Q \rightarrow P) \rightarrow (\neg P \rightarrow \neg Q)$.

Proof using. *tautop*. Qed.

Negation is injective

Lemma injective_not : $\forall P Q, (\neg P) = (\neg Q) \rightarrow (P = Q)$.

Proof using. *tautop*. Qed.

End DoubleNeg.

4.4.4 Distribution of negation on quantifiers

Section SimplNotQuantifiers.

Variables ($A : \text{Type}$).

Implicit Types $P : A \rightarrow \text{Prop}$.

Three auxiliary facts (private lemmas)

Lemma exists_of_not_forall : $\forall P,$

$$\neg(\forall x, \neg P x) \rightarrow (\exists x, P x).$$

Proof using. intros. apply \times not_not_inv. Qed.

Lemma forall_of_not_exists : $\forall P,$

$$\neg(\exists x, \neg P x) \rightarrow (\forall x, P x).$$

Proof using. intros. apply \times not_not_inv. Qed.

Lemma not_not_pred_eq : $\forall A (P:A \rightarrow \text{Prop}),$

$$P = (\text{fun } x \Rightarrow \neg \neg (P x)).$$

Proof using. intros. extens. intros. rewrite \times not_not_eq. Qed.

Rewriting rules for quantifiers

Lemma not_forall_eq : $\forall P,$

$$(\neg(\forall x, P x)) = (\exists x, \neg P x).$$

Proof using.

extens. iff.

{ apply exists_of_not_forall. rewrite \neg (not_not_pred_eq P) in $H.$ }

{ intros M . destruct $\neg H$ as [$x Cx$]. }

Qed.

Lemma not_exists_eq : $\forall P,$

$$(\neg(\exists x, P x)) = (\forall x, \neg P x).$$

Proof using.

intros. apply injective_not. rewrite not_forall_eq.

rewrite not_not_eq. set ($P' := P$) at 1.

rewrite \neg (not_not_pred_eq P').

Qed.

Lemma not_forall_not_eq : $\forall P,$

$$(\neg(\forall x, \neg P x)) = (\exists x, P x).$$

Proof using.

intros. rewrite not_forall_eq.

```
set (P':=P) at 2. rewrite  $\neg$  (not_not_pred_eq P').  
Qed.
```

```
Lemma not_exists_not_eq :  $\forall P,$   
 $(\neg (\exists x, \neg P x)) = (\forall x, P x).$ 
```

Proof using.

```
intros. rewrite not_exists_eq.  
set (P':=P) at 2. rewrite  $\neg$  (not_not_pred_eq P').
```

Qed.

End SimplNotQuantifiers.

4.4.5 Propositions equal to **True** or **False**

Section EqTrueFalse.

Implicit Types $P Q : \text{Prop}.$

Propositions equal to **True**

```
Lemma prop_eq_True_eq :  $\forall P,$   
 $(P = \text{True}) = P.$ 
```

Proof using. *tautop*. Qed.

```
Lemma prop_eq_True :  $\forall P,$   
 $P \rightarrow$   
 $P = \text{True}.$ 
```

Proof using. *tautop*. Qed.

```
Lemma prop_eq_True_inv :  $\forall P,$   
 $P = \text{True} \rightarrow$   
 $P.$ 
```

Proof using. *tautop*. Qed.

Propositions equal to **False**

```
Lemma prop_eq_False_eq :  $\forall P,$   
 $(P = \text{False}) = \neg P.$ 
```

Proof using. *tautop*. Qed.

```
Lemma prop_eq_False :  $\forall P,$   
 $\neg P \rightarrow$   
 $P = \text{False}.$ 
```

Proof using. *tautop*. Qed.

```
Lemma prop_eq_False_inv :  $\forall P,$   
 $P = \text{False} \rightarrow$   
 $\neg P.$ 
```

Proof using. *tautop*. Qed.

End EqTrueFalse.

Hint Resolve *prop_eq_True* *prop_eq_False*.

4.4.6 Disequal propositions

Section NeqProp.

Implicit Types P Q : Prop.

Propositions not **True** or not **False**

Lemma *prop_neq_True_eq* : $\forall P,$

$$(P \neq \text{True}) = \neg P.$$

Proof using. *tautop*. Qed.

Lemma *prop_neq_False_eq* : $\forall P,$

$$(P \neq \text{False}) = P.$$

Proof using. *tautop*. Qed.

Proving two propositions not equal

Lemma *prop_neq_of_iff_l* : $\forall P Q,$

$$(P \leftrightarrow \neg Q) \rightarrow$$

$$P \neq Q.$$

Proof using. *tautop*. Qed.

Lemma *prop_neq_of_iff_r* : $\forall P Q,$

$$(\neg P \leftrightarrow Q) \rightarrow$$

$$P \neq Q.$$

Proof using. *tautop*. Qed.

Lemma *prop_neq_inv_iff_l* : $\forall P Q,$

$$P \neq Q \rightarrow$$

$$(P \leftrightarrow \neg Q).$$

Proof using. *tautop*. Qed.

Lemma *prop_neq_inv_iff_r* : $\forall P Q,$

$$P \neq Q \rightarrow$$

$$(\neg P \leftrightarrow Q).$$

Proof using. *tautop*. Qed.

True is not False

Lemma *True_neq_False* : **True** \neq **False**.

Proof using. *tautop*. Qed.

End NeqProp.

Hint Resolve *True_neq_False*.

4.4.7 Peirce rule and similar

Section ClassicOr.

Implicit Types $P \ Q : \text{Prop}$.

Peirce's result: proving a fact by assuming its negation

```
Lemma assume_not :  $\forall P,$ 
   $(\neg P \rightarrow P) \rightarrow$ 
   $P.$ 
```

Proof using. *tautop*. Qed.

Proving a disjunction, assuming the negation of the other branch

```
Lemma or_classic_l :  $\forall P \ Q,$ 
   $(\neg Q \rightarrow P) \rightarrow$ 
   $P \vee Q.$ 
```

Proof using. *tautop*. Qed.

```
Lemma or_classic_r :  $\forall P \ Q,$ 
   $(\neg P \rightarrow Q) \rightarrow$ 
   $P \vee Q.$ 
```

Proof using. *tautop*. Qed.

Same, but in forward style

```
Lemma or_inv_classic_l :  $\forall P \ Q,$ 
   $P \vee Q \rightarrow$ 
   $(P \vee (\neg P \wedge Q)).$ 
```

Proof using. *tautop*. Qed.

```
Lemma or_inv_classic_r :  $\forall P \ Q,$ 
   $P \vee Q \rightarrow$ 
   $(Q \vee (P \wedge \neg Q)).$ 
```

Proof using. *tautop*. Qed.

End ClassicOr.

4.4.8 Properties of logical equivalence

Section *IffProp*.

Implicit Types $P \ Q \ R : \text{Prop}$.

Introduction

```
Lemma iff_eq_and :  $\forall P \ Q : \text{Prop},$ 
   $(P \leftrightarrow Q) = ((P \rightarrow Q) \wedge (Q \rightarrow P)).$ 
```

Proof using. *tautop*. Qed.

```
Lemma iff_intro :  $\forall P \ Q : \text{Prop},$ 
   $(P \rightarrow Q) \rightarrow$ 
   $(Q \rightarrow P) \rightarrow$ 
   $(P \leftrightarrow Q).$ 
```

Proof using. intros. rewrite× iff_eq_and. Qed.

Reflexivity: `refl iff`

```
Lemma iff_refl : ∀ P,  
  P ↔ P.
```

Proof using. *tautop*. Qed.

Symmetry: `sym iff`

```
Lemma iff_sym : ∀ P Q,  
  (P ↔ Q) →  
  (Q ↔ P).
```

Proof using. *tautop*. Qed.

Transitivity: `trans iff`

```
Lemma iff_trans : ∀ P Q R,  
  (P ↔ Q) →  
  (Q ↔ R) →  
  (P ↔ R).
```

Proof using. *tautop*. Qed.

First projection

```
Lemma iff_l : ∀ P Q,  
  (P ↔ Q) →  
  P →  
  Q.
```

Proof using. *tautop*. Qed.

Second projection

```
Lemma iff_r : ∀ P Q,  
  (P ↔ Q) →  
  Q →  
  P.
```

Proof using. *tautop*. Qed.

Contrapose of the first projection

```
Lemma iff_not_l : ∀ P Q,  
  (P ↔ Q) →  
  ¬ P →  
  ¬ Q.
```

Proof using. *tautop*. Qed.

Contrapose of the second projection

```
Lemma iff_not_r : ∀ P Q,  
  (P ↔ Q) →  
  ¬ Q →  
  ¬ P.
```

Proof using. *tautop*. Qed.

Negation can change side of an equivalence

```
Lemma iff_not_swap_eq : ∀ P Q,
  ((¬ P) ↔ Q) = (P ↔ (¬ Q)).
```

Proof using. *tautop*. Qed.

Negation can be cancelled on both sides

```
Lemma iff_not_not_eq : ∀ P Q,
  ((¬ P) ↔ (¬ Q)) = (P ↔ Q).
```

Proof using. *tautop*. Qed.

End IffProp.

4.5 Tactics

4.5.1 Tactic for simplifying expressions

The tactic *rew_logic* can be used to automatically simplify logical expressions. Syntax *rew_logic in H* and *rew_logic in ** are also available.

Hint Rewrite

```
not_not_eq not_and_eq not_or_eq not_impl_eq not_True_eq not_False_eq
not_forall_eq not_forall_not_eq
not_exists_eq not_exists_not_eq not_impl_eq
prop_eq_True_eq prop_eq_False_eq eq_prop_eq_iff
and_True_l_eq and_True_r_eq and_False_l_eq and_False_r_eq
or_True_l_eq or_True_r_eq or_False_l_eq or_False_r_eq
not_False_eq not_True_eq
: rew_logic.
```

```
Tactic Notation "rew_logic" :=
  autorewrite with rew_logic.
```

```
Tactic Notation "rew_logic" "in" hyp(H) :=
  autorewrite with rew_logic in H.
```

```
Tactic Notation "rew_logic" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_logic).
```

```
Tactic Notation "rew_logic" "˜" :=
  rew_logic; auto_tilde.
```

```
Tactic Notation "rew_logic" "˜" "in" hyp(H) :=
  rew_logic in H; auto_tilde.
```

```
Tactic Notation "rew_logic" "˜" "in" "*" :=
  rew_logic in *; auto_tilde.
```

```
Tactic Notation "rew_logic" "*" :=
  rew_logic; auto_star.
```

```
Tactic Notation "rew_logic" "*" "in" hyp(H) :=
```

```

rew_logic in H; auto_star.

Tactic Notation "rew_logic" "*" "in" "*" :=
  rew_logic in *; auto_star.

```

4.5.2 Tactic tests: P for classical disjunction on P .

The tactic *tests*: P can be used to tests whether the proposition P is true or not. If P is an equality, it is substituted. Use the tactic *tests_nosubst*: P to disable the automated substitution. Use the tactic *tests_basic*: P to moreover disable simplification of logical expressions. To name the resulting hypotheses use *tests I*: P , or *tests I1 I2*: P to assign different names to both sides. (In LibReflect, the tactic is extended so that P can be a boolean.)

```

Ltac tests_ssum_base E H1 H2 :=
  destruct E as [H1|H2].

Ltac tests_prop_base E H1 H2 :=
  tests_ssum_base (classicT E) H1 H2.

Ltac tests_dispatch E H1 H2 :=
  match type of E with
  | Prop => tests_prop_base E H1 H2
  | {_}+{_} => tests_ssum_base E H1 H2
  end.

Ltac tests_post H introstac :=
  tryfalse; rew_logic in H; revert H;
  introstac tt; tryfalse.

Ltac tests_post_subst H I :=
  tests_post H ltac:(fun _ => first [ intro_subst_hyp | intros I ]).

Ltac tests_post_nosubst H I :=
  tests_post H ltac:(fun _ => intros I).

Ltac tests_base E I1 I2 tests_post :=
  let H1 := fresh "TEMP" in
  let H2 := fresh "TEMP" in
  tests_dispatch E H1 H2;
  [ tests_post H1 I1
  | tests_post H2 I2 ].

Tactic Notation "tests" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E)
:=
  tests_base E I1 I2 ltac:(tests_post_subst).

Tactic Notation "tests" simple_intropattern(I) ":" constr(E) :=
  tests I I: E.

Tactic Notation "tests" ":" constr(E) :=

```

```

let I := fresh "C" in tests I: E.
Tactic Notation "tests" "˜" ":" constr(E) :=
  tests: E; auto_tilde.
Tactic Notation "tests" "*" ":" constr(E) :=
  tests: E; auto_star.
Tactic Notation "tests_nosubst" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E)
:=
  tests_base E I1 I2 ltac:(tests_post_nosubst).
Tactic Notation "tests_nosubst" simple_intropattern(I) ":" constr(E) :=
  tests_nosubst I I: E.
Tactic Notation "tests_nosubst" ":" constr(E) :=
  let I := fresh "C" in tests I: E.
Tactic Notation "tests_nosubst" "˜" ":" constr(E) :=
  tests: E; auto_tilde.
Tactic Notation "tests_nosubst" "*" ":" constr(E) :=
  tests: E; auto_star.
Tactic Notation "tests_basic" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E)
:=
  tests_dispatch E I1 I2.
Tactic Notation "tests_basic" simple_intropattern(I1) ":" constr(E) :=
  tests_basic I1 I1: E.
Tactic Notation "tests_basic" ":" constr(E) :=
  let C := fresh "C" in tests_basic C: E.

```

4.5.3 Tactic *absurds*

absurds applies to a goal G and replaces it with $(\sim G) \rightarrow \text{False}$. The expression $\neg G$ is simplified using *rew_logic*.

absurds ;=> H introduces H as the negation of the goal, and leaves False is the goal.
absurds_nosimpl is similar, but it does not perform simplifications.

```

Lemma absurds_lemma : ∀ (G:Prop),
  ((¬ G) → False) →
  G.

```

Proof using. intros. applys¬ not_not_inv. Qed.

```

Ltac absurds_nosimpl_core tt :=
  applys absurds_lemma.

```

```

Tactic Notation "absurds_nosimpl" :=
  absurds_nosimpl_core tt.

```

```

Ltac absurds_post tt :=
  rew_logic.

```

```

Ltac absurds_core tt :=

```

```

applys absurd_lemma;
absurd_post tt.

Tactic Notation "absurd" :=
  absurd_core tt.

```

4.6 Predicate combinators, comparison and compatibility

4.6.1 Definition of predicate combinators

```

Definition pred_true {A : Type} :=
  fun (_:A) => True.

Definition pred_false {A : Type} :=
  fun (_:A) => False.

Definition pred_not (A : Type) (P : A → Prop) :=
  fun x =>  $\neg(P\ x)$ .

Definition pred_and (A : Type) (P Q : A → Prop) :=
  fun x => P x  $\wedge$  Q x.

Definition pred_or (A : Type) (P Q : A → Prop) :=
  fun x => P x  $\vee$  Q x.

Definition pred_impl (A : Type) (P Q : A → Prop) :=
  fun x => P x  $\rightarrow$  Q x.

Hint Unfold pred_true pred_false.

```

4.6.2 Properties of combinators

```

Lemma pred_conj_forall_distrib :  $\forall A (P\ Q : A \rightarrow \text{Prop}),$ 
   $((\forall x, P\ x) \wedge (\forall x, Q\ x)) = (\forall x, P\ x \wedge Q\ x).$ 
Proof using. intros. apply prop_ext. iff H. autos×. split; intros x; apply× (H x).
Qed.

```

4.6.3 Order on predicates

```

Definition pred_incl (A : Type) (P Q : A → Prop) :=
   $\forall x, P\ x \rightarrow Q\ x.$ 

```

```

Lemma pred_eq_forall_impl :  $\forall A (P\ Q : A \rightarrow \text{Prop}),$ 
  pred_incl P Q =  $(\forall x, P\ x \rightarrow Q\ x).$ 
Proof using. auto. Qed.

```

```
Lemma pred_incl_refl : ∀ A (P : A → Prop),
  pred_incl P P.
```

Proof using. *unfold*s \neg pred_incl. Qed.

```
Lemma pred_incl_trans : ∀ A (Q P R : A → Prop),
  pred_incl P Q →
  pred_incl Q R →
  pred_incl P R.
```

Proof using. *unfold*s \neg pred_incl. Qed.

```
Lemma pred_incl_antisym : ∀ A (P Q : A → Prop),
  pred_incl P Q →
  pred_incl Q P →
  P = Q.
```

Proof using. *extens* \times . Qed.

See also LibRelation and LibOrder for higher-level statements of these results

4.7 Existentials

4.8 Properties of unique existentials

`unique_st P x` asserts that x is the unique element for which $P x$ holds.

```
Definition unique_st (A : Type) (P : A → Prop) (x : A) :=
  P x ∧ ∀ y, P y → y = x.
```

Hint Unfold unique_st.

`ex_unique P` asserts that there exists a unique element for which $P x$ holds.

```
Definition ex_unique (A : Type) (P : A → Prop) :=
  ex (unique_st P).
```

$\exists! x, P$ is the notation for `ex_unique P`.

```
Notation "'exists' ! x , P" := (ex_unique (fun x ⇒ P))
  (at level 200, x ident, right associativity,
  format "[ 'exists' ! / 'x , '/ 'P ']") : type_scope.
```

```
Notation "'exists' ! x : A , P" := 
  (ex_unique (fun x:A ⇒ P))
  (at level 200, x ident, right associativity,
  format "[ 'exists' ! / 'x : A , '/ 'P ']") : type_scope.
```

`at_most_one P` asserts that there exists at most one element for which $P x$ holds. In other words, P has 0 or 1 inhabitant.

```
Definition at_most_one (A : Type) (P : A → Prop) :=
  ∀ x y, P x → P y → x = y.
```

```

Section UniqueProp.
Variables (A : Type).
Implicit Types (P : A → Prop).

 $\exists! x, P$  entails  $\exists x, P$ 

Lemma ex_of_ex_unique : ∀ P,
  ex_unique P →
  ex P.

Proof using. introv (x&H&U). eauto. Qed.

 $\exists! x, P$  entails at_most_one  $P$ 

Lemma at_most_one_of_ex_unique : ∀ P,
  ex_unique P →
  at_most_one P.

Proof using.
  introv (a&H&U) Px Py. applys¬ eq_trans a. rewrite¬ ← (U y).
Qed.

 $\exists x, P$  and at_most_one  $P$  entail  $\exists! x, P$ 

Lemma ex_unique_of_ex_at_most_one : ∀ P,
  ex P →
  at_most_one P →
  ex_unique P.

Proof using. introv [x Px] H. ∃ x. split¬. Qed.

End UniqueProp.

```

4.9 Conjunctions

4.9.1 Changing the order of branches

```

Lemma conj_swap: ∀ (P Q: Prop),
  P →
  Q →
  Q ∧ P.

```

Proof using. autos×. Qed.

```

Lemma conj_dup_r : ∀ P Q : Prop,
  Q →
  (Q → P) →
  P ∧ Q.

```

Proof using. autos×. Qed.

```

Lemma conj_dup_l : ∀ P Q : Prop,
  P →

```

$$(P \rightarrow Q) \rightarrow \\ P \wedge Q.$$

Proof using. *auto*s×. Qed.

4.9.2 Parallel strengthening of a conjunction

Lemma conj_strengthen_2 : $\forall (Q1\ Q2\ P1\ P2 : \text{Prop}),$

$$(Q1 \wedge Q2) \rightarrow \\ (Q1 \rightarrow P1) \rightarrow \\ (Q2 \rightarrow P2) \rightarrow \\ (P1 \wedge P2).$$

Proof using. *auto*s×. Qed.

Lemma conj_strengthen_3 : $\forall (Q1\ Q2\ Q3\ P1\ P2\ P3 : \text{Prop}),$

$$(Q1 \wedge Q2 \wedge Q3) \rightarrow \\ (Q1 \rightarrow P1) \rightarrow \\ (Q2 \rightarrow P2) \rightarrow \\ (Q3 \rightarrow P3) \rightarrow \\ (P1 \wedge P2 \wedge P3).$$

Proof using. *auto*s×. Qed.

Lemma conj_strengthen_4 : $\forall (Q1\ Q2\ Q3\ Q4\ P1\ P2\ P3\ P4 : \text{Prop}),$

$$(Q1 \wedge Q2 \wedge Q3 \wedge Q4) \rightarrow \\ (Q1 \rightarrow P1) \rightarrow \\ (Q2 \rightarrow P2) \rightarrow \\ (Q3 \rightarrow P3) \rightarrow \\ (Q4 \rightarrow P4) \rightarrow \\ (P1 \wedge P2 \wedge P3 \wedge P4).$$

Proof using. *auto*s×. Qed.

Lemma conj_strengthen_5 : $\forall (Q1\ Q2\ Q3\ Q4\ Q5\ P1\ P2\ P3\ P4\ P5 : \text{Prop}),$

$$(Q1 \wedge Q2 \wedge Q3 \wedge Q4 \wedge Q5) \rightarrow \\ (Q1 \rightarrow P1) \rightarrow \\ (Q2 \rightarrow P2) \rightarrow \\ (Q3 \rightarrow P3) \rightarrow \\ (Q4 \rightarrow P4) \rightarrow \\ (Q5 \rightarrow P5) \rightarrow \\ (P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5).$$

Proof using. *auto*s×. Qed.

4.9.3 Projections of lemmas concluding on a conjunction

Section ProjLemma.

Variables ($A1 : \text{Type}$).

```

Variables (A2 : ∀ (x1 : A1), Type).
Variables (A3 : ∀ (x1 : A1) (x2 : A2 x1), Type).
Variables (A4 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), Type).
Variables (A5 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), Type).
Variables (A6 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), Type).
Variables (A7 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5), Type).
Variables (A8 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6), Type).
Variables (A9 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6) (x8 : A8 x7), Type).
Variables (A10 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6) (x8 : A8 x7) (x9 : A9 x8), Type).
Variables (A11 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6) (x8 : A8 x7) (x9 : A9 x8) (x10 : A10 x9), Type).

```

```

Ltac prove_forall_conj_inv :=
  intros;
  match goal with H: context [_ ∧ _] ⊢ _ ⇒
    split; intros; apply H end.

```

```

Lemma forall_conj_inv_1 : ∀ (P Q : ∀ (x1:A1), Prop),
  (forall x1, P x1 ∧ Q x1) →
  (forall x1, P x1) ∧
  (forall x1, Q x1).

```

Proof using. *prove_forall_conj_inv*. Qed.

```

Lemma forall_conj_inv_2 : ∀ (P Q : ∀ (x1:A1) (x2:A2 x1), Prop),
  (forall x1 x2, P x1 x2 ∧ Q x1 x2) →
  (forall x1 x2, P x1 x2) ∧
  (forall x1 x2, Q x1 x2).

```

Proof using. *prove_forall_conj_inv*. Qed.

```

Lemma forall_conj_inv_3 : ∀ (P Q : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2), Prop),
  (forall x1 x2 x3, P x1 x2 x3 ∧ Q x1 x2 x3) →
  (forall x1 x2 x3, P x1 x2 x3) ∧
  (forall x1 x2 x3, Q x1 x2 x3).

```

Proof using. *prove_forall_conj_inv*. Qed.

```

Lemma forall_conj_inv_4 : ∀ (P Q : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2) (x4:A4 x3), Prop),
  (forall x1 x2 x3 x4, P x1 x2 x3 x4 ∧ Q x1 x2 x3 x4) →
  (forall x1 x2 x3 x4, P x1 x2 x3 x4) ∧
  (forall x1 x2 x3 x4, Q x1 x2 x3 x4).

```

Proof using. *prove_forall_conj_inv*. Qed.

```

Lemma forall_conj_inv_5 : ∀ (P Q : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2))

```

$(x4:A4\ x3)\ (x5:A5\ x4)$, Prop),
 $(\forall x1\ x2\ x3\ x4\ x5, P\ x1\ x2\ x3\ x4\ x5 \wedge Q\ x1\ x2\ x3\ x4\ x5) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5, P\ x1\ x2\ x3\ x4\ x5) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5, Q\ x1\ x2\ x3\ x4\ x5)$.

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_6 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5))$, Prop),
 $(\forall x1\ x2\ x3\ x4\ x5\ x6, P\ x1\ x2\ x3\ x4\ x5\ x6 \wedge Q\ x1\ x2\ x3\ x4\ x5\ x6) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6, P\ x1\ x2\ x3\ x4\ x5\ x6) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6, Q\ x1\ x2\ x3\ x4\ x5\ x6)$.

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_7 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6))$, Prop),
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7 \wedge Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7)$.

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_8 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6) (x8:A8\ x7))$, Prop),
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8 \wedge Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8)$.

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_9 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6) (x8:A8\ x7) (x9:A9\ x8))$, Prop),
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9 \wedge Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9)$.

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_10 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6) (x8:A8\ x7) (x9:A9\ x8) (x10:A10\ x9))$, Prop),
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10 \wedge Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10)$.

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_11 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6) (x8:A8\ x7) (x9:A9\ x8) (x10:A10\ x9))$

$(x11:A11\ x10), \text{Prop},$

$(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11 \wedge Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11) \rightarrow$

$(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11) \wedge$

$(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11).$

Proof using. *prove_forall_conj_inv*. Qed.

End ProjLemma.

Arguments forall_conj_inv_1 [A1] [P] [Q].

Arguments forall_conj_inv_2 [A1] [A2] [P] [Q].

Arguments forall_conj_inv_3 [A1] [A2] [A3] [P] [Q].

Arguments forall_conj_inv_4 [A1] [A2] [A3] [A4] [P] [Q].

Arguments forall_conj_inv_5 [A1] [A2] [A3] [A4] [A5] [P] [Q].

Arguments forall_conj_inv_6 [A1] [A2] [A3] [A4] [A5] [A6] [P] [Q].

Arguments forall_conj_inv_7 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [P] [Q].

Arguments forall_conj_inv_8 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [P] [Q].

Arguments forall_conj_inv_9 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [A9] [P] [Q].

Arguments forall_conj_inv_10 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [A9] [A10] [P] [Q].

Arguments forall_conj_inv_11 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [A9] [A10] [A11] [P] [Q].

Chapter 5

Library SLF.LibOperation

```
Set Implicit Arguments.  
From SLF Require Import LibTactics.
```

5.1 Definitions

```
Section Definitions.  
Variables (A : Type).  
Implicit Types f g : A → A → A.  
Implicit Types i : A → A.
```

5.1.1 Commutativity, associativity

Commutativity

```
Definition comm f := ∀ x y,  
  f x y = f y x.
```

Associativity

```
Definition assoc f := ∀ x y z,  
  f x (f y z) = f (f x y) z.
```

Combined associativity commutativity

```
Definition comm_assoc f := ∀ x y z,  
  f x (f y z) = f y (f x z).
```

5.1.2 Distributivity

Distributivity of unary operator

```
Definition distrib i f := ∀ x y,
```

$i(f x y) = f(i x)(i y).$

comm distributivity of unary operator

Definition distrib_comm $i f := \forall x y,$

$i(f x y) = f(i y)(i x).$

Left distributivity

Definition distrib_l $f g := \forall x y z,$

$f x(g y z) = g(f x y)(f x z).$

Right distributivity

Definition distrib_r $f g := \forall x y z,$

$f(g y z)x = g(f y x)(f z x).$

5.1.3 Neutral and absorbant

Left Neutral

Definition neutral_l $f e := \forall x,$

$f e x = x.$

Right Neutral

Definition neutral_r $f e := \forall x,$

$f x e = x.$

Left Absorbant

Definition absorb_l $f a := \forall x,$

$f a x = a.$

Right Absorbant

Definition absorb_r $f a := \forall x,$

$f x a = a.$

Idempotence

Definition idempotent $i := \forall x,$

$i(i x) = i x.$

Idempotence

Definition involutive $i := \forall x,$

$i(i x) = x.$

Idempotence for binary operators

Definition idempotent2 $f := \forall x,$

$f x x = x.$

Self Neutral

Definition self_neutral $f e x :=$

$f x x = e.$

5.1.4 Inverses

Left Inverse

```
Definition inverse_for_l f e a b :=  
  f a b = e.
```

Right Inverse

```
Definition inverse_for_r f e a b :=  
  f b a = e.
```

Left Inverse function – todo arguments in order f i e ?

```
Definition inverse_l f e i := ∀ x,  
  f (i x) x = e.
```

Right Inverse function

```
Definition inverse_r f e i := ∀ x,  
  f x (i x) = e.
```

Self Inverse

```
Definition self_inverse i x :=  
  i x = x.
```

End Definitions.

5.1.5 Morphism and automorphism

Morphism

```
Definition morphism (A B : Type) (h : A → B) (f : A → A → A) (g : B → B → B) :=  
  ∀ x y, h (f x y) = g (h x) (h y).
```

Auto-morphism

```
Definition automorphism A := @morphism A A.  
Arguments automorphism [A].
```

5.1.6 Injectivity

```
Definition injective A B (f : A → B) :=  
  ∀ x y, f x = f y → x = y.
```

5.2 Lemmas

Section OpProperties.

Variables (A : Type).

Implicit Types h : A → A.

```
Implicit Types  $f\ g : A \rightarrow A \rightarrow A$ .
```

```
Lemma idempotent_inv :  $\forall h\ x\ y,$   
  idempotent  $h \rightarrow$   
   $y = h\ x \rightarrow$   
   $h\ y = y$ .
```

```
Proof using. intros. subst $\times$ . Qed.
```

For comm operators, right-properties can be derived from corresponding left-properties

```
Lemma neutral_r_of_comm_neutral_l :  $\forall f\ e,$   
  comm  $f \rightarrow$   
  neutral_l  $f\ e \rightarrow$   
  neutral_r  $f\ e$ .
```

```
Proof using. introv C N. intros_all. rewrite $\times$  C. Qed.
```

```
Lemma inverse_r_of_comm_inverse_l :  $\forall f\ e\ i,$   
  comm  $f \rightarrow$   
  inverse_l  $f\ e\ i \rightarrow$   
  inverse_r  $f\ e\ i$ .
```

```
Proof using. introv C I. intros_all. rewrite $\times$  C. Qed.
```

```
Lemma distrib_r_of_comm_distrib_l :  $\forall f\ g,$   
  comm  $f \rightarrow$   
  distrib_l  $f\ g \rightarrow$   
  distrib_r  $f\ g$ .
```

```
Proof using.
```

```
  introv C N. intros_all. unfolds distrib_l.  
  do 3 rewrite  $\leftarrow (C\ x)$ . auto.
```

```
Qed.
```

```
Lemma comm_assoc_of_comm_and_assoc :  $\forall f,$   
  comm  $f \rightarrow$   
  assoc  $f \rightarrow$   
  comm_assoc  $f$ .
```

```
Proof using.
```

```
  introv C S. intros_all. rewrite C.  
  rewrite  $\leftarrow S$ . rewrite $\neg (C\ x)$ .
```

```
Qed.
```

```
End OpProperties.
```

Chapter 6

Library `SLF.LibBool`

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibOperation.

6.1 Boolean type

6.1.1 Definition

From the Prelude:

```
Inductive bool : Type := | true : bool | false : bool.
```

6.1.2 Inhabited

Instance Inhab_bool : **Inhab bool**.

Proof using constructor apply (Inhab_of_val **true**). Qed.

6.1.3 Extensionality

See LibReflect for extensionality of booleans.

6.2 Boolean Operations

6.2.1 Comparison

```
Definition eqb (x y:bool) : bool :=
  match x, y with
  | true, true => true
  | false, false => true
```

```
| _, _ ⇒ false  
end.
```

Negation

```
Definition neg (x:bool) : bool :=  
  match x with  
  | true ⇒ false  
  | false ⇒ true  
  end.
```

Conjunction

```
Definition and (x y:bool) : bool :=  
  match x, y with  
  | true, true ⇒ true  
  | _, _ ⇒ false  
  end.
```

Disjunction

```
Definition or (x y:bool) : bool :=  
  match x, y with  
  | false, false ⇒ false  
  | _, _ ⇒ true  
  end.
```

Implication

```
Definition impl (x y:bool) : bool :=  
  or x (neg y).
```

Exclusive or

```
Definition xor (x y:bool) : bool :=  
  neg (eqb x y).
```

Notations

Declare Scope Bool_scope.

Bind Scope Bool_scope with bool.

Open Scope Bool_scope.

Notation "!" x := (neg x)
(at level 35, right associativity) : Bool_scope.

Infix "&&" := and
(at level 40, left associativity) : Bool_scope.

Infix "||" := or
(at level 50, left associativity) : Bool_scope.

Notation "!" x := (neg x)
(at level 35, right associativity) : Bool_scope.

```

Infix "&&" := and
  (at level 40, left associativity) : Bool_scope.
Infix "||" := or
  (at level 50, left associativity) : Bool_scope.

```

6.3 Boolean Decision Procedure : tactic working by exponential case

analysis on all variables of type bool.

6.3.1 Tactic *tautob*

tautob introduces all variables that it can, performs a case analysis on all the boolean variables, then splits all subgoals and attempts resolution using intuition

```

Ltac tautob_post tt :=
  simpls; try split; intros; try discriminate;
  try solve [ intuition auto_false ].
```

```

Ltac tautob_core tt :=
  let rec aux tt :=
    (try intros_all); match goal with
    | b : bool ⊢ _ ⇒ destruct b; clear b; aux tt
    | _ ⇒ tautob_post tt
  end in
aux tt.
```

```
Tactic Notation "tautob" :=
  tautob_core tt.
```

6.4 Properties of boolean operators

6.4.1 Properties of *eqb*

```
Lemma eqb_same : ∀ x,
  eqb x x = true.
```

Proof using. *tautob*. Qed.

```
Lemma eqb_true_l : neutral_l eqb true.
```

Proof using. *tautob*. Qed.

```
Lemma eqb_true_r : neutral_r eqb true.
```

Proof using. *tautob*. Qed.

```
Lemma eqb_false_l : ∀ x,
```

`eqb false x = neg x.`

Proof using. *tautob*. Qed.

Lemma `eqb_false_r : ∀ x,`

`eqb x false = neg x.`

Proof using. *tautob*. Qed.

Lemma `eqb_comm : comm eqb.`

Proof using. *tautob*. Qed.

6.4.2 Properties of and

Lemma `and_same : idempotent2 and.`

Proof using. *tautob*. Qed.

Lemma `and_true_l : neutral_l and true.`

Proof using. *tautob*. Qed.

Lemma `and_true_r : neutral_r and true.`

Proof using. *tautob*. Qed.

Lemma `and_false_l : absorb_l and false.`

Proof using. *tautob*. Qed.

Lemma `and_false_r : absorb_r and false.`

Proof using. *tautob*. Qed.

Lemma `and_comm : comm and.`

Proof using. *tautob*. Qed.

Lemma `and_assoc : assoc and.`

Proof using. *tautob*. Qed.

Lemma `and_or_l : distrib_r and or.`

Proof using. *tautob*. Qed.

Lemma `and_or_r : distrib_l and or.`

Proof using. *tautob*. Qed.

6.4.3 Properties of or

Lemma `or_same : idempotent2 or.`

Proof using. *tautob*. Qed.

Lemma `or_false_l : neutral_l or false.`

Proof using. *tautob*. Qed.

Lemma `or_false_r : neutral_r or false.`

Proof using. *tautob*. Qed.

Lemma `or_true_l : absorb_l or true.`

```

Proof using. tautob. Qed.

Lemma or_true_r : absorb_r or true.
Proof using. tautob. Qed.

Lemma or_comm : comm or.
Proof using. tautob. Qed.

Lemma or_assoc : assoc or.
Proof using. tautob. Qed.

Lemma or_and_l : distrib_r or and.
Proof using. tautob. Qed.

Lemma or_and_r : distrib_l or and.
Proof using. tautob. Qed.

```

6.4.4 Properties of neg

```

Lemma neg_false : ! false = true.
Proof using. auto. Qed.

Lemma neg_true : ! true = false.
Proof using. auto. Qed.

Lemma neg_and : automorphism neg and or.
Proof using. tautob. Qed.

Lemma neg_or : automorphism neg or and.
Proof using. tautob. Qed.

Lemma neg_neg : involutive neg.
Proof using. tautob. Qed.

```

6.4.5 Properties of if then else

```

Section PropertiesIf.

Implicit Types x y z : bool.

Lemma if_true :  $\forall x y,$ 
   $(\text{if } \text{true} \text{ then } x \text{ else } y) = x.$ 
Proof using. auto. Qed.

Lemma if_false :  $\forall x y,$ 
   $(\text{if } \text{false} \text{ then } x \text{ else } y) = y.$ 
Proof using. auto. Qed.

Lemma if_then_else_same :  $\forall x y,$ 
   $(\text{if } x \text{ then } y \text{ else } y) = y.$ 
Proof using. tautob. Qed.

```

```

Lemma if_then_true_else_false : ∀ x,
  (if x then true else false) = x.
Proof using. tautob. Qed.

Lemma if_then_false_else_true : ∀ x,
  (if x then false else true) = !x.
Proof using. tautob. Qed.

Lemma if_then_true : ∀ x y,
  (if x then true else y) = x || y.
Proof using. tautob. Qed.

Lemma if_then_false : ∀ x y,
  (if x then false else y) = (!x) && y.
Proof using. tautob. Qed.

Lemma if_else_false : ∀ x y,
  (if x then y else false) = x && y.
Proof using. tautob. Qed.

Lemma if_else_true : ∀ x y,
  (if x then y else true) = (!x) || y.
Proof using. tautob. Qed.

End PropertiesIf.

```

6.4.6 Properties of `impl` and `xor`

We do not provide lemmas for `impl` and `xor` because these functions can be easily expressed in terms of the other operators.

6.4.7 Opacity

`Opaque eqb neg and or.`

6.5 Tactics

6.5.1 Tactic `rew_neg_neg`

`rew_neg_neg` is a tactic that simplifies all double negations of booleans, i.e. replaces `!!b` with `b`.

`Hint Rewrite neg_neg : rew_neg_neg.`

`Tactic Notation "rew_neg_neg" :=`

`autorewrite with rew_neg_neg.`

`Tactic Notation "rew_neg_neg" "¬¬" :=`

```

rew_neg_neg; auto_tilde.
Tactic Notation "rew_neg_neg" "*" :=
  rew_neg_neg; auto_star.
Tactic Notation "rew_neg_neg" "in" hyp(H) :=
  autorewrite with rew_neg_neg in H.
Tactic Notation "rew_neg_neg" "~~" "in" hyp(H) :=
  rew_neg_neg in H; auto_tilde.
Tactic Notation "rew_neg_neg" "*" "in" hyp(H) :=
  rew_neg_neg in H; auto_star.
Tactic Notation "rew_neg_neg" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_neg_neg).
Tactic Notation "rew_neg_neg" "~~" "in" "*" :=
  rew_neg_neg in *; auto_tilde.
Tactic Notation "rew_neg_neg" "*" "in" "*" :=
  rew_neg_neg in *; auto_star.

```

6.5.2 Tactic *rew_bool*

rew_bool simplifies boolean expressions, using rewriting lemmas in the database *rew_bool* defined below.

```

Hint Rewrite
  eqb_same eqb_true_l eqb_true_r eqb_false_l eqb_false_r
  neg_false neg_true neg_neg neg_and neg_or
  and_true_l and_true_r and_false_l and_false_r
  or_false_l or_false_r or_true_l or_true_r
  if_true if_false if_then_else_same
  if_then_true_else_false if_then_false_else_true
  if_then_true_if_else_false
  if_then_false_if_else_true
  : rew_bool.

Tactic Notation "rew_bool" :=
  autorewrite with rew_bool.
Tactic Notation "rew_bool" "~~" :=
  rew_bool; auto_tilde.
Tactic Notation "rew_bool" "*" :=
  rew_bool; auto_star.
Tactic Notation "rew_bool" "in" hyp(H) :=
  autorewrite with rew_bool in H.
Tactic Notation "rew_bool" "~~" "in" hyp(H) :=
  rew_bool in H; auto_tilde.
Tactic Notation "rew_bool" "*" "in" hyp(H) :=
  rew_bool in H; auto_star.

```

```
Tactic Notation "rew_bool" "in" "*" :=  
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_bool).  
Tactic Notation "rew_bool" "~" "in" "*" :=  
  rew_bool in *; auto_tilde.  
Tactic Notation "rew_bool" "*" "in" "*" :=  
  rew_bool in *; auto_star.
```

Chapter 7

Library `SLF.LibReflect`

```
Set Implicit Arguments.
```

```
From SLF Require Import LibTactics.
```

```
From SLF Require Export LibBool LibLogic.
```

```
Implicit Type P : Prop.
```

```
Implicit Type b : bool.
```

7.1 Reflection between booleans and propositions

- `istrue b` produces a proposition that is `True` if and only if the boolean `b` is equal to `true`.
- `isTrue P` produces a boolean expression that is `true` if and only if the proposition `P` is equal to `True`.

7.1.1 Translation from booleans into propositions

Any boolean `b` can be viewed as a proposition through the relation `b = true`.

```
Coercion istrue (b : bool) : Prop := (b = true).
```

Specification

```
Lemma istrue_eq_eq_true : ∀ b,  
  istrue b = (b = true).
```

```
Proof using. reflexivity. Qed.
```

```
Lemma istrue_true_eq :  
  istrue true = True.
```

```
Proof using. rewrite istrue_eq_eq_true. extens×. Qed.
```

```
Lemma istrue_false_eq :
```

`istrue false = False.`

`Proof using. rewrite istrue_eq_eq_true. extens. iff; auto_false. Qed.`

`Global Opaque istrue.`

Proving the goals `true` and $\neg \text{false}$

`Lemma istrue_true : istrue true. Proof using. reflexivity. Qed.`

`Lemma not_istrue_false : $\neg (\text{istrue false})$. Proof using. rewrite istrue_false_eq. intuition. Qed.`

Equivalence of `false` and `False`

`Lemma false_of_False :`

`False \rightarrow
false.`

`Proof using. intros K. false. Qed.`

`Lemma False_of_false :`

`false \rightarrow
False.`

`Proof using. intros K. rewrite \neg istrue_false_eq in K. Qed.`

Hints for proving `false` and `False`

`Hint Resolve istrue_true not_istrue_false.`

`Hint Extern 1 (istrue false) \Rightarrow
apply false_of_False.`

`Hint Extern 1 (False) \Rightarrow match goal with
| H : istrue false $\vdash \dots$ \Rightarrow apply (not_istrue_false H) end.`

7.1.2 Translation from propositions into booleans

The expression `isTrue P` evaluates to `true` if and only if the proposition P is `True`.

`Definition isTrue (P : Prop) : bool :=`

`If P then true else false.`

Specification

`Lemma isTrue_eq_if : $\forall P,$
isTrue $P = \text{If } P \text{ then true else false}.$`

`Proof using. reflexivity. Qed.`

`Lemma isTrue_True :`

`isTrue True = true.`

`Proof using. unfolds. case_if; auto_false \neg . Qed.`

`Lemma isTrue_False :`

`isTrue False = false.`

`Proof using. unfolds. case_if; auto_false \neg . Qed.`

```
Global Opaque isTrue.
```

Lemmas

```
Lemma isTrue_eq_true : ∀ P,  
  P →  
  isTrue P = true.
```

Proof using. intros. rewrite isTrue_eq_if. case_if×. Qed.

```
Lemma isTrue_eq_false : ∀ P,  
  ¬ P →  
  isTrue P = false.
```

Proof using. intros. rewrite isTrue_eq_if. case_if×. Qed.

7.1.3 Extensionality for boolean equality, stated using `istrue`

```
Lemma bool_ext : ∀ b1 b2,  
  (b1 ↔ b2) →  
  b1 = b2.
```

Proof using.

```
destruct b1; destruct b2; intros; auto_false.  
destruct H. false H; auto.  
destruct H. false H0; auto.
```

Qed.

```
Lemma bool_ext_eq : ∀ b1 b2,  
  (b1 = b2) = (b1 ↔ b2).
```

Proof using.

```
intros. extens. iff M. { subst×. } { applys× bool_ext. }
```

Qed.

Instance Extensionality_bool : Extensionality **bool**.

Proof using. apply (Extensionality_make bool_ext). Defined.

7.1.4 Specification of boolean equality

```
Definition is_beq A (beq:A→A→bool) :=  
  ∀ x y, beq x y = isTrue (x = y).
```

7.2 Rewriting rules

7.2.1 Rewriting rules for distributing `istrue`

```
Lemma istrue_isTrue_eq : ∀ P,  
  istrue (isTrue P) = P.
```

Proof using. *extens.* *rewrite* *isTrue_eq_if.* *case_if;* *auto_false* \times . Qed.

Lemma *istrue_neg_eq* : $\forall b,$
 $\text{istrue}(\neg b) = \neg(\text{istrue } b).$

Proof using. *extens.* *tautob.* Qed.

Lemma *istrue_and_eq* : $\forall b1 b2,$
 $\text{istrue}(b1 \& b2) = (\text{istrue } b1 \wedge \text{istrue } b2).$

Proof using. *extens.* *tautob.* Qed.

Lemma *istrue_or_eq* : $\forall b1 b2,$
 $\text{istrue}(b1 \vee b2) = (\text{istrue } b1 \vee \text{istrue } b2).$

Proof using. *extens.* *tautob.* Qed.

Corollary

Lemma *istrue_neg_isTrue* : $\forall P,$
 $\text{istrue}(\neg P) = \neg P.$

Proof using. *intros.* *rewrite* *istrue_neg_eq.* *rewrite* \neg *istrue_isTrue_eq.* Qed.

istrue and conditionals

Lemma *If_istrue* : $\forall b A (x y : A),$
 $(\text{If } \text{istrue } b \text{ then } x \text{ else } y)$
 $= (\text{if } b \text{ then } x \text{ else } y).$

Proof using. *intros.* *case_if* as *C*; *case_if* as *D*; *auto.* Qed.

Lemma *istrue_if_eq* : $\forall P b1 b2,$
 $\text{istrue}(\text{If } P \text{ then } b1 \text{ else } b2)$
 $= (\text{If } P \text{ then } \text{istrue } b1 \text{ else } \text{istrue } b2).$

Proof using. *extens.* *case_if* \times . Qed.

Lemma *istrue_if_eq* : $\forall b1 b2 b3,$
 $\text{istrue}(\text{if } b1 \text{ then } b2 \text{ else } b3)$
 $= (\text{If } \text{istrue } b1 \text{ then } \text{istrue } b2 \text{ else } \text{istrue } b3).$

Proof using. *intros.* do 2 *case_if*; *auto.* Qed.

7.2.2 Rewriting rules for distributing *isTrue*

Lemma *isTrue_istrue* : $\forall b,$
 $\text{isTrue}(\text{istrue } b) = b.$

Proof using. *extens.* *rewrite* \times *istrue_isTrue_eq.* Qed.

Lemma *isTrue_not* : $\forall P,$
 $\text{isTrue}(\neg P) = \neg \text{isTrue } P.$

Proof using. *extens.* do 2 *rewrite* *isTrue_eq_if.* do 2 *case_if;* *auto_false* \times . Qed.

Lemma *isTrue_and* : $\forall P1 P2,$
 $\text{isTrue}(P1 \wedge P2) = (\text{isTrue } P1 \wedge \text{isTrue } P2).$

Proof using. *extens.* do 3 *rewrite* *isTrue_eq_if.* do 3 *case_if;* *auto_false* \times . Qed.

```

Lemma isTrue_or : ∀ P1 P2,
  isTrue (P1 ∨ P2) = (isTrue P1 || isTrue P2).
Proof using. extens. do 3 rewrite isTrue_eq_if. do 3 case_if; auto_false×. Qed.

```

Corollary

```

Lemma isTrue_not_istrue : ∀ b,
  isTrue (¬ istrue b) = !b.

```

```
Proof using. intros. rewrite isTrue_not. rewrite¬ isTrue_istrue. Qed.
```

Simplification of equalities involving isTrue

Section IsTrueEqualities.

```

Ltac prove_isTrue_lemma :=
  intros; try extens; try iff; rewrite isTrue_eq_if in *; case_if; auto_false×.

```

```

Lemma true_eq_isTrue_eq : ∀ P,
  (true = isTrue P) = P.

```

```
Proof using. prove_isTrue_lemma. Qed.
```

```

Lemma isTrue_eq_true_eq : ∀ P,
  (isTrue P = true) = P.

```

```
Proof using. prove_isTrue_lemma. Qed.
```

```

Lemma false_eq_isTrue_eq : ∀ P,
  (false = isTrue P) = ¬ P.

```

```
Proof using. prove_isTrue_lemma. Qed.
```

```

Lemma isTrue_eq_false_eq : ∀ P,
  (isTrue P = false) = ¬ P.

```

```
Proof using. prove_isTrue_lemma. Qed.
```

```

Lemma isTrue_eq_isTrue_eq : ∀ P1 P2,
  (isTrue P1 = isTrue P2) = (P1 ↔ P2).

```

Proof using.

```

  intros. extens. iff; repeat rewrite isTrue_eq_if in *;
  repeat case_if; auto_false×.

```

Qed.

End IsTrueEqualities.

isTrue and conditionals

```

Lemma if_isTrue : ∀ P A (x y : A),
  (if isTrue P then x else y)
  = (If P then x else y).

```

Proof using.

```

  intros. case_if as C; case_if as D; auto.
  { rewrite× isTrue_eq_true_eq in C. }
  { rewrite× isTrue_eq_false_eq in C. }

```

Qed.

```

Lemma isTrue_if : ∀ P1 P2 P3,
  isTrue (If P1 then P2 else P3)
  = If P1 then isTrue P2 else isTrue P3.
Proof using. extens. case_if ×. Qed.

```

```

Lemma isTrue_if_eq_if_isTrue : ∀ P1 P2 P3,
  isTrue (If P1 then P2 else P3)
  = (if isTrue P1 then isTrue P2 else isTrue P3).
Proof using. intros. rewrite if_isTrue. rewrite ← isTrue_if. Qed.

```

7.2.3 Lemmas for testing booleans

```

Lemma bool_inv_or : ∀ b,
  b ∨ !b.

```

Proof using. tautob. Qed.

```

Lemma bool_inv_or_eq : ∀ b,
  b = true ∨ b = false.

```

Proof using. tautob. Qed.

```

Lemma xor_inv_or : ∀ b1 b2,
  xor b1 b2 →
    (b1 = true ∧ b2 = false)
  ∨ (b1 = false ∧ b2 = true).

```

Proof using. tautob; auto_false ×. Qed.

Arguments xor_inv_or [b1] [b2].

7.2.4 Lemmas for normalizing **b = true** and **b = false** terms

```

Lemma bool_eq_true_eq : ∀ b,
  (b = true) = istrue b.

```

Proof using. extens. tautob. Qed.

```

Lemma bool_eq_false_eq : ∀ b,
  (b = false) = istrue (!b).

```

Proof using. extens. tautob. Qed.

```

Lemma true_eq_bool_eq : ∀ b,
  (true = b) = istrue b.

```

Proof using. extens. tautob. Qed.

```

Lemma false_eq_bool_eq : ∀ b,
  (false = b) = istrue (!b).

```

Proof using. extens. tautob. Qed.

7.3 Tactics

7.3.1 Tactics *rew_istrue* to distribute **istrue**

rew_istrue distributes **istrue**. It is useful to replace all boolean operators with corresponding logical operators.

```
Hint Rewrite istrue_true_eq istrue_false_eq istrue_isTrue_eq  
      istrue_neg_eq istrue_and_eq istrue_or_eq  
      If_istrue istrue_If_eq istrue_if_eq: rew_istrue.  
  
Tactic Notation "rew_istrue" :=  
  autorewrite with rew_istrue.  
Tactic Notation "rew_istrue" "in" hyp(H) :=  
  autorewrite with rew_istrue in H.  
Tactic Notation "rew_istrue" "in" "*" :=  
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_istrue).  
Tactic Notation "rew_istrue" "~~" :=  
  rew_istrue; auto_tilde.  
Tactic Notation "rew_istrue" "~~" "in" hyp(H) :=  
  rew_istrue in H; auto_tilde.  
Tactic Notation "rew_istrue" "~~" "in" "*" :=  
  rew_istrue in *; auto_tilde.  
Tactic Notation "rew_istrue" "*" :=  
  rew_istrue; auto_star.  
Tactic Notation "rew_istrue" "*" "in" hyp(H) :=  
  rew_istrue in H; auto_star.  
Tactic Notation "rew_istrue" "*" "in" "*" :=  
  rew_istrue in *; auto_star.
```

7.3.2 Tactics *rew_isTrue* to distribute **isTrue**

rew_isTrue distributes **isTrue**. This tactic is probably much less useful than *rew_istrue*, since logical operators are often simpler to work with.

```
Hint Rewrite isTrue_True isTrue_False isTrue_istrue  
      isTrue_not isTrue_and isTrue_or  
      if_isTrue isTrue_If : rew_isTrue.  
  
Tactic Notation "rew_isTrue" :=  
  autorewrite with rew_isTrue.  
Tactic Notation "rew_isTrue" "in" hyp(H) :=  
  autorewrite with rew_isTrue in H.  
Tactic Notation "rew_isTrue" "in" "*" :=  
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_isTrue).
```

```

Tactic Notation "rew_isTrue" "~~" :=
  rew_isTrue; auto_tilde.
Tactic Notation "rew_isTrue" "~~" "in" hyp(H) :=
  rew_isTrue in H; auto_tilde.
Tactic Notation "rew_isTrue" "~~" "*" "in" "*" :=
  rew_isTrue in *; auto_tilde.
Tactic Notation "rew_isTrue" "*" "in" "*" :=
  rew_isTrue; auto_star.
Tactic Notation "rew_isTrue" "*" "in" "*" "in" hyp(H) :=
  rew_isTrue in H; auto_star.
Tactic Notation "rew_isTrue" "*" "*" "in" "*" "in" "*" :=
  rew_isTrue in *; auto_star.

```

7.3.3 Tactics useful for program verification, when reasoning about

the result of if-statements over boolean expressions, i.e. an expression of the form $b = ..$ or $.. = b$, which produces hypotheses of the form `true = ..` and `false = ..` or symmetric. It is used as post-treatment for tactic `case_if`.

```

Hint Rewrite
  true_eq_isTrue_eq isTrue_eq_true_eq
  false_eq_isTrue_eq isTrue_eq_false_eq
  isTrue_eq_isTrue_eq
  not_not_eq
  istrue_true_eq istrue_false_eq istrue_isTrue_eq
  istrue_neg_eq istrue_and_eq istrue_or_eq
  bool_eq_true_eq bool_eq_false_eq true_eq_bool_eq false_eq_bool_eq
  : rew_bool_eq.

Tactic Notation "rew_bool_eq" :=
  autorewrite with rew_bool_eq.
Tactic Notation "rew_bool_eq" "~~" :=
  rew_bool_eq; auto_tilde.
Tactic Notation "rew_bool_eq" "*" :=
  rew_bool_eq; auto_star.
Tactic Notation "rew_bool_eq" "in" hyp(H) :=
  autorewrite with rew_bool_eq in H.
Tactic Notation "rew_bool_eq" "~~" "in" hyp(H) :=
  rew_bool_eq in H; auto_tilde.
Tactic Notation "rew_bool_eq" "*" "*" "in" hyp(H) :=
  rew_bool_eq in H; auto_star.
Tactic Notation "rew_bool_eq" "in" "*" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_bool_eq).

```

```

Tactic Notation "rew_bool_eq" "~~" "in" "*" :=
  rew_bool_eq; auto_tilde.
Tactic Notation "rew_bool_eq" "*" "in" "*" :=
  rew_bool_eq; auto_star.

```

7.3.4 Tactics extended for reflection

Extension of the tactic *case_if* to automatically performs simplification using *logics*.

For less aggressive introduction of *istrue*, consider rewriting without the lemmas: `bool_eq_true_eq`
`bool_eq_false_eq true_eq_bool_eq false_eq_bool_eq`

```

Ltac case_if_post H ::= 
  rew_bool_eq in H; tryfalse.

```

Extension of the tactic *test_dispatch* from LibLogic.v, so as to be able to call the tactic *tests* directly on boolean expressions

```

Ltac tests_bool_base E H1 H2 ::= 
  tests_prop_base (istrue E) H1 H2.

```

```

Ltac tests_dispatch E H1 H2 ::= 
  match type of E with
  | bool => tests_bool_base E H1 H2
  | Prop => tests_prop_base E H1 H2
  | {_}+{_} => tests_ssum_base E H1 H2
  end.

```

Extension of the tactic *apply_to_head_of* (see LibTactics).

```

Ltac apply_to_head_of E cont ::= 
  let go E := let P := get_head E in cont P in
  match E with
  | istrue ?A => go A
  | istrue (neg ?A) => go A
  | ?A = ?B => first [ go A | go B ]
  | ?A => go A
  end.

```

Chapter 8

Library SLF.LibProd

```
Set Implicit Arguments.
```

```
From SLF Require Import LibTactics LibLogic LibReflect.
```

```
Generalizable Variables A B.
```

8.1 Product type

8.1.1 Definition

From the Prelude:

```
Inductive prod A B : Type := | pair : A -> B -> prod A B.
```

```
Hint Constructors prod : core.
```

```
Add Printing Let prod. Notation "x * y" := (prod x y) : type_scope. Notation "( x , y , ... , z )" := (pair .. (pair x y) .. z) : core_scope.
```

```
Definition fst A B (p:A*B) : A := match p with (x,y) => x end.
```

```
Definition snd A B (p:A*B) : B := match p with (x,y) => y end.
```

```
Remark: to follow conventions pair should be renamed to prod_intro.
```

8.1.2 Inhabited

```
Global Instance Inhab_prod : ∀ {Inhab A, Inhab B}, Inhab (A × B).  
Proof using intros. apply (Inhab_of_val (arbitrary, arbitrary)). Qed.
```

8.2 Structure

Decomposition as projection

```
Lemma prod2_eq_tuple_proj : ∀ A1 A2 (x:A1×A2),  
x = (fst x, snd x).
```

Proof using. intros. destruct $\neg x$. Qed.

Structural equality

Section Properties.

Variables ($A_1 A_2 A_3 A_4 : \text{Type}$).

Lemma eq_prod2 : $\forall (x_1 y_1:A_1) (x_2 y_2:A_2),$

$x_1 = y_1 \rightarrow$

$x_2 = y_2 \rightarrow$

$(x_1, x_2) = (y_1, y_2)$.

Proof using. intros. subst \neg . Qed.

Lemma eq_prod3 : $\forall (x_1 y_1:A_1) (x_2 y_2:A_2) (x_3 y_3:A_3),$

$x_1 = y_1 \rightarrow$

$x_2 = y_2 \rightarrow$

$x_3 = y_3 \rightarrow$

$(x_1, x_2, x_3) = (y_1, y_2, y_3)$.

Proof using. intros. subst \neg . Qed.

Lemma eq_prod4 : $\forall (x_1 y_1:A_1) (x_2 y_2:A_2) (x_3 y_3:A_3) (x_4 y_4:A_4),$

$x_1 = y_1 \rightarrow$

$x_2 = y_2 \rightarrow$

$x_3 = y_3 \rightarrow$

$x_4 = y_4 \rightarrow$

$(x_1, x_2, x_3, x_4) = (y_1, y_2, y_3, y_4)$.

Proof using. intros. subst \neg . Qed.

End Properties.

Hint Immediate eq_prod2 eq_prod3 eq_prod4.

8.3 Operations

8.3.1 Definition of projections

`fst` and `snd` are defined in the Prelude

Definition `fst` $A B$ ($p:A*B$) : $A := \text{match } p \text{ with } (x,y) \Rightarrow x \text{ end.}$

Definition `snd` $A B$ ($p:A*B$) : $B := \text{match } p \text{ with } (x,y) \Rightarrow y \text{ end.}$

Arguments `fst` { A } { B }.

Arguments `snd` { A } { B }.

8.3.2 Notation for projections

N-ary projections are defined as notations and not as definitions, which has appeared to be more flexible with respect to type inference. TODO: investigate the possibility of using definitions.

```

Notation "'proj1' P := P (at level 69, only parsing).
Notation "'proj2' P := (proj1 P) (at level 69, only parsing).
Notation "'proj22' P := (proj2 P) (at level 69, only parsing).
Notation "'proj31' P := (proj1 P) (at level 69).
Notation "'proj32' P := (proj1 (proj2 P)) (at level 69).
Notation "'proj33' P := (proj2 (proj2 P)) (at level 69).
Notation "'proj41' P := (proj1 P) (at level 69).
Notation "'proj42' P := (proj1 (proj2 P)) (at level 69).
Notation "'proj43' P := (proj1 (proj2 (proj2 P))) (at level 69).
Notation "'proj44' P := (proj2 (proj2 (proj2 P))) (at level 69).
Notation "'proj51' P := (proj1 P) (at level 69).
Notation "'proj52' P := (proj1 (proj2 P)) (at level 69).
Notation "'proj53' P := (proj1 (proj2 (proj2 P))) (at level 69).
Notation "'proj54' P := (proj1 (proj2 (proj2 (proj2 P)))) (at level 69).
Notation "'proj55' P := (proj2 (proj2 (proj2 (proj2 P)))) (at level 69).

```

8.3.3 Currying

```

Section Currying.
Variables (A1 A2 A3 A4 A5 B : Type).
Definition curry1 f : A1 → B :=
  f.
Definition curry2 f : A1 → A2 → B :=
  fun x1 x2 ⇒ f (x1 , x2).
Definition curry3 f : A1 → A2 → A3 → B :=
  fun x1 x2 x3 ⇒ f (x1 , x2 , x3).
Definition curry4 f : A1 → A2 → A3 → A4 → B :=
  fun x1 x2 x3 x4 ⇒ f (x1 , x2 , x3 , x4).
Definition curry5 f : A1 → A2 → A3 → A4 → A5 → B :=
  fun x1 x2 x3 x4 x5 ⇒ f (x1 , x2 , x3 , x4 , x5).
End Currying.

```

8.3.4 Uncurrying

```

Section Uncurrying.
Variables (A1 A2 A3 A4 A5 B : Type).
Definition uncurry1 f : A1 → B :=
  f.
Definition uncurry2 f : A1 × A2 → B :=
  fun p ⇒ match p with (x1 , x2) ⇒
    f x1 x2 end.

```

```

Definition uncurry3 f : A1 × A2 × A3 → B :=
  fun p ⇒ match p with (x1, x2, x3) ⇒
    f x1 x2 x3 end.

Definition uncurry4 f : A1 × A2 × A3 × A4 → B :=
  fun p ⇒ match p with (x1, x2, x3, x4) ⇒
    f x1 x2 x3 x4 end.

Definition uncurry5 f : A1 × A2 × A3 × A4 × A5 → B :=
  fun p ⇒ match p with (x1, x2, x3, x4, x5) ⇒
    f x1 x2 x3 x4 x5 end.

End Uncurrying.

```

8.3.5 Uncurrying for relations

uncurrypN turns a function of type $A_1 \rightarrow A_1 \rightarrow \dots \rightarrow A_N \rightarrow A_N \rightarrow B$ into a function of type $(A_1^{*..*} A_N) \rightarrow (A_1^{*..*} A_N) \rightarrow B$.

Section Uncurryp.

Variables (A1 A2 A3 A4 B : Type).

```

Definition uncurryp1 f : A1 → A1 → B :=
  f.

```

```

Definition uncurryp2 f : A1 × A2 → A1 × A2 → B :=
  fun p1 p2 ⇒ match p1, p2 with (x1, x2), (y1, y2) ⇒
    f x1 y1 x2 y2 end.

```

```

Definition uncurryp3 f : A1 × A2 × A3 → A1 × A2 × A3 → B :=
  fun p1 p2 ⇒ match p1, p2 with (x1, x2, x3), (y1, y2, y3) ⇒
    f x1 x2 x3 y1 y2 y3 end.

```

```

Definition uncurryp4 f : A1 × A2 × A3 × A4 → A1 × A2 × A3 × A4 → B :=
  fun p1 p2 ⇒ match p1, p2 with (x1, x2, x3, x4), (y1, y2, y3, y4) ⇒
    f x1 x2 x3 x4 y1 y2 y3 y4 end.

```

End Uncurryp.

Unfolding

```

Tactic Notation "unfold_uncurryp" :=
  unfold uncurryp1, uncurryp2, uncurryp3, uncurryp4.

```

```

Tactic Notation "unfolds_uncurryp" :=
  unfold uncurryp1, uncurryp2, uncurryp3, uncurryp4 in *.

```

8.3.6 Inverse projections for relations

unprojNK turns a function of type $AK \rightarrow AK \rightarrow B$ into a function of type $(A_1^{*..*} A_N) \rightarrow (A_1^{*..*} A_N) \rightarrow B$.

Section Unproj.

Variables (A1 A2 A3 A4 A5 B : Type).

```

Definition unproj21 f : A1×A2 → A1×A2 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ),(y1 ,y2 ) ⇒
    f x1 y1 end.

Definition unproj22 f : A1×A2 → A1×A2 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ),(y1 ,y2 ) ⇒
    f x2 y2 end.

Definition unproj31 f : A1×A2×A3 → A1×A2×A3 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ),(y1 ,y2 ,y3 ) ⇒
    f x1 y1 end.

Definition unproj32 f : A1×A2×A3 → A1×A2×A3 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ),(y1 ,y2 ,y3 ) ⇒
    f x2 y2 end.

Definition unproj33 f : A1×A2×A3 → A1×A2×A3 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ),(y1 ,y2 ,y3 ) ⇒
    f x3 y3 end.

Definition unproj41 f : A1×A2×A3×A4 → A1×A2×A3×A4 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ),(y1 ,y2 ,y3 ,y4 ) ⇒
    f x1 y1 end.

Definition unproj42 f : A1×A2×A3×A4 → A1×A2×A3×A4 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ),(y1 ,y2 ,y3 ,y4 ) ⇒
    f x2 y2 end.

Definition unproj43 f : A1×A2×A3×A4 → A1×A2×A3×A4 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ),(y1 ,y2 ,y3 ,y4 ) ⇒
    f x3 y3 end.

Definition unproj44 f : A1×A2×A3×A4 → A1×A2×A3×A4 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ),(y1 ,y2 ,y3 ,y4 ) ⇒
    f x4 y4 end.

Definition unproj51 f : A1×A2×A3×A4×A5 → A1×A2×A3×A4×A5 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ,x5 ),(y1 ,y2 ,y3 ,y4 ,y5 ) ⇒
    f x1 y1 end.

Definition unproj52 f : A1×A2×A3×A4×A5 → A1×A2×A3×A4×A5 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ,x5 ),(y1 ,y2 ,y3 ,y4 ,y5 ) ⇒
    f x2 y2 end.

Definition unproj53 f : A1×A2×A3×A4×A5 → A1×A2×A3×A4×A5 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ,x5 ),(y1 ,y2 ,y3 ,y4 ,y5 ) ⇒
    f x3 y3 end.

Definition unproj54 f : A1×A2×A3×A4×A5 → A1×A2×A3×A4×A5 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ,x5 ),(y1 ,y2 ,y3 ,y4 ,y5 ) ⇒
    f x4 y4 end.

Definition unproj55 f : A1×A2×A3×A4×A5 → A1×A2×A3×A4×A5 → B :=
  fun p1 p2 ⇒ match p1,p2 with (x1 ,x2 ,x3 ,x4 ,x5 ),(y1 ,y2 ,y3 ,y4 ,y5 ) ⇒
    f x5 y5 end.

```

End Unproj.

Arguments unproj21 [A1] A2 [B].
Arguments unproj22 A1 [A2] [B].
Arguments unproj31 [A1] A2 A3 [B].
Arguments unproj32 A1 [A2] A3 [B].
Arguments unproj33 A1 A2 [A3] [B].
Arguments unproj41 [A1] A2 A3 A4 [B].
Arguments unproj42 A1 [A2] A3 A4 [B].
Arguments unproj43 A1 A2 [A3] A4 [B].
Arguments unproj44 A1 A2 A3 [A4] [B].
Arguments unproj51 [A1] A2 A3 A4 A5 [B].
Arguments unproj52 A1 [A2] A3 A4 A5 [B].
Arguments unproj53 A1 A2 [A3] A4 A5 [B].
Arguments unproj54 A1 A2 A3 [A4] A5 [B].
Arguments unproj55 A1 A2 A3 A4 [A5] [B].

Unfolding

Tactic Notation "unfold_unproj" :=
unfold unproj21, unproj22, unproj31, unproj32, unproj33,
unproj41, unproj42, unproj43, unproj44,
unproj51.

Tactic Notation "unfolds_unproj" :=
unfold unproj21, unproj22, unproj31, unproj32, unproj33,
unproj41, unproj42, unproj43, unproj44,
unproj51 in *.

Chapter 9

Library **SLF.LibSum**

```
Set Implicit Arguments.  
From SLF Require Import LibTactics LibLogic LibBool.  
Generalizable Variables A B.
```

9.1 Sum type

9.1.1 Definition

From the Prelude:

Inductive sum A B : Type := | inl : A -> sum A B | inr : B -> sum A B.

Hint Constructors sum : core. Notation “x + y” := (sum x y) : type_scope.

Remark: ideally, constructors would be renamed to *sum_l* and *sum_r*; to follow conventions.

Arguments **inl** {A} {B}.

Arguments **inr** {A} {B}.

9.1.2 Inhabited

Instance **sum_inhab_l** : $\forall \{Inhab\ A\} B, \mathbf{Inhab}\ (A + B)$.

Proof using. intros. apply (**Inhab_of_val** (**inl** arbitrary)). Qed.

Instance **sum_inhab_r** : $\forall \{Inhab\ B\} A, \mathbf{Inhab}\ (A + B)$.

Proof using. intros. apply (**Inhab_of_val** (**inr** arbitrary)). Qed.

Definition **Inhab_sum** : $\forall \{Inhab\ A, Inhab\ B\}, \mathbf{Inhab}\ (A + B)$.

Proof using. *typeclass*. Qed.

9.2 Operations

9.2.1 Testing the branch of the sum

```
Definition is_inl (A B : Type) (x : A + B) : bool :=  
  match x with  
  | inl _ => true  
  | inr _ => false  
  end.
```

```
Definition is_inr (A B : Type) (x : A + B) : bool :=  
  match x with  
  | inl _ => false  
  | inr _ => true  
  end.
```

Section lsln.

Variables (A B : Type).

Implicit Type x : A + B.

```
Lemma is_inl_neg_is_inr : ∀ x,  
  is_inl x = ! (is_inr x).
```

Proof using. intros x. destruct x. Qed.

```
Lemma is_inr_neg_is_inl : ∀ x,  
  is_inr x = ! (is_inl x).
```

Proof using. intros x. destruct x. Qed.

End lsln.

9.2.2 Stripping of the branch tag

Section Get.

Context ‘{IA1:**Inhab** A1} ‘{IA2:**Inhab** A2}.

Implicit Types x : A1+A2.

```
Definition get21 x :=  
  match x with  
  | inl x1 => x1  
  | inr x2 => arbitrary  
  end.
```

```
Definition get22 x :=  
  match x with  
  | inl x1 => arbitrary  
  | inr x2 => x2  
  end.
```

End Get.

9.2.3 Lifting functions over sum types

Section Fget.

Context {A1:Type} {A2:Type} ‘{IB1:**Inhab** B1} ‘{IB2:**Inhab** B2}.

Implicit Types f : A1+A2 → B1+B2.

Definition fun_get21 f :=
 fun x ⇒ get21 (f (**inl** x)).

Definition fun_get22 f :=
 fun x ⇒ get22 (f (**inr** x)).

End Fget.

Chapter 10

Library `SLF.LibRelation`

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibBool LibLogic LibProd LibSum.

From *SLF* Require Export LibOperation.

10.1 Type of binary relations

10.1.1 Type of endorelation, i.e. homogeneous binary relations

Definition `binary` ($A : \text{Type}$) := $A \rightarrow A \rightarrow \text{Prop}$.

10.1.2 Inhabited

Instance `Inhab_binary` : $\forall A, \text{Inhab}(\text{binary } A)$.

Proof using. intros. apply (`Inhab_of_val` (fun _ _ \Rightarrow `True`)). Qed.

10.1.3 Extensionality

Lemma `binary_ext` : $\forall A (R1 R2:\text{binary } A),$

$(\forall x y, R1 x y \leftrightarrow R2 x y) \rightarrow$

$R1 = R2$.

Proof using. extens \times . Qed.

Instance `Extensionality_binary` : $\forall A,$

Extensionality ($\text{binary } A$).

Proof using. intros. apply (`Extensionality_make` (@`binary_ext` A)). Defined.

10.2 Properties of relations

10.2.1 Reflexivity

Definition refl A (R :binary A) :=

$$\forall x, R x x.$$

Section Refl.

Variables (A : Type).

Implicit Types R : binary A .

Lemma refl_inv : $\forall x y R,$

$$\text{refl } R \rightarrow$$

$$x = y \rightarrow$$

$$R x y.$$

Proof using. intros_all. subst \neg . Qed.

End Refl.

10.2.2 Irreflexivity

Definition irrefl A (R :binary A) :=

$$\forall x, \neg (R x x).$$

Section Irrefl.

Variables (A : Type).

Implicit Types R : binary A .

Lemma irrefl_inv : $\forall x R,$

$$\text{irrefl } R \rightarrow$$

$$R x x \rightarrow$$

False.

Proof using. introv H P. apply \times H. Qed.

Lemma irrefl_eq_forall_neq : $\forall R,$

$$\text{irrefl } R = (\forall x y, R x y \rightarrow x \neq y).$$

Proof using.

unfold irrefl. extens. iff M.

{ introv H E. subst \times . }

{ autos \times . }

Qed.

Lemma irrefl_inv_neq : $\forall x y R,$

$$\text{irrefl } R \rightarrow$$

$$R x y \rightarrow$$

$$x \neq y.$$

Proof using. introv H M. rewrite \times irrefl_eq_forall_neq in H. Qed.

```
End Irrefl.
```

10.2.3 Symmetry

```
Definition sym A (R:binary A) :=
```

```
  ∀ x y, R x y → R y x.
```

```
Section Sym.
```

```
Variables (A : Type).
```

```
Implicit Types R : binary A.
```

```
Lemma sym_inv : ∀ x y R,
```

```
  sym R →
```

```
  R x y →
```

```
  R y x.
```

```
Proof using. introv Sy R1. apply× Sy. Qed.
```

```
Lemma sym_inv_eq : ∀ x y R,
```

```
  sym R →
```

```
  R x y = R y x.
```

```
Proof using. unfold sym. extens×. Qed.
```

```
Lemma sym_eq_forall_eq : ∀ R,
```

```
  sym R = (forall x y, R x y = R y x).
```

```
Proof using.
```

```
  unfold sym. extens. iff M.
```

```
  { extens×. }
```

```
  { intros. rewrite× M. }
```

```
Qed.
```

```
End Sym.
```

10.2.4 Asymmetry

```
Definition asym A (R:binary A) :=
```

```
  ∀ x y, R x y → ¬ R y x.
```

```
Section Asym.
```

```
Variables (A : Type).
```

```
Implicit Types R : binary A.
```

```
Lemma asym_eq_forall_false : ∀ R,
```

```
  asym R = (forall x y, R x y → R y x → False).
```

```
Proof using. unfold asym. extens×. Qed.
```

```
Lemma asym_inv : ∀ x y R,
```

```
  asym R →
```

```
  R x y →
```

$R y x \rightarrow$

False.

Proof using. *introv H M1 M2. apply× H. Qed.*

End Asym.

10.2.5 Antisymmetry

Definition antisym A (R:binary A) :=

$\forall x y, R x y \rightarrow R y x \rightarrow x = y.$

Section Antisym.

Variables (A : Type).

Implicit Types R : binary A.

Lemma antisym_eq_forall_eq : $\forall R,$

$\text{antisym } R = (\forall x y, R x y \rightarrow R y x \rightarrow x = y).$

Proof using. unfold antisym. *extens×. Qed.*

Lemma antisym_inv : $\forall x y R,$

$\text{antisym } R \rightarrow$

$R x y \rightarrow$

$R y x \rightarrow$

$x \neq y \rightarrow$

False.

Proof using. *intros_all×. Qed.*

End Antisym.

10.2.6 Antisymmetry with respect to an equivalence relation

Definition antisym_wrt A (E:binary A) R :=

$\forall x y, R x y \rightarrow R y x \rightarrow E x y.$

10.2.7 Transitivity

Definition trans A (R:binary A) :=

$\forall y x z, R x y \rightarrow R y z \rightarrow R x z.$

Section Trans.

Variables (A : Type).

Implicit Types R : binary A.

Lemma trans_eq_forall_impl : $\forall R,$

$\text{trans } R = (\forall y x z, R x y \rightarrow R y z \rightarrow R x z).$

Proof using. unfold trans. *extens×. Qed.*

```

Lemma trans_inv : ∀ y x z R,
  trans R →
  R x y →
  R y z →
  R x z.

```

Proof using. introv Tr R1 R2. apply× Tr. Qed.

```

Lemma trans_inv_swap : ∀ y x z R,
  trans R →
  R y z →
  R x y →
  R x z.

```

Proof using. introv Tr R1 R2. apply× Tr. Qed.

trans + sym

Definition trans_sym_ll := trans_inv.

```

Lemma trans_sym_lr : ∀ y x z R,
  trans R →
  sym R →
  R x y →
  R z y →
  R x z.

```

Proof using. introv Tr Sy R1 R2. apply× Tr. Qed.

```

Lemma trans_sym_rr : ∀ y x z R,
  trans R →
  sym R →
  R y x →
  R z y →
  R x z.

```

Proof using. introv Tr Sy R1 R2. apply× Tr. Qed.

```

Lemma trans_sym_rl : ∀ y x z R,
  trans R →
  sym R →
  R y x →
  R y z →
  R x z.

```

Proof using. introv Tr Sy R1 R2. apply× Tr. Qed.

End Trans.

Arguments trans_inv [A] y [x] [z] [R].

Arguments trans_inv_swap [A] y [x] [z] [R].

Arguments trans_sym_rr [A] y [x] [z] [R].

Arguments trans_sym_lr [A] y [x] [z] [R].

Arguments trans_sym_rl [A] y [x] [z] [R].

10.2.8 Equivalence relation

```
Record equiv A (R:binary A) :=  
{ equiv_refl : refl R;  
  equiv_sym : sym R;  
  equiv_trans : trans R }.
```

Section Equiv.

Variables (A : Type).
Implicit Types R : binary A.

End Equiv.

10.2.9 Inclusion

Definition rel_incl A B (R1 R2:A→B→Prop) :=
 $\forall x y, R1 x y \rightarrow R2 x y.$

Section Incl.

Variables (A B : Type).
Implicit Types R : A→B→Prop.

Lemma rel_incl_eq_forall_impl : $\forall R1 R2,$
 $rel_incl R1 R2 = (\forall x y, R1 x y \rightarrow R2 x y).$

Proof using. auto. Qed.

Lemma refl_rel_incl :
refl (@rel_incl A B).

Proof using. unfolds refl, rel_incl. autos×. Qed.

Lemma refl_rel_incl' : $\forall R,$
rel_incl R R.

Proof using. intros. applys refl_rel_incl. Qed.

Hint Resolve refl_rel_incl refl_rel_incl'.

Lemma antisym_rel_incl :
antisym (@rel_incl A B).

Proof using. unfolds rel_incl. extens×. Qed.

Lemma trans_rel_incl :
trans (@rel_incl A B).

Proof using. unfold trans, rel_incl. autos×. Qed.

End Incl.

10.2.10 Totality

Definition total A (R:binary A) :=

```

 $\forall x y, R x y \vee R y x.$ 
Section Total.
Variables (A : Type).
Implicit Types R : binary A.
Lemma total_eq_forall_or :  $\forall R,$ 
  total R = ( $\forall x y, R x y \vee R y x$ ).
Proof using. auto. Qed.

Lemma total_inv :  $\forall x y R,$ 
  total R  $\rightarrow$ 
   $R x y \vee R y x.$ 
Proof using. introv H. apply $\times$  H. Qed.

Lemma total_inv_not_l :  $\forall x y R,$ 
  total R  $\rightarrow$ 
   $\neg R x y \rightarrow$ 
   $R y x.$ 
Proof using. introv H N. destruct $\times$  (H x y). Qed.

Lemma total_inv_not_r :  $\forall x y R,$ 
  total R  $\rightarrow$ 
   $\neg R y x \rightarrow$ 
   $R x y.$ 
Proof using. introv H N. destruct $\times$  (H x y). Qed.

End Total.

```

10.2.11 Trichotomy

```

Inductive trichotomy A (R:binary A) : binary A :=
| trichotomy_left :  $\forall x y,$ 
   $R x y \rightarrow$ 
   $x \neq y \rightarrow$ 
   $\neg R y x \rightarrow$ 
  trichotomy R x y
| trichotomy_eq :  $\forall x,$ 
   $\neg R x x \rightarrow$ 
  trichotomy R x x
| trichotomy_right :  $\forall x y,$ 
   $\neg R x y \rightarrow$ 
   $x \neq y \rightarrow$ 
   $R y x \rightarrow$ 
  trichotomy R x y.

```

```

Definition trichotomous A (R:binary A) :=
   $\forall x y, \text{trichotomy } R x y.$ 

```

10.2.12 Definedness

```
Definition defined A B (R:A→B→Prop) :=  
  ∀ x, ∃ y, R x y.  
Section Defined.  
Variables (A : Type).  
Implicit Types R : binary A.  
Lemma total_eq_forall_exists : ∀ R,  
  defined R = (∀ x, ∃ y, R x y).  
Proof using. auto. Qed.  
Lemma defined_inv : ∀ x R,  
  defined R →  
  ∃ y, R x y.  
Proof using. introv H. apply× H. Qed.  
Lemma defined_inv_not : ∀ x R,  
  defined R →  
  (∀ y, ¬ R x y) →  
  False.  
Proof using. introv H N. forwards× (?&?): H. Qed.  
End Defined.
```

10.2.13 Functionality

```
Definition functional A B (R:A→B→Prop) :=  
  ∀ x y z, R x y → R x z → y = z.  
Section Functional.  
Variables (A : Type).  
Implicit Types R : binary A.  
Lemma functional_eq_forall_eq : ∀ R,  
  functional R = (∀ x y z, R x y → R x z → y = z).  
Proof using. auto. Qed.  
Lemma functional_inv : ∀ x z y R,  
  functional R →  
  R x y →  
  R x z →  
  y = z.  
Proof using. introv H N1 N2. apply× H. Qed.  
End Functional.
```

10.2.14 Criteria for equality

Section Equality.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

If $R1$ is defined, $R2$ is functional, and $R1$ is a subset of $R2$, then $R1$ equals $R2$. In that case, $R1$ and $R2$ represent the graph of a total function.

```
Lemma eq_of_incl_defined_functional : ∀ R1 R2,
  rel_incl R1 R2 →
  defined R1 →
  functional R2 →
  R1 = R2.
```

Proof using.

```
  introv Hincl Hdef Hfun extens. intros x y. iff M.
  { eauto. }
  { forwards (w'&M1): Hdef x.
    forwards M2: Hincl M1.
    forwards: Hfun M M2. subst×. }
```

Qed.

End Equality.

10.2.15 Properties of the equality relation

These results are not in LibEqual because the definitions from LibRelation are not yet available from that file.

Section Eq.

Variables ($A : \text{Type}$).

```
Lemma refl_eq :
  refl (@eq A).
```

Proof using. intros_all; subst¬. Qed.

```
Lemma sym_eq :
  sym (@eq A).
```

Proof using. intros_all; subst¬. Qed.

```
Lemma trans_eq :
  trans (@eq A).
```

Proof using. intros_all; subst¬. Qed.

```
Lemma equiv_eq :
  equiv (@eq A).
```

Proof using. intros. constructor; intros_all; subst¬. Qed.

End Eq.

10.2.16 Properties of the equality relation

These results are not in LibLogic because the definitions from LibRelation are not yet available from that file. See also LibOrder for a package of these properties.

```
Section Pred_incl.
```

```
Variables (A : Type).
```

```
Lemma refl_pred_incl :
```

```
  refl (@pred_incl A).
```

```
Proof using. unfold refl, pred_incl. autos×. Qed.
```

```
Lemma antisym_pred_incl :
```

```
  antisym (@pred_incl A).
```

```
Proof using. unfold antisym, pred_incl. extens×. Qed.
```

```
Lemma trans_pred_incl :
```

```
  trans (@pred_incl A).
```

```
Proof using. unfold trans, pred_incl. autos×. Qed.
```

```
End Pred_incl.
```

10.2.17 Properties of the equivalence relation

These results are not in LibLogic because the definitions from LibRelation are not yet available from that file. See also LibOrder for a package of these properties.

```
Section Iff.
```

```
Lemma refl_iff :
```

```
  refl iff.
```

```
Proof using. unfold refl, iff. autos×. Qed.
```

```
Lemma antisym_iff :
```

```
  antisym iff.
```

```
Proof using. unfold antisym, iff. extens×. Qed.
```

```
Lemma trans_iff :
```

```
  trans iff.
```

```
Proof using. unfold trans, iff. autos×. Qed.
```

```
End Iff.
```

10.3 Basic constructions

10.3.1 The empty relation

```
Definition empty A : binary A :=
```

```
  fun x y => False.
```

```

Section Empty.
Variables (A : Type).
Implicit Types x y : A.
Lemma empty_eq : ∀ x y,
  empty x y = False.
Proof using. auto. Qed.
Lemma empty_inv : ∀ x y,
  empty x y →
  False.
Proof using. auto. Qed.
Lemma functional_empty :
  functional (@empty A).
Proof using. unfolds× empty, functional. Qed.
End Empty.

```

10.3.2 Union of two relations

```

Definition union A (R1 R2:binary A) : binary A :=
  fun x y ⇒ R1 x y ∨ R2 x y.

Section Union.
Variables (A : Type).
Implicit Types R : binary A.
Lemma union_l : ∀ R1 R2 x y,
  R1 x y →
  union R1 R2 x y.
Proof using. unfold union. eauto. Qed.
Lemma union_r : ∀ R1 R2 x y,
  R2 x y →
  union R1 R2 x y.
Proof using. unfold union. eauto. Qed.
Lemma rel_incl_union_l : ∀ R1 R2,
  rel_incl R1 (union R1 R2).
Proof using. unfold rel_incl, union. eauto. Qed.
Lemma rel_incl_union_r : ∀ R1 R2,
  rel_incl R2 (union R1 R2).
Proof using. unfold rel_incl, union. eauto. Qed.
Lemma refl_union_l : ∀ R1 R2,
  refl R1 →
  refl (union R1 R2).

```

Proof using. unfold refl, union. eauto. Qed.

Lemma refl_union_r : $\forall R1 R2,$

refl $R2 \rightarrow$

refl (union $R1 R2$).

Proof using. unfold refl, union. eauto. Qed.

Lemma comm_union :

comm (@union A).

Proof using. unfold union. extens \times . Qed.

Lemma comm_union_args : $\forall R1 R2 x y,$

union $R2 R1 x y \rightarrow$

union $R1 R2 x y.$

Proof using. intros. rewrite \neg comm_union. Qed.

Union is functional provided disjoint domains Lemma functional_union : $\forall R1 R2,$

functional $R1 \rightarrow$

functional $R2 \rightarrow$

($\forall x y z, R1 x y \rightarrow R2 x z \rightarrow \text{False}$) \rightarrow

functional (union $R1 R2$).

Proof using.

intros. unfold union. intros $x y z Hxy Hxz.$

destruct Hxy ; destruct Hxz ; auto_false \times .

Qed.

Lemma covariant_union : $\forall R1 R2 S1 S2,$

rel_incl $R1 S1 \rightarrow$

rel_incl $R2 S2 \rightarrow$

rel_incl (union $R1 R2$) (union $S1 S2$).

Proof using. unfold rel_incl, union. autos \times . Qed.

End Union.

10.3.3 Intersection of two relations

Definition inter A ($R1 R2:\text{binary } A$) : $\text{binary } A :=$

fun $x y \Rightarrow R1 x y \wedge R2 x y.$

10.3.4 Complement of a relation

Definition compl A ($R:\text{binary } A$) : $\text{binary } A :=$

fun $x y \Rightarrow \neg R y x.$

10.3.5 Inverse of a relation

```
Definition inverse A (R:binary A) : binary A :=  
  fun x y => R y x.
```

Section Inverse.

Variables (A : Type).

Implicit Types R : binary A.

```
Lemma inverse_eq_fun : ∀ R,  
  inverse R = (fun x y => R y x).
```

Proof using. auto. Qed.

```
Lemma inverse_eq : ∀ R x y,  
  inverse R x y = R y x.
```

Proof using. auto. Qed.

```
Lemma injective_inverse :  
  injective (@inverse A).
```

Proof using.

```
  intros R1 R2 E. extens. intros x y.  
  unfolds inverse. rewrite× (fun_eq_2 y x E).
```

Qed.

```
Lemma inverse_sym : ∀ R,  
  sym R →  
  inverse R = R.
```

Proof using. intros. unfold inverse. extens×. Qed.

```
Lemma inverse_inverse : ∀ R,  
  inverse (inverse R) = R.
```

Proof using. extens×. Qed.

```
Lemma inverse_eq_l : ∀ R2 R1,  
  R1 = inverse R2 →  
  inverse R1 = R2.
```

Proof using. intros. apply injective_inverse. rewrite¬ inverse_inverse. Qed.

```
Lemma inverse_eq_r : ∀ R2 R1,  
  inverse R1 = R2 →  
  R1 = inverse R2.
```

Proof using. intros. apply injective_inverse. rewrite¬ inverse_inverse. Qed.

```
Lemma refl_inverse : ∀ R,  
  refl R →  
  refl (inverse R).
```

Proof using. intros_all. unfolds inverse. auto. Qed.

```
Lemma trans_inverse : ∀ R,  
  trans R →
```

trans (inverse R).

Proof using. intros_all. unfolds inverse. eauto. Qed.

Lemma antisym_inverse : $\forall R,$

antisym $R \rightarrow$

antisym (inverse R).

Proof using. intros_all. unfolds inverse. auto. Qed.

Lemma antisym_wrt_inverse : $\forall E R,$

antisym_wrt $E R \rightarrow$

antisym_wrt E (inverse R).

Proof using. intros_all. unfolds inverse. auto. Qed.

Lemma asym_inverse : $\forall R,$

asym $R \rightarrow$

asym (inverse R).

Proof using. intros_all. unfolds inverse. apply \times H. Qed.

Lemma total_inverse : $\forall R,$

total $R \rightarrow$

total (inverse R).

Proof using. intros_all. unfolds inverse. auto. Qed.

Lemma trichotomous_inverse : $\forall R,$

trichotomous $R \rightarrow$

trichotomous (inverse R).

Proof using.

intros H. intros x y. destruct (H x y).

apply \neg trichotomy_right.

apply \neg trichotomy_eq.

apply \neg trichotomy_left.

Qed.

Lemma inverse_equiv : $\forall A (E:\text{binary } A),$

equiv $E \rightarrow$

equiv (inverse E).

Proof using.

intros Equi. unfold inverse. constructor; intros_all;

dintuition eauto.

Qed.

Lemma inverse_union : $\forall R1 R2,$

inverse (union $R1 R2$) = union (inverse $R1$) (inverse $R2$).

Proof using.

unfold inverse, union. constructor; intros_all;

dintuition eauto.

Qed.

End Inverse.

10.3.6 preimage

Preimage, a.k.a. inverse image

```
Definition rel_preimage (A B:Type) (R:binary B) (f:A→B) : binary A :=  
  fun x y ⇒ R (f x) (f y).
```

10.3.7 rel_seq

Composition of two relations, usually written $R_1; R_2$.

```
Definition rel_seq (A B C:Type) (R1:A→B→Prop) (R2:B→C→Prop) : A→C→Prop :=  
  fun x z ⇒ ∃ y, R1 x y ∧ R2 y z.
```

A relation R is functional if and only if inverse R composed with R is a subset of the diagonal relation **eq**.

```
Lemma functional_eq_seq_inverse_incl_eq : ∀ A (R:binary A),  
  functional R = rel_incl (rel_seq (inverse R) R) eq.
```

Proof using.

unfold functional, rel_incl, rel_seq, inverse. extens. iff M; jauto.

Qed.

10.3.8 Turning a function into a relation.

```
Definition rel_fun A B (f:A→B) :=  
  fun x y ⇒ (y = f x).
```

Section Rel_fun.

Variables (A : Type).

Implicit Types R : binary A.

End Rel_fun.

10.4 Products

10.4.1 Pointwise product

```
Definition prod2 (A1 A2:Type)  
  (R1:binary A1) (R2:binary A2)  
  : binary (A1 × A2) :=  
  fun p1 p2 : A1 × A2 ⇒ match p1,p2 with (x1,x2),(y1,y2) ⇒  
    R1 x1 y1 ∧ R2 x2 y2 end.
```

```
Definition prod3 (A1 A2 A3:Type)  
  (R1:binary A1) (R2:binary A2) (R3:binary A3)
```

```
: binary (A1 × A2 × A3) :=
prod2 (prod2 R1 R2) R3.
```

```
Definition prod4 (A1 A2 A3 A4:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4)
: binary (A1 × A2 × A3 × A4) :=
prod2 (prod3 R1 R2 R3) R4.
```

Tactics

```
Tactic Notation "unfold_prod" :=
unfold prod4, prod3, prod2.
```

```
Tactic Notation "unfolds_prod" :=
unfold prod4, prod3, prod2 in *.
```

Equivalence

```
Lemma prod2_equiv : ∀ A1 A2 (E1:binary A1) (E2:binary A2),
equiv E1 →
equiv E2 →
equiv (prod2 E1 E2).
```

Proof using.

```
intros [R1 S1 T1] [R2 S2 T2]. constructor.
{ intros [x1 x2]. simple×. }
{ intros [x1 x2] [y1 y2]. simple×. }
{ intros [x1 x2] [y1 y2] [z1 z2]. simple×. }
```

Qed.

10.4.2 Lexicographical product

```
Definition lexico2 {A1 A2} (R1:binary A1) (R2:binary A2)
: binary (A1 × A2) :=
fun p1 p2 : A1 × A2 ⇒ let (x1,x2) := p1 in let (y1,y2) := p2 in
(R1 x1 y1) ∨ (x1 = y1) ∧ (R2 x2 y2).
```

```
Definition lexico3 {A1 A2 A3}
(R1:binary A1) (R2:binary A2) (R3:binary A3) : binary (A1 × A2 × A3) :=
lexico2 (lexico2 R1 R2) R3.
```

```
Definition lexico4 {A1 A2 A3 A4}
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4)
: binary (A1 × A2 × A3 × A4) :=
lexico2 (lexico3 R1 R2 R3) R4.
```

Tactics

```
Tactic Notation "unfold_lexico" :=
unfold lexico4, lexico3, lexico2.
```

Tactic Notation "unfolds_lexico" :=
unfold lexico4, lexico3, lexico2 in *.

Elimination

Section Lexico.

Variables (A1 A2 A3 A4:Type).

Variables (R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4).

Lemma lexico2_1 : $\forall x1 x2 y1 y2,$

$R1 x1 y1 \rightarrow$

lexico2 R1 R2 (x1, x2) (y1, y2).

Proof using. intros. left \neg . Qed.

Lemma lexico2_2 : $\forall x1 x2 y1 y2,$

$x1 = y1 \rightarrow$

$R2 x2 y2 \rightarrow$

lexico2 R1 R2 (x1, x2) (y1, y2).

Proof using. intros. right \neg . Qed.

Lemma lexico3_1 : $\forall x1 x2 x3 y1 y2 y3,$

$R1 x1 y1 \rightarrow$

lexico3 R1 R2 R3 (x1, x2, x3) (y1, y2, y3).

Proof using. intros. left. left \neg . Qed.

Lemma lexico3_2 : $\forall x1 x2 x3 y1 y2 y3,$

$x1 = y1 \rightarrow$

$R2 x2 y2 \rightarrow$

lexico3 R1 R2 R3 (x1, x2, x3) (y1, y2, y3).

Proof using. intro. left. right \neg . Qed.

Lemma lexico3_3 : $\forall x1 x2 x3 y1 y2 y3,$

$x1 = y1 \rightarrow$

$x2 = y2 \rightarrow$

$R3 x3 y3 \rightarrow$

lexico3 R1 R2 R3 (x1, x2, x3) (y1, y2, y3).

Proof using. intros. right \neg . Qed.

Lemma lexico4_1 : $\forall x1 x2 x3 x4 y1 y2 y3 y4,$

$R1 x1 y1 \rightarrow$

lexico4 R1 R2 R3 R4 (x1, x2, x3, x4) (y1, y2, y3, y4).

Proof using. intros. left. left. left \neg . Qed.

Lemma lexico4_2 : $\forall x1 x2 x3 x4 y1 y2 y3 y4,$

$x1 = y1 \rightarrow$

$R2 x2 y2 \rightarrow$

lexico4 R1 R2 R3 R4 (x1, x2, x3, x4) (y1, y2, y3, y4).

Proof using. intros. left. left. right \neg . Qed.

Lemma lexico4_3 : $\forall x1 x2 x3 x4 y1 y2 y3 y4,$

```

x1 = y1 →
x2 = y2 →
R3 x3 y3 →
lexico4 R1 R2 R3 R4 (x1 ,x2 ,x3 ,x4) (y1 ,y2 ,y3 ,y4).

```

Proof using. intros. left. right \neg . Qed.

```

Lemma lexico4_4 : ∀ x1 x2 x3 x4 y1 y2 y3 y4,
x1 = y1 →
x2 = y2 →
x3 = y3 →
R4 x4 y4 →
lexico4 R1 R2 R3 R4 (x1 ,x2 ,x3 ,x4) (y1 ,y2 ,y3 ,y4).

```

Proof using. intros. right \neg . Qed.

End Lexico.

Transitivity

```

Lemma trans_lexico2 : ∀ A1 A2
(R1:binary A1) (R2:binary A2),
trans R1 →
trans R2 →
trans (lexico2 R1 R2).

```

Proof using.

```

intros Tr1 Tr2. intros [x1 x2] [y1 y2] [z1 z2] Rxy Ryz.
simpls. destruct Rxy as [L1|[Eq1 L1]];
destruct Ryz as [M2|[Eq2 M2]]; subst $\times$ .

```

Qed.

```

Lemma trans_lexico3 : ∀ A1 A2 A3
(R1:binary A1) (R2:binary A2) (R3:binary A3),
trans R1 →
trans R2 →
trans R3 →
trans (lexico3 R1 R2 R3).

```

Proof using.

```

intros Tr1 Tr2 Tr3. applys $\neg$  trans_lexico2. applys $\neg$  trans_lexico2.

```

Qed.

```

Lemma trans_lexico4 : ∀ A1 A2 A3 A4
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
trans R1 →
trans R2 →
trans R3 →
trans R4 →
trans (lexico4 R1 R2 R3 R4).

```

Proof using.

introv Tr1 Tr2 Tr3. applys \sqsubset trans_lexico3. applys \sqsubset trans_lexico2.
 Qed.

Inclusion

Lemma rel_incl_lexico2 : $\forall A1 A2$
 $(R1 R1':\text{binary } A1) (R2 R2':\text{binary } A2),$
 $\text{rel_incl } R1 R1' \rightarrow$
 $\text{rel_incl } R2 R2' \rightarrow$
 $\text{rel_incl } (\text{lexico2 } R1 R2) (\text{lexico2 } R1' R2').$

Proof using.

introv I1 I2. intros [x1 x2] [y1 y2] [H1|[H1 H2]].
 $\{\text{left}\sqsubset.\} \{\text{subst. right}\sqsubset.\}$

Qed.

10.5 Closures

10.5.1 Reflexive closure

Inductive rclosure A ($R:\text{binary } A$) : binary A :=
 $| \text{rclosure_once} : \forall x y,$
 $R x y \rightarrow$
 $\text{rclosure } R x y$
 $| \text{rclosure_refl} : \forall x,$
 $\text{rclosure } R x x.$

Section Rclosure.

Variables (A : Type).

Implicit Types R : binary A.

Hint Constructors rclosure.

Equivalent definition

Lemma rclosure_eq_fun : $\forall R,$
 $\text{rclosure } R = (\text{fun } x y \Rightarrow R x y \vee x = y).$

Proof using. extens. iff M; destruct M; subst x. Qed.

Lemma rclosure_eq : $\forall R x y,$
 $\text{rclosure } R x y = (R x y \vee x = y).$

Proof using. extens. iff M; destruct M; subst x. Qed.

Lemma rclosure_inv : $\forall R x y,$
 $\text{rclosure } R x y \rightarrow$
 $R x y \vee x = y.$

Proof using. introv M; rewrite x rclosure_eq in M. Qed.

Properties

Lemma `refl_rclosure` : $\forall R,$

`refl (rclosure R).`

Proof using. *unfolds* `refl`. `Qed.`

Lemma `sym_rclosure` : $\forall R,$

`sym R →`

`sym (rclosure R).`

Proof using. *unfolds* `sym`. `introv M N. destruct × N.` `Qed.`

Lemma `antisym_rclosure` : $\forall R,$

`antisym R →`

`antisym (rclosure R).`

Proof using.

unfolds `antisym`. `introv M N1 N2.`

`destruct N1; destruct N2; subst ×.`

`Qed.`

Lemma `antisym_wrt_rclosure` : $\forall E R,$

`antisym_wrt E R →`

`antisym_wrt (rclosure E) (rclosure R).`

Proof using.

unfolds `antisym_wrt`. `introv M N1 N2.`

`destruct N1; destruct N2; subst ×.`

`Qed.`

Lemma `trans_rclosure` : $\forall R,$

`trans R →`

`trans (rclosure R).`

Proof using.

unfolds `trans`. `introv H M1 M2.`

`destruct M1; destruct M2; subst ×.`

`Qed.`

Lemma `total_rclosure` : $\forall R,$

`total R →`

`total (rclosure R).`

Proof using.

unfolds `total`. `introv H. intros x y. destruct × (H x y).`

`Qed.`

Lemma `rclosure_eq_of_refl` : $\forall R,$

`refl R →`

`rclosure R = R.`

Proof using.

unfolds `refl`. `introv H. extens. iff M.`

`{ destruct M; subst ×. } { auto. }`

`Qed.`

Lemma `rclosure_inverse_eq` : $\forall R,$
 $\text{rclosure}(\text{inverse } R) \equiv \text{inverse}(\text{rclosure } R).$

Proof using. `unfold inverse. extens. iff M; destruct× M.` **Qed.**

Constructors

TODO: rename to `rclosure_of_rclosure_step` `rclosure_of_step_rclosure` `trans_inv_rclosure_step` `trans_inv_step_rclosure`

Lemma `rclosure_trans_l` : $\forall y x z R,$
 $\text{trans } R \rightarrow$
 $\text{rclosure } R x y \rightarrow$
 $R y z \rightarrow$
 $\text{rclosure } R x z.$

Proof using. `introv T M N. rewrite rclosure_eq in *. destruct M; subst×.` **Qed.**

Lemma `rclosure_trans_r` : $\forall y x z R,$
 $\text{trans } R \rightarrow$
 $R x y \rightarrow$
 $\text{rclosure } R y z \rightarrow$
 $\text{rclosure } R x z.$

Proof using. `introv T M N. rewrite rclosure_eq in *. destruct N; subst×.` **Qed.**

Lemma `trans_rclosure_l` : $\forall y x z R,$
 $\text{trans } R \rightarrow$
 $\text{rclosure } R x y \rightarrow$
 $R y z \rightarrow$
 $R x z.$

Proof using. `introv T M H. rewrite rclosure_eq in *. destruct M; subst×.` **Qed.**

Lemma `trans_rclosure_r` : $\forall y x z R,$
 $\text{trans } R \rightarrow$
 $R x y \rightarrow$
 $\text{rclosure } R y z \rightarrow$
 $R x z.$

Proof using. `introv T M N. rewrite rclosure_eq in *. destruct N; subst×.` **Qed.**

Negation

Lemma `not_rclosure_inv` : $\forall R x y,$
 $\neg \text{rclosure } R x y \rightarrow$
 $\neg R x y \wedge x \neq y.$

Proof using. `introv M. rewrite× rclosure_eq in M.` **Qed.**

Lemma `not_rclosure_inv_rel` : $\forall R x y,$
 $\neg \text{rclosure } R x y \rightarrow$
 $\neg R x y.$

Proof using. `introv M. rewrite× rclosure_eq in M.` **Qed.**

Lemma `not_rclosure_inv_neq` : $\forall R x y,$

$\neg \text{rclosure } R \ x \ y \rightarrow$

$x \neq y.$

Proof using. *introv M. rewrite_x rclosure_eq in M.* Qed.

Inclusions

Lemma rel_incl_rclosure : $\forall R,$
 $\text{rel_incl } R (\text{rclosure } R).$

Proof using. *unfolds_x rel_incl.* Qed.

Lemma rclosure_of_rel : $\forall R \ x \ y,$
 $R \ x \ y \rightarrow$
 $\text{rclosure } R \ x \ y.$

Proof using. *intros. applys_x rel_incl_rclosure.* Qed.

Lemma covariant_rclosure : $\forall R1 \ R2,$
 $\text{rel_incl } R1 \ R2 \rightarrow$
 $\text{rel_incl } (\text{rclosure } R1) (\text{rclosure } R2).$

Proof using. *introv H M. destruct_x M.* Qed.

Lemma rel_incl_rclosure_rclosure : $\forall R1 \ R2,$
 $\text{rel_incl } R1 (\text{rclosure } R2) \rightarrow$
 $\text{rel_incl } (\text{rclosure } R1) (\text{rclosure } R2).$

Proof using. *introv H M. destruct_x M.* Qed.

End Rclosure.

Hint Constructors **rclosure** : *rclosure.*

10.5.2 Symmetric closure

Inductive **sclosure** A (*R:binary A*) : binary A :=
| sclosure_once : $\forall x \ y,$
 $R \ x \ y \rightarrow$
 $\text{sclosure } R \ x \ y$
| sclosure_sym : $\forall x \ y,$
 $\text{sclosure } R \ y \ x \rightarrow$
 $\text{sclosure } R \ x \ y.$

Section Sclosure.

Variables (A : Type).

Implicit Types R : binary A.

Hint Constructors **sclosure**.

Equivalent definition

Lemma sclosure_eq : $\forall R \ x \ y,$
 $\text{sclosure } R \ x \ y = (R \ x \ y \vee R \ y \ x).$

Proof using.

extens. iff M.
 { induction× M. }
 { destruct M; subst×. }
 Qed.

Lemma sclosure_inv : ∀ R x y,
sclosure R x y →
 $R x y \vee R y x$.

Proof using. introv M; rewrite× sclosure_eq in M. Qed.

Properties

Lemma refl_sclosure : ∀ R,
refl R →
refl (**sclosure** R).

Proof using. unfolds refl. Qed.

Lemma sym_sclosure : ∀ R,
sym (**sclosure** R).

Proof using. unfolds sym. introv N. destruct× N. Qed.

Lemma total_sclosure : ∀ R,
total R →
total (**sclosure** R).

Proof using.

unfolds total. introv H. intros x y. destruct× (H x y).

Qed.

Lemma sclosure_eq_of_sym : ∀ R,
sym R →
sclosure R = R.

Proof using.

unfolds sym. introv H. extens. intros. rewrite sclosure_eq.
iff M. { destruct M; subst×. } { auto. }

Qed.

Lemma sclosure_inverse_eq : ∀ R,
sclosure (**inverse** R) = **sclosure** R.

Proof using.

unfolds inverse. extens. intros. do 2 rewrite sclosure_eq. autos×.

Qed.

Negation

Lemma not_sclosure_inv : ∀ R x y,
 $\neg \text{sclosure} R x y \rightarrow$
 $\neg R x y \wedge \neg R y x$.

Proof using. introv M. rewrite× sclosure_eq in M. Qed.

Lemma not_sclosure_inv_l : ∀ R x y,

$\neg \text{sclosure } R \ x \ y \rightarrow$
 $R \ x \ y \rightarrow$
False.

Proof using. *introv M. rewrite \times sclosure_eq in M.* Qed.

Lemma not_sclosure_inv_r : $\forall R \ x \ y,$
 $\neg \text{sclosure } R \ x \ y \rightarrow$
 $R \ y \ x \rightarrow$
False.

Proof using. *introv M. rewrite \times sclosure_eq in M.* Qed.

Inclusions

Lemma rel_incl_sclosure : $\forall R,$
 $\text{rel_incl } R (\text{sclosure } R).$

Proof using. *unfolds \times rel_incl.* Qed.

Lemma rel_incl_inverse_sclosure : $\forall R,$
 $\text{rel_incl } (\text{inverse } R) (\text{sclosure } R).$

Proof using. *unfolds \times rel_incl, inverse.* Qed.

Lemma covariant_sclosure : $\forall R1 \ R2,$
 $\text{rel_incl } R1 \ R2 \rightarrow$
 $\text{rel_incl } (\text{sclosure } R1) (\text{sclosure } R2).$

Proof using. *introv H M. induction \times M.* Qed.

Lemma rel_incl_sclosure_sclosure : $\forall R1 \ R2,$
 $\text{rel_incl } R1 (\text{sclosure } R2) \rightarrow$
 $\text{rel_incl } (\text{sclosure } R1) (\text{sclosure } R2).$

Proof using. *introv H M. induction \times M.* Qed.

End Sclosure.

Hint Constructors **sclosure** : *sclosure.*

10.5.3 Reflexive-symmetric closure

Inductive rsclosure A (*R:binary A*) : binary A :=
 $| \text{rsclosure_once} : \forall x \ y,$
 $R \ x \ y \rightarrow$
rsclosure R x y
 $| \text{rsclosure_refl} : \forall x,$
rsclosure R x x
 $| \text{rsclosure_sym} : \forall x \ y,$
 $\text{rsclosure } R \ x \ y \rightarrow$
rsclosure R y x.

Section Rsclosure.

Variables (A : Type).

Implicit Types R : binary A .
 Hint Constructors **rsclosure**.

Equivalent definition

Lemma **rsclosure_eq** : $\forall R x y,$
 $\text{rsclosure } R x y = (R x y \vee R y x \vee x = y).$

Proof using.

extens. iff M.
 $\{ \text{induction} \times M. \}$
 $\{ \text{destruct } M \text{ as } [M|M|M]; \text{subst} \times. \}$

Qed.

Lemma **rsclosure_inv** : $\forall R x y,$
 $\text{rsclosure } R x y \rightarrow$
 $R x y \vee R y x \vee x = y.$

Proof using. *introv M; rewrite \times rsclosure_eq in M.* Qed.

Properties

Lemma **refl_rsclosure** : $\forall R,$
 $\text{refl } (\text{rsclosure } R).$

Proof using. *unfolds \times refl.* Qed.

Lemma **sym_rsclosure** : $\forall R,$
 $\text{sym } (\text{rsclosure } R).$

Proof using. *unfolds sym. introv N. destruct \times N.* Qed.

Lemma **total_rsclosure** : $\forall R,$
 $\text{total } R \rightarrow$
 $\text{total } (\text{rsclosure } R).$

Proof using.

unfolds total. introv H. intros x y. destruct \times (H x y).

Qed.

Lemma **rsclosure_inverse_eq** : $\forall R,$
 $\text{rsclosure } (\text{inverse } R) = \text{rsclosure } R.$

Proof using.

unfold inverse. extens. intros. do 2 rewrite rsclosure_eq. autos \times .

Qed.

Negation

Lemma **not_rsclosure_inv** : $\forall R x y,$
 $\neg \text{rsclosure } R x y \rightarrow$
 $\neg R x y \wedge \neg R y x \wedge \neg x = y.$

Proof using. *introv M. rewrite \times rsclosure_eq in M.* Qed.

Lemma **not_rsclosure_inv_l** : $\forall R x y,$
 $\neg \text{rsclosure } R x y \rightarrow$

$R x y \rightarrow$

False.

Proof using. *introv M. rewrite \times rsclosure_eq in M.* Qed.

Lemma not_rsclosure_inv_r : $\forall R x y,$

$\neg \text{rsclosure } R x y \rightarrow$

$R y x \rightarrow$

False.

Proof using. *introv M. rewrite \times rsclosure_eq in M.* Qed.

Lemma not_rsclosure_inv_neq : $\forall R x y,$

$\neg \text{rsclosure } R x y \rightarrow$

$x \neq y.$

Proof using. *introv M. rewrite \times rsclosure_eq in M.* Qed.

Inclusions

Lemma rel_incl_rsclosure : $\forall R,$

$\text{rel_incl } R (\text{rsclosure } R).$

Proof using. *unfold \times rel_incl.* Qed.

Lemma rel_incl_inverse_rsclosure : $\forall R,$

$\text{rel_incl } (\text{inverse } R) (\text{rsclosure } R).$

Proof using. *unfold \times rel_incl, inverse.* Qed.

Lemma covariant_rsclosure : $\forall R1 R2,$

$\text{rel_incl } R1 R2 \rightarrow$

$\text{rel_incl } (\text{rsclosure } R1) (\text{rsclosure } R2).$

Proof using. *introv H M. induction \times M.* Qed.

Lemma rel_incl_rsclosure_rsclosure : $\forall R1 R2,$

$\text{rel_incl } R1 (\text{rsclosure } R2) \rightarrow$

$\text{rel_incl } (\text{rsclosure } R1) (\text{rsclosure } R2).$

Proof using. *introv H M. induction \times M.* Qed.

End Rsclosure.

Hint Constructors **rsclosure** : *rsclosure.*

10.5.4 Transitive closure (**R+**), defined as $R \setminus\!\! \circ R \times$

Inductive **tclosure** A (*R:binary A*) : binary A :=

| tclosure_once : $\forall x y,$

$R x y \rightarrow$

tclosure R x y

| tclosure_trans : $\forall y x z,$

tclosure R x y \rightarrow

tclosure R y z \rightarrow

tclosure R x z.

Section Tclosure.

Variables (A : Type).

Implicit Types R : binary A .

Hint Constructors **tclosure**.

Properties

Lemma refl_tclosure : $\forall R,$

refl $R \rightarrow$

refl (**tclosure** R).

Proof using. unfolds \times refl. Qed.

Lemma sym_tclosure : $\forall R,$

sym $R \rightarrow$

sym (**tclosure** R).

Proof using. unfolds sym. introv $H N$. induction $\times N$. Qed.

Lemma trans_tclosure : $\forall R,$

trans (**tclosure** R).

Proof using. unfolds \times trans. Qed.

Lemma total_tclosure : $\forall R,$

total $R \rightarrow$

total (**tclosure** R).

Proof using.

unfolds total. introv H . intros $x y$. destruct $\times (H x y)$.

Qed.

Lemma tclosure_eq_of_trans : $\forall R,$

trans $R \rightarrow$

tclosure $R = R$.

Proof using.

unfolds trans. introv H . extens. iff M .

{ induction M ; subst \times . }

{ auto. }

Qed.

Lemma tclosure_inverse_eq : $\forall R,$

tclosure ($\text{inverse } R$) $= \text{inverse } (\text{tclosure } R)$.

Proof using.

unfolds inverse. extens. intros $x y$. iff M ; induction $\times M$.

Qed.

Constructors

Lemma tclosure_l : $\forall R y x z,$

$R x y \rightarrow$

tclosure $R y z \rightarrow$

tclosure $R x z$.

Proof using. *autos* \times . Qed.

Lemma **tclosure_r** : $\forall R y x z,$
tclosure $R x y \rightarrow$
 $R y z \rightarrow$
tclosure $R x z.$

Proof using. *autos* \times . Qed.

Inclusions

Lemma **rel_incl_tclosure** : $\forall R,$
rel_incl $R (\mathbf{tclosure} R).$

Proof using. *unfold*s \times **rel_incl**. Qed.

Lemma **covariant_tclosure** : $\forall A (R1 R2 : \text{binary } A),$
rel_incl $R1 R2 \rightarrow$
rel_incl (**tclosure** $R1$) (**tclosure** $R2$).

Proof using. *intros* $H M$. *induction* \times M . Qed.

Lemma **rel_incl_tclosure_tclosure** : $\forall R1 R2,$
rel_incl $R1 (\mathbf{tclosure} R2) \rightarrow$
rel_incl (**tclosure** $R1$) (**tclosure** $R2$).

Proof using. *intros* $H M$. *induction* \times M . Qed.

Induction principle with steps at head or tail

Section Ind.

Inductive **tclosure'l** $A (R:\text{binary } A) : \text{binary } A :=$
| **tclosure'l_once** : $\forall x y,$
 $R x y \rightarrow$
tclosure'l $R x y$
| **tclosure'l_step** : $\forall y x z,$
 $R x y \rightarrow$
tclosure'l $R y z \rightarrow$
tclosure'l $R x z.$

Lemma **trans_tclosure'l** : $\forall R,$
trans (**tclosure'l** R).

Proof using.

Hint Constructors **tclosure'l**.

intros $R y x z M1$. *gen* z . *induction* $M1$; *intros* $M2$; *autos* \times .

Qed.

Lemma **tclosure_eq_tclosure'l** : $\forall R,$
tclosure $R = \mathbf{tclosure}'l R.$

Proof using.

extens. *intros* $x y$. *iff* M .

{ *induction* \times M . *apply*s \times **trans_tclosure'l** y . }
{ *induction* \times M . }

Qed.

Lemma tclosure_ind_l : $\forall R (P : A \rightarrow A \rightarrow \text{Prop}),$
 $(\forall x y, R x y \rightarrow P x y) \rightarrow$
 $(\forall y x z, R x y \rightarrow \text{tclosure } R y z \rightarrow P y z \rightarrow P x z) \rightarrow$
 $(\forall x y, \text{tclosure } R x y \rightarrow P x y).$

Proof using.

intros H1 H2 M. rewrite tclosure_eq_tclosure'l in *. induction× M.

Qed.

Inductive tclosure'r A (R:binary A) : binary A :=
| tclosure'r_once : $\forall x y,$
 $R x y \rightarrow$
 $\text{tclosure}'r R x y$
| tclosure'r_step : $\forall y x z,$
 $\text{tclosure}'r R x y \rightarrow$
 $R y z \rightarrow$
 $\text{tclosure}'r R x z.$

Lemma trans_tclosure'r : $\forall R,$
 $\text{trans } (\text{tclosure}'r R).$

Proof using.

Hint Constructors tclosure'r.

intros R y x z M1 M2. gen x. induction M2; introv M1; autos×.

Qed.

Lemma tclosure_eq_tclosure'r : $\forall R,$
 $\text{tclosure } R = \text{tclosure}'r R.$

Proof using.

extens. intros x y. iff M.
{ induction× M. applys× trans_tclosure'r y. }
{ induction× M. }

Qed.

Lemma tclosure_ind_r : $\forall R (P : A \rightarrow A \rightarrow \text{Prop}),$
 $(\forall x y, R x y \rightarrow P x y) \rightarrow$
 $(\forall y x z, \text{tclosure } R x y \rightarrow P x y \rightarrow R y z \rightarrow P x z) \rightarrow$
 $(\forall x y, \text{tclosure } R x y \rightarrow P x y).$

Proof using.

intros H1 H2 M. rewrite tclosure_eq_tclosure'r in *. induction× M.

Qed.

End Ind.

Inversion principle with steps at head or tail

Lemma tclosure_inv_l : $\forall R x z,$
 $\text{tclosure } R x z \rightarrow$

$(R x z) \vee (\exists y, R x y \wedge \text{tclosure } R y z)$.

Proof using. intros R . applys \times tclosure_ind_l. Qed.

Lemma tclosure_inv_r : $\forall R x z,$

$\text{tclosure } R x z \rightarrow$

$(R x z) \vee (\exists y, \text{tclosure } R x y \wedge R y z)$.

Proof using. intros R . applys \times tclosure_ind_r. Qed.

End Tclosure.

Hint Resolve tclosure_once tclosure_l tclosure_r
: rtclosure.

10.5.5 Reflexive-transitive closure (R^*)

Inductive rtclosure A ($R:\text{binary } A$) : binary $A :=$

| rtclosure_once : $\forall x y,$

$R x y \rightarrow$

$\text{rtclosure } R x y$

| rtclosure_refl : $\forall x,$

$\text{rtclosure } R x x$

| rtclosure_trans : $\forall y x z,$

$\text{rtclosure } R x y \rightarrow$

$\text{rtclosure } R y z \rightarrow$

$\text{rtclosure } R x z.$

Section Rtclosure.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

Hint Constructors rtclosure.

Properties

Lemma refl_rtclosure : $\forall R,$

$\text{refl } (\text{rtclosure } R)$.

Proof using. unfolds \times refl. Qed.

Lemma sym_rtclosure : $\forall R,$

$\text{sym } R \rightarrow$

$\text{sym } (\text{rtclosure } R)$.

Proof using. unfolds sym. introv $M N$. induction $\times N$. Qed.

Lemma trans_rtclosure : $\forall R,$

$\text{trans } (\text{rtclosure } R)$.

Proof using. unfolds \times trans. Qed.

Lemma total_rtclosure : $\forall R,$

$\text{total } R \rightarrow$

$\text{total } (\text{rtclosure } R)$.

Proof using.

unfold total. introv H. intros x y. destruct \times (H x y).
Qed.

Lemma tclosure_eq_of_refl_trans : $\forall R,$

refl $R \rightarrow$
trans $R \rightarrow$
rtclosure $R = R.$

Proof using.

unfolds refl, trans. introv H1 H2. extens. iff M.
{ induction M; subst \times . }
{ autos \times . }

Qed.

Lemma rtclosure_inverse_eq : $\forall R,$

rtclosure (inverse R) = inverse (**rtclosure** R).

Proof using. unfold inverse. extens. iff M; induction \times M. Qed.

Constructors

Lemma rtclosure_l : $\forall R y x z,$

$R x y \rightarrow$
rtclosure $R y z \rightarrow$
rtclosure $R x z.$

Proof using. autos \times . Qed.

Lemma rtclosure_r : $\forall R y x z,$

rtclosure $R x y \rightarrow$
 $R y z \rightarrow$
rtclosure $R x z.$

Proof using. autos \times . Qed.

Lemma rtclosure_r' : $\forall R y x z,$

$R y z \rightarrow$
rtclosure $R x y \rightarrow$
rtclosure $R x z.$

Proof using. autos \times . Qed.

Inclusion

Lemma rel_incl_rtclosure : $\forall R,$

rel_incl R (**rtclosure** R).

Proof using. unfolds \times rel_incl. Qed.

Lemma covariant_rtclosure : $\forall R1 R2,$

rel_incl $R1 R2 \rightarrow$
rel_incl (**rtclosure** $R1$) (**rtclosure** $R2$).

Proof using. unfolds rel_incl. introv H M. induction \times M. Qed.

Lemma rel_incl_rtclosure_rtclosure : $\forall R1 R2,$

```
rel_incl R1 (rtclosure R2) →
  rel_incl (rtclosure R1) (rtclosure R2).
```

Proof using. unfolds rel_incl. introv H M. induction× M. Qed.

```
Lemma rel_incl_union_rtclosure : ∀ R1 R2,
  rel_incl (union (rtclosure R1) (rtclosure R2))
    (rtclosure (union R1 R2)).
```

Proof using.

```
hint rel_incl_union_l, rel_incl_union_r. introv [M|M].
{ applys× covariant_rtclosure R1. }
{ applys× covariant_rtclosure R2. }
```

Qed.

Negation

```
Lemma not_rtclosure_inv_neq : ∀ R x y,
  ¬ rtclosure R x y →
  x ≠ y.
```

Proof using. introv M E. subst. induction× M. Qed.

Induction principle with steps at head or tail

Section Ind.

```
Inductive rtclosure'l A (R:binary A) : binary A :=
| rtclosure'l_refl : ∀ x,
  rtclosure'l R x x
| rtclosure'l_step : ∀ y x z,
  R x y →
  rtclosure'l R y z →
  rtclosure'l R x z.
```

```
Lemma trans_rtclosure'l : ∀ R,
  trans (rtclosure'l R).
```

Proof using.

```
Hint Constructors rtclosure'l.
intros R y x z M1. gen z. induction M1; introv M2; autos×.
```

Qed.

```
Lemma rtclosure_eq_rtclosure'l : ∀ R,
  rtclosure R = rtclosure'l R.
```

Proof using.

```
extens. intros x y. iff M.
{ induction× M. applys× trans_rtclosure'l y. }
{ induction× M. }
```

Qed.

```
Lemma rtclosure_ind_l : ∀ R (P : A → A → Prop),
  (forall x, P x x) →
```

$(\forall y x z, R x y \rightarrow \text{rtclosure } R y z \rightarrow P y z \rightarrow P x z) \rightarrow$
 $(\forall x y, \text{rtclosure } R x y \rightarrow P x y).$

Proof using.

intros H1 H2 M. rewrite rtclosure_eq_rtclosure'l in *. induction× M.

Qed.

Inductive rtclosure'r A (R:binary A) : binary A :=
| rtclosure'r_refl : $\forall x,$
 $\text{rtclosure}'r R x x$
| rtclosure'r_step : $\forall y x z,$
 $\text{rtclosure}'r R x y \rightarrow$
 $R y z \rightarrow$
 $\text{rtclosure}'r R x z.$

Lemma trans_rtclosure'r : $\forall R,$
 $\text{trans } (\text{rtclosure}'r R).$

Proof using.

Hint Constructors rtclosure'r.

intros R y x z M1 M2. gen x. induction M2; introv M1; autos×.

Qed.

Lemma rtclosure_eq_rtclosure'r : $\forall R,$
 $\text{rtclosure } R = \text{rtclosure}'r R.$

Proof using.

extens. intros x y. iff M.

{ induction× M. applys× trans_rtclosure'r y. }
{ induction× M. }

Qed.

Lemma rtclosure_ind_r : $\forall R (P : A \rightarrow A \rightarrow \text{Prop}),$
 $(\forall x, P x x) \rightarrow$
 $(\forall y x z, \text{rtclosure } R x y \rightarrow P x y \rightarrow R y z \rightarrow P x z) \rightarrow$
 $(\forall x y, \text{rtclosure } R x y \rightarrow P x y).$

Proof using.

intros H1 H2 M. rewrite rtclosure_eq_rtclosure'r in *. induction× M.

Qed.

End Ind.

Inversion principle with steps at head or tail

Lemma rtclosure_inv_l : $\forall R x z,$
 $\text{rtclosure } R x z \rightarrow$
 $(x = z) \vee (\exists y, R x y \wedge \text{rtclosure } R y z).$

Proof using. intros R. applys× rtclosure_ind_l. Qed.

Lemma rtclosure_inv_r : $\forall R x z,$
 $\text{rtclosure } R x z \rightarrow$

$(x = z) \vee (\exists y, \text{rtclosure } R x y \wedge R y z)$.

Proof using. intros R . applys \times `rtclosure_ind_r`. Qed.

End Rtclosure.

Hint Resolve `rtclosure_refl` `rtclosure_once`
`rtclosure_l` `rtclosure_r'` : `rtclosure`.

10.5.6 Symmetric-transitive closure

Inductive **stclosure** A ($R:\text{binary } A$) : $\text{binary } A :=$

| `stclosure_once` : $\forall x y,$
 $R x y \rightarrow$
 stclosure $R x y$
| `stclosure_sym` : $\forall x y,$
 stclosure $R x y \rightarrow$
 stclosure $R y x$
| `stclosure_trans` : $\forall y x z,$
 stclosure $R x y \rightarrow$
 stclosure $R y z \rightarrow$
 stclosure $R x z$.

Section Stclosure.

Variables (A : Type).

Implicit Types R : $\text{binary } A$.

Hint Constructors **stclosure**.

Properties

Lemma `refl_stclosure` : $\forall R,$

`refl` $R \rightarrow$
 `refl` (**stclosure** R).

Proof using. `unfolds` \times `refl`. Qed.

Lemma `sym_stclosure` : $\forall R,$

`sym` (**stclosure** R).

Proof using. `unfolds` `sym`. `intros` N . `induction` $\times N$. Qed.

Lemma `trans_stclosure` : $\forall R,$

`trans` (**stclosure** R).

Proof using. `unfolds` \times `trans`. Qed.

Lemma `total_stclosure` : $\forall R,$

`total` $R \rightarrow$
 `total` (**stclosure** R).

Proof using.

`unfolds` `total`. `intros` H . `intros` $x y$. `destruct` $\times (H x y)$.

Qed.

```

Lemma stclosure_eq_of_sym_trans : ∀ R,
  sym R →
  trans R →
  stclosure R = R.

```

Proof using.

```

  unfolds sym, trans. introv H1 H2. extens. iff M.
  { induction M; subst×. }
  { autos×. }

```

Qed.

```

Lemma stclosure_inverse_eq : ∀ R,
  stclosure (inverse R) = inverse (stclosure R).

```

Proof using.

```

  unfolds inverse. extens. intros x y. iff M; induction× M.

```

Qed.

Constructors

```

Lemma stclosure_l : ∀ R y x z,
  R x y →
  stclosure R y z →
  stclosure R x z.

```

Proof using. autos×. Qed.

```

Lemma stclosure_r : ∀ R y x z,
  stclosure R x y →
  R y z →
  stclosure R x z.

```

Proof using. autos×. Qed.

Inclusion

```

Lemma rel_incl_stclosure : ∀ R,
  rel_incl R (stclosure R).

```

Proof using. unfolds× rel_incl. Qed.

```

Lemma covariant_stclosure : ∀ R1 R2,
  rel_incl R1 R2 →
  rel_incl (stclosure R1) (stclosure R2).

```

Proof using. introv H M. induction× M. Qed.

```

Lemma rel_incl_stclosure_stclosure : ∀ R1 R2,
  rel_incl R1 (stclosure R2) →
  rel_incl (stclosure R1) (stclosure R2).

```

Proof using. introv H M. induction× M. Qed.

End Stclosure.

Hint Constructors stclosure : stclosure.

10.5.7 Reflexive-symmetric-transitive closure

```
Inductive rstclosure A (R:binary A) : binary A :=
| rstclosure_once :  $\forall x y,$ 
   $R x y \rightarrow$ 
  rstclosure R x y
| rstclosure_refl :  $\forall x,$ 
  rstclosure R x x
| rstclosure_sym :  $\forall x y,$ 
  rstclosure R x y  $\rightarrow$ 
  rstclosure R y x
| rstclosure_trans :  $\forall y x z,$ 
  rstclosure R x y  $\rightarrow$ 
  rstclosure R y z  $\rightarrow$ 
  rstclosure R x z.
```

Section Rstclosure.

Variables (A : Type).

Implicit Types R : binary A.

Hint Constructors **rstclosure**.

Properties

Lemma **refl_rstclosure** : $\forall R,$
 $\text{refl } (\mathbf{rstclosure} R).$

Proof using. *unfolds* \times **refl**. Qed.

Lemma **sym_rstclosure** : $\forall R,$
 $\text{sym } (\mathbf{rstclosure} R).$

Proof using. *unfolds* \times **sym**. Qed.

Lemma **trans_rstclosure** : $\forall R,$
 $\text{trans } (\mathbf{rstclosure} R).$

Proof using. *unfolds* \times **trans**. Qed.

Lemma **total_rstclosure** : $\forall R,$
 $\text{total } R \rightarrow$
 $\text{total } (\mathbf{rstclosure} R).$

Proof using.

unfolds total. introv H. intros x y. destruct \times (H x y).

Qed.

Lemma **rstclosure_eq_of_refl_sym_trans** : $\forall R,$
 $\text{refl } R \rightarrow$
 $\text{sym } R \rightarrow$
 $\text{trans } R \rightarrow$
 $\mathbf{rstclosure} R = R.$

Proof using.

unfolds refl, sym, trans. introv H. extens. iff M.
 { induction M; subst×. }
 { auto. }
 Qed.

Lemma **rstclosure_inverse_eq** : $\forall R,$
rstclosure (inverse R) = **rstclosure** R .

Proof using.

unfolds inverse. extens. intros x y. iff M; induction× M.
 Qed.

Constructors

Lemma **rstclosure_l** : $\forall R y x z,$
 $R x y \rightarrow$
rstclosure $R y z \rightarrow$
rstclosure $R x z.$

Proof using. autos×. Qed.

Lemma **rstclosure_r** : $\forall R y x z,$
rstclosure $R x y \rightarrow$
 $R y z \rightarrow$
rstclosure $R x z.$

Proof using. autos×. Qed.

Inclusion

Lemma **rel_incl_rstclosure** : $\forall R,$
 $\text{rel_incl } R (\text{rstclosure } R).$

Proof using. unfolds× rel_incl. Qed.

Lemma **rel_incl_inverse_rstclosure** : $\forall R,$
 $\text{rel_incl} (\text{inverse } R) (\text{rstclosure } R).$

Proof using. unfolds× rel_incl, inverse. Qed.

Lemma **covariant_rstclosure** : $\forall R1 R2,$
 $\text{rel_incl } R1 R2 \rightarrow$
 $\text{rel_incl} (\text{rstclosure } R1) (\text{rstclosure } R2).$

Proof using. introv H M. induction× M. Qed.

Lemma **rel_incl_rstclosure_rstclosure** : $\forall R1 R2,$
 $\text{rel_incl } R1 (\text{rstclosure } R2) \rightarrow$
 $\text{rel_incl} (\text{rstclosure } R1) (\text{rstclosure } R2).$

Proof using. introv H M. induction× M. Qed.

Lemma **rel_incl_union_rstclosure** : $\forall R1 R2,$
 $\text{rel_incl} (\text{union} (\text{rstclosure } R1) (\text{rstclosure } R2))$
 $(\text{rstclosure} (\text{union } R1 R2)).$

Proof using.

hint **rel_incl_union_l**, **rel_incl_union_r**. introv [M|M].

```

{ applys× covariant_rstclosure R1. }
{ applys× covariant_rstclosure R2. }
Qed.
```

End Rstclosure.

Hint Constructors **rstclosure** : *rstclosure*.

10.5.8 Relationship between closures

Section ClosuresRel.

Variables (A : Type).

Implicit Types R : binary A .

Hint Constructors **rtclosure rsclosure stclosure rstclosure**.

rclosure to **rtclosure**

```

Lemma rtclosure_of_rclosure : ∀ R x y,
  rclosure R x y →
  rtclosure R x y.
```

Proof using. intros. destruct \times H. Qed.

```

Lemma rel_incl_rclosure_rtclosure : ∀ R,
  rel_incl (rclosure R) (rtclosure R).
```

Proof using. intros. *applys*× **rtclosure_of_rclosure**. Qed.

tclosure to **rtclosure**

```

Lemma rtclosure_of_tclosure : ∀ R x y,
  tclosure R x y →
  rtclosure R x y.
```

Proof using. intros. induction \times H. Qed.

```

Lemma rel_incl_tclosure_rtclosure : ∀ R,
  rel_incl (tclosure R) (rtclosure R).
```

Proof using. intros. *applys*× **rtclosure_of_tclosure**. Qed.

rclosure to **rsclosure**

```

Lemma rsclosure_of_rclosure : ∀ R x y,
  rclosure R x y →
  rsclosure R x y.
```

Proof using. intros. destruct \times H. Qed.

```

Lemma rel_incl_rclosure_rsclosure : ∀ R,
  rel_incl (rclosure R) (rsclosure R).
```

Proof using. intros. *applys*× **rsclosure_of_rclosure**. Qed.

sclosure to **rsclosure**

```

Lemma rsclosure_of_sclosure : ∀ R x y,
```

sclosure $R x y \rightarrow$

rsclosure $R x y.$

Proof using. intros. induction $\times H$. Qed.

Lemma rel_incl_sclosure_rsclosure : $\forall R,$

 rel_incl (**sclosure** R) (**rsclosure** R).

Proof using. intros. applys \times rsclosure_of_sclosure. Qed.

sclosure to stclosure

Lemma stclosure_of_sclosure : $\forall R x y,$

sclosure $R x y \rightarrow$

stclosure $R x y.$

Proof using. intros. induction $\times H$. Qed.

Lemma rel_incl_sclosure_stclosure : $\forall R,$

 rel_incl (**sclosure** R) (**stclosure** R).

Proof using. intros. applys \times stclosure_of_sclosure. Qed.

tclosure to stclosure

Lemma stclosure_of_tclosure : $\forall R x y,$

tclosure $R x y \rightarrow$

stclosure $R x y.$

Proof using. intros. induction $\times H$. Qed.

Lemma rel_incl_tclosure_stclosure : $\forall R,$

 rel_incl (**tclosure** R) (**stclosure** R).

Proof using. intros. applys \times stclosure_of_tclosure. Qed.

rclosure to rsclosure

Lemma rsclosure_of_rclosure : $\forall R x y,$

rclosure $R x y \rightarrow$

rsclosure $R x y.$

Proof using. intros. destruct $\times H$. Qed.

Lemma rel_incl_rclosure_rsclosure : $\forall R,$

 rel_incl (**rclosure** R) (**rsclosure** R).

Proof using. intros. applys \times rsclosure_of_rclosure. Qed.

sclosure to rsclosure

Lemma rsclosure_of_sclosure : $\forall R x y,$

sclosure $R x y \rightarrow$

rsclosure $R x y.$

Proof using. intros. induction $\times H$. Qed.

Lemma rel_incl_sclosure_rsclosure : $\forall R,$

 rel_incl (**sclosure** R) (**rsclosure** R).

Proof using. intros. applys \times rsclosure_of_sclosure. Qed.

tclosure to *tstclosure*

Lemma `rstclosure_of_tclosure : ∀ R x y,`

tclosure R x y \rightarrow

rstclosure R x y .

Proof using. intros. induction× H . Qed.

Lemma `rel_incl_tclosure_rstclosure : ∀ R,`

`rel_incl (tclosure R) (rstclosure R)`.

Proof using. intros. `applys`× `rstclosure_of_tclosure`. Qed.

rsclosure to **rstclosure**

Lemma `rstclosure_of_rsclosure : ∀ R x y,`

rsclosure R x y \rightarrow

rstclosure R x y .

Proof using. intros. induction× H . Qed.

Lemma `rel_incl_rsclosure_rstclosure : ∀ R,`

`rel_incl (rsclosure R) (rstclosure R)`.

Proof using. intros. `applys`× `rstclosure_of_rsclosure`. Qed.

rtclosure to **rstclosure**

Lemma `rstclosure_of_rtclosure : ∀ R x y,`

rtclosure R x y \rightarrow

rstclosure R x y .

Proof using. intros. induction× H . Qed.

Lemma `rel_incl_rtclosure_rstclosure : ∀ R,`

`rel_incl (rtclosure R) (rstclosure R)`.

Proof using. intros. `applys`× `rstclosure_of_rtclosure`. Qed.

stclosure to **rstclosure**

Lemma `rstclosure_of_stclosure : ∀ R x y,`

stclosure R x y \rightarrow

rstclosure R x y .

Proof using. intros. induction× H . Qed.

Lemma `rel_incl_stclosure_rstclosure : ∀ R,`

`rel_incl (stclosure R) (rstclosure R)`.

Proof using. intros. `applys`× `rstclosure_of_stclosure`. Qed.

End ClosuresRel.

10.5.9 Iterated closures

Section IterClosures.

Variables (A : Type).

Implicit Types R : binary A .

```
Hint Constructors rclosure sclosure tclosure  
  rtclosure rsclosure stclosure rstclosure.
```

```
Hint Resolve sym_tclosure sym_sclosure sym_rclosure  
  sym_rtclosure sym_rsclosure sym_rtclosure sym_rstclosure.
```

```
Lemma rclosure_sclosure_eq_rsclosure :  $\forall R,$   
  rclosure (sclosure R)  $=$  rsclosure R.
```

Proof using.

```
extens. intros x y iff M.  
{ destruct $\times$  M. { applys $\times$  rsclosure_of_sclosure. } }  
{ induction $\times$  M. { applys $\times$  sym_inv. } }
```

Qed.

```
Lemma sclosure_rclosure_eq_rsclosure :  $\forall R,$   
  sclosure (rclosure R)  $=$  rsclosure R.
```

Proof using.

```
extens. intros x y iff M.  
{ induction $\times$  M. { applys $\times$  rsclosure_of_rclosure. } }  
{ induction $\times$  M. }
```

Qed.

```
Lemma rclosure_tclosure_eq_rtclosure :  $\forall R,$   
  rclosure (tclosure R)  $=$  rtclosure R.
```

Proof using.

```
extens. intros x y iff M.  
{ induction $\times$  M. { applys $\times$  rtclosure_of_tclosure. } }  
{ induction $\times$  M. { destruct IHM1; destruct IHM2; autos $\times$ . } }
```

Qed.

```
Lemma tclosure_rclosure_eq_rtclosure :  $\forall R,$   
  tclosure (rclosure R)  $=$  rtclosure R.
```

Proof using.

```
extens. intros x y iff M.  
{ induction $\times$  M. { applys $\times$  rtclosure_of_rclosure. } }  
{ induction $\times$  M. }
```

Qed.

```
Lemma tclosure_sclosure_eq_stclosure :  $\forall R,$   
  tclosure (sclosure R)  $=$  stclosure R.
```

Proof using.

```
extens. intros x y iff M.  
{ induction $\times$  M. { applys $\times$  stclosure_of_sclosure. } }  
{ induction $\times$  M. { applys $\times$  sym_inv. } }
```

Qed.

```
Lemma rclosure_sclosure_eq_rstclosure :  $\forall R,$   
  rclosure (sclosure R)  $=$  rstclosure R.
```

Proof using.

```
extens. intros x y. iff M.  
{ induction× M. { applys× rstclosure_of_stclosure. } }  
{ induction× M.  
  { destruct IHM; autos×. }  
  { destruct IHM1; destruct IHM2; autos×. } }
```

Qed.

Lemma stclosure_rclosure_eq_rstclosure : $\forall R,$
stclosure (**rclosure** R) $=$ **rstclosure** R .

Proof using.

```
extens. intros x y. iff M.  
{ induction× M. { applys× rstclosure_of_rclosure. } }  
{ induction× M. }
```

Qed.

Lemma rtclosure_sclosure_eq_rstclosure : $\forall R,$
rtclosure (**sclosure** R) $=$ **rstclosure** R .

Proof using.

```
extens. intros x y. iff M.  
{ induction× M. { applys× rstclosure_of_sclosure. } }  
{ induction× M. { applys× sym_inv. } }
```

Qed.

Lemma tclosure_rsclosure_eq_rstclosure : $\forall R,$
tclosure (**rsclosure** R) $=$ **rstclosure** R .

Proof using.

```
extens. intros x y. iff M.  
{ induction× M. { applys× rstclosure_of_rsclosure. } }  
{ induction× M. { applys× sym_inv. } }
```

Qed.

End IterClosures.

10.5.10 Other lemmas – TODO

Section EquivClosures.

Variables (A : Type).

Implicit Types R : binary A .

Hint Constructors **rclosure** **sclosure** **tclosure** **rsclosure** **stclosure**.

Lemma rsclosure_eq_union_rclosure_sclosure : $\forall R,$
rsclosure R $=$ union (**rclosure** R) (**sclosure** R).

Proof using.

```
extens. intros x y. unfold union. iff M.  
{ induction× M. destruct× IHM as [H|H]. destruct× H. }
```

{ destruct \times M as [H|H]. destruct \times H. applys \times rsclosure_of_sclosure. }

Qed.

Lemma rtclosure_eq_union_rclosure : $\forall R,$
rtclosure R = union (**rclosure** R) (**tclosure** R).

Proof using.

```
extens. intros x y. unfold union. iff M.
{ induction $\times$  M. destruct IHM1 as [H1|H1]; destruct IHM2 as [H2|H2].
 { destruct H1; destruct $\times$  H2. }
 { destruct $\times$  H1. }
 { destruct $\times$  H2. }
 { autos $\times$ . } }
{ destruct $\times$  M.
 { applys $\times$  rtclosure_of_rclosure. }
 { applys $\times$  rtclosure_of_tclosure. } }
```

Qed.

Lemma rtclosure_inv_rclosure_or_tclosure : $\forall R x y,$
rtclosure R x y \rightarrow
 $x = y \vee \text{tclosure } R x y.$

Proof using.

```
intros M. rewrite rtclosure_eq_union_rclosure in M.
destruct M as [M|M]. { destruct $\times$  M. } { auto. }
```

Qed.

Lemma stclosure_eq_rstclosure_of_refl : $\forall R,$
refl R \rightarrow
stclosure R = **rstclosure** R.

Proof using.

```
intros H. extens. intros x y. iff M.
{ applys $\times$  rstclosure_of_stclosure. }
{ induction $\times$  M. }
```

Qed.

Lemma rel_incl_tclosure_stclosure_l : $\forall R1 R2,$
rel_incl R1 (**stclosure** R2) \rightarrow
rel_incl (**tclosure** R1) (**stclosure** R2).

Proof using. *intros H M. induction \times M.* Qed.

End EquivClosures.

10.5.11 Mixed transitivity between closures

Section MixedClosures.
Variables (A : Type).

```
Implicit Types R : binary A.
```

```
Hint Constructors tclosure.
```

```
Lemma tclosure_of_rtclosure_l : ∀ R x y z,
```

```
  rtclosure R x y →
```

```
  R y z →
```

```
  tclosure R x z.
```

Proof using.

```
  introv H M. destruct (rtclosure_inv_rclosure_or_tclosure H).
```

```
  { subst×. }
```

```
  { applys× tclosure_r. }
```

Qed.

```
Lemma tclosure_of_rtclosure_r : ∀ R x y z,
```

```
  R x y →
```

```
  rtclosure R y z →
```

```
  tclosure R x z.
```

Proof using.

```
  introv H M. destruct (rtclosure_inv_rclosure_or_tclosure H).
```

```
  { subst×. }
```

```
  { applys× tclosure_l. }
```

Qed.

```
Lemma tclosure_of_rtclosure_tclosure : ∀ R y x z,
```

```
  rtclosure R x y →
```

```
  tclosure R y z →
```

```
  tclosure R x z.
```

Proof using.

```
  introv H M. destruct (rtclosure_inv_rclosure_or_tclosure H); subst×.
```

Qed.

```
Lemma tclosure_of_tclosure_rtclosure : ∀ R y x z,
```

```
  tclosure R x y →
```

```
  rtclosure R y z →
```

```
  tclosure R x z.
```

Proof using.

```
  introv M H. destruct (rtclosure_inv_rclosure_or_tclosure H); subst×.
```

Qed.

End MixedClosures.

10.5.12 Irreflexive restriction of a relation

```
Definition strict A (R:binary A) : binary A :=
```

```
  fun x y ⇒ R x y ∧ x ≠ y.
```

```

Section Strict.

Variables (A : Type).
Implicit Types R : binary A.
Hint Unfold strict.

Lemma strict_eq_fun : ∀ R,
  strict R = (fun x y ⇒ R x y ∧ x ≠ y).
Proof using. auto. Qed.

Lemma strict_eq : ∀ R x y,
  strict R x y = (R x y ∧ x ≠ y).
Proof using. auto. Qed.

Lemma inverse_strict : ∀ R,
  inverse (strict R) = strict (inverse R).
Proof using. intros. unfold inverse, strict. extens×. Qed.

Lemma trans_strict_l : ∀ y x z R,
  trans R →
  strict R x y →
  R y z →
  R x z.
Proof using. introv T (E&H) H'; subst×. Qed.

Lemma trans_strict_r : ∀ y x z R,
  trans R →
  R x y →
  strict R y z →
  R x z.
Proof using. introv T H (E&H'); subst×. Qed.

Lemma irrefl_strict : ∀ R,
  irrefl (strict R).
Proof using. unfold strict, irrefl. intros. rew_logic×. Qed.

Lemma antisym_strict : ∀ R,
  antisym R →
  antisym (strict R).
Proof using. unfolds× antisym, strict. Qed.

Lemma trans_strict : ∀ R,
  trans R →
  antisym R →
  trans (strict R).
Proof using.
  introv T S. unfold strict. introv [H1 H2] [H3 H4]. split.
  { apply× T. }
  { intros K. subst. apply H2. apply¬ S. }

```

Qed.

```
Lemma strict_rclosure : ∀ R,
  irrefl R →
  strict (rclosure R) = R.
```

Proof using.

```
unfold strict. extens. intros x y. iff (K1&K2) K.
{ destruct× K1. }
{ split. { left×. } { apply× irrefl_inv_neq. } }
```

Qed.

```
Lemma rclosure_strict : ∀ R,
  refl R →
  rclosure (strict R) = R.
```

Proof using.

```
Hint Constructors rclosure.
unfold strict. extens. intros x y. iff K.
{ destruct K; subst×. }
{ tests: (x = y); subst×. }
```

Qed.

End Strict.

10.5.13 Inclusion of a function in a relation

`fun_in_rel f R` asserts that input-output pairs of `f` are included in the relation `R`.

Definition `fun_in_rel A B (f:A→B) (R:A→B→Prop) :=`
 $\forall x, R x (f x)$.

Section `Fun_in_rel`.

Variables (`A B : Type`).

Implicit Type `f : A→B`.

Implicit Type `R : A→B→Prop`.

```
Lemma defined_of_fun_in_rel : ∀ f R,
  fun_in_rel f R →
  defined R.
```

Proof using. `unfolds× fun_in_rel, defined`. Qed.

The relation built from a function `f` is included in a relation `R` iff the function `f` is included in `R`

```
Lemma rel_incl_rel_fun_eq_fun_in_rel : ∀ f R,
  rel_incl (rel_fun f) R = fun_in_rel f R.
```

Proof using.

```
extens. unfold rel_fun, fun_in_rel. iff H; intros x; specializes H x.
{ applys× H. }
```

```

{ intros y Hy. subst ⊥. }
Qed.

End Fun_in_rel.
```

10.5.14 Inclusion of a relation in a function

`rel_in_fun R f` asserts that input-output pairs of R are input-output for f .

Definition $\text{rel_in_fun } A \ B \ (R:A \rightarrow B \rightarrow \text{Prop}) \ (f:A \rightarrow B) :=$
 $\forall x \ y, R \ x \ y \rightarrow f \ x = y.$

`Section Rel_in_fun.`

`Variables (A B : Type).`

`Implicit Type f : A → B.`

`Implicit Type R : A → B → Prop.`

`Lemma functional_of_rel_in_fun : ∀ R f,`

`rel_in_fun R f →`

`functional R.`

`Proof using.`

`unfold rel_in_fun, functional. introv M N1 N2.`

`lets: M N1. lets: M N2. congruence.`

`Qed.`

`Lemma rel_in_fun_of_fun_in_rel_functional : ∀ f R,`

`fun_in_rel f R →`

`functional R →`

`rel_in_fun R f.`

`Proof using.`

`introv h1 h2. intros a b H. forwards M: h1 a. forwards*: h2 H M.`

`Qed.`

`End Rel_in_fun.`

Chapter 11

Library SLF.LibOrder

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibReflect LibOperation LibRelation.
Generalizable Variables A .

Definition

```
Record preorder  $A$  ( $R$ :binary  $A$ ) : Prop := {  
  preorder_refl : refl  $R$ ;  
  preorder_trans : trans  $R$  }.
```

Arguments preorder_trans [A] [R] [p] y [x] [z].

Transformations

```
Lemma preorder_inverse :  $\forall A$  ( $R$ :binary  $A$ ),  
  preorder  $R \rightarrow$   
  preorder (inverse  $R$ ).
```

Proof using. hint trans_inverse. introv [Re Tr]. constructor \neg . Qed.

```
Lemma preorder_rclosure :  $\forall A$  ( $R$ :binary  $A$ ),  
  preorder  $R \rightarrow$   
  preorder (rclosure  $R$ ).
```

Proof using. hint refl_rclosure, trans_rclosure. introv [Re Tr]. constructor \neg . Qed.

Definition of total preorder relations

```
Record total_preorder  $A$  ( $R$ :binary  $A$ ) : Prop := {  
  total_preorder_trans : trans  $R$ ;  
  total_preorder_total : total  $R$  }.
```

Arguments total_preorder_trans [A] [R] t y [x] [z].

Conversion to preorder

```
Lemma total_preorder_refl :  $\forall A$  ( $le$ :binary  $A$ ),  
  total_preorder  $le \rightarrow$ 
```

`refl le.`

`Proof using. introv [Tr To]. intros x. destruct¬ (To x x). Qed.`

`Hint Resolve total_preorder_refl.`

`Coercion total_preorder_to_preorder A (R:binary A)
(O:total_preorder R) : preorder R.`

`Proof using. lets [M _]: O. constructor¬. Qed.`

`Hint Resolve total_preorder_to_preorder.`

Transformations

`Lemma total_preorder_inverse : ∀ A (R:binary A),
total_preorder R →
total_preorder (inverse R).`

`Proof using. hint trans-inverse, total-inverse. introv [Tr To]. constructor¬. Qed.`

`Lemma total_preorder_rclosure : ∀ A (R:binary A),
total_preorder R →
total_preorder (rclosure R).`

`Proof using. hint trans_rclosure, total_rclosure. introv [Re Tr]. constructor¬. Qed.`

Properties

`Lemma inverse_of_not : ∀ A (R:binary A) x y,
total R →
¬ R x y →
inverse R x y.`

`Proof using. introv T H. destruct (T x y); auto_false¬. Qed.`

`Lemma inverse_strict_of_not : ∀ A (R:binary A) x y,
total R →
¬ R x y →
inverse (strict R) x y.`

`Proof using.`

`introv T H. destruct (T x y). auto_false¬.
 hnf. split¬. intro_subst¬.`

`Qed.`

Definition

`Record order A (R:binary A) : Prop := {
 order_refl : refl R;
 order_trans : trans R;
 order_antisym : antisym R }.`

`Arguments order_trans [A] [R] [o] y [x] [z].`

`Arguments order_antisym [A] [R] [o] [x] [y].`

Conversion to preorder

Coercion `order_to_preorder A (R:binary A)`

`(O:order R) : preorder R.`

Proof using. `destruct` \times `O.constructors` \times . Qed.

Hint Resolve `order_to_preorder`.

Transformations

Lemma `order_inverse : ∀ A (R:binary A),`

`order R →`

`order (inverse R).`

Proof using.

`hint trans_inverse, antisym_inverse.`

`intros [Re Tr An]. constructor ⊥.`

Qed.

Lemma `order_rclosure : ∀ A (R:binary A),`

`order R →`

`order (rclosure R).`

Proof using.

`hint refl_rclosure, trans_rclosure, antisym_rclosure.`

`intros [Re Tr An]. constructor ⊥.`

Qed.

Properties

11.1 Order relation upto an equivalence relation

Note: this is used in LibFix

Record `order_wrt (A:Type) (E:binary A) (R:binary A) : Prop := {`

`order_wrt_refl : refl R;`

`order_wrt_trans : trans R;`

`order_wrt_antisym : antisym_wrt E R }.`

Arguments `order_wrt_trans [A] [E] [R] [o] y [x] [z]`.

Arguments `order_wrt_antisym [A] [E] [R] [o] [x] [y]`.

Conversion to preorder

Coercion `order_wrt_to_preorder A (E:binary A) (R:binary A)`

`(O:order_wrt E R) : preorder R.`

Proof using. `destruct` \times `O.constructors` \times . Qed.

Hint Resolve `order_wrt_to_preorder`.

Transformations

Lemma `order_wrt_inverse : ∀ A (E:binary A) (R:binary A),`

`order_wrt E R →`

order_wrt E (inverse R).

Proof using.

hint trans_inverse, antisym_wrt_inverse.
introv [Re Tr An]. constructor \neg .

Qed.

Lemma **order_wrt_rclosure** : $\forall A (E:\text{binary } A) (R:\text{binary } A)$,
order_wrt $E R \rightarrow$
order_wrt (**rclosure** E) (**rclosure** R).

Proof using.

hint refl_rclosure, trans_rclosure, antisym_wrt_rclosure.
introv [Re Tr An]. constructor \neg .

Qed.

Properties

Definition

Record **total_order** $A (R:\text{binary } A) : \text{Prop} := \{$
total_order_order : **order** R ;
total_order_total : total $R \}$.

Projections

Definition total_order_refl := order_refl.

Definition total_order_trans := order_trans.

Definition total_order_antisym := order_antisym.

Arguments total_order_trans [A] [R] [o] y [x] [z].

Arguments total_order_antisym [A] [R] [o] [x] [y].

Construction

Lemma **total_order_intro** : $\forall A (R:\text{binary } A)$,
trans $R \rightarrow$
antisym $R \rightarrow$
total $R \rightarrow$
total_order R .

Proof using.

introv Tra Ant Tot. constructor \neg . constructor \neg .
intros_all. destruct \neg (Tot x x).

Qed.

Conversion to order

Coercion **total_order_to_total_preorder** $A (R:\text{binary } A)$
($O:\text{total_order } R$) : **total_preorder** R .

Proof using. destruct \times O . constructors \times . applys \times order_trans. Qed.

Definition **total_order_to_order** := total_order_order.

Hint Resolve **total_order_to_order** **total_order_to_total_preorder**.

Transformations

```
Lemma total_order_inverse : ∀ A (R:binary A),  
  total_order R →  
  total_order (inverse R).
```

Proof using.

```
  hint total_inverse, order_inverse.  
  introv [Or To]. constructor¬.
```

Qed.

```
Lemma total_order_rclosure : ∀ A (R:binary A),  
  total_order R →  
  total_order (rclosure R).
```

Proof using.

```
  hint total_rclosure, order_rclosure.  
  introv [Or To]. constructor¬.
```

Qed.

Properties

Section TotalOrderProp.

```
Variables (A:Type) (R : binary A).
```

WARNING: notations here are not typeclass operators TODO: is this really what we want? perhaps it would be clearer to inline these notations.

```
Notation "'le'" := (R).  
Notation "'ge'" := (inverse R).  
Notation "'lt'" := (strict R).  
Notation "'gt'" := (inverse lt).
```

```
Ltac total_order_normalize :=  
  repeat rewrite rclosure_eq_fun;  
  repeat rewrite inverse_eq_fun;  
  repeat rewrite strict_eq_fun.
```

```
Lemma total_order_le_is_rclosure_lt : ∀ (To:total_order R),  
  le = rclosure lt.
```

Proof using.

```
  extens. intros. total_order_normalize. iff M.  
  tests~: (x = y).  
  destruct M. autos×. subst×. dintuition eauto.
```

Qed.

```
Lemma total_order_lt_is_strict_le : ∀ (To:total_order R),  
  lt = strict le.
```

Proof using.

```
  auto.
```

Qed.

Lemma total_order_ge_is_rclosure_gt : $\forall (To:\text{total_order } R)$,
 $\text{ge} = \text{rclosure } \text{gt}$.

Proof using.

```
extens. intros. total_order_normalize. iff M.
tests $\sim$ : (x = y).
destruct M. autos $\times$ . subst $\times$ . dintuition eauto.
```

Qed.

Lemma total_order_gt_is_strict_ge : $\forall (To:\text{total_order } R)$,
 $\text{gt} = \text{strict ge}$.

Proof using.

```
extens. intros. total_order_normalize. iff M.
tests $\sim$ : (x = y).
destruct M. autos $\times$ .
destruct M. autos $\times$ .
```

Qed.

Lemma total_order_lt_or_eq_or_gt : $\forall (To:\text{total_order } R) x y$,
 $\text{lt } x y \vee x = y \vee \text{gt } x y$.

Proof using.

```
introv H. intros. total_order_normalize. tests: (x = y).
branch $\neg$  2.
destruct (total_order_total H x y).
branch $\neg$  1.
branch $\neg$  3.
```

Qed.

Lemma total_order_lt_or_ge : $\forall (To:\text{total_order } R) x y$,
 $\text{lt } x y \vee \text{ge } x y$.

Proof using.

```
intros. branches (total_order_lt_or_eq_or_gt To x y).
left $\neg$ .
right $\neg$ . subst. hnf. apply $\neg$  total_order_refl.
right $\neg$ . rewrite $\neg$  total_order_ge_is_rclosure_gt.
total_order_normalize. hnfs $\neg$ .
```

Qed.

Lemma total_order_le_or_gt : $\forall (To:\text{total_order } R) x y$,
 $\text{le } x y \vee \text{gt } x y$.

Proof using.

```
intros. branches $\neg$  (total_order_lt_or_eq_or_gt To x y).
left $\neg$ . rewrite $\neg$  total_order_le_is_rclosure_lt. total_order_normalize. hnfs $\neg$ .
left $\neg$ . subst. apply $\neg$  total_order_refl.
```

Qed.

End TotalOrderProp.

Definition

```
Record strict_order A (R:binary A) : Prop := {  
    strict_order_irrefl : irrefl R;  
    strict_order_asym : asym R;  
    strict_order_trans : trans R }.
```

Arguments strict_order_trans [A] [R] [s] y [x] [z].

Transformations

```
Lemma strict_order_inverse : ∀ A (R:binary A),  
    strict_order R →  
    strict_order (inverse R).
```

Proof using.

```
  hint antisym-inverse, trans-inverse, asym-inverse.  
  introv [Ir As Tr]. constructor¬.
```

Qed.

```
Lemma strict_order_strict : ∀ A (R:binary A),  
    order R →  
    strict_order (strict R).
```

Proof using.

```
  introv [Re As Tr]. unfold strict. constructor; intros_all; simpls.  
  destruct× H.  
  applys× antisym_inv x y.  
  split. applys× As. intros E. subst. applys× antisym_inv y z.
```

Qed.

```
Lemma order_rclosure_of.strict_order : ∀ A (R:binary A),  
    strict_order R →  
    order (rclosure R).
```

Proof using.

```
  introv [Re As Tr]. rewrite rclosure_eq_fun. constructor; simpl.  
  intros_all¬.  
  introv [H1|E1] [H2|E2]; subst; auto.  
  left. applys× trans_inv.  
  introv [H1|E1] [H2|E2]; try subst; auto.  
  false. applys× As.
```

Qed.

Definition

```
Record strict_total_order A (R:binary A) : Prop := {  
    strict_total_order_trans : trans R;  
    strict_total_order_trichotomous : trichotomous R }.
```

Arguments strict_total_order_trans [A] [R] [s] y [x] [z].

Conversion to strict order and back

```
Lemma strict_total_order_irrefl : ∀ A (R:binary A),  
  strict_total_order R →  
  irrefl R.
```

Proof using. introv [Tr Tk]. intros x. lets: (Tk x x). inverts¬ H. Qed.

```
Lemma strict_total_order_asym : ∀ A (R:binary A),  
  strict_total_order R →  
  asym R.
```

Proof using. introv [Tr Tk]. intros x y. lets: (Tk x y). inverts¬ H. Qed.

Coercion strict_total_order_to_strict_order A (R:binary A)
(O:strict_total_order R) : strict_order R.

Proof using.

```
lets [M _]: O. constructor;  
autos¬ strict_total_order_irrefl strict_total_order_asym.
```

Qed.

Hint Resolve strict_total_order_to_strict_order.

Transformation

```
Lemma strict_total_order_inverse : ∀ A (R:binary A),  
  strict_total_order R →  
  strict_total_order (inverse R).
```

Proof using.

```
introv [Tr Tk]. constructor. apply¬ trans_inverse.  
apply¬ trichotomous_inverse.
```

Qed.

From total order

```
Lemma strict_total_order_of_total_order : ∀ A (R:binary A),  
  total_order R →  
  strict_total_order (strict R).
```

Proof using.

```
introv [[Re Tr As] To]. constructor.  
apply¬ trans_strict.  
intros x y. tests: (x = y).  
subst. apply trichotomy_eq. unfolds× strict.  
unfold strict. destruct (To x y).  
apply× trichotomy_left.  
apply× trichotomy_right.
```

Qed.

11.2 Definition of order operators

11.2.1 Classes and notation for comparison operators

Operators

```
Class Le ( $A : \text{Type}$ ) := { le : binary  $A$  }.
Class Ge ( $A : \text{Type}$ ) := { ge : binary  $A$  }.
Class Lt ( $A : \text{Type}$ ) := { lt : binary  $A$  }.
Class Gt ( $A : \text{Type}$ ) := { gt : binary  $A$  }.
```

```
Global Opaque le lt ge gt.
```

Structures

```
Class Le_preorder '{Le A} : Prop :=
  { le_preorder : preorder le }.

Class Le_total_preorder '{Le A} : Prop :=
  { le_total_preorder : total_preorder le }.

Class Le_order '{Le A} : Prop :=
  { le_order : order le }.

Class Le_total_order '{Le A} : Prop :=
  { le_total_order : total_order le }.

Class Lt_strict_order '{Lt A} : Prop :=
  { lt_strict_order : strict_order lt }.

Class Lt_strict_total_order '{Lt A} : Prop :=
  { lt_strict_total_order : strict_total_order lt }.
```

Notation

Declare Scope comp_scope.

```
Notation " $x \leq y$ " := (le  $x y$ )
  (at level 70, no associativity) : comp_scope.
Notation " $x \geq y$ " := (ge  $x y$ )
  (at level 70, no associativity) : comp_scope.
Notation " $x < y$ " := (lt  $x y$ )
  (at level 70, no associativity) : comp_scope.
Notation " $x > y$ " := (gt  $x y$ )
  (at level 70, no associativity) : comp_scope.
```

Open Scope comp_scope.

```
Notation " $x \leq y \leq z$ " := ( $x \leq y \wedge y \leq z$ )
  (at level 70,  $y$  at next level) : comp_scope.
Notation " $x \leq y < z$ " := ( $x \leq y \wedge y < z$ )
  (at level 70,  $y$  at next level) : comp_scope.
```

```

Notation " $x < y \leq z$ " := ( $x < y \wedge y \leq z$ )
  (at level 70,  $y$  at next level) : comp_scope.
Notation " $x < y < z$ " := ( $x < y \wedge y < z$ )
  (at level 70,  $y$  at next level) : comp_scope.

```

11.2.2 The operators ge, lt and gt are deduced from le

```

Instance ge_of_le : ∀ ‘{Le A}, Ge A.
  constructor. apply (inverse le). Defined.
Instance lt_of_le : ∀ ‘{Le A}, Lt A.
  constructor. apply (strict le). Defined.
Instance gt_of_le : ∀ ‘{Le A}, Gt A.
  constructor. apply (inverse lt). Defined.

Lemma ge_is_inverse_le : ∀ ‘{Le A}, ge = inverse le.
Proof using. extens×. Qed.

Lemma lt_is_strict_le : ∀ ‘{Le A}, lt = strict le.
Proof using. extens×. Qed.

Lemma gt_is_inverse_lt : ∀ ‘{Le A}, gt = inverse lt.
Proof using. extens×. Qed.

Lemma gt_is_inverse_strict_le : ∀ ‘{Le A}, gt = inverse (strict le).
Proof using. extens. intros. rewrite gt_is_inverse_lt. rewrite× lt_is_strict_le. Qed.

Global Opaque ge_of_le lt_of_le gt_of_le.

Local tactic rew_to_le

Hint Rewrite @gt_is_inverse_strict_le @ge_is_inverse_le @lt_is_strict_le : rew_to_le.

Tactic Notation "rew_to_le" :=
  autorewrite with rew_to_le in *.

Hint Rewrite @ge_is_inverse_le @gt_is_inverse_lt : rew_to_le_lt.

Tactic Notation "rew_to_le_lt" :=
  autorewrite with rew_to_le_lt in *.

Lemma gt_is_strict_inverse_le : ∀ ‘{Le A},
  gt = strict (inverse le).
Proof using. intros. rew_to_le. apply inverse_strict. Qed.

Lemma le_is_rclosure_lt : ∀ ‘{Le A},
  refl le →
  le = rclosure lt.
Proof using. intros. rew_to_le. rewrite¬ rclosure_strict. Qed.

Lemma le_is_inverse_ge : ∀ ‘{Le A},
  le = inverse ge.

```

```

Proof using. intros. rew_to_le. rewrite¬ inverse_inverse. Qed.

Lemma lt_is_inverse_gt :  $\forall \{Le A\},$ 
  lt = inverse gt.
Proof using. intros. rew_to_le. rewrite¬ inverse_inverse. Qed.

Lemma gt_is_strict_ge :  $\forall \{Le A\},$ 
  gt = strict ge.
Proof using. intros. rew_to_le. apply inverse_strict. Qed.

Lemma ge_is_rclosure_gt :  $\forall \{Le A\},$ 
  refl le →
  ge = rclosure gt.
Proof using. intros. rewrite gt_is_strict_ge. rewrite¬ rclosure_strict. Qed.

```

11.3 Classes for comparison properties

11.3.1 Definition of classes

symmetric structure

```

Class Ge_preorder ‘{Le A} : Prop :=
  { ge_preorder : preorder ge }.
Class Ge_total_preorder ‘{Le A} : Prop :=
  { ge_total_preorder : total_preorder ge }.
Class Ge_order ‘{Le A} : Prop :=
  { ge_order : order ge }.
Class Ge_total_order ‘{Le A} : Prop :=
  { ge_total_order : total_order le }.
Class Gt_strict_order ‘{Le A} : Prop :=
  { gt_strict_order : strict_order gt }.
Class Gt_strict_total_order ‘{Le A} : Prop :=
  { gt_strict_total_order : strict_total_order gt }.

```

properties of le

```

Class Le_refl ‘{Le A} :=
  { le_refl : refl le }.
Class Le_trans ‘{Le A} :=
  { le_trans : trans le }.
Class Le_antisym ‘{Le A} :=
  { le_antisym : antisym le }.
Class Le_total ‘{Le A} :=
  { le_total : total le }.

```

properties of ge

```

Class Ge_refl ‘{Ge A} :=

```

```

{ ge_refl : refl ge }.
Class Ge_trans '{Ge A} :=
{ ge_trans : trans ge }.
Class Ge_antisym '{Ge A} :=
{ ge_antisym : antisym ge }.
Class Ge_total '{Ge A} :=
{ ge_total : total ge }.

properties of lt

Class Lt_irrefl '{Lt A} :=
{ lt_irrefl : irrefl lt }.
Class Lt_trans '{Lt A} :=
{ lt_trans : trans lt }.

properties of gt

Class Gt_irrefl '{Gt A} :=
{ gt_irrefl : irrefl gt }.
Class Gt_trans '{Gt A} :=
{ gt_trans : trans gt }.

mixed transitivity results

Class Lt_le_trans '{Le A} :=
{ lt_le_trans :  $\forall y \ x \ z, x < y \rightarrow y \leq z \rightarrow x < z$  }.
Class Le_lt_trans '{Le A} :=
{ le_lt_trans :  $\forall y \ x \ z, x \leq y \rightarrow y < z \rightarrow x < z$  }.
Class Gt_ge_trans '{Le A} :=
{ gt_ge_trans :  $\forall y \ x \ z, x > y \rightarrow y \geq z \rightarrow x > z$  }.
Class Ge_gt_trans '{Le A} :=
{ ge_gt_trans :  $\forall y \ x \ z, x \geq y \rightarrow y > z \rightarrow x > z$  }.

Arguments lt_irrefl [A] [H] [Lt_irrefl].
Arguments le_trans {A} {H} {Le_trans} y [x] [z].
Arguments ge_trans {A} {H} {Ge_trans} y [x] [z].
Arguments lt_trans {A} {H} {Lt_trans} y [x] [z].
Arguments gt_trans {A} {H} {Gt_trans} y [x] [z].
```

conversion between operators

```

Class Ge_as_sle '{Le A} : Prop :=
{ ge_as_sle :  $\forall x \ y : A, (x \geq y) = (y \leq x)$  }.
Class Gt_as_slt '{Le A} : Prop :=
{ gt_as_slt :  $\forall x \ y : A, (x > y) = (y < x)$  }.
Class Ngt_as_sle '{Le A} : Prop :=
{ ngt_as_sle :  $\forall x \ y : A, (\neg x < y) = (y \leq x)$  }.
Class Nlt_as_ge '{Le A} : Prop :=
```

```

{ nlt_as_ge :  $\forall x y : A, (\neg x < y) = (x \geq y)$  }.

Class Ngt_as_le '{Le A} : Prop :=
{ ngt_as_le :  $\forall x y : A, (\neg x > y) = (x \leq y)$  }.

Class Nle_as_gt '{Le A} : Prop :=
{ nle_as_gt :  $\forall x y : A, (\neg x \leq y) = (x > y)$  }.

Class Nge_as_lt '{Le A} : Prop :=
{ nge_as_lt :  $\forall x y : A, (\neg x \geq y) = (x < y)$  }.

inclusion between operators

Class Lt_to_le '{Le A} : Prop :=
{ lt_to_le :  $\forall x y : A, (x < y) \rightarrow (x \leq y)$  }.

Class Gt_to_ge '{Le A} : Prop :=
{ gt_to_ge :  $\forall x y : A, (x > y) \rightarrow (x \geq y)$  }.

Class Nle_to_sle '{Le A} : Prop :=
{ nle_to_sle :  $\forall x y : A, (\neg x \leq y) \rightarrow (y \leq x)$  }.

Class Nle_to_slt '{Le A} : Prop :=
{ nle_to_slt :  $\forall x y : A, (\neg x \leq y) \rightarrow (y < x)$  }.

case analysis

Class Case_eq_lt_gt '{Le A} : Prop :=
{ case_eq_lt_gt :  $\forall x y : A, x = y \vee x < y \vee x > y$  }.

Class Case_eq_lt_slt '{Le A} : Prop :=
{ case_eq_lt_slt :  $\forall x y : A, x = y \vee x < y \vee y < x$  }.

Class Case_le_gt '{Le A} : Prop :=
{ case_le_gt :  $\forall x y : A, x \leq y \vee x > y$  }.

Class Case_le_slt '{Le A} : Prop :=
{ case_le_slt :  $\forall x y : A, x \leq y \vee y < x$  }.

Class Case_lt_ge '{Le A} : Prop :=
{ case_lt_ge :  $\forall x y : A, x < y \vee x \geq y$  }.

Class Case_lt_sle '{Le A} : Prop :=
{ case_lt_sle :  $\forall x y : A, x < y \vee y \leq x$  }.

case analysis under one assumption

Class Neq_case_lt_gt '{Le A} : Prop :=
{ neq_case_lt_gt :  $\forall x y : A, x \neq y \rightarrow x < y \vee x > y$  }.

Class Neq_case_lt_slt '{Le A} : Prop :=
{ neq_case_lt_slt :  $\forall x y : A, x \neq y \rightarrow x < y \vee y < x$  }.

Class Le_case_eq_lt '{Le A} : Prop :=
{ le_case_eq_lt :  $\forall x y : A, x \leq y \rightarrow x = y \vee x < y$  }.

Class Ge_case_eq_gt '{Le A} : Prop :=

```

```
{ ge_case_eq_gt : ∀ x y : A, x ≥ y → x = y ∨ x > y }.
```

case analysis under two assumptions

```
Class Le_neq_to_lt '{Le A} : Prop :=  
{ le_neq_to_lt : ∀ x y : A, x ≤ y → x ≠ y → x < y }.
```

```
Class Ge_neq_to_gt '{Le A} : Prop :=  
{ ge_neq_to_gt : ∀ x y : A, x ≥ y → x ≠ y → x > y }.
```

```
Class Nlt_nslt_to_eq '{Le A} : Prop :=  
{ nlt_nslt_to_eq : ∀ x y : A, ¬ (lt x y) → ¬ (lt y x) → x = y }.
```

contradiction from case analysis

```
Class Lt_ge_false '{Le A} : Prop :=  
{ lt_ge_false : ∀ x y : A, x < y → x ≥ y → False }.
```

```
Class Lt_gt_false '{Le A} : Prop :=  
{ lt_gt_false : ∀ x y : A, x < y → x > y → False }.
```

```
Class Lt_slt_false '{Le A} : Prop :=  
{ lt_slt_false : ∀ x y : A, x < y → y < x → False }.
```

Section Instances.

Context '{Le A}.

```
Ltac auto_star ::= try solve [ dauto ].
```

derived structures

```
Global Instance Le_preorder_of_Le_order :
```

```
Le_order →
```

```
Le_preorder.
```

Proof using. constructor. intros. apply× order_to_preorder. Qed.

```
Global Instance Le_total_preorder_of_Le_total_order :
```

```
Le_total_order →
```

```
Le_total_preorder.
```

Proof using. constructor. intros. apply× total_order_to_total_preorder. Qed.

```
Global Instance Le_preorder_of_Total_preorder :
```

```
Le_total_preorder →
```

```
Le_preorder.
```

Proof using. constructor. intros. apply× total_preorder_to_preorder. Qed.

```
Global Instance Le_order_of_Le_total_order :
```

```
Le_total_order →
```

```
Le_order.
```

Proof using. constructor. intros. apply× total_order_to_order. Qed.

```
Global Instance Lt_strict_order_of_Lt_strict_total_order :
```

```
Lt_strict_total_order →
```

Lt.strict_order.

Proof using. constructor. intros. apply strict_total_order_to_strict_order. Qed.

Global Instance Lt.strict_order_of_Le_order :

Le_order →

Lt.strict_order.

Proof using. constructor. intros. rew_to_le. apply strict_order_strict. Qed.

Global Instance Lt.strict_total_order_of_Le_total_order :

Le_total_order →

Lt.strict_total_order.

Proof using. constructor. intros. rew_to_le. apply strict_total_order_of_total_order.

Qed.

symmetric structures

Global Instance Ge_preorder_of_Le_order :

Le_order →

Ge_preorder.

Proof using. constructor. rew_to_le. apply preorder_inverse. apply le_preorder. Qed.

Global Instance Ge_total_preorder_of_Le_total_order :

Le_total_order →

Ge_total_preorder.

Proof using. constructor. rew_to_le. apply total_preorder_inverse. apply le_total_preorder.

Qed.

Global Instance Ge_preorder_of_Total_preorder :

Le_total_preorder →

Ge_preorder.

Proof using. constructor. rew_to_le. apply preorder_inverse. apply le_preorder. Qed.

Global Instance Ge_order_of_Le_total_order :

Le_total_order →

Ge_order.

Proof using. constructor. rew_to_le. apply order_inverse. apply le_order. Qed.

Global Instance Gt.strict_order_of_lt.strict_total_order :

Lt.strict_total_order →

Gt.strict_order.

Proof using. constructor. rewrite gt_is_inverse_lt. apply strict_order_inverse. apply lt.strict_order. Qed.

Global Instance Gt.strict_order_of_Le_order :

Le_order →

Gt.strict_order.

Proof using. constructor. rewrite gt_is_inverse_lt. apply strict_order_inverse. apply lt.strict_order. Qed.

Global Instance Gt.strict_total_order_of_Le_total_order :

```

Le_total_order →
Gt_strict_total_order.
Proof using. constructor. rewrite gt_is_inverse_lt. apply strict_total_order_inverse.
apply lt_strict_total_order. Qed.

properties of le

Global Instance Le_refl_of_Le_preorder :
Le_preorder →
Le_refl.
Proof using. intros [[Re Tr]]. constructor¬. Qed.

Global Instance Le_trans_of_Le_preorder :
Le_preorder →
Le_trans.
Proof using. intros [[Re Tr]]. constructor¬. Qed.

Global Instance Le_antisym_of_Le_order :
Le_order →
Le_antisym.
Proof using. constructor. intros. apply× order_antisym. Qed.

Global Instance Le_total_of_Le_total_order :
Le_total_order →
Le_total.
Proof using. constructor. intros. apply× total_order_total. Qed.

properties of ge

Global Instance Ge_refl_of_Le_preorder :
Le_preorder →
Ge_refl.
Proof using. constructor. rew_to_le. apply refl_inverse. apply le_refl. Qed.

Global Instance Ge_trans_of_Le_preorder :
Le_preorder →
Ge_trans.
Proof using. constructor. rew_to_le. apply trans_inverse. apply le_trans. Qed.

Global Instance Ge_antisym_of_Le_order :
Le_order →
Ge_antisym.
Proof using. constructor. rew_to_le. apply antisym_inverse. apply le_antisym. Qed.

Global Instance Ge_total_of_Le_total_order :
Le_total_order →
Ge_total.
Proof using. constructor. rew_to_le. apply total_inverse. apply le_total. Qed.

properties of lt

```

```

Global Instance Lt_irrefl_of_Le_order :
  Le_order →
  Lt_irrefl.

Proof using. constructor. apply strict_order_irrefl. apply lt_strict_order. Qed.

Global Instance Lt_trans_of_Le_order :
  Le_order →
  Lt_trans.

Proof using. constructor. apply strict_order_trans. apply lt_strict_order. Qed.

  properties of gt

Global Instance Gt_irrefl_of_Le_order :
  Le_order →
  Gt_irrefl.

Proof using. constructor. apply strict_order_irrefl. apply gt_strict_order. Qed.

Global Instance Gt_trans_of_Le_order :
  Le_order →
  Gt_trans.

Proof using. constructor. apply strict_order_trans. apply gt_strict_order. Qed.

  mixed transitivity results

Global Instance Lt_le_trans_of_Le_order :
  Le_order →
  Lt_le_trans.

Proof using.
  constructor. introv K L. rew_to_le. destruct K as [U V].
  split¬. apply× le_trans. intro_subst. apply V. apply× le_antisym.

Qed.

Global Instance Le_lt_trans_of_Le_order :
  Le_order →
  Le_lt_trans.

Proof using.
  constructor. introv K L. rew_to_le. destruct L as [U V].
  split¬. apply× le_trans. intro_subst. apply V. apply× le_antisym.

Qed.

Global Instance gt_ge_trans_of_Le_order :
  Le_order →
  Gt_ge_trans.

Proof using.
  constructor. introv K L. rew_to_le_lt. hnf in *. apply× le_lt_trans.

Qed.

Global Instance ge_gt_trans_of_Le_order :
  Le_order →

```

Ge_gt_trans.

Proof using.

```
constructor. introv K L. rew_to_le_lt. hnf in *. apply× lt_le_trans.
```

Qed.

conversion between operators

Global Instance Ge_as_sle_of :

Ge_as_sle.

Proof using. constructor. intros. rew_to_le. auto. Qed.

Global Instance Gt_as_slt_of :

Gt_as_slt.

Proof using. constructor. intros. rew_to_le. auto. Qed.

Global Instance Ngt_as_sle_of_Le_total_order :

Le_total_order →

Ngt_as_sle.

Proof using.

```
constructor. intros. rew_to_le. unfold strict. rew_logic. iff M.
```

destruct M.

forwards K:(inverse_strict_of_not (R:=le)); eauto.

apply le_total. apply (proj1 K).

subst. apply le_refl.

apply or_classic_l. intros P Q. apply P. apply× le_antisym.

Qed.

Global Instance Nlt_as_ge_of_Le_total_order :

Le_total_order →

Nlt_as_ge.

Proof using. constructor. intros. rew_to_le_lt. unfold inverse. apply ngt_as_sle. Qed.

Global Instance Ngt_as_le_of_Le_total_order :

Le_total_order →

Ngt_as_le.

Proof using. constructor. intros. rew_to_le_lt. unfold inverse. apply ngt_as_sle. Qed.

Global Instance Nle_as_gt_of_Le_total_order :

Le_total_order →

Nle_as_gt.

Proof using.

```
constructor. intros. rew_to_le_lt. unfold inverse.
```

rewrite ← ngt_as_sle. rewrite¬ not_not_eq.

Qed.

Global Instance Nge_as_lt_of_Le_total_order :

Le_total_order →

Nge_as_lt.

Proof using.

```

constructor. intros. rew_to_le_lt. unfold inverse.
rewrite nle_as_gt. rewrite¬ gt_is_inverse_lt.
Qed.

inclusion between operators

Global Instance Lt_to_le_of :
  Lt_to_le.

Proof using. constructor. intros. rew_to_le. unfolds× strict. Qed.

Global Instance Gt_to_ge_of :
  Gt_to_ge.

Proof using. constructor. intros. rew_to_le. unfolds× inverse, strict. Qed.

Global Instance Nle_to_sle_of_Le_total_order :
  Le_total_order →
  Nle_to_sle.

Proof using.
  constructor. introv K. rewrite nle_as_gt in K.
  rew_to_le. unfolds× inverse, strict.
Qed.

Global Instance Nle_to_slt_of_Le_total_order :
  Le_total_order →
  Nle_to_slt.

Proof using.
  constructor. introv K. rewrite nle_as_gt in K.
  rew_to_le. unfolds× inverse, strict.
Qed.

case analysis under no assumption

Global Instance Case_eq_lt_gt_of_Le_total_order :
  Le_total_order →
  Case_eq_lt_gt.

Proof using.
  introv K. constructor. intros.
  let [(M1&M2)|(M|(M1&M2))]: (total_order_lt_or_eq_or_gt le_total_order x y).
    rewrite le_is_rclosure_lt in M1 by applys× total_order_refl. destruct× M1.
    autos×.
    rewrite le_is_rclosure_lt in M1 by applys× total_order_refl. destruct× M1.
Qed.

Global Instance Case_eq_lt_slt_of_Le_total_order :
  Le_total_order →
  Case_eq_lt_slt.

Proof using.
  constructor. intros. pattern lt at 2. rewrite lt_is_inverse_gt.

```

```

apply case_eq_lt_gt.
Qed.

Global Instance Case_le_gt_of_Le_total_order :
  Le_total_order →
  Case_le_gt.
Proof using.
  constructor. intros.
  rewrite le_is_rclosure_lt by applys× total_order_refl. rewrite rclosure_eq.
  branches (total_order_lt_or_eq_or_gt le_total_order x y); eauto.
Qed.

Global Instance Case_eq_lt_ge_of_Le_total_order :
  Le_total_order →
  Case_lt_ge.
Proof using.
  constructor. intros.
  rewrite ge_is_rclosure_gt by applys× total_order_refl. rewrite rclosure_eq.
  branches (total_order_lt_or_eq_or_gt le_total_order x y); eauto.
Qed.

Global Instance Case_le_slt_of_Le_total_order :
  Le_total_order →
  Case_le_slt.
Proof using. constructor. intros. rewrite lt_is_inverse_gt. apply case_le_gt. Qed.

Global Instance Case_eq_lt_sle_of_Le_total_order :
  Le_total_order →
  Case_lt_sle.
Proof using. constructor. intros. rewrite le_is_inverse_ge. apply case_lt_ge. Qed.
  case analysis under one assumption

Global Instance Neq_case_lt_gt_of_Le_total_order :
  Le_total_order →
  Neq_case_lt_gt.
Proof using. constructor. intros. destruct× (case_eq_lt_gt x y). Qed.

Global Instance Neq_case_lt_slt_of_Le_total_order :
  Le_total_order →
  Neq_case_lt_slt.
Proof using. constructor. intros. destruct× (case_eq_lt_gt x y). Qed.

Global Instance Le_case_eq_lt_of_Le_total_order :
  Le_total_order →
  Le_case_eq_lt.
Proof using. constructor. intros. rew_to_le. unfold strict. tests*: (x = y). Qed.

Global Instance Ge_case_eq_gt_of_Le_total_order :

```

Le_total_order →

Ge_case_eq_gt.

Proof using. constructor. intros. *rew_to_le*. unfold inverse, strict. *tests**: ($x = y$). Qed.

case analysis under two assumptions

Global Instance Le_neq_to_lt_of_Le_total_order :

Le_total_order →

Le_neq_to_lt.

Proof using. constructor. intros. *rew_to_le*. *hnfs* ×. Qed.

Global Instance Ge_neq_to_gt_of_Le_total_order :

Le_total_order →

Ge_neq_to_gt.

Proof using. constructor. intros. *rew_to_le*. *hnfs* ×. Qed.

Global Instance Nlt_nslt_to_eq_of_Le_total_order :

Le_total_order →

Nlt_nslt_to_eq.

Proof using. constructor. intros. *branches* × (case_eq_lt_gt $x y$). Qed.

contradiction from case analysis

Global Instance Lt_ge_false_of_Le_total_order :

Le_total_order →

Lt_ge_false.

Proof using. constructor. *introv H1 H2*. *rewrite* ↵ ← *nlt_as_ge* in $H2$. Qed.

Global Instance Lt_gt_false_of_Le_total_order :

Le_total_order →

Lt_gt_false.

Proof using.

constructor. *introv H1 H2*. *rewrite* ↵ ← *nle_as_gt* in $H2$.

apply $H2$. apply × *lt_to_le*.

Qed.

Global Instance Lt_slt_false_of_Le_total_order :

Le_total_order →

Lt_slt_false.

Proof using.

constructor. *introv H1 H2*. *rewrite* ↵ ← *gt_as_slt* in $H2$.

apply × *lt_gt_false*.

Qed.

End Instances.

Other lemmas needs arguments to be implicit?

11.4 Boolean comparison

Module BOOLEANCOMPARISON.

Open Scope comp_scope.

Additional notation for reflected boolean comparison. Use Open Scope *comp_scope_reflect* to use them.

Declare Scope comp_scope_reflect.

```
Notation "x "≤" y" := (isTrue (@le _ _ x y))
  (at level 70, no associativity) : comp_scope_reflect.
Notation "x "≥" y" := (isTrue (@ge _ _ x y))
  (at level 70, no associativity) : comp_scope_reflect.
Notation "x "⟨" y" := (isTrue (@lt _ _ x y))
  (at level 70, no associativity) : comp_scope_reflect.
Notation "x "⟩" y" := (isTrue (@gt _ _ x y))
  (at level 70, no associativity) : comp_scope_reflect.
```

End BOOLEANCOMPARISON.

11.5 Order on relations and on predicates

Lemma order_rel_incl : $\forall A B,$
order (@rel_incl *A B*).

Proof using.

hint refl_rel_incl, antisym_rel_incl, trans_rel_incl.
constructors \times .

Qed.

Lemma order_pred_incl : $\forall A B,$
order (@rel_incl *A B*).

Proof using.

hint refl_rel_incl, antisym_rel_incl, trans_rel_incl.
constructors \times .

Qed.

Definition min ‘{Le *A*} (*n m:A*) : *A* :=
If $n \leq m$ then *n* else *m*.

Lemma min_l : \forall ‘{Le *A*} (*n m:A*),
antisym le \rightarrow
 $n \leq m \rightarrow$
min *n m* = *n*.

Proof using. introv *T M*. unfold min. case_if \times . Qed.

Lemma min_r : \forall ‘{Le *A*} (*n m:A*),

antisym le \rightarrow

$m \leq n \rightarrow$

$\min n m = m.$

Proof using. introv T M. unfold min. case_if \times . Qed.

Definition max ‘{Le A} (n m:A) : A :=

If $n \leq m$ then m else n .

Lemma max_l : \forall ‘{Le A} (n m:A),

antisym le \rightarrow

$n \geq m \rightarrow$

$\max n m = n.$

Proof using. introv T M. unfold max. case_if \times . Qed.

Lemma max_r : \forall ‘{Le A} (n m:A),

antisym le \rightarrow

$n \leq m \rightarrow$

$\max n m = m.$

Proof using. introv T M. unfold max. case_if \times . Qed.

Chapter 12

Library SLF.LibNat

```
Set Implicit Arguments.
Require Export Coq.Arith.Arith Coq.micromega.Lia.
From SLF Require Import LibTactics LibReflect LibBool LibOperation LibRelation LibOrder.
From SLF Require Export LibOrder.
Global Close Scope positive_scope.
```

12.1 Nat type

12.1.1 Definition

From the Prelude:

Inductive nat : Set := | O : nat | S : nat -> nat.

Remark: ideally, constructors would be renamed to *zero* and *succ*, or *nat_zero* and *nat_succ*, with the notations *O* or *0%nat*, and *S n* or *succ n*. It is indeed problematic to prevent the use of single letter variables in pattern matching to the user who does not care about *nat*.

12.1.2 Inhabited

Instance Inhab_nat : Inhab nat.

Proof using. intros. apply (Inhab_of_val 0). Qed.

12.2 Order on natural numbers

12.2.1 Definition

The typeclass instance of *le* on *nat* is defined to be the *le* relation on Peano numbers from Coq's standard library.

```
Instance le_nat_inst : Le nat := Build_Le Peano.le.
```

12.2.2 Translating typeclass instances to Peano relations

These lemmas and tactics are useful to transform arithmetic goals into a form on which the *lia* decision procedure may apply.

```
Lemma le_peano : le = Peano.le.
```

```
Proof using. extens x. Qed.
```

```
Global Opaque le_nat_inst.
```

```
Lemma lt_peano : lt = Peano.lt.
```

```
Proof using.
```

```
  extens. rew_to_le. rewrite le_peano.  
  unfold strict. intros. lia.
```

```
Qed.
```

```
Lemma ge_peano : ge = Peano.ge.
```

```
Proof using.
```

```
  extens. rew_to_le. rewrite le_peano.  
  unfold inverse. intros. lia.
```

```
Qed.
```

```
Lemma gt_peano : gt = Peano.gt.
```

```
Proof using.
```

```
  extens. rew_to_le. rewrite le_peano.  
  unfold strict, inverse. intros. lia.
```

```
Qed.
```

```
Hint Rewrite le_peano lt_peano ge_peano gt_peano : rew_nat_comp.
```

```
Ltac nat_comp_to_peano :=  
  autorewrite with rew_nat_comp in *.
```

nat_math calls *lia* after basic pre-processing (*intros* and *split*) and after replacing comparison operators with the ones defined in *Peano* library.

```
Ltac nat_math_setup :=  
  intros;  
  try match goal with  $\vdash \_ \wedge \_ \Rightarrow$  split end;  
  try match goal with  $\vdash \_ = \_ :> \text{Prop} \Rightarrow$  apply prop_ext; iff end;  
  nat_comp_to_peano.
```

```
Ltac nat_math :=  
  nat_math_setup; lia.
```

12.2.3 The *nat_maths* database is used for registering automation on mathematical goals.

```

Ltac nat_math_hint := nat_math.

Hint Extern 3 (_ = _ :> nat) => nat_math_hint : nat_maths.
Hint Extern 3 (_ ≠ _ :> nat) => nat_math_hint : nat_maths.
Hint Extern 3 (istrue (isTrue (_ = _ :> nat))) => nat_math_hint : nat_maths.
Hint Extern 3 (istrue (isTrue (_ ≠ _ :> nat))) => nat_math_hint : nat_maths.
Hint Extern 3 ((_ ≤ _)%nat) => nat_math_hint : nat_maths.
Hint Extern 3 ((_ ≥ _)%nat) => nat_math_hint : nat_maths.
Hint Extern 3 ((_ < _)%nat) => nat_math_hint : nat_maths.
Hint Extern 3 ((_ > _)%nat) => nat_math_hint : nat_maths.
Hint Extern 3 (@le nat _ _ _) => nat_math_hint : nat_maths.
Hint Extern 3 (@lt nat _ _ _) => nat_math_hint : nat_maths.
Hint Extern 3 (@ge nat _ _ _) => nat_math_hint : nat_maths.
Hint Extern 3 (@gt nat _ _ _) => nat_math_hint : nat_maths.

```

12.2.4 Total order instance

Instance nat_le_total_order : **Le_total_order** ($A:=\text{nat}$).

Proof using.

```

constructor. constructor. constructor; unfolds.
nat_math. nat_math. nat_math.
unfolds. intros. tests: (x ≤ y). left¬. right. nat_math.
Qed.

```

12.3 Induction

Lemma peano_induction :

$$\begin{aligned} \forall (P:\text{nat} \rightarrow \text{Prop}), \\ (\forall n, (\forall m, m < n \rightarrow P m) \rightarrow P n) \rightarrow \\ (\forall n, P n). \end{aligned}$$

Proof using.

```

intros H. cuts× K: (\forall n m, m < n → P m).
nat_comp_to_peano.
induction n; introv Le. inversion Le. apply H.
intros. apply IHn. nat_math.
Qed.

```

Lemma measure_induction :

$$\begin{aligned} \forall (A:\text{Type}) (mu:A \rightarrow \text{nat}) (P:A \rightarrow \text{Prop}), \\ (\forall x, (\forall y, mu y < mu x \rightarrow P y) \rightarrow P x) \rightarrow \\ (\forall x, P x). \end{aligned}$$

Proof using.

```

intros IH. intros x. gen_eq n: (mu x). gen x.

```

induction n using peano_induction. introv Eq. subst \times .
 Qed.

12.4 Simplification lemmas

12.4.1 Trivial monoid simplifications

Lemma plus_zero_r : $\forall n,$
 $n + 0 = n.$

Proof using. nat_math. Qed.

Lemma plus_zero_l : $\forall n,$
 $0 + n = n.$

Proof using. nat_math. Qed.

Lemma minus_zero_r : $\forall n,$
 $n - 0 = n.$

Proof using. nat_math. Qed.

Lemma plus_succ : $\forall n1\ n2,$
 $n1 + S\ n2 = S\ (n1 + n2).$

Proof using. nat_math. Qed.

Lemma minus_zero : $\forall n,$
 $n - 0 = n.$

Proof using. nat_math. Qed.

Lemma succ_minus_succ : $\forall n1\ n2,$
 $S\ n1 - S\ n2 = n1 - n2.$

Proof using. nat_math. Qed.

Lemma minus_same : $\forall n,$
 $n - n = 0.$

Proof using. nat_math. Qed.

Lemma plus_minus_same : $\forall n1\ n2,$
 $n1 + n2 - n1 = n2.$

Proof using. nat_math. Qed.

Lemma mult_zero_l : $\forall n,$
 $0 \times n = 0.$

Proof using. nat_math. Qed.

Lemma mult_zero_r : $\forall n,$
 $n \times 0 = 0.$

Proof using. nat_math. Qed.

Lemma mult_one_l : $\forall n,$
 $1 \times n = n.$

Proof using. *nat_math*. Qed.

Lemma mult_one_r : $\forall n,$
 $n \times 1 = n$.

Proof using. *nat_math*. Qed.

12.4.2 Simplification tactic

rew_nat performs some basic simplification on expressions involving natural numbers

Hint Rewrite plus_zero_r plus_zero_l minus_zero_r
plus_succ minus_zero succ_minus_succ minus_same plus_minus_same
mult_zero_l mult_zero_r mult_one_l mult_one_r : *rew_nat*.

Tactic Notation "rew_nat" :=
autorewrite with *rew_nat*.

Tactic Notation "rew_nat" " \sim " :=
rew_nat; auto_tilde.

Tactic Notation "rew_nat" " $*$ " :=
rew_nat; auto_star.

Tactic Notation "rew_nat" "in" " $*$ " :=
autorewrite_in_star_patch ltac:(fun tt \Rightarrow autorewrite with *rew_nat*).

Tactic Notation "rew_nat" " \sim " "in" " $*$ " :=
rew_nat in *; auto_tilde.

Tactic Notation "rew_nat" " $*$ " "in" " $*$ " :=
rew_nat in *; auto_star.

Tactic Notation "rew_nat" "in" *hyp(H)* :=
autorewrite with *rew_nat* in *H*.

Tactic Notation "rew_nat" " \sim " "in" *hyp(H)* :=
rew_nat in *H*; auto_tilde.

Tactic Notation "rew_nat" " $*$ " "in" *hyp(H)* :=
rew_nat in *H*; auto_star.

12.5 Executable functions

Fixpoint beq (*x y* : **nat**) :=
match *x y* with
| O, O \Rightarrow true
| S *x'*, S *y'* \Rightarrow beq *x' y'*
| -, - \Rightarrow false
end.

Lemma beq_eq : $\forall n1\ n2,$
beq *n1 n2* = isTrue (*n1 = n2*).

Proof using.

```
intros n1. induction n1; intros; destruct n2; simpl; rew_bool_eq; auto_false.  
rewrite IHn1. extens. rew_istrue. nat_math.  
Qed.
```

Chapter 13

Library SLF.LibEpsilon

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibRelation.

Generalizable Variables A B .

13.1 Definition and specification of Hilbert's epsilon operator

13.1.1 Definition of epsilon

epsilon P where P is a predicate over an inhabited type A , returns a value x of type A that satisfies P , if there exists one such value, else it returns an arbitrary value of type A .

Definition epsilon_def : $\forall A \{IA:\mathbf{Inhab} A\} (P:A \rightarrow \text{Prop})$,
 $\{x : A \mid (\exists y, P y) \rightarrow P x\}$.

Proof using.

```
intros A IA P. destruct (classicT ( $\exists y, P y$ )) as [H|H].
{ apply indefinite_description. destruct H as [x H].
   $\exists x.$  intros .. apply H.
  {  $\exists$  (@arbitrary A IA). intros N. false H. apply N. }}
```

Qed.

Definition epsilon A {IA: **Inhab** A} (P:A → Prop) : A :=
sig_val (epsilon_def P).

Lemma pred_epsilon : $\forall A \{IA:\mathbf{Inhab} A\} (P:A \rightarrow \text{Prop})$,
 $(\exists x, P x) \rightarrow P (\text{epsilon } P)$.

Proof using. intros. apply \neg (sig_proof (epsilon_def P)). Qed.

Opaque epsilon.

Definition epsilon_def' A {IA: **Inhab** A} (P:A → Prop) :

```

{ x : A | ( $\exists$  y, P y)  $\rightarrow$  P x } := 
match classicT ( $\exists$  y, P y) with
| left H  $\Rightarrow$ 
  indefinite_description
    (let (x,H0) := H in
      ex_intro (fun x0  $\Rightarrow$  ( $\exists$  y, P y)  $\rightarrow$  P x0)
        x
        (fun N  $\Rightarrow$  H0))
| right H  $\Rightarrow$ 
  exist (fun x  $\Rightarrow$  ( $\exists$  y, P y)  $\rightarrow$  P x)
    arbitrary
    (fun N  $\Rightarrow$  False_ind (P arbitrary) (H N))
end.

```

13.1.2 Lemmas about epsilon

Lemma pred_epsilon_weaken : $\forall A \{IA:\mathbf{Inhab} A\} (P Q : A \rightarrow \text{Prop})$,
 $(\exists x, P x) \rightarrow$
 $(\forall x, P x \rightarrow Q x) \rightarrow$
 $Q (\text{epsilon } P)$.

Proof using. introv E M. apply M. apply× pred_epsilon. Qed.

Lemma pred_epsilon_of_val : $\forall A (x:A) (P:A \rightarrow \text{Prop}) \{IA:\mathbf{Inhab} A\}$,
 $P x \rightarrow$
 $P (\text{epsilon } P)$.

Proof using. intros. apply× pred_epsilon. Qed.

Lemma pred_epsilon_of_val_weaken : $\forall A (x:A) (P Q:A \rightarrow \text{Prop}) \{IA:\mathbf{Inhab} A\}$,
 $P x \rightarrow$
 $(\forall x, P x \rightarrow Q x) \rightarrow$
 $Q (\text{epsilon } P)$.

Proof using. introv Px W. apply W. apply× pred_epsilon. Qed.

Lemma epsilon_eq : $\forall A \{IA:\mathbf{Inhab} A\} (P Q:A \rightarrow \text{Prop})$,
 $(\forall x, P x \leftrightarrow Q x) \rightarrow$
 $\text{epsilon } P = \text{epsilon } Q$.

Proof using. introv H. fequals. extens×. Qed.

13.1.3 (Private) tactic *epsilon_find*

epsilon_find cont locates an expression of the form `epsilon P` in the goal and invokes the continuation *cont* on *P*.

epsilon_find_in H cont is similar but looks for the expression only in the hypothesis named *H*.

```

Ltac epsilon_find cont :=
  match goal with
  | ⊢ context [epsilon ?P] ⇒ cont P
  | H: context [epsilon ?P] ⊢ _ ⇒ cont P
  end.

Ltac epsilon_find_in H cont :=
  match type of H with context [epsilon ?P] ⇒ cont P end.

```

13.1.4 Tactics *epsilon_name*

epsilon_name X assigns a name *X* to an expression of the form *epsilon P* that appears in the goal or an hypothesis, by calling **set** (*X := epsilon P*).

epsilon_name X in H assignes a name *X* to an expression of the form *epsilon P* that appears in hypothesis *H*.

```

Ltac epsilon_name_core X :=
  epsilon_find ltac:(fun P ⇒ sets X: (epsilon P)).

```

```

Ltac epsilon_name_in_core X H :=
  epsilon_find_in H ltac:(fun P ⇒ sets X: (epsilon P)).

```

```

Tactic Notation "epsilon_name" ident(X) :=
  epsilon_name_core X.

```

```

Tactic Notation "epsilon_name" ident(X) "in" hyp(H) :=
  epsilon_name_in_core X H.

```

13.1.5 Tactics to work with *epsilon*

epsilon X locates an expression of the form *epsilon P* in the goal, names *X* this expression (like *epsilon_name X*), then produces a subgoal $\exists x, P x$, and leaves at the head of the main goal an hypothesis *P X*.

epsilon X in H is similar, but looks for *epsilon P* only in hypothesis *H*.

```

Lemma pred_epsilon' : ∀ A (P:A→Prop) (IA:Inhab A),
  ( $\exists x, P x$ ) →
  P (epsilon P).

```

Proof using. intros. *applys*× *pred_epsilon*. Qed.

```

Ltac epsilon_cont X P :=
  let I := fresh "H" X in
  let I: (@pred_epsilon' _ P) _ _ _ _;
  [ | sets X: (epsilon P); revert I ].
```

```

Ltac epsilon_core X :=
  epsilon_find ltac:(fun P ⇒ epsilon_cont X P).

```

```

Ltac epsilon_in_core X H :=

```

```

epsilon_find_in H ltac:(fun P => epsilon_cont X P).

Tactic Notation "epsilon" ident(X) :=
  epsilon_core X.

Tactic Notation "epsilon" ident(X) "in" hyp(H) :=
  epsilon_in_core X H.

Tactic Notation "epsilon" "~~" ident(X) :=
  epsilon X; auto_tilde.

Tactic Notation "epsilon" "~~" ident(X) "in" hyp(H) :=
  epsilon X in H; auto_tilde.

Tactic Notation "epsilon" "*" ident(X) :=
  epsilon X; auto_star.

Tactic Notation "epsilon" "*" ident(X) "in" hyp(H) :=
  epsilon X in H; auto_star.

```

13.2 Construction of a function from a relation, using epsilon

Definition rel_to_fun A '{IB:**Inhab** B} (R:A→B→Prop) : A → B :=
 fun (a:A) => epsilon (fun b => R a b).

Section Rel_to_fun.

Context (A B : Type) {IB:**Inhab** B}.

Implicit Types R : A → B → Prop.

Lemma rel_rel_to_fun_of_exists : ∀ R a,
 (exists b, R a b) →
 R a (rel_to_fun R a).

Proof using IB. introv [x H]. unfold rel_to_fun. epsilon× y. Qed.

Lemma rel_rel_to_fun_of_rel : ∀ R a b,
 R a b →
 R a (rel_to_fun R a).

Proof using IB. intros. applys× rel_rel_to_fun_of_exists. Qed.

Lemma rel_rel_to_fun_of_not_forall : ∀ R a,
 ¬ (forall b, ¬ R a b) →
 R a (rel_to_fun R a).

Proof using IB.
 introv. rew_logic. intros [x H]. applys× rel_rel_to_fun_of_exists.

Qed.

Lemma rel_in_fun_rel_to_fun : ∀ R,
 functional R →

`rel_in_fun R (rel_to_fun R).`

`Proof using IB. unfold rel_in_fun, rel_to_fun. introv M H. epsilon× z. Qed.`

Reformulation of above

`Lemma rel_to_fun_eq_of_functional : ∀ x y R,`

`functional R →`

`R x y →`

`rel_to_fun R x = y.`

`Proof using IB. introv M E. applys× rel_in_fun_rel_to_fun. Qed.`

`End Rel_to_fun.`

Chapter 14

Library SLF.LibInt

```
Set Implicit Arguments.
Require Export Coq.ZArith.ZArith.
From SLF Require Import LibTactics LibLogic LibReflect LibRelation.
Export LibTacticsCompatibility.
From SLF Require Export LibNat.
```

14.1 Parsing of integers and operations

14.1.1 Notation for type and operation

Define *int* as an alias for *Z*, the type of integers from Coq's stdlib.

Declare Scope Int_scope.

Notation "'int'" := Z : Int_scope.

Delimit Scope Int_scope with I.

Local Open Scope Int_scope.

14.1.2 Inhabited type

Instance Inhab_int : Inhab int.

Proof using. intros. apply (Inhab_of_val 0%Z). Qed.

14.1.3 Coercion from nat

Remark: we cannot simply use the coercion: Coercion Z_of_nat : nat >-> Z. because otherwise when we try to make the coercion opaque using: Opaque Z_of_nat. the lia fails to work. Thus, we introduce an alias, called *nat_to_Z* for *Z_of_nat*, and we register *nat_to_Z* as coercion.

```

Definition nat_to_Z := Z_of_nat.

Lemma nat_to_Z_eq_Z_of_nat : nat_to_Z = Z_of_nat.
Proof using reflexivity. Qed.

Global Opaque nat_to_Z.

Coercion nat_to_Z : nat >-> Z.

```

14.1.4 Order relation

The comparison operators on integers are those from `LibOrder`, not the ones from Coq's `ZArith`.

`Open Scope Z_scope.`

`Open Scope comp_scope.`

The typeclass `le` on type `int` is bound to `Zle`, from Coq's standard library

```
Instance le_int_inst : Le int := Build_Le Z.le.
```

14.2 Conversion to natural numbers, for tactic programming

These tactics are helpful to convert a number passed to a Ltac tactic into a `nat`, regardless of whether it is a `nat` or an `int`.

```

Definition ltac_int_to_nat (x:Z) : nat :=
  match x with
  | Z0 => 0%nat
  | Zpos p => nat_of_P p
  | Zneg p => 0%nat
  end.

Ltac number_to_nat N :=
  match type of N with
  | nat => constr:(N)
  | int => let N' := constr:(ltac_int_to_nat N) in eval compute in N'
  | Z => let N' := constr:(ltac_int_to_nat N) in eval compute in N'
  end.

```

14.3 Decision procedure

A lot of hacks to allow calling the `lia` tactic

14.3.1 Translation from typeclass order to ZArith, for using *lia*

```
Lemma le_zarith : le = Z.le.
Proof using. extens x. Qed.

Global Opaque le_int_inst.

Lemma lt_zarith : lt = Z.lt.
Proof using.
  extens. rew_to_le. rewrite le_zarith.
  unfold strict. intros. lia.
Qed.

Lemma ge_zarith : ge = Z.ge.
Proof using.
  extens. rew_to_le. rewrite le_zarith.
  unfold inverse. intros. lia.
Qed.

Lemma gt_zarith : gt = Z.gt.
Proof using.
  extens. rew_to_le. rewrite le_zarith.
  unfold strict, inverse. intros. lia.
Qed.

Hint Rewrite le_zarith lt_zarith ge_zarith gt_zarith : rew_int_comp.

Ltac int_comp_to_zarith :=
  autorewrite with rew_int_comp in *.
```

14.3.2 Hypothesis selection

is_arith_type T returns a boolean indicating whether *T* is equal to **nat** or *int*

```
Ltac is_arith_type T :=
  match T with
  | nat => constr:(true)
  | int => constr:(true)
  | _ => constr:(false)
  end.
```

is_arith E returns a boolean indicating whether *E* is an arithmetic expression

```
Ltac is_arith E :=
  match E with
  | _ = _ :> ?T => is_arith_type T
  | _ ≠ _ :> ?T => is_arith_type T
  | ¬ ?E' => is_arith E'
  | ?E' → False => is_arith E'
```

```

| @le ?T _ _ _ ⇒ is_arith_type T
| @ge ?T _ _ _ ⇒ is_arith_type T
| @lt ?T _ _ _ ⇒ is_arith_type T
| @gt ?T _ _ _ ⇒ is_arith_type T
| (_ ≤ _)%Z ⇒ constr:(true)
| (_ ≥ _)%Z ⇒ constr:(true)
| (_ < _)%Z ⇒ constr:(true)
| (_ > _)%Z ⇒ constr:(true)
| (_ ≤ _)%nat ⇒ constr:(true)
| (_ ≥ _)%nat ⇒ constr:(true)
| (_ < _)%nat ⇒ constr:(true)
| (_ > _)%nat ⇒ constr:(true)
| _ ⇒ constr:(false)
end.

```

arith_goal_or_false looks at the current goal and replaces it with **False** if it is not an arithmetic goal

```

Ltac arith_goal_or_false :=
  match goal with ⊢ ?E ⇒
    match is_arith E with
    | true ⇒ idtac
    | false ⇒ false
    end
  end.

```

generalize_arith generalizes all hypotheses which correspond to some arithmetic goal. It destructs conjunctions on the fly.

```

Lemma istrue_isTrue_forw : ∀ (P:Prop),
  istrue (isTrue P) →
  P.

```

Proof using. *introv H. rew_istrue¬ in H. Qed.*

```

Ltac generalize_arith :=
  repeat match goal with
  | H: istrue (isTrue _) ⊢ _ ⇒ generalize (@istrue_isTrue_forw _ H); clear H; intro
  | H:?E1 ∧ ?E2 ⊢ _ ⇒ destruct H
  | H: ?E → False ⊢ _ ⇒
    match is_arith E with
    | true ⇒ change (E → False) with (¬ E) in H
    | false ⇒ clear H
    end
  | H:?E ⊢ _ ⇒
    match is_arith E with
    | true ⇒ generalize H; clear H
    
```

```

| false ⇒ clear  $H$ 
end
end.
```

14.3.3 Normalization of arithmetic expressions

Two lemmas to help lia out

Lemma Z_of_nat_O :

$$\text{Z_of_nat } 0 = 0.$$

Proof using. reflexivity. Qed.

Lemma Z_of_nat_S : $\forall n,$

$$\text{Z_of_nat } (S n) = \text{Z.succ} (\text{Z.of_nat } n).$$

Proof using. intros. rewrite \leftarrow **Zpos_P_of_succ_nat**. Qed.

Lemma Z_of_nat_plus1 : $\forall n,$

$$\text{Z.of_nat } (1 + n) = \text{Z.succ} (\text{Z.of_nat } n).$$

Proof using. intros. rewrite \leftarrow **Z_of_nat_S.** *fequals* \leftarrow . Qed.

rew_maths rewrite any lemma in the base *rew_maths*. The goal should not contain any evar, otherwise tactic might loop.

Hint Rewrite nat_to_Z_eq_Z_of_nat Z_of_nat_O Z_of_nat_S Z_of_nat_plus1 : *rew_maths*.

Ltac *rew_maths* :=

autorewrite with *rew_maths* in *.

14.3.4 Setting up the goal for *lia*

math_setup_goal does introduction, splits, and replace the goal by **False** if it is not arithmetic. If the goal is of the form $P1 = P2 :> \text{Prop}$, then the goal is changed to $P1 \leftrightarrow P2$.

Ltac *math_setup_goal_step tt* :=

```

match goal with
|  $\vdash \_ \rightarrow \_ \Rightarrow \text{intro}$ 
|  $\vdash \_ \leftrightarrow \_ \Rightarrow \text{iff}$ 
|  $\vdash \forall \_, \_ \Rightarrow \text{intro}$ 
|  $\vdash \_ \wedge \_ \Rightarrow \text{split}$ 
|  $\vdash \_ = \_ :> \text{Prop} \Rightarrow \text{apply prop_ext; iff}$ 
end.
```

Ltac *math_setup_goal* :=

```

repeat (math_setup_goal_step tt);
arith_goal_or_false.
```

math tactics performs several preprocessing step, selects all arithmetic hypotheses, and the call *lia*.

14.3.5 Main driver for the set up process to goal *lia*

```

Ltac check_noevar_goal ::= 
  match goal with ⊢ ?G ⇒ first [ has_evar G; fail 1 | idtac ] end.

Ltac math_0 := idtac.
Ltac math_1 := intros; generalize_arith.
Ltac math_2 := instantiate; check_noevar_goal.
Ltac math_3 := autorewrite with rew_maths rew_int_comp rew_nat_comp; intros.
Ltac math_4 := math_setup_goal.
Ltac math_5 := lia.

Ltac math_setup := math_0; math_1; math_2; math_3; math_4.
Ltac math_base := math_setup; math_5.

Tactic Notation "math" := math_base.

Tactic Notation "math" simple_intropattern(I) ":" constr(E) :=
  asserts I : E; [ math | ].
Tactic Notation "maths" constr(E) :=
  let H := fresh "H" in asserts H : E; [ math | ].
```

14.3.6 *math* tactic restricted to arithmetic goals

math_only calls *math* but only on goals which have an arithmetic form. Thus, contrasty to *math*, it does not attempt to look for a contradiction in the hypotheses if the conclusion is not an arithmetic goal. Useful for efficiency.

```

Ltac math_only_if_arith_core tt :=
  match goal with ⊢ ?T ⇒
    match is_arith T with true ⇒ math end end.

Tactic Notation "math_only_if_arith" :=
  math_only_if_arith_core tt.
```

14.3.7 Elimination of multiplication, to call *lia*

```

Lemma mult_2_eq_plus : ∀ x, 2 × x = x + x.
Proof using. intros. ring. Qed.

Lemma mult_3_eq_plus : ∀ x, 3 × x = x + x + x.
Proof using. intros. ring. Qed.
```

14.3.8 Hint externs for calling *math* in the hint base *maths*

```
Ltac math_hint := math.
```

```

Hint Extern 3 (_ = _ :> nat) ⇒ math_hint : maths.
Hint Extern 3 (_ = _ :> int) ⇒ math_hint : maths.
Hint Extern 3 (_ ≠ _ :> nat) ⇒ math_hint : maths.
Hint Extern 3 (_ ≠ _ :> int) ⇒ math_hint : maths.
Hint Extern 3 (istrue (isTrue (_ = _ :> nat))) ⇒ math_hint : maths.
Hint Extern 3 (istrue (isTrue (_ = _ :> int))) ⇒ math_hint : maths.
Hint Extern 3 (istrue (isTrue (_ ≠ _ :> nat))) ⇒ math_hint : maths.
Hint Extern 3 (istrue (isTrue (_ ≠ _ :> int))) ⇒ math_hint : maths.
Hint Extern 3 ((_ ≤ _)%nat) ⇒ math_hint : maths.
Hint Extern 3 ((_ ≥ _)%nat) ⇒ math_hint : maths.
Hint Extern 3 ((_ < _)%nat) ⇒ math_hint : maths.
Hint Extern 3 ((_ > _)%nat) ⇒ math_hint : maths.
Hint Extern 3 ((_ ≤ _)%Z) ⇒ math_hint : maths.
Hint Extern 3 ((_ ≥ _)%Z) ⇒ math_hint : maths.
Hint Extern 3 ((_ < _)%Z) ⇒ math_hint : maths.
Hint Extern 3 ((_ > _)%Z) ⇒ math_hint : maths.
Hint Extern 3 (@le nat _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (@lt nat _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (@ge nat _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (@gt nat _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (@le int _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (@lt int _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (@ge int _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (@gt int _ _ _) ⇒ math_hint : maths.
Hint Extern 3 (¬ @le int _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @lt int _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @ge int _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @gt int _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @eq int _ _ ) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (@le int _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (@lt int _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (@ge int _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (@gt int _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (@eq int _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (¬ @le nat _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @lt nat _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @ge nat _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @gt nat _ _ _) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (¬ @eq nat _ _ ) ⇒ unfold not; math_hint : maths.
Hint Extern 3 (@le nat _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (@lt nat _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (@ge nat _ _ → False) ⇒ math_hint : maths.

```

```

Hint Extern 3 (@gt nat _ _ _ → False) ⇒ math_hint : maths.
Hint Extern 3 (@eq nat _ _ → False) ⇒ math_hint : maths.

```

14.4 Rewriting on arithmetic expressions

14.4.1 Rewriting equalities provable by the *math* tactic

math_rewrite ($E = F$) replaces all occurrences of E with the expression F . It produces the equality $E = F$ as subgoal, and tries to solve it using the tactic *math*

```

Tactic Notation "math_rewrite" constr(E) :=
  asserts_rewrite E; [ try math | ].
Tactic Notation "math_rewrite" constr(E) "in" hyp(H) :=
  asserts_rewrite E in H; [ try math | ].
Tactic Notation "math_rewrite" constr(E) "in" "*" :=
  asserts_rewrite E in *; [ try math | ].
Tactic Notation "math_rewrite" "~" constr(E) :=
  math_rewrite E; auto_tilde.
Tactic Notation "math_rewrite" "~" constr(E) "in" hyp(H) :=
  math_rewrite E in H; auto_tilde.
Tactic Notation "math_rewrite" "~" constr(E) "in" "*" :=
  math_rewrite E in *; auto_tilde.
Tactic Notation "math_rewrite" "*" constr(E) :=
  math_rewrite E; auto_star.
Tactic Notation "math_rewrite" "*" constr(E) "in" hyp(H) :=
  math_rewrite E in H; auto_star.
Tactic Notation "math_rewrite" "*" constr(E) "in" "*" :=
  math_rewrite E in *; auto_star.

```

14.4.2 Addition and subtraction

```

Lemma plus_zero_r : ∀ n,
  n + 0 = n.

```

Proof using. math. Qed.

```

Lemma plus_zero_l : ∀ n,
  0 + n = n.

```

Proof using. math. Qed.

```

Lemma minus_zero_r : ∀ n,
  n - 0 = n.

```

Proof using. math. Qed.

```

Lemma minus_zero_l : ∀ n,

```

```

0 - n = (-n).
Proof using. math. Qed.

Lemma mult_zero_l : ∀ n,
  0 × n = 0.
Proof using. math. Qed.

Lemma mult_zero_r : ∀ n,
  n × 0 = 0.
Proof using. math. Qed.

Lemma mult_one_l : ∀ n,
  1 × n = n.
Proof using. math. Qed.

Lemma mult_one_r : ∀ n,
  n × 1 = n.
Proof using. math. Qed.

```

14.4.3 Simplification tactic

`rew_int` performs some basic simplification on expressions involving integers

```

Hint Rewrite plus_zero_r plus_zero_l minus_zero_r minus_zero_l
      mult_zero_l mult_zero_r mult_one_l mult_one_r : rew_int.

Tactic Notation "rew_int" :=
  autorewrite with rew_int.

Tactic Notation "rew_int" "~~" :=
  rew_int; auto_tilde.

Tactic Notation "rew_int" "*" :=
  rew_int; auto_star.

Tactic Notation "rew_int" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_int).

Tactic Notation "rew_int" "~~" "in" "*" :=
  rew_int in *; auto_tilde.

Tactic Notation "rew_int" "*" "in" "*" :=
  rew_int in *; auto_star.

Tactic Notation "rew_int" "in" hyp(H) :=
  autorewrite with rew_int in H.

Tactic Notation "rew_int" "~~" "in" hyp(H) :=
  rew_int in H; auto_tilde.

Tactic Notation "rew_int" "*" "in" hyp(H) :=
  rew_int in H; auto_star.

```

14.5 Conversions of operations from **nat** to *int* and back

LATER: make proofs below no longer depend on Coq's stdlib

14.5.1 Lifting of comparisons from **nat** to *int*

```
Lemma eq_nat_of_eq_int : ∀ (n m:nat),  
  (n:int) = (m:int) →  
  n = m :> nat.
```

Proof using. math. Qed.

```
Lemma neq_nat_of_neq_int : ∀ (n m:nat),  
  (n:int) ≠ (m:int) →  
  (n ≠ m)%nat.
```

Proof using. math. Qed.

```
Lemma eq_int_of_eq_nat : ∀ (n m:nat),  
  n = m :> nat →  
  (n:int) = (m:int).
```

Proof using. math. Qed.

```
Lemma neq_int_of_neq_nat : ∀ (n m:nat),  
  (n ≠ m)%nat →  
  (n:int) ≠ (m:int).
```

Proof using. math. Qed.

14.5.2 Lifting of inequalities from **nat** to *int*

```
Lemma le_nat_of_le_int : ∀ (n m:nat),  
  (n:int) ≤ (m:int) →  
  (n ≤ m).
```

Proof using. math. Qed.

```
Lemma le_int_of_le_nat : ∀ (n m:nat),  
  (n ≤ m) →  
  (n:int) ≤ (m:int).
```

Proof using. math. Qed.

```
Lemma lt_nat_of_lt_int : ∀ (n m:nat),  
  (n:int) < (m:int) →  
  (n < m).
```

Proof using. math. Qed.

```
Lemma lt_int_of_lt_nat : ∀ (n m:nat),  
  (n < m) →  
  (n:int) < (m:int).
```

Proof using. *math*. Qed.

Lemma ge_nat_of_ge_int : $\forall (n m:\text{nat}),$
 $(n:\text{int}) \geq (m:\text{int}) \rightarrow$
 $(n \geq m).$

Proof using. *math*. Qed.

Lemma ge_int_of_ge_nat : $\forall (n m:\text{nat}),$
 $(n \geq m) \rightarrow$
 $(n:\text{int}) \geq (m:\text{int}).$

Proof using. *math*. Qed.

Lemma gt_nat_of_gt_int : $\forall (n m:\text{nat}),$
 $(n:\text{int}) > (m:\text{int}) \rightarrow$
 $(n > m).$

Proof using. *math*. Qed.

Lemma gt_int_of_gt_nat : $\forall (n m:\text{nat}),$
 $(n > m) \rightarrow$
 $(n:\text{int}) > (m:\text{int}).$

Proof using. *math*. Qed.

14.5.3 Lifting of operations from **nat** to *int*

Lemma plus_nat_eq_plus_int : $\forall (n m:\text{nat}),$
 $(n+m)\%{\text{nat}} = (n:\text{int}) + (m:\text{int}) :> \text{int}.$

Proof using.

Transparent nat_to_Z.
intros. unfold nat_to_Z. *applys* Nat2Z.inj_add.

Qed.

Lemma minus_nat_eq_minus_int : $\forall (n m:\text{nat}),$
 $(n \geq m)\%{\text{nat}} \rightarrow$
 $(n-m)\%{\text{nat}} = (n:\text{int}) - (m:\text{int}) :> \text{int}.$

Proof using.

Transparent nat_to_Z.
intros. unfold nat_to_Z. *applys* Nat2Z.inj_sub. *math*.

Qed.

Hint Rewrite plus_nat_eq_plus_int : *rew_maths*.

14.5.4 Properties of comparison

Lemma antisym_le_int :
antisym (le (A:=int)).

Proof using. intros x y L1 L2. *math*. Qed.

14.5.5 Absolute function in `nat`

Notation "'abs'" := `Z.abs_nat` (at level 0).

Global Arguments `Z.abs` : simpl never.

Global Arguments `Z.abs_nat` : simpl never.

Lemma `abs_nat` : $\forall (n:\text{nat})$,
 $\text{abs } n = n$.

Proof using. exact `Zabs_nat_Z_of_nat`. Qed.

Lemma `abs_nonneg` : $\forall (x:\text{int})$,
 $x \geq 0 \rightarrow$
 $\text{abs } x = x :> \text{int}$.

Proof using.

```
intros. rewrite inj_Zabs_nat.
rewrite Z.abs_eq. math. math.
```

Qed.

Lemma `abs_eq_nat_eq` : $\forall (x:\text{int}) (y:\text{nat})$,
 $x \geq 0 \rightarrow$
 $(\text{abs } x = y :> \text{nat}) = (x = Z_of_nat y :> \text{int})$.

Proof using.

```
introv M. extens. iff E.
{ subst. rewrite Zabs2Nat.id_abs, Z.abs_eq; math. }
{ subst. rewrite¬ Zabs2Nat.id. }
```

Qed.

Lemma `lt_abs_abs` : $\forall (n m : \text{int})$,
 $(0 \leq n) \rightarrow$
 $(n < m) \rightarrow$
 $(\text{abs } n < \text{abs } m)$.

Proof using.

```
intros. nat_comp_to_peano. apply Zabs_nat_lt. math.
```

Qed.

Lemma `abs_to_int` : $\forall (n : \text{int})$,
 $0 \leq n \rightarrow$
 $Z_of_nat (\text{abs } n) = n$.

Proof using. intros. rewrite¬ abs_nonneg. Qed.

Lemma `abs_le_nat_le` : $\forall (x:\text{int}) (y:\text{nat})$,
 $(0 \leq x) \rightarrow$
 $(\text{abs } x \leq y) = (x \leq y) \% Z$.

Proof.

```
intros. extens. iff E.
{ rewrites¬ <- (» abs_to_int x). math. }
{ rewrites¬ <- (» abs_to_int x) in E. math. }
```

Qed.

Lemma abs_ge_nat_ge : $\forall (x:\text{int}) (y:\text{nat}),$
 $(0 \leq x) \rightarrow$
 $(\text{abs } x \geq y) = (x \geq y)\%Z .$

Proof.

intros. extens. iff E.
{ rewrites $\neg \leftarrow (\text{abs_to_int } x)$. math. }
{ rewrites $\neg \leftarrow (\text{abs_to_int } x)$ in E. math. }

Qed.

TODO: many useful lemmas missing

14.5.6 **abs** distribute on constants and operators

Lemma abs_0 : $\text{abs } 0 = 0\%\text{nat} :> \text{nat}.$

Proof using. reflexivity. Qed.

Lemma abs_1 : $\text{abs } 1 = 1\%\text{nat} :> \text{nat}.$

Proof using. reflexivity. Qed.

Lemma abs_plus : $\forall (x y:\text{int}),$

$(x \geq 0) \rightarrow$
 $(y \geq 0) \rightarrow$
 $\text{abs } (x + y) = (\text{abs } x + \text{abs } y)\%\text{nat} :> \text{nat}.$

Proof using. intros. applys Zabs2Nat.inj_add; math. Qed.

Lemma abs_minus : $\forall (x y:\text{int}),$

$(x \geq y) \rightarrow$
 $(y \geq 0) \rightarrow$
 $\text{abs } (x - y) = (\text{abs } x - \text{abs } y)\%\text{nat} :> \text{nat}.$

Proof using. intros. applys Zabs2Nat.inj_sub; math. Qed.

Lemma abs_nat_plus_nonneg : $\forall (n:\text{nat}) (x:\text{int}),$

$x \geq 0 \rightarrow$
 $\text{abs } (n + x)\%Z = (n + \text{abs } x)\%\text{nat}.$

Proof using.

introv N. applys eq_nat_of_eq_int.
rewrite plus_nat_eq_plus_int.
do 2 (rewrite abs_nonneg; [|math|]). auto.

Qed.

Lemma abs_gt_minus_nat : $\forall (n:\text{nat}) (x:\text{int}),$

$(x \geq n)\%Z \rightarrow$
 $\text{abs } (x - n)\%Z = (\text{abs } x - n)\%\text{nat}.$

Proof using.

introv N. applys eq_nat_of_eq_int.

```

rewrite minus_nat_eq_minus_int.
do 2 (rewrite abs_nonneg; [|math|]). auto.
applys ge_nat_of_ge_int. rewrite abs_nonneg; math.
Qed.

```

Lemma succ_abs_eq_abs_one_plus : $\forall (x:\text{int}),$
 $x \geq 0 \rightarrow$
 $S(\text{abs } x) = \text{abs}(1 + x) :> \text{nat}.$

Proof using.

```

intros x. pattern x. applys (@measure_induction _ abs). clear x.
intros x IH Pos. rewrite ← Zabs_nat_Zsucc. fequals. math. math.

```

Qed.

Lemma abs_eq_succ_abs_minus_one : $\forall (x:\text{int}),$
 $x > 0 \rightarrow$
 $\text{abs } x = S(\text{abs } (x - 1)) :> \text{nat}.$

Proof using.

```

intros. apply eq_nat_of_eq_int.
rewrite abs_nonneg; try math.

```

Qed.

14.5.7 Tactic *rew_abs_nonneg* to normalize expressions involving **abs**

issuing side-conditions that arguments are nonnegative

Hint Rewrite abs_nat abs_0 abs_1 abs_plus abs_nonneg : *rew_abs_nonneg*.

Tactic Notation "rew_abs_nonneg" :=
autorewrite with *rew_abs_nonneg*.

Tactic Notation "rew_abs_nonneg" " \sim " :=
autorewrite with *rew_abs_nonneg*; try *math*; *autos* \neg .

14.5.8 Positive part of an integer. Returns 0 on negative values.

Notation "'to_nat'" := **Z.to_nat** (at level 0).

Lemma to_nat_nat : $\forall (n:\text{nat}),$
 $\text{to_nat } n = n.$

Proof using. exact **Nat2Z.id**. Qed.

Lemma to_nat_nonneg : $\forall (x:\text{int}),$
 $x \geq 0 \rightarrow$
 $\text{to_nat } x = x :> \text{int}.$

Proof using. intros. apply \neg **Z2Nat.id**. Qed.

Lemma to_nat_neg : $\forall (x:\text{int}),$

$x \leq 0 \rightarrow$
 $\text{to_nat } x = 0\%nat.$

Proof using.

```
intros x H. destruct x.
assert (Z.pos p = 0) as →. { forwards: Zle_0_pos p. math. }
reflexivity.
```

Qed.

Lemma to_nat_eq_nat_eq : $\forall (x:\text{int}) (y:\text{nat}),$
 $x \geq 0 \rightarrow$
 $(\text{to_nat } x = y :> \text{nat}) = (x = \text{Z_of_nat } y :> \text{int}).$

Proof using.

```
introv M. extens. iff E.
{ subst. rewrite ↵ Z2Nat.id. }
{ subst. rewrite ↵ Nat2Z.id. }
```

Qed.

Lemma lt_to_nat_to_nat : $\forall (n m : \text{int}),$
 $(0 \leq n) \rightarrow$
 $(n < m) \rightarrow$
 $(\text{to_nat } n < \text{to_nat } m).$

Proof using.

```
intros. nat_comp_to_peano.
rewrite <-!Zabs2Nat.abs_nat_nonneg by math.
apply ↵ Zabs_nat_lt. math.
```

Qed.

Lemma to_nat_to_int : $\forall (n : \text{int}),$
 $0 \leq n \rightarrow$
 $\text{Z_of_nat } (\text{to_nat } n) = n.$

Proof using. intros. rewrite ↵ to_nat_nonneg. Qed.

Lemma to_nat_le_nat_le : $\forall (x:\text{int}) (y:\text{nat}),$
 $(0 \leq x) \rightarrow$
 $(\text{to_nat } x \leq y) = (x \leq y)\%Z.$

Proof.

```
intros. extens. iff E.
{ rewrites ↵ <-(» to_nat_to_int x). math. }
{ rewrites ↵ <-(» to_nat_to_int x) in E. math. }
```

Qed.

Lemma to_nat_ge_nat_ge : $\forall (x:\text{int}) (y:\text{nat}),$
 $(0 \leq x) \rightarrow$
 $(\text{to_nat } x \geq y) = (x \geq y)\%Z .$

Proof.

```
intros. extens. iff E.
```

```

{ rewrites¬ <-(» to_nat_to_int x). math. }
{ rewrites¬ <-(» to_nat_to_int x) in E. math. }
Qed.

```

14.5.9 **to_nat** distribute on constants and operators

Lemma `to_nat_0 : to_nat 0 = 0%nat :> nat.`

Proof using. reflexivity. Qed.

Lemma `to_nat_1 : to_nat 1 = 1%nat :> nat.`

Proof using. reflexivity. Qed.

Lemma `to_nat_plus : ∀ (x y:int),`

`(x ≥ 0) →`

`(y ≥ 0) →`

`to_nat (x + y) = (to_nat x + to_nat y)%nat :> nat.`

Proof using. intros. apply¬ `Z2Nat.inj_add`. Qed.

Lemma `to_nat_minus : ∀ (x y:int),`

`(x ≥ y) →`

`(y ≥ 0) →`

`to_nat (x - y) = (to_nat x - to_nat y)%nat :> nat.`

Proof using. intros. apply¬ `Z2Nat.inj_sub`. Qed.

14.5.10 Tactic *rew_to_nat_nonneg* to normalize expressions involving **to_nat**

issuing side-conditions that arguments are nonnegative

```
Hint Rewrite to_nat_nat to_nat_0 to_nat_1 to_nat_plus to_nat_minus
      to_nat_nonneg : rew_to_nat_nonneg.
```

```
Tactic Notation "rew_to_nat_nonneg" :=
  autorewrite with rew_to_nat_nonneg.
```

```
Tactic Notation "rew_to_nat_nonneg" "~~" :=
  autorewrite with rew_to_nat_nonneg; try math; autos¬.
```

Chapter 15

Library `SLF.LibMonoid`

DISCLAIMER: the current presentation of monoids uses typeclasses, but in fact it's not obvious that typeclasses are needed/useful here. Indeed, there is no overloading involved. Thus, the interface might change in the near future.

15.1

15.2

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibOperation.
Generalizable Variables A B .

15.3 Monoids

15.4 Structures

Monoid structure: binary operator and neutral element

```
Record monoid_op ( $A$ :Type) : Type := monoid_make {  
  monoid_oper :  $A \rightarrow A \rightarrow A$ ;  
  monoid_neutral :  $A$  }.
```

Monoid properties Note that field names are suffixed by $_prop$ because the corresponding properties are also available through typeclass instances.

```
Class Monoid  $A$  ( $m$ :monoid_op  $A$ ) : Prop := Monoid_make {  
  monoid_assoc_prop : let  $(o,n)$  :=  $m$  in assoc  $o$ ;  
  monoid_neutral_l_prop : let  $(o,n)$  :=  $m$  in neutral_l  $o$   $n$ ;
```

```

monoid_neutral_r_prop : let (o,n) := m in neutral_r o n }.

Commutative monoid

Class Comm_monoid A (m:monoid_op A) : Prop := Comm_monoid_make {
  comm_monoid_monoid : Monoid m;
  comm_monoid_comm : let (o,n) := m in comm o }.

```

15.5 Examples

Example:

Instance monoid_plus_zero: Monoid (monoid_make plus 0). Proof using. constructor; repeat intro; lia. Qed.

15.6 Properties

```

Section MonoidProp.

Context {A:Type}.

Class Monoid_assoc {m:monoid_op A} := {
  monoid_assoc : assoc (monoid_oper m) }.

Class Monoid_neutral_l {m:monoid_op A} := {
  monoid_neutral_l : neutral_l (monoid_oper m) (monoid_neutral m) }.

Class Monoid_neutral_r {m:monoid_op A} := {
  monoid_neutral_r : neutral_r (monoid_oper m) (monoid_neutral m) }.

Class Monoid_comm {m:monoid_op A} := {
  monoid_comm : comm (monoid_oper m) }.

End MonoidProp.

```

15.7 Derived Properties

```

Section MonoidInst.

Variables (A:Type).

Implicit Types m : monoid_op A.

Global Instance Monoid_assoc_of_Monoid : ∀ m (M:Monoid m),
  Monoid_assoc (m:=m).

Proof using.
  introv M. constructor. destruct M as [U ? ?]. destruct m. simpl. apply U.
Qed.

Global Instance Monoid_neutral_l_of_Monoid : ∀ m (M:Monoid m),

```

Monoid_neutral_l ($m := m$).

Proof using.

introv M . constructor. destruct M as [? U ?]. destruct m . simpl. apply U .

Qed.

Global Instance Monoid_neutral_r_of_Monoid : $\forall m (M:\mathbf{Monoid} m),$

Monoid_neutral_r ($m := m$).

Proof using.

introv M . constructor. destruct M as [? ? U]. destruct m . simpl. apply U .

Qed.

Global Instance Monoid_of_Comm_monoid : $\forall m (M:\mathbf{Comm_monoid} m),$

Monoid m .

Proof using.

introv M . destruct M as [U ?]. destruct m . simpl. apply U .

Qed.

Global Instance Monoid_comm_of_Comm_Monoid : $\forall m (M:\mathbf{Comm_monoid} m),$

Monoid_comm ($m := m$).

Proof using.

introv M . constructor. destruct M as [? U]. destruct m . simpl. apply U .

Qed.

End MonoidInst.

Chapter 16

Library SLF.LibContainer

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibReflect

LibRelation LibOperation LibInt LibMonoid.

Generalizable Variables $A B K T$.

16.1 Operators

16.1.1 Definitions

```
Class BagEmpty  $T := \{ \text{empty} : T \}$ .
Class BagSingle  $A T := \{ \text{single} : A \rightarrow T \}$ .
Class BagSingleBind  $A B T := \{ \text{single\_bind} : A \rightarrow B \rightarrow T \}$ .
Class BagIn  $A T := \{ \text{is\_in} : A \rightarrow T \rightarrow \text{Prop} \}$ .
Class BagBinds  $A B T := \{ \text{binds} : T \rightarrow A \rightarrow B \rightarrow \text{Prop} \}$ .
Class BagRead  $A B T := \{ \text{read} : T \rightarrow A \rightarrow B \}$ .
Class BagUpdate  $A B T := \{ \text{update} : T \rightarrow A \rightarrow B \rightarrow T \}$ .
Class BagUnion  $T := \{ \text{union} : T \rightarrow T \rightarrow T \}$ .
Class BagInter  $T := \{ \text{inter} : T \rightarrow T \rightarrow T \}$ .
Class BagIncl  $T := \{ \text{incl} : \text{binary } T \}$ .
Class BagDisjoint  $T := \{ \text{disjoint} : \text{binary } T \}$ .
Class BagRestrict  $T K := \{ \text{restrict} : T \rightarrow K \rightarrow T \}$ .
Class BagRemove  $T K := \{ \text{remove} : T \rightarrow K \rightarrow T \}$ .
Class BagFold  $I F T := \{ \text{fold} : \text{monoid\_op } I \rightarrow F \rightarrow T \rightarrow I \}$ .
Class BagCard  $T := \{ \text{card} : T \rightarrow \text{nat} \}$ .
Class BagDom  $T K := \{ \text{dom} : T \rightarrow K \}$ .
Class BagImg  $T K := \{ \text{img} : T \rightarrow K \}$ .
Class BagIndex  $A T := \{ \text{index} : T \rightarrow A \rightarrow \text{Prop} \}$ .
```

Definition notin ‘ $\{\text{BagIn } A T\}$ ’ $x m :=$

$\neg (\text{is_in } x \ m).$

16.1.2 Notation

Declare Scope `container_scope`.

Notation " $\{\}$ " := (`empty`)

: `container_scope`.

Notation " $\{ \ x \ \}$ " := (`single` x)

: `container_scope`.

Notation " $x \backslash\text{in}' m$ " := (`is_in` $x \ m$)

(at level 39) : `container_scope`.

Notation " $x \backslash\text{notin}' E$ " := (`notin` $x \ E$)

(at level 39) : `container_scope`.

Notation " $x \ \backslash:= v$ " := (`single_bind` $x \ v$)

(at level 29) : `container_scope`.

Notation " $m \ [\ x \]$ " := (`read` $m \ x$)

(at level 7, format " $m \ [\ x \]$ ", left associativity).

Notation " $m \ [\ x := v \]$ " := (`update` $m \ x \ v$)

(at level 7, format " $m \ [\ x := v \]$ ", left associativity).

Notation " $m1 \ \backslash c' m2$ " := (`incl` $m1 \ m2$)

(at level 38) : `container_scope`.

Notation " $m1 \ \backslash u' m2$ " := (`union` $m1 \ m2$)

(at level 37, right associativity) : `container_scope`.

Notation " $m1 \ \backslash n' m2$ " := (`inter` $m1 \ m2$)

(at level 36, right associativity) : `container_scope`.

Notation " $m1 \ \backslash -' m2$ " := (`remove` $m1 \ m2$)

(at level 35) : `container_scope`.

Notation " $m1 \ \backslash |' m2$ " := (`restrict` $m1 \ m2$)

(at level 34) : `container_scope`.

Notation " $m1 \ \backslash \#' m2$ " := (`disjoint` $m1 \ m2$)

(at level 37, right associativity) : `container_scope`.

Open Scope `container_scope`.

Notation " $\{ \ x \ \}$ " := (`single` x) (format " $\{ \ x \ \}$ ")

: `container_scope`.

Notation " $M \ \backslash - i$ " := ($M \ \backslash - \ \backslash \{i\}$) (at level 35) : `container_scope`.

16.1.3 $\forall x \ \backslash\text{in} \ E, P$ x notation

Notation "'forall_-' x 'in' E ',' P" :=

($\forall x, x \ \backslash\text{in} \ E \rightarrow P$)

```

(at level 200, x ident) : container_scope.
Notation "'forall_-' x y '\in' E ',' P" :=
  ( $\forall x y, x \in E \rightarrow y \in E \rightarrow P$ )
(at level 200, x ident, y ident) : container_scope.
Notation "'forall_-' x y z '\in' E ',' P" :=
  ( $\forall x y z, x \in E \rightarrow y \in E \rightarrow z \in E \rightarrow P$ )
(at level 200, x ident, y ident, z ident) : container_scope.

```

16.1.4 $\exists x \in E \text{ st } P$ x notation

```

Notation "'exists_-' x '\in' E ',' P" :=
  ( $\exists x, x \in E \wedge P$ )
(at level 200, x ident) : container_scope.
Notation "'exists_-' x y '\in' E ',' P" :=
  ( $\exists x, x \in E \wedge y \in E \wedge P$ )
(at level 200, x ident, y ident) : container_scope.
Notation "'exists_-' x y z '\in' E ',' P" :=
  ( $\exists x, x \in E \wedge y \in E \wedge z \in E \wedge P$ )
(at level 200, x ident, y ident, z ident) : container_scope.

```

16.1.5 Foreach

```

Definition foreach '{BagIn A T} (P:A→Prop) (E:T) :=
   $\forall x, x \in E \rightarrow P x.$ 

```

16.1.6 index for natural numbers

Local Open Scope Int_scope.

Instance int_index : BagIndex int int.

Proof using. intros. constructor. exact (fun n (i:int) ⇒ 0 ≤ i < n). Defined.

Lemma int_index_eq : $\forall (n i : \text{int})$,
 $\text{index } n i = (0 \leq i < n).$

Proof using. auto. Qed.

Global Opaque int_index.

Lemma int_index_le : $\forall i n m : \text{int}$,
 $\text{index } n i \rightarrow n \leq m \rightarrow \text{index } m i.$

Proof using. introv. do 2 rewrite @int_index_eq. math. Qed.

Lemma int_index_prove : $\forall (n i : \text{int})$,
 $0 \leq i \rightarrow i < n \rightarrow \text{index } n i.$

Proof using. intros. rewrite ← int_index_eq. Qed.

```
Lemma int_index_succ : ∀ n i, n ≥ 0 →
  index (n + 1) i = (index n i ∨ i = n).
```

Proof using.

```
  introv P. do 2 rewrite int_index_eq. extens. iff H.
```

```
  apply or_classic_l. math.
```

```
  destruct H; math.
```

Qed.

16.1.7 Derivable

Bag update can be defined as bag union with a singleton bag

```
Instance bag_update_as_union_single : ∀ A B T
  ‘{BagSingleBind A B T} ‘{BagUnion T},
  BagUpdate A B T.
constructor. apply (fun m k v ⇒ m \u2228 (k \u2228 v)). Defined.
```

```
Global Opaque bag_update_as_union_single.
```

16.1.8 Properties

Section Properties.

```
Context {A T:Type}
  {BI: BagIn A T} {BE: BagEmpty T} {BS: BagSingle A T}
  {BN: BagInter T} {BU: BagUnion T} {BR: BagRemove T T} {BC: BagCard T}
  {BL: BagIncl T} {BD: BagDisjoint T}.
```

In

```
Class In_empty_eq :=
  { in_empty_eq : ∀ x, x \in \{} = False }.
Class In_empty :=
  { in_empty : ∀ x, x \in \{} → False }.
Class Notin_eq :=
  { notin_eq : ∀ x E, (x \notin E) = ¬ (x \in E) }.
Class Notin_empty :=
  { notin_empty : ∀ x, x \notin \{} }.
Class In_single_eq :=
  { in_single_eq : ∀ x y, x \in \{y\} = (x = y) }.
Class In_single :=
  { in_single : ∀ x y, x \in \{y\} → x = y }.
Class In_single_self :=
  { in_single_self : ∀ x, x \in \{x\} }.
Class In_extens_eq :=
```

```

{ in_extens_eq :  $\forall E F, (\forall x, x \in E = x \in F) \rightarrow E = F$  }.
Class In_extens :=
{ in_extens :  $\forall E F, (\forall x, x \in E \leftrightarrow x \in F) \rightarrow E = F$  }.

Class Is_empty_eq :=
{ is_empty_eq :  $\forall E, (E = \emptyset) = (\forall x, x \in E \rightarrow \text{False})$  }.
Class Is_empty_prove :=
{ is_empty_prove :  $\forall E, (\forall x, x \in E \rightarrow \text{False}) \rightarrow E = \emptyset$  }.

Class Is_empty_inv :=
{ is_empty_inv :  $\forall x E, E = \emptyset \rightarrow x \in E \rightarrow \text{False}$  }.

Class Is_nonempty_prove :=
{ is_nonempty_prove :  $\forall x E, x \in E \rightarrow E \neq \emptyset$  }.

Class Is_single_eq :=
{ is_single_eq :  $\forall x E, (E = \{x\}) = (x \in E \wedge (\forall y, y \in E \rightarrow y = x))$  }.

Class Is_single_prove :=
{ is_single_prove :  $\forall x E, x \in E \rightarrow (\forall y, y \in E \rightarrow y = x) \rightarrow E = \{x\}$  }.

Class Is_single_inv :=
{ is_single_inv :  $\forall x y E, E = \{x\} \rightarrow y \in E \rightarrow y = x$  }.

Class Notin_single_eq :=
{ notin_single_eq :  $\forall x y, x \notin \{y\} = (x \neq y)$  }.

Class In_inter_eq :=
{ in_inter_eq :  $\forall x E F, x \in (E \cap F) = (x \in E \wedge x \in F)$  }.

Class In_inter :=
{ in_inter :  $\forall x E F, x \in E \rightarrow x \in F \rightarrow x \in (E \cap F)$  }.

Class In_inter_inv :=
{ in_inter_inv :  $\forall x E F, x \in (E \cap F) \rightarrow x \in E \wedge x \in F$  }.

Class Notin_inter_eq :=
{ notin_inter_eq :  $\forall x E F, x \notin (E \cap F) = (x \notin E \vee x \notin F)$  }.

Class Notin_inter_l :=
{ notin_inter_l :  $\forall x E F, x \notin E \rightarrow x \notin (E \cap F)$  }.

Class Notin_inter_r :=
{ notin_inter_r :  $\forall x E F, x \notin F \rightarrow x \notin (E \cap F)$  }.

Class Notin_inter_inv :=
{ notin_inter_inv :  $\forall x E F, x \notin (E \cap F) \rightarrow x \notin E \vee x \notin F$  }.

Class In_union_eq :=
{ in_union_eq :  $\forall x (E F : T), x \in (E \cup F) = (x \in E \vee x \in F)$  }.

Class In_union_l :=
{ in_union_l :  $\forall x E F, x \in E \rightarrow x \in (E \cup F)$  }.

Class In_union_r :=
{ in_union_r :  $\forall x E F, x \in F \rightarrow x \in (E \cup F)$  }.

Class In_union_inv :=
{ in_union_inv :  $\forall x (E F : T), x \in (E \cup F) \rightarrow (x \in E \vee x \in F)$  }.

```

```

Class Notin_union_eq :=
{notin_union_eq : ∀ x E F, x \notin (E ∪ F) = (x \notin E ∧ x \notin F)}.

Class Notin_union :=
{notin_union : ∀ x E F, x \notin E → x \notin F → x \notin (E ∪ F)}.

Class Notin_union_inv :=
{notin_union_inv : ∀ x E F, x \notin (E ∪ F) → x \notin E ∧ x \notin F}.

Class In_remove_eq :=
{in_remove_eq : ∀ x (E F : T), x \in (E \setminus F) = (x \in E ∧ x \notin F)}.

Class Remove_incl :=
{remove_incl : ∀ (E F : T), (E \setminus F) \subset E}.

Class Remove_disjoint :=
{remove_disjoint : ∀ (E F : T), F \# (E \setminus F)}.

Incl

Class Incl_in_eq :=
{incl_in_eq : ∀ E F, (E \subset F) = (∀ x, x \in E → x \in F)}.

Class Incl_prove :=
{incl_prove : ∀ E F, (∀ x, x \in E → x \in F) → E \subset F}.

Class Incl_inv :=
{incl_inv : ∀ x E F, E \subset F → x \in E → x \in F}.

Class Incl_refl :=
{incl_refl : refl incl}.

Class Incl_trans :=
{incl_trans : trans incl}.

Class Incl_antisym :=
{incl_antisym : antisym incl}.

Class Incl_order :=
{incl_order : LibOrder.order incl}.

Class Empty_incl :=
{empty_incl : ∀ E, \{\} \subset E}.

Class Incl_empty :=
{incl_empty : ∀ E, (E \subset \{\}) = (E = \{\})}.

Class Incl_empty_inv :=
{incl_empty_inv : ∀ E, E \subset \{} → E = \{}}.

Class Single_incl_r_eq :=
{single_incl_r_eq : ∀ x E, (\{x\} \subset E) = (x \in E)}.

Class Single_incl_r :=
{single_incl_r : ∀ x E, x \in E → \{x\} \subset E}.

Class Single_incl_l_eq :=
{single_incl_l_eq : ∀ x E, (E \subset \{x\}) = (E = \{\} ∨ E = \{x\})}.

Class Incl_union_l :=

```

```

{ incl_union_l :  $\forall E F G, E \setminus\!\! c F \rightarrow E \setminus\!\! c (F \cup G) \}.$ 
Class Incl_union_r :=
{ incl_union_r :  $\forall E F G, E \setminus\!\! c G \rightarrow E \setminus\!\! c (F \cup G) \}.$ 

Class Union_incl_eq :=
{ union_incl_eq :  $\forall E F G, ((E \cup F) \setminus\!\! c G) = (E \setminus\!\! c G \wedge F \setminus\!\! c G) \}.$ 
Class Union_incl :=
{ union_incl :  $\forall E F G, E \setminus\!\! c G \rightarrow F \setminus\!\! c G \rightarrow (E \cup F) \setminus\!\! c G \}.$ 
Class Union_incl_inv :=
{ union_incl_inv :  $\forall E F G, (E \cup F) \setminus\!\! c G \rightarrow E \setminus\!\! c G \wedge F \setminus\!\! c G \}.$ 

Class Incl_inter_eq :=
{ incl_inter_eq :  $\forall E F G, E \setminus\!\! c (F \cap G) = (E \setminus\!\! c F \wedge E \setminus\!\! c G) \}.$ 
Class Incl_inter :=
{ incl_inter :  $\forall E F G, E \setminus\!\! c F \rightarrow E \setminus\!\! c G \rightarrow E \setminus\!\! c (F \cap G) \}.$ 
Class Incl_inter_inv :=
{ incl_inter_inv :  $\forall E F G, E \setminus\!\! c (F \cap G) \rightarrow E \setminus\!\! c F \wedge E \setminus\!\! c G \}.$ 

Union

Class Union_assoc :=
{ union_assoc : assoc union }.
Class Union_comm :=
{ union_comm : comm union }.
Class Union_comm_assoc :=
{ union_comm_assoc : comm_assoc union }.
Class Union_empty_l :=
{ union_empty_l : neutral_l union empty }.
Class Union_empty_r :=
{ union_empty_r : neutral_r union empty }.
Class Union_empty_inv :=
{ union_empty_inv :  $\forall E F, E \cup F = \{\} \rightarrow E = \{\} \wedge F = \{\} \}.$ 
Class Union_self :=
{ union_self : idempotent2 union }.

Intersection

Class Inter_assoc :=
{ inter_assoc : assoc inter }.
Class Inter_comm :=
{ inter_comm : comm inter }.
Class Inter_comm_assoc :=
{ inter_comm_assoc : comm_assoc inter }.
Class Inter_empty_l :=
{ inter_empty_l : absorb_l inter empty }.
Class Inter_empty_r :=
{ inter_empty_r : absorb_r inter empty }.

```

```

Class Inter_self :=
{ inter_self : idempotent2 inter }.

Removal

Cardinal

Class Card_empty :=
{ card_empty : card \{\} = 0%nat }.

Class Card_single :=
{ card_single :  $\forall X, \text{card } \{X\} = 1\%nat$  }.

Class Card_union :=
{ card_union :  $\forall E F, \text{card } (E \setminus u F) = (\text{card } E + \text{card } F)\%nat$  }.

Class Card_disjoin_union :=
{ card_disjoin_union :  $\forall E F, E \setminus \# F \rightarrow \text{card } (E \setminus u F) = (\text{card } E + \text{card } F)\%nat$  }.

Class Card_union_le :=
{ card_union_le :  $\forall E F, \text{card } (E \setminus u F) \leq (\text{card } E + \text{card } F)\%nat$  }.

Disjointness

Class Disjoint_sym :=
{ disjoint_sym : sym disjoint }.

Class Disjoint_eq :=
{ disjoint_eq :  $\forall E F, (E \setminus \# F) = (\forall x, x \in E \rightarrow x \in F \rightarrow \text{False})$  }.

Class Disjoint_not_eq :=
{ disjoint_not_eq :  $\forall E F, (\neg (E \setminus \# F)) = (\exists x, x \in E \wedge x \notin F)$  }.

Class Disjoint_prove :=
{ disjoint_prove :  $\forall E F, (\forall x, x \in E \rightarrow x \in F \rightarrow \text{False}) \rightarrow E \setminus \# F$  }.

Class Disjoint_inv :=
{ disjoint_inv :  $\forall x E F, (E \setminus \# F) \rightarrow x \in E \rightarrow x \in F \rightarrow \text{False}$  }.

Class Disjoint_single_l_eq :=
{ disjoint_single_l_eq :  $\forall x E, (\setminus \{x\} \setminus \# E) = x \notin E$  }.

Class Disjoint_single_r_eq :=
{ disjoint_single_r_eq :  $\forall x E, (E \setminus \# \setminus \{x\}) = x \notin E$  }.

Class Inter_disjoint :=
{ inter_disjoint :  $\forall E F, E \setminus \# F \rightarrow E \setminus n F = \{\}$  }.

```

End Properties.

Lemmas with premises and operators in the conclusion need additional Arguments

```

Arguments is_empty_inv {A} {T} {BI} {BE} {Is_empty_inv} [x] [E].
Arguments is_nonempty_prove {A} {T} {BI} {BE} {Is_nonempty_prove} [x] [E].
Arguments in_single A T {BI} {BS} {In_single} [x] [y].
Arguments is_single_inv {A} {T} {BI} {BS} {Is_single_inv} [x] y [E].
Arguments in_inter {A} {T} {BI} {BN} {In_inter} [x] [E] [F].
Arguments in_inter_inv {A} {T} {BI} {BN} {In_inter_inv} [x] [E] [F].

```

```

Arguments notin_inter_l {A} {T} {BI} {BN} {Notin_inter_l} [x] [E] [F].
Arguments notin_inter_r {A} {T} {BI} {BN} {Notin_inter_r} [x] [E] [F].
Arguments notin_inter_inv {A} {T} {BI} {BN} {Notin_inter_inv} [x] [E] [F].
Arguments in_union_l {A} {T} {BI} {BU} {In_union_l} [x] [E] [F].
Arguments in_union_r {A} {T} {BI} {BU} {In_union_r} [x] [E] [F].
Arguments in_union_inv {A} {T} {BI} {BU} {In_union_inv} [x] [E] [F].
Arguments notin_union {A} {T} {BI} {BU} {Notin_union} [x] [E] [F].
Arguments notin_union_inv {A} {T} {BI} {BU} {Notin_union_inv} [x] [E] [F].
Arguments incl_probe {A} {T} {BI} {BL} {Incl_probe} [E] [F].
Arguments incl_inv {A} {T} {BI} {BL} {Incl_inv} [x] [E] [F].
Arguments incl_trans {T} {BL} {Incl_trans} y [x] [z].
Arguments incl_empty_inv {T} {BE} {BL} {Incl_empty_inv} [E].
Arguments incl_union_l {T} {BU} {BL} {Incl_union_l} [E] [F] [G].
Arguments incl_union_r {T} {BU} {BL} {Incl_union_r} [E] [F] [G].
Arguments incl_inter {T} {BN} {BL} {Incl_inter} [E] [F] [G].
Arguments incl_inter_inv {T} {BN} {BL} {Incl_inter_inv} [E] [F] [G].
Arguments union_empty_inv {T} {BE} {BU} {Union_empty_inv} [E] [F].
Arguments disjoint_sym {T} {BD} {Disjoint_sym}.
Arguments disjoint_probe {A} {T} {BI} {BD} {Disjoint_probe} [E] [F].
Arguments disjoint_inv {A} {T} {BI} {BD} {Disjoint_inv} [x] [E] [F].
Arguments disjoint_single_l_eq {A} {T} {BI} {BS} {BD} {Disjoint_single_l_eq}.
Arguments disjoint_single_r_eq {A} {T} {BI} {BS} {BD} {Disjoint_single_r_eq}.

```

16.1.9 Derived Properties

Section DerivedProperties.

Context {A T:Type}

```

{BI: BagIn A T} {BE: BagEmpty T} {BS: BagSingle A T}
{BN: BagInter T} {BU: BagUnion T} {BR: BagRemove T T} {BC: BagCard T}
{BL: BagIncl T} {BD: BagDisjoint T}.

```

In

Global Instance in_empty_of_in_empty_eq :

```

In_empty_eq →
In_empty.

```

Proof using. constructor. introv I. rewrite¬ in_empty_eq in I. Qed.

Global Instance notin_eq_of_nothing :

```

Notin_eq.

```

Proof using. constructor. intros. unfold notin. auto. Qed.

Global Instance notin_empty_of_in_empty_eq :

In_empty_eq →
Notin_empty.

Proof using. constructor. introv I . rewrite¬ in_empty_eq in I . Qed.

Global Instance in_single_of_in_single_eq :

In_single_eq →
In_single.

Proof using. constructor. introv I . rewrite¬ in_single_eq in I . Qed.

Global Instance in_single_self_of_in_single_eq :

In_single_eq →
In_single_self.

Proof using. constructor. intros. rewrite¬ in_single_eq. Qed.

Global Instance in_extens_eq_of_in_extens :

In_extens →
In_extens_eq.

Proof using. constructor. introv I . apply in_extens. intros. rewrite× I . Qed.

Global Instance is_empty_eq_of_in_empty_eq :

In_extens →
In_empty_eq →
Is_empty_eq.

Proof using.

constructor. intros. extens. iff M .

subst. introv N . rewrite× in_empty_eq in N .
apply in_extens. iff N . false× M . rewrite× in_empty_eq in N .

Qed.

Global Instance is_empty_prove_of_is_empty_eq :

Is_empty_eq →
Is_empty_prove.

Proof using. constructor. introv I . rewrite× is_empty_eq. Qed.

Global Instance is_empty_inv_of_is_empty_eq :

Is_empty_eq →
Is_empty_inv.

Proof using. constructor. introv $I1 I2$. rewrite× is_empty_eq in $I1$. Qed.

Global Instance is_nonempty_prove_of_is_empty_eq :

Is_empty_eq →
Is_nonempty_prove.

Proof using. constructor. introv $I1 I2$. rewrite is_empty_eq in $I2$. eauto. Qed.

Global Instance is_single_eq_of_in_single_eq :

In_extens →
In_single_eq →
Is_single_eq.

Proof using.

```
constructor. intros. extens. iff M (M1&M2).
  subst. split. rewrite× in_single_eq. introv N. rewrite× in_single_eq in N.
  apply in_extens. iff N.
    rewrite× (M2 x0). rewrite× in_single_eq.
    rewrite× in_single_eq in N. subst×.
```

Qed.

Global Instance is_single_prove_of_is_single_eq :

```
  Is_single_eq →
  Is_single_prove.
```

Proof using. constructor. introv I. rewrite× is_single_eq. Qed.

Global Instance is_single_inv_of_is_single_eq :

```
  Is_single_eq →
  Is_single_inv.
```

Proof using. constructor. introv I1 I2. rewrite× is_single_eq in I1. Qed.

Global Instance notin_single_eq_of_in_single_eq :

```
  In_single_eq →
  Notin_single_eq.
```

Proof using. constructor. intros. unfold notin. rewrite in_single_eq. eauto. Qed.

Global Instance in_inter_of_in_inter_eq :

```
  In_inter_eq →
  In_inter.
```

Proof using. constructor. introv I1 I2. rewrite× in_inter_eq. Qed.

Global Instance in_inter_inv_of_in_inter_eq :

```
  In_inter_eq →
  In_inter_inv.
```

Proof using. constructor. introv I. rewrite¬← in_inter_eq. Qed.

Global Instance notin_inter_l_of_notin_inter_eq :

```
  Notin_inter_eq →
  Notin_inter_l.
```

Proof using. constructor. introv I. rewrite¬ notin_inter_eq. Qed.

Global Instance notin_inter_r_of_notin_inter_eq :

```
  Notin_inter_eq →
  Notin_inter_r.
```

Proof using. constructor. introv I. rewrite¬ notin_inter_eq. Qed.

Global Instance notin_inter_inv_of_notin_inter_eq :

```
  Notin_inter_eq →
  Notin_inter_inv.
```

Proof using. constructor. introv I. rewrite¬ notin_inter_eq in I. Qed.

Global Instance in_union_l_of_in_union_eq :

In_union_eq →

In_union_l.

Proof using. constructor. introv I . rewrite \times in_union_eq. Qed.

Global Instance in_union_r_of_in_union_eq :

In_union_eq →

In_union_r.

Proof using. constructor. introv I . rewrite \times in_union_eq. Qed.

Global Instance in_union_inv_of_in_union_eq :

In_union_eq →

In_union_inv.

Proof using. constructor. introv I . rewrite \neg @in_union_eq in I . Qed.

Global Instance notin_union_of_notin_union_eq :

Notin_union_eq →

Notin_union.

Proof using. constructor. introv $I1 I2$. rewrite \neg notin_union_eq. Qed.

Global Instance notin_union_inv_of_notin_union_eq :

Notin_union_eq →

Notin_union_inv.

Proof using. constructor. introv I . rewrite \neg notin_union_eq in I . Qed.

Incl

Global Instance incl_prove_of_in_eq :

Incl_in_eq →

Incl_prove.

Proof using. constructor. introv I . rewrite \times incl_in_eq. Qed.

Global Instance incl_inv_of_in_eq :

Incl_in_eq →

Incl_inv.

Proof using. constructor. introv $I1 I2$. rewrite \times incl_in_eq in $I1$. Qed.

Global Instance incl_order_of_incl_in_eq :

In_extens →

Incl_in_eq →

Incl_order.

Proof using.

constructor. constructor.

intros x . rewrite \times incl_in_eq.

intros $E F G I1 I2$. rewrite incl_in_eq. rewrite incl_in_eq in $I1, I2$. autos \times .

intros $E F I1 I2$. rewrite incl_in_eq in $I1, I2$. apply \times in_extens.

Qed.

Global Instance incl_refl_of_incl_order :

Incl_order →

Incl_refl.

Proof using. constructor. apply order_refl. apply incl_order. Qed.

Global Instance incl_trans_of_incl_order :

Incl_order →

Incl_trans.

Proof using. constructor. apply order_trans. apply incl_order. Qed.

Global Instance incl_antisym_of_incl_order :

Incl_order →

Incl_antisym.

Proof using. constructor. apply order_antisym. apply incl_order. Qed.

Global Instance empty_incl_inv_of_incl_in_eq_and_in_empty_eq :

Incl_in_eq →

In_empty_eq →

Empty_incl.

Proof using.

constructor. intros. rewrite incl_in_eq. introv M.

rewrite in_empty_eq in M. false.

Qed.

Global Instance incl_empty_of_in_empty_eq_and_incl_in_eq :

In_extens →

In_empty_eq →

Incl_in_eq →

Incl_empty.

Proof using.

constructor. intros. extens. rewrite incl_in_eq. iff M.

apply in_extens. iff N. applys× M. rewrite in_empty_eq in N. false.

subst. introv N. rewrite in_empty_eq in N. false.

Qed.

Global Instance incl_empty_inv_of_incl_empty :

Incl_empty →

Incl_empty_inv.

Proof using. constructor. introv I. rewrite¬ incl_empty in I. Qed.

Global Instance single_incl_r_eq_of_in_single_eq_and_and_incl_in_eq :

In_extens →

In_single_eq →

Incl_in_eq →

Single_incl_r_eq.

Proof using.

constructor. intros. extens. rewrite incl_in_eq. iff M.

applys× M. rewrite¬ in_single_eq.

introv N. rewrite in_single_eq in N. subst¬.

Qed.

Global Instance single_incl_r_of_single_incl_r_eq:

Single_incl_r_eq →

Single_incl_r.

Proof using.

 constructor. intros. rewrite single_incl_r_eq. assumption.

Qed.

Global Instance single_incl_l_eq_of_in_empty_eq_and_in_single_eq_and_and_incl_in_eq :

In_extens →

In_empty_eq →

In_single_eq →

Incl_in_eq →

Single_incl_l_eq.

Proof using.

 constructor. intros. extens. rewrite incl_in_eq. iff M.

 tests: ($x \in E$).

 right. apply is_single_prove. introv N. forwards $\neg R : M y$.

 rewrite in_single_eq in R.

 left. apply in_extens. iff N. forwards $\neg R : M x_0$.

 rewrite in_single_eq in R. subst. false.

 rewrite in_empty_eq in N. false.

 introv N. rewrite in_single_eq. destruct M.

 subst. rewrite in_empty_eq in N. false.

 subst. rewrite in_single_eq in N. auto.

Qed.

Global Instance union_incl_eq_of_in_union_eq_and_and_incl_in_eq :

In_extens →

In_union_eq →

Incl_in_eq →

Union_incl_eq.

Proof using.

 constructor. intros. extens. repeat rewrite incl_in_eq. iff M ($M_1 \& M_2$).

 split. intros x N. specializes M x. rewrite in_union_eq in M.

 intros x N. specializes M x. rewrite in_union_eq in M.

 intros x N. specializes M1 x. specializes M2 x. rewrite in_union_eq in N.

Qed.

Global Instance incl_union_l_of_incl_in_eq_and_in_union_eq :

Incl_in_eq →

In_union_eq →

Incl_union_l.

Proof using.

```
constructor. introv I. rewrite incl_in_eq. rewrite incl_in_eq in I.
introv N. rewrite× in_union_eq.
```

Qed.

```
Global Instance incl_union_r_of_incl_union_l :
```

```
Incl_union_l →
Union_comm →
Incl_union_r.
```

Proof using. constructor. introv I. rewrite union_comm. apply× @incl_union_l. Qed.

```
Global Instance union_incl_of_union_incl_eq :
```

```
Union_incl_eq →
Union_incl_eq.
```

Proof using. constructor. intros_all. rewrite× union_incl_eq. Qed.

```
Global Instance union_incl_inv_of_union_incl_eq :
```

```
Union_incl_eq →
Union_incl_inv.
```

Proof using. constructor. introv I. rewrite union_incl_eq in I. destruct× I. Qed.

```
Global Instance incl_inter_eq_of_in_inter_eq_and_and_incl_in_eq :
```

```
In_extens →
In_inter_eq →
Incl_in_eq →
Incl_inter_eq.
```

Proof using.

```
constructor. intros. extens. repeat rewrite incl_in_eq. iff M (M1&M2).
split. intros x N. specializes M x. rewrite× in_inter_eq in M.
      intros x N. specializes M x. rewrite× in_inter_eq in M.
intros x N. specializes M1 N. specializes M2 N. rewrite× in_inter_eq.
```

Qed.

```
Global Instance incl_inter_of_incl_inter_eq :
```

```
Incl_inter_eq →
Incl_inter.
```

Proof using. constructor. intros. rewrite× incl_inter_eq. Qed.

```
Global Instance incl_inter_inv_of_incl_inter_eq :
```

```
Incl_inter_eq →
Incl_inter_inv.
```

Proof using. constructor. introv N. rewrite× incl_inter_eq in N. Qed.

Local tactic *contain_by_in_double* to prove inclusion

```
Hint Rewrite @in_union_eq @in_inter_eq
@in_empty_eq @in_single_eq : rew_in_eq.
```

```
Ltac contain_by_in_double :=
intros_all; apply in_extens; intros;
```

```

autorewrite with rew_in_eq;
intuition (try solve [auto|eauto|auto_false|false]).
```

Union

```

Global Instance union_comm_form_in_union_eq :
  In_extens →
  In_union_eq →
  Union_comm.
```

Proof using. constructor. contain_by_in_double. Qed.

```

Global Instance union_assoc_form_in_union_eq :
  In_extens →
  In_union_eq →
  Union_assoc.
```

Proof using. constructor. contain_by_in_double. Qed.

```

Global Instance union_comm_assoc_of_union_comm_and_union_assoc :
  Union_comm →
  Union_assoc →
  Union_comm_assoc.
```

Proof using.

```

constructor. intros_all. do 2 rewrite union_assoc.
  rewrite (union_comm _ x). auto.
```

Qed.

```

Global Instance union_empty_l_of_in_union_eq_and_in_empty_eq :
  In_extens →
  In_union_eq →
  In_empty_eq →
  Union_empty_l.
```

Proof using. constructor. contain_by_in_double. Qed.

```

Global Instance union_empty_r_of_union_empty_l :
  Union_empty_l → Union_comm → Union_empty_r.
```

Proof using. constructor. intros_all. rewrite union_comm. apply union_empty_l. Qed.

```

Global Instance union_empty_inv_of_in_union_eq :
  In_extens →
  In_empty_eq →
  In_union_eq →
  Union_empty_inv.
```

Proof using.

```

constructor. introv N. split.
  apply in_extens. iff R. rewrite ← N. rewrite× in_union_eq. rewrite× in_empty_eq
in R.
  apply in_extens. iff R. rewrite ← N. rewrite× in_union_eq. rewrite× in_empty_eq
in R.
```

Qed.

Global Instance union_self_of_in_union_eq :

In_extens →
In_union_eq →
Union_self.

Proof using. constructor. *contain_by_in_double*. Qed.

Global Instance notin_union_eq_of_in_union_eq :

In_union_eq →
Notin_union_eq.

Proof using. constructor. intros. unfold notin. rewrite @in_union_eq. *extens* ×. eauto. Qed.

Inter

Global Instance inter_comm_form_in_inter_eq :

In_extens →
In_inter_eq →
Inter_comm.

Proof using. constructor. *contain_by_in_double*. Qed.

Global Instance inter_assoc_form_in_inter_eq :

In_extens →
In_inter_eq →
Inter_assoc.

Proof using. constructor. *contain_by_in_double*. Qed.

Global Instance inter_comm_assoc_of_inter_comm_and_inter_assoc :

Inter_comm →
Inter_assoc →
Inter_comm_assoc.

Proof using.

constructor. *intros_all*. do 2 rewrite inter_assoc.
rewrite (inter_comm _ x). auto.

Qed.

Global Instance inter_empty_l_of_in_inter_eq_and_in_empty_eq :

In_extens →
In_inter_eq →
In_empty_eq →
Inter_empty_l.

Proof using. constructor. *contain_by_in_double*. Qed.

Global Instance inter_empty_r_of_inter_empty_l :

Inter_empty_l →
Inter_comm →
Inter_empty_r.

Proof using. constructor. *intros_all*. rewrite *inter_comm*. apply *inter_empty_l*. Qed.

Global Instance *inter_self_of_in_inter_eq* :

In_extens →
In_inter_eq →
Inter_self.

Proof using. constructor. *contain_by_in_double*. Qed.

Remove

Global Instance *remove_incl_of_in_remove_eq_and_incl_in_eq* :

In_remove_eq →
Incl_in_eq →
Remove_incl.

Proof using.

constructor. intros. rewrite *incl_in_eq*. *introv N*. *rewrite* \times *in_remove_eq* in *N*.
Qed.

Global Instance *remove_disjoint_of_in_remove_eq_and_disjoint_eq* :

In_remove_eq →
Disjoint_eq →
Remove_disjoint.

Proof using.

constructor. intros. rewrite *disjoint_eq*. *introv N M*.
rewrite \times *in_remove_eq* in *M*.

Qed.

Disjoint

Global Instance *disjoint_not_eq_of_disjoint_eq* :

Disjoint_eq →
Disjoint_not_eq.

Proof using.

constructor. intros. rewrite *disjoint_eq*. *extens iff M*.
{ *rew_logic* in *M*. *destruct M* as [*x M*]. *rew_logic* \times in *M*. }
{ *destruct M* as [*x M*]. *rew_logic*. $\exists x$. *rew_logic* \times . }

Qed.

Global Instance *disjoint_prove_of_disjoint_eq* :

Disjoint_eq →
Disjoint_prove.

Proof using. constructor. intros. *rewrite* \times *disjoint_eq*. Qed.

Global Instance *disjoint_inv_of_disjoint_eq* :

Disjoint_eq →
Disjoint_inv.

Proof using. constructor. *introv I I1 I2*. *rewrite* \times *disjoint_eq* in *I*. Qed.

Global Instance *disjoint_single_l_eq_of_disjoint_eq_and_in_single_eq* :

Disjoint_eq →
In_single_eq →
Disjoint_single_l_eq.

Proof using.

```
constructor. intros. rewrite disjoint_eq. unfold notin. extens. iff M.  
introv N. specializes M N. rewrite× in_single_eq. false.  
introv N1 N2. rewrite in_single_eq in N1. subst. false.
```

Qed.

Global Instance disjoint_sym_of_disjoint_eq :

Disjoint_eq →
Disjoint_sym.

Proof using. constructor. intros x y. do 2 rewrite× disjoint_eq. Qed.

Global Instance disjoint_single_r_eq_of_disjoint_single_l :

Disjoint_single_l_eq →
Disjoint_sym →
Disjoint_single_r_eq.

Proof using.

```
constructor. intros_all.  
rewrites (» sym_inv_eq disjoint_sym). apply disjoint_single_l_eq.
```

Qed.

Global Instance inter_disjoint_of_disjoint_eq_and_in_inter_eq :

In_extens →
In_empty_eq →
Disjoint_eq →
In_inter_eq →
Inter_Disjoint.

Proof using.

```
constructor. introv M. apply in_extens. rewrite disjoint_eq in M. iff N.  
rewrite in_inter_eq in N. false×.  
rewrite in_empty_eq in N. false.
```

Qed.

End DerivedProperties.

Chapter 17

Library SLF.LibOption

```
Set Implicit Arguments.  
From SLF Require Import LibTactics LibReflect.  
Generalizable Variables A.
```

17.1 Option type

17.1.1 Definition

From the Prelude:

```
Inductive option A : Type := | Some : A -> option A | None : option A.
```

Arguments Some {A}.

Arguments None {A}.

17.1.2 Inhabited

```
Instance Inhab_option : ∀ A, Inhab (option A).  
Proof using intros. apply (Inhab_of_val None). Qed.
```

17.2 Operations

17.2.1 is_some

is_some o holds when o is of the form Some x

```
Definition is_some A (o:option A) :=  
  match o with  
  | None ⇒ false  
  | Some _ ⇒ true
```

```
end.
```

17.2.2 unsome_default

`unsome_default d o` returns the content of the option, and returns `d` in case the option in `None`.

```
Definition unsome_default A d (o:option A) :=  
  match o with  
  | None => d  
  | Some x => x  
  end.
```

17.2.3 unsome

`unsome o` returns the content of the option, and returns an arbitrary value in case the option in `None`.

```
Definition unsome '{Inhab A} :=  
  unsome_default (arbitrary (A:=A)).
```

17.2.4 map

`map f o` takes an option and returns an option, and maps the function `f` to the content of the option if it has one.

```
Definition map A B (f : A → B) (o : option A) : option B :=  
  match o with  
  | None => None  
  | Some x => Some (f x)  
  end.
```

```
Lemma map_eq_none_inv : ∀ A B (f : A → B) o,  
  map f o = None →  
  o = None.
```

Proof using. introv H. destruct o; simpl; inverts× H. Qed.

```
Lemma map_eq_some_inv : ∀ A B (f : A → B) o x,  
  map f o = Some x →  
  ∃ z, o = Some z ∧ x = f z.
```

Proof using. introv H. destruct o; simpl; inverts× H. Qed.

Arguments map_eq_none_inv [A] [B] [f] [o].

Arguments map_eq_some_inv [A] [B] [f] [o] [x].

17.2.5 map_on

Definition `map_on A B (o : option A) (f : A → B) : option B :=`
`map f o.`

Lemma `map_on_eq_none_inv : ∀ A B (f : A → B) o,`
`map f o = None →`
`o = None.`

Proof using. `introv H. destruct o; tryfalse. Qed.`

Lemma `map_on_eq_some_inv : ∀ A B (f : A → B) o x,`
`map_on o f = Some x →`
`∃ z, o = Some z ∧ x = f z.`

Proof using. `introv H. destruct o; simpl; inverts x H. Qed.`

Arguments `map_eq_none_inv [A] [B] [f] [o].`

Arguments `map_on_eq_some_inv [A] [B] [f] [o] [x].`

17.2.6 apply

`apply f o` optionnaly applies a function of type $A \rightarrow \text{option } B$

Definition `apply A B (f : A → option B) (o : option A) : option B :=`
`match o with`
| `None ⇒ None`
| `Some x ⇒ f x`
`end.`

17.2.7 apply_on

Definition `apply_on A B (o : option A) (f : A → option B) : option B :=`
`apply f o.`

Lemma `apply_on_eq_none_inv : ∀ A B (f : A → option B) o,`
`apply_on o f = None →`
`(o = None)`
`∨ (∃ x, o = Some x ∧ f x = None).`

Proof using. `introv H. destruct o; simpl; inverts x H. Qed.`

Lemma `apply_on_eq_some_inv : ∀ A B (f : A → option B) o x,`
`apply_on o f = Some x →`
`∃ z, o = Some z ∧ f z = Some x.`

Proof using. `introv H. destruct o; simpl; inverts x H. Qed.`

Arguments `apply_on_eq_none_inv [A] [B] [f] [o].`

Arguments `apply_on_eq_some_inv [A] [B] [f] [o] [x].`

Chapter 18

Library SLF.LibWf

```
Set Implicit Arguments.
```

```
From SLF Require Import LibTactics LibLogic  
LibProd LibSum LibRelation LibNat LibInt.
```

18.1 Compatibility

Coq's stdlib Prelude defines:

```
Inductive Acc A (R:A->A->Prop) (x:A) : Prop := | Acc_intro : (forall (y:A), R y x ->  
Acc y) -> Acc x.
```

Definition well_founded A (R:A->A->Prop) := forall (x:A), Acc x.

TLC introduces **wf** as a shorter name for **well_founded**, both for conciseness and for tactics to specifically recognize this symbol.

```
Definition wf := well_founded.
```

18.1.1 Tactics

auto with wf attempts to unfold the names of the relations given as argument to **wf**.

```
Hint Extern 1 (wf ?R) => progress (unfold R) : wf.
```

solve_wf is a shorthand for solving goals using **auto with wf**, aimed to prove goals of the form **wf R**.

```
Tactic Notation "solve_wf" :=  
  solve [ auto with wf ].
```

18.1.2 Definition

measure f is a well-founded binary relation which relates x to y when $f x < f y$, at type **nat**.

```
Definition measure A (f:A->nat) : binary A :=  
  fun x1 x2 => (f x1 < f x2).
```

18.1.3 Properties

Section Measure.

Variables ($A : \text{Type}$).

Implicit Type $f : A \rightarrow \text{nat}$.

Lemma wf_measure : $\forall f,$
 $\text{wf}(\text{measure } f)$.

Proof using.

```
intros f a. gen_eq n: (f a). gen a. pattern n.  
apply peano_induction. clear n. introv IH Eq.  
apply Acc_intro. introv H. unfolds in H.  
rewrite ← Eq in H. apply× IH.
```

Qed.

Lemma trans_measure : $\forall (f : A \rightarrow \text{nat}),$
 $\text{trans}(\text{measure } f)$.

Proof using. intros. unfold measure, trans. intros. nat_math. Qed.

End Measure.

Hint Resolve wf_measure : wf.

18.1.4 Measure on pairs

Definition measure2 $A1\ A2\ (f : A1 \rightarrow A2 \rightarrow \text{nat}) : \text{binary}(A1 \times A2) :=$
 $\text{fun } p1\ p2 \Rightarrow \text{let } (x1,y1) := p1 \text{ in}$
 $\quad \text{let } (x2,y2) := p2 \text{ in}$
 $\quad (f\ x1\ y1 < f\ x2\ y2)$.

Lemma wf_measure2 : $\forall A1\ A2\ (f : A1 \rightarrow A2 \rightarrow \text{nat}),$
 $\text{wf}(\text{measure2 } f)$.

Proof using.

```
intros A1 A2 f [x1 x2]. apply (@measure_induction _ (uncurry2 f)). clear x1 x2.  
intros [x1 x2] H. apply Acc_intro. intros [y1 y2] Lt. apply¬ H.
```

Qed.

Hint Resolve wf_measure2 : wf.

Ltac induction_wf_process_wf_hyp tt ::=
match goal with
| $\vdash \text{wf } _ \Rightarrow \text{auto with wf}$
| $\vdash \text{well_founded } _ \Rightarrow \text{change well_founded with wf; auto with wf}$
end.

Ltac induction_wf_process_measure E ::=
applys well_founded_ind (wf_measure E).

18.2 Construction of well-founded relations

18.2.1 Empty relation

```
Lemma wf_empty : ∀ A,  
  wf (@empty A).  
Proof using. intros_all. constructor. introv H. false. Qed.  
Hint Resolve wf_empty : wf.
```

18.2.2 Inclusion

Well-foundedness preserved by inclusion

```
Lemma wf_of_rel_incl : ∀ A (R1 R2 : binary A),  
  wf R1 →  
  rel_incl R2 R1 →  
  wf R2.  
Proof using.  
  introv W1 Inc. intros x.  
  pattern x. apply (well_founded_ind W1). clear x.  
  intros x IH. constructor. intros. apply IH. apply¬ Inc.  
Qed.
```

18.2.3 Peano.lt on nat

The relation “less than” on natural numbers is well_founded.

```
Lemma wf_peano_lt : wf Peano.lt.  
Proof using.  
  intros x.  
  induction x using peano_induction. apply¬ Acc_intro.  
  intros. applys H. math.  
Qed.  
Hint Resolve wf_peano_lt : wf.
```

18.2.4 lt on nat

The relation “less than” on natural numbers is well_founded.

```
Lemma wf_lt : @wf nat lt.  
Proof using.  
  intros x.  
  induction x using peano_induction. apply¬ Acc_intro.  
Qed.
```

Hint Resolve wf_lt : wf.

18.2.5 The relation “greater than” on the set of natural number lower than a fixed upper bound.

Definition nat upto ($b:\text{nat}$) :=
fun ($n m:\text{nat}$) \Rightarrow ($n \leq b$)%nat \wedge ($m < n$)%nat.

Lemma nat upto_eq : $\forall (b n m:\text{nat})$,
 $\text{nat_upto } b n m = ((n \leq b)\%nat \wedge (m < n)\%nat)$.

Proof using. auto. Qed.

Lemma wf_nat upto : $\forall (b:\text{nat})$,
wf (nat upto b).

Proof using.

```
intros b n.  
induction_wf IH: (wf_measure (fun n  $\Rightarrow$  (b-n)%nat)) n.  
apply Acc_intro. introv [H1 H2]. apply IH.  
hnf. nat_math.
```

Qed.

18.2.6 The relation “less than” on the set of integers greater than a fixed lower bound.

Definition downto ($b:\mathbf{Z}$) :=
fun ($n m:\mathbf{Z}$) \Rightarrow ($b \leq n$) \wedge ($n < m$).

Lemma downto_eq : $\forall (b n m:\mathbf{Z})$,
 $\text{downto } b n m = (b \leq n \wedge n < m)$.

Proof using. auto. Qed.

Lemma downto_intro : $\forall (b n m:\mathbf{Z})$,
 $b \leq n \rightarrow$
 $n < m \rightarrow$
 $\text{downto } b n m$.

Proof using. split \neg . Qed.

Lemma wf_downto : $\forall (b:\mathbf{Z})$,
wf (downto b).

Proof using.

```
intros b n.  
induction_wf IH: (wf_measure (fun n  $\Rightarrow$  Z.abs_nat (n-b))) n.  
apply Acc_intro. introv [H1 H2]. apply IH.  
unfolds. applys lt_abs_abs; math.
```

Qed.

```

Hint Resolve wf_downto : wf.
Hint Unfold downto.
Hint Extern 1 (downto _ _ _) ⇒ math : maths.

```

18.2.7 The relation “greater than” on the set of integers lower than a fixed upper bound.

```

Definition upto (b:Z) :=
  fun (n m:Z) ⇒ (n ≤ b) ∧ (m < n).

Lemma upto_eq : ∀ b n m,
  upto b n m = ((n ≤ b) ∧ (m < n)).
Proof using. auto. Qed.

Lemma upto_intro : ∀ b n m,
  n ≤ b →
  m < n →
  upto b n m.
Proof using. split¬. Qed.

Lemma wf_uppto : ∀ n,
  wf (upto n).
Proof using.
  intros b n.
  induction_wf IH: (wf_measure (fun n ⇒ Z.abs_nat (b-n))) n.
  apply Acc_intro. introv [H1 H2]. apply IH.
  applys lt_abs_abs; math.

Qed.

Hint Resolve wf_uppto : wf.
Hint Unfold upto.
Hint Extern 1 (upto _ _ _) ⇒ math : maths.

```

18.3 Inverse projections

```

Section UnprojWf.
Variables (A1 A2 A3 A4 A5 : Type).

Lemma wf_unproj21 : ∀ (R:binary A1),
  wf R →
  wf (unproj21 A2 R).
Proof using.
  intros R H [x1 x2]. gen x2.
  induction_wf IH: H x1. constructor. intros [y1 y2]. auto.

Qed.

```

Lemma wf_unproj22 : $\forall (R:\text{binary } A2),$

 wf $R \rightarrow$

 wf (unproj22 A1 R).

Proof using.

 intros R H [x1 x2]. gen x1.

 induction_wf IH: H x2. constructor. intros [y1 y2]. auto.

Qed.

Lemma wf_unproj31 : $\forall (R:\text{binary } A1),$

 wf $R \rightarrow$

 wf (unproj31 A2 A3 R).

Proof using.

 intros R H [[x1 x2] x3]. gen x2 x3.

 induction_wf IH: H x1. constructor. intros [[y1 y2] y3]. auto.

Qed.

Lemma wf_unproj32 : $\forall (R:\text{binary } A2),$

 wf $R \rightarrow$

 wf (unproj32 A1 A3 R).

Proof using.

 intros R H [[x1 x2] x3]. gen x1 x3.

 induction_wf IH: H x2. constructor. intros [[y1 y2] y3]. auto.

Qed.

Lemma wf_unproj33 : $\forall (R:\text{binary } A3),$

 wf $R \rightarrow$

 wf (unproj33 A1 A2 R).

Proof using.

 intros R H [[x1 x2] x3]. gen x1 x2.

 induction_wf IH: H x3. constructor. intros [[y1 y2] y3]. auto.

Qed.

Lemma wf_unproj41 : $\forall (R:\text{binary } A1),$

 wf $R \rightarrow$

 wf (unproj41 A2 A3 A4 R).

Proof using.

 intros R H [[[x1 x2] x3] x4]. gen x2 x3 x4.

 induction_wf IH: H x1. constructor. intros [[[y1 y2] y3] y4]. auto.

Qed.

Lemma wf_unproj42 : $\forall (R:\text{binary } A2),$

 wf $R \rightarrow$

 wf (unproj42 A1 A3 A4 R).

Proof using.

 intros R H [[[x1 x2] x3] x4]. gen x1 x3 x4.

 induction_wf IH: H x2. constructor. intros [[[y1 y2] y3] y4]. auto.

Qed.

Lemma wf_unproj43 : $\forall (R:\text{binary } A3),$
wf $R \rightarrow$
wf (unproj43 A1 A2 A4 R).

Proof using.

intros R H [[[x1 x2] x3] x4]. gen x1 x2 x4.
induction_wf IH: H x3. constructor. intros [[[y1 y2] y3] y4]. auto.

Qed.

Lemma wf_unproj44 : $\forall (R:\text{binary } A4),$
wf $R \rightarrow$
wf (unproj44 A1 A2 A3 R).

Proof using.

intros R H [[[x1 x2] x3] x4]. gen x1 x2 x3.
induction_wf IH: H x4. constructor. intros [[[y1 y2] y3] y4]. auto.

Qed.

Lemma wf_unproj51 : $\forall (R:\text{binary } A1),$
wf $R \rightarrow$
wf (unproj51 A2 A3 A4 A5 R).

Proof using.

intros R H [[[x1 x2] x3] x4] x5]. gen x2 x3 x4 x5.
induction_wf IH: H x1. constructor. intros [[[y1 y2] y3] y4] y5]. auto.

Qed.

End UnprojWf.

Hint Resolve

wf_unproj21 wf_unproj22
wf_unproj31 wf_unproj32 wf_unproj33
wf_unproj41 wf_unproj42 wf_unproj43 wf_unproj44
wf_unproj51 : wf.

18.4 Lexicographical product

Lemma wf_lexico2 : $\forall A1 A2$
($R1:\text{binary } A1$) ($R2:\text{binary } A2$),
wf $R1 \rightarrow$
wf $R2 \rightarrow$
wf (lexico2 R1 R2).

Proof using.

introv W1 W2. intros [x1 x2]. gen x2.
induction_wf IH1: W1 x1. intros.
induction_wf IH2: W2 x2. constructor. intros [y1 y2] H.

```

simples. destruct H as [H1|[H1 H2]].
apply  $\neg$  IH1. rewrite H1. apply  $\neg$  IH2.
Qed.

```

```

Lemma wf_lexico3 :  $\forall A1 A2 A3$ 
(R1:binary A1) (R2:binary A2) (R3:binary A3),
wf R1  $\rightarrow$ 
wf R2  $\rightarrow$ 
wf R3  $\rightarrow$ 
wf (lexico3 R1 R2 R3).

```

Proof using.

```

intros. apply  $\neg$  wf_lexico2. apply  $\neg$  wf_lexico2.
Qed.

```

```

Lemma wf_lexico4 :  $\forall A1 A2 A3 A4$ 
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
wf R1  $\rightarrow$ 
wf R2  $\rightarrow$ 
wf R3  $\rightarrow$ 
wf R4  $\rightarrow$ 
wf (lexico4 R1 R2 R3 R4).

```

Proof using.

```

intros. apply  $\neg$  wf_lexico3. apply  $\neg$  wf_lexico2.
Qed.

```

```
Hint Resolve wf_lexico2 wf_lexico3 wf_lexico4 : wf.
```

18.5 Symmetric product

```

Lemma wf_prod2_of_wf_1 :  $\forall (A1 A2:\text{Type})$ 
(R1:binary A1) (R2:binary A2),
wf R1  $\rightarrow$ 
wf (prod2 R1 R2).

```

Proof using.

```

introv W1. intros [x1 x2].
gen x2. induction_wf IH: W1 x1. intros.
constructor. intros [y1 y2] [E1 E2]. apply  $\neg$  IH.

```

Qed.

```

Lemma wf_prod2_of_wf_2 :  $\forall (A1 A2:\text{Type})$ 
(R1:binary A1) (R2:binary A2),
wf R2  $\rightarrow$ 
wf (prod2 R1 R2).

```

Proof using.

```

introv W2. intros [x1 x2].

```

```

gen x1. induction_wf IH: W2 x2. intros.
constructor. intros [y1 y2] [E1 E2]. apply¬ IH.
Qed.

Lemma wf_prod3_of_wf_1 : ∀ (A1 A2 A3:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3),
wf R1 →
wf (prod3 R1 R2 R3).
Proof using. intros. apply wf_prod2_of_wf_1. apply¬ wf_prod2_of_wf_1. Qed.

Lemma wf_prod3_of_wf_2 : ∀ (A1 A2 A3:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3),
wf R2 →
wf (prod3 R1 R2 R3).
Proof using. intros. apply wf_prod2_of_wf_1. apply¬ wf_prod2_of_wf_2. Qed.

Lemma wf_prod3_of_wf_3 : ∀ (A1 A2 A3:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3),
wf R3 →
wf (prod3 R1 R2 R3).
Proof using. intros. apply¬ wf_prod2_of_wf_2. Qed.

Lemma wf_prod4_of_wf_1 : ∀ (A1 A2 A3 A4:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
wf R1 →
wf (prod4 R1 R2 R3 R4).
Proof using. intros. apply wf_prod2_of_wf_1. apply¬ wf_prod3_of_wf_1. Qed.

Lemma wf_prod4_of_wf_2 : ∀ (A1 A2 A3 A4:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
wf R2 →
wf (prod4 R1 R2 R3 R4).
Proof using. intros. apply wf_prod2_of_wf_1. apply¬ wf_prod3_of_wf_2. Qed.

Lemma wf_prod4_of_wf_3 : ∀ (A1 A2 A3 A4:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
wf R3 →
wf (prod4 R1 R2 R3 R4).
Proof using. intros. apply wf_prod2_of_wf_1. apply¬ wf_prod3_of_wf_3. Qed.

Lemma wf_prod4_of_wf_4 : ∀ (A1 A2 A3 A4:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
wf R4 →
wf (prod4 R1 R2 R3 R4).
Proof using. intros. apply¬ wf_prod2_of_wf_2. Qed.

Hint Resolve
wf_prod2_of_wf_1 wf_prod2_of_wf_2

```

```
wf_prod3_of_wf_1 wf_prod3_of_wf_2 wf_prod3_of_wf_3
wf_prod4_of_wf_1 wf_prod4_of_wf_2 wf_prod4_of_wf_3 wf_prod4_of_wf_4 : wf.
```

18.6 Well-foundedness of a function image

Lemma `wf_rel_preimage` : $\forall A B (R:\text{binary } B) (f:A \rightarrow B),$
 $\text{wf } R \rightarrow$
 $\text{wf } (\text{rel_preimage } R f).$

Proof using.

```
intros W. intros x. gen_eq a: (f x). gen x.
induction_wf IH: W a. introv E. constructors.
intros y Hy. subst a. hnf in Hy. applys× IH.
```

Qed.

Hint Resolve `wf_rel_preimage` : `wf`.

18.7 Transitive closure

Lemma `wf_tclosure` : $\forall A (R:\text{binary } A),$
 $\text{wf } R \rightarrow$
 $\text{wf } (\text{tclosure } R).$

Proof using.

```
unfold wf, well_founded.
introv HAcc. intro a. specializes HAcc a. generalize dependent a.
induction 1 as [ a _ IH ].
constructor. intros b Hba.
generalize a b Hba IH. clear a b Hba IH.
induction 1; eauto using Acc_inv.
```

Qed.

Chapter 19

Library SLF.LibList

Set Implicit Arguments.

Require Import Coq.Classes.Morphisms. From SLF Require Import LibTactics LibLogic LibReflect LibOperation

LibProd LibOption LibNat LibInt LibWf LibMonoid LibRelation.

Generalizable Variables A B.

Local Open Scope nat_scope.

Local Open Scope comp_scope.

Global Close Scope list_scope.

Arguments nil {A}.

Arguments cons {A}.

Declare Scope liblist_scope.

Open Scope liblist_scope.

Delimit Scope liblist_scope with list.

Bind Scope liblist_scope with list.

Infix "::" := cons (at level 60, right associativity) : liblist_scope.

19.1 Inhabited

Instance Inhab_list : $\forall A, \text{Inhab}(\text{list } A)$.

Proof using. intros. apply (Inhab_of_val nil). Qed.

19.2 Normalization tactics

19.2.1 *rew_list* for basic list properties

Normalize

- ++
- length
- rev

```

Tactic Notation "rew_list" :=
  autorewrite with rew_list.
Tactic Notation "rew_list" "˜" :=
  rew_list; auto_tilde.
Tactic Notation "rew_list" "*" :=
  rew_list; auto_star.
Tactic Notation "rew_list" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_list).
Tactic Notation "rew_list" "˜" "in" "*" :=
  rew_list in *; auto_tilde.
Tactic Notation "rew_list" "*" "in" "*" :=
  rew_list in *; auto_star.
Tactic Notation "rew_list" "in" hyp(H) :=
  autorewrite with rew_list in H.
Tactic Notation "rew_list" "˜" "in" hyp(H) :=
  rew_list in H; auto_tilde.
Tactic Notation "rew_list" "*" "in" hyp(H) :=
  rew_list in H; auto_star.

```

19.2.2 *rew_listx* for all other operations on lists

```

Tactic Notation "rew_listx" :=
  autorewrite with rew_listx.
Tactic Notation "rew_listx" "˜" :=
  rew_listx; auto_tilde.
Tactic Notation "rew_listx" "*" :=
  rew_listx; auto_star.
Tactic Notation "rew_listx" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_listx).
Tactic Notation "rew_listx" "˜" "in" "*" :=
  rew_listx in *; auto_tilde.
Tactic Notation "rew_listx" "*" "in" "*" :=
  rew_listx in *; auto_star.
Tactic Notation "rew_listx" "in" hyp(H) :=
  autorewrite with rew_listx in H.
Tactic Notation "rew_listx" "˜" "in" hyp(H) :=
  rew_listx in H; auto_tilde.

```

```
Tactic Notation "rew_listx" "*" "in" hyp(H) :=
  rew_listx in H; auto_star.
```

19.3 Properties of operations

19.3.1 Definitions of Fold-right and App

```
Fixpoint fold_right A B (f:A→B→B) (i:B) (l:list A) :=
  match l with
  | nil ⇒ i
  | x::L' ⇒ f x (fold_right f i L')
  end.
```

```
Definition app A (l1 l2 : list A) :=
  fold_right (fun x (acc:list A) ⇒ x::acc) l2 l1.
```

$|l_1 ++ l_2$ concatenates two lists

Infix "++" := app (right associativity, at level 60) : liblist_scope.

$| & x$ extends the list $|$ with the value x at the right end

```
Notation "| & x" := (l ++ (x::nil))
(at level 28, left associativity) : liblist_scope.
```

19.3.2 App

Section App.

Variables A B : Type.

Implicit Types x : A.

Implicit Types l : list A.

Lemma app_cons_l : $\forall x l1 l2, (x::l1) ++ l2 = x :: (l1 ++ l2)$.

Proof using. auto. Qed.

Lemma app_nil_l : $\forall l, \text{nil} ++ l = l$.

Proof using. auto. Qed.

Lemma app_nil_r : $\forall l, l ++ \text{nil} = l$.

Proof using. induction l. auto. rewrite app_cons_l. f_equal. Qed.

Lemma app_assoc : $\forall l1 l2 l3, (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3)$.

Proof using.

```

intros. induction l1.
{ rewrite_all¬ app_nil_l. }
{ rewrite_all¬ app_cons_l. fequals¬. }
Qed.

Lemma app_cons_r : ∀ x l1 l2,
l1 ++ (x::l2) = (l1 & x) ++ l2.
Proof using. intros. rewrite¬ app_assoc. Qed.

Lemma app_cons_one_r : ∀ x l,
(x::nil) ++ l = x::l.
Proof using. auto. Qed.

Lemma app_cons_one_l : ∀ x l,
l ++ (x::nil) = l&x. Proof using. auto. Qed.

Lemma app_last_l : ∀ x l1 l2,
(l1 & x) ++ l2 = l1 ++ (x::l2).
Proof using. intros. rewrite¬ ← app_cons_r. Qed.

Lemma app_last_r : ∀ x l1 l2,
l1 ++ (l2 & x) = (l1 ++ l2) & x.
Proof using. intros. rewrite¬ ← app_assoc. Qed.

Lemma last_nil : ∀ x,
nil & x = x::nil.
Proof using. auto. Qed.

Lemma last_cons : ∀ x y l,
(x::l) & y = x::(l&y).
Proof using. auto. Qed.

Lemma last_one : ∀ x y,
(x::nil) & y = x::y::nil.
Proof using. auto. Qed.

Lemma last_app : ∀ x l1 l2,
(l1 ++ l2) & x = l1 ++ (l2 & x).
Proof using. intros. rewrite¬ app_last_r. Qed.

End App.

Global Opaque app.

Hint Rewrite app_cons_l app_nil_l app_nil_r app_assoc
app_cons_one_r : rew_list.

Hint Rewrite app_cons_l app_nil_l app_nil_r app_assoc
app_cons_one_r : rew_listx.

Hint Rewrite app_cons_l app_nil_l app_nil_r app_assoc
app_cons_one_r : rew_lists.

```

19.3.3 FoldRight

```
Section FoldRight.  
Variables A B : Type.  
Implicit Types x : A.  
Implicit Types l : list A.  
Implicit Types (f : A → B → B) (i : B).  
Lemma fold_right_nil : ∀ f i,  
  fold_right f i nil = i.  
Proof using. auto. Qed.  
Lemma fold_right_cons : ∀ f i x l,  
  fold_right f i (x::l) = f x (fold_right f i l) .  
Proof using. auto. Qed.  
Lemma fold_right_app : ∀ f i l1 l2,  
  fold_right f i (l1 ++ l2) = fold_right f (fold_right f i l2) l1.  
Proof using.  
  intros. induction l1.  
  { rew_list. do 2 rewrite fold_right_cons. fequals-. }  
Qed.  
Lemma fold_right_last : ∀ f i x l,  
  fold_right f i (l & x) = fold_right f (f x i) l.  
Proof using. intros. rewrite fold_right_app. Qed.  
End FoldRight.  
Global Opaque fold_right.  
Hint Rewrite fold_right_nil fold_right_cons fold_right_last : rew_listx.
```

19.3.4 FoldLeft

```
Fixpoint fold_left A B (f:A→B→B) (i:B) (l:list A) : B :=  
  match l with  
  | nil ⇒ i  
  | x::L' ⇒ fold_left f (f x i) L'  
  end.
```

```
Section FoldLeft.  
Variables A B : Type.  
Implicit Types x : A.  
Implicit Types l : list A.  
Implicit Types (f : A → B → B) (i : B).  
Lemma fold_left_nil : ∀ f i,  
  fold_left f i nil = i.
```

```

Proof using. auto. Qed.

Lemma fold_left_cons :  $\forall f i x l,$ 
   $\text{fold\_left } f i (x :: l) = \text{fold\_left } f (f x i) l.$ 
Proof using. auto. Qed.

Lemma fold_left_app :  $\forall f i l1 l2,$ 
   $\text{fold\_left } f i (l1 ++ l2) = \text{fold\_left } f (\text{fold\_left } f i l1) l2.$ 
Proof using.
  intros. gen i. induction l1.
  { intros. rew_list. do 2 rewrite fold_left_cons. rewrite IHl1. }
Qed.

Lemma fold_left_last :  $\forall f i x l,$ 
   $\text{fold\_left } f i (l & x) = f x (\text{fold\_left } f i l).$ 
Proof using. intros. rewrite fold_left_app. Qed.

End FoldLeft.

Global Opaque fold_left.

Hint Rewrite fold_left_nil fold_left_cons
  fold_left_last : rew_listx.

```

19.3.5 Length

```

Definition length A (l:list A) : nat :=
  fold_right (fun x acc => 1+acc) 0 l.

Section Length.
Variables (A : Type).
Implicit Types l : list A.

Lemma length_nil :
  length (@nil A) = 0.
Proof using. auto. Qed.

Lemma length_cons :  $\forall x l,$ 
   $\text{length } (x :: l) = 1 + \text{length } l.$ 
Proof using. auto. Qed.

Lemma length_app :  $\forall l1 l2,$ 
   $\text{length } (l1 ++ l2) = \text{length } l1 + \text{length } l2.$ 
Proof using.
  intros. unfold length at 1. rewrite fold_right_app.
  fold (length l2). induction l1; rew_listx.
  { auto. }
  { rewrite length_cons. rewrite IHl1. math. }
Qed.

```

```

Lemma length_last : ∀ x l,
  length (l & x) = 1 + length l.

Proof using.
  intros. rewrite length_app, length_cons, length_nil.
  simpl. math.

Qed.

```

End Length.

Global Opaque length.

```

Hint Rewrite length_nil length_cons length_app
  length_last : rew_list.

Hint Rewrite length_nil length_cons length_app
  length_last : rew_listx.

```

19.4 Inversion lemmas for structural composition

Section ApplInversion.

Variables A : Type.

Implicit Types x : A.

Implicit Types l : list A.

```

Lemma length_zero_inv : ∀ l,
  length l = 0%nat →
  l = nil.

```

Proof using.

```
  intros l. destruct l; rew_list; introv E. { auto. } { false. }
```

Qed.

```

Lemma length_zero_eq_eq_nil : ∀ l,
  (length l = 0) = (l = nil).

```

Proof using.

```
  extens. iff M. destruct l; rew_list; auto_false×. { subst×. }
```

Qed.

```

Lemma length_one_inv : ∀ l,
  length l = 1 →
  ∃ x, l = x::nil.

```

Proof using.

```
  introv N. destruct l as [|x []]; rew_list in *; tryfalse. eauto.
```

Qed.

```

Lemma length_neq_inv : ∀ l1 l2,
  length l1 ≠ length l2 →
  (l1 ≠ l2).

```

Proof using. *introv N E. subst* \times . Qed.

Lemma length_pos_inv_cons : $\forall l,$
 $(\text{length } l > 0\%nat) \rightarrow$
 $\exists x l', l = x :: l'.$

Proof using.

*introv E. destruct l; rew_list in *.*
 $\{ \text{false. math.} \}$
 $\{ \text{eauto.} \}$

Qed.

Lemma length_pos_inv_last : $\forall l,$
 $(\text{length } l > 0\%nat) \rightarrow$
 $\exists x l', l = l' & x.$

Proof using.

intros l. induction l; rew_list; introv H.
 $\{ \text{false. math.} \}$
 $\{ \text{destruct } l.$
 $\{ \exists a (@nil A). \}$
 $\{ \text{destruct } IHl \text{ as } (x \& l' \& E).$
 $\{ \text{rew_list in *. math.} \}$
 $\{ \exists x (a :: l'). \text{ rewrite } \neg E. \} \}$

Qed.

Lemma list_same_length_inv_nil : $\forall l1 l2,$
 $\text{length } l1 = \text{length } l2 \rightarrow$
 $l1 = \text{nil} \leftrightarrow l2 = \text{nil}.$

Proof using. *intros. destruct l1; destruct l2; auto_false* \times . Qed.

Lemma cons_case : $\forall l,$
 $l = \text{nil} \vee \exists x l', l = x :: l'.$

Proof using. *intros. destruct* $\times l$. Qed.

Lemma cons_eq_nil_inv : $\forall x l,$
 $x :: l = \text{nil} \rightarrow$
False.

Proof using. *auto_false*. Qed.

Lemma nil_eq_cons_inv : $\forall x l,$
 $\text{nil} = x :: l \rightarrow$
False.

Proof using. *auto_false*. Qed.

Lemma list_neq_nil_inv_cons : $\forall l,$
 $l \neq \text{nil} \rightarrow$
 $\exists x q, l = x :: q.$

Proof using. *introv N. destruct* $\times l$. Qed.

Lemma `cons_eq_cons_inv` : $\forall x_1 x_2 l_1 l_2,$
 $x_1 :: l_1 = x_2 :: l_2 \rightarrow$
 $x_1 = x_2 \wedge l_1 = l_2.$

Proof using. `introv H. inverts` \times `H. Qed.`

Lemma `app_eq_nil_inv` : $\forall l_1 l_2,$
 $l_1 ++ l_2 = \text{nil} \rightarrow$
 $l_1 = \text{nil} \wedge l_2 = \text{nil}.$

Proof using. `intros. destruct` l_1 ; `destruct` l_2 ; `intros; tryfalse~`; `auto. Qed.`

Lemma `nil_eq_app_inv` : $\forall l_1 l_2,$
 $\text{nil} = l_1 ++ l_2 \rightarrow$
 $l_1 = \text{nil} \wedge l_2 = \text{nil}.$

Proof using. `intros. symmetry in` $H.$ `apply` \times `app_eq_nil_inv.` `Qed.`

Lemma `app_not_empty_l` : $\forall l_1 l_2,$
 $l_1 \neq \text{nil} \rightarrow$
 $l_1 ++ l_2 \neq \text{nil}.$

Proof using. `introv NE K. apply NE. forwards*`: `app_eq_nil_inv K. Qed.`

Lemma `app_not_empty_r` : $\forall l_1 l_2,$
 $l_2 \neq \text{nil} \rightarrow$
 $l_1 ++ l_2 \neq \text{nil}.$

Proof using. `introv NE K. apply NE. forwards*`: `app_eq_nil_inv K. Qed.`

Lemma `app_l_eq_self_inv` : $\forall l_1 l_2,$
 $l_1 ++ l_2 = l_1 \rightarrow$
 $l_2 = \text{nil}.$

Proof using.

`introv E. apply length_zero_inv.`
`lets:` $(\text{args_eq_1 } (@\text{length } A) E).$ `rew_list in` $H.$ `math.`

`Qed.`

Lemma `self_eq_app_l_inv` : $\forall l_1 l_2,$
 $l_1 = l_1 ++ l_2 \rightarrow$
 $l_2 = \text{nil}.$

Proof using. `intros. apply` \times `app_l_eq_self_inv.` `Qed.`

Lemma `app_r_eq_self_inv` : $\forall l_1 l_2,$
 $l_1 ++ l_2 = l_2 \rightarrow$
 $l_1 = \text{nil}.$

Proof using.

`introv E. apply length_zero_inv.`
`lets:` $(\text{args_eq_1 } (@\text{length } A) E).$ `rew_list in` $H.$ `math.`

`Qed.`

Lemma `self_eq_app_r_inv` : $\forall l_1 l_2,$
 $l_2 = l_1 ++ l_2 \rightarrow$

$l1 = \text{nil}.$

Proof using. intros. applys \times app_r_eq_self_inv. Qed.

Lemma app_cancel_l : $\forall l1 l2 l3,$

$l1 ++ l2 = l1 ++ l3 \rightarrow$

$l2 = l3.$

Proof using.

introv E. induction l1; rew_list in *. auto. inverts \times E.

Qed.

Lemma app_cancel_r : $\forall l1 l2 l3,$

$l1 ++ l3 = l2 ++ l3 \rightarrow$

$l1 = l2.$

Proof using.

intros l1. induction l1; introv E; rew_list in *.

{ rewrites $\neg (\gg \text{self_eq_app_r_inv } E)$. }

{ destruct l2; rew_list in *.

{ rewrite $\leftarrow \text{app_cons_l}$ in E. rewrites $\neg (\gg \text{self_eq_app_r_inv } (\text{eq_sym } E))$. }

{ inverts E. fequals. applys \times IHl1. }

Qed.

Lemma last_case : $\forall l,$

$l = \text{nil} \vee \exists x l', l = l' \& x.$

Proof using.

intros. destruct l. { left \times . }

{ right. forwards $\times (x \& l' \& H) : (\text{length_pos_inv_last } (a :: l))$.

rew_list. math. }

Qed.

Lemma last_eq_nil_inv : $\forall a l,$

$l \& a = \text{nil} \rightarrow$

False.

Proof using. introv E. induction l; rew_list; false. Qed.

Lemma nil_eq_last_inv : $\forall a l,$

$\text{nil} = l \& a \rightarrow$

False.

Proof using. intros. applys last_eq_nil_inv. Qed.

Lemma list_neq_nil_inv_last : $\forall l,$

$l \neq \text{nil} \rightarrow$

$\exists x q, l = q \& x.$

Proof using. introv N. destruct $\times (@\text{last_case } l)$. Qed.

Lemma last_eq_last_inv : $\forall x1 x2 l1 l2,$

$l1 \& x1 = l2 \& x2 \rightarrow$

$l1 = l2 \wedge x1 = x2.$

Proof using.

```
intros H. gen l2. induction l1; introv E; rew_list in E.  
{ destruct l2; rew_list in E; inverts E as E.  
  { auto. } { false nil_eq_last_inv E. } }  
{ destruct l2; rew_list in E.  
  { inverts E as E. false last_eq_nil_inv E. } }  
{ inverts E. forwards× [? ?]: IHl1.  
  split; congruence. } }
```

Qed.

Lemma nil_eq_middle_inv : $\forall x l1 l2,$

$\text{nil} = l1 \& x ++ l2 \rightarrow$

False.

Proof using. intros. destruct l1; inverts H. Qed.

Lemma cons_eq_middle_inv : $\forall x y l1 l2 l,$

$x :: l = l1 \& y ++ l2 \rightarrow$

$(l1 = \text{nil} \wedge x = y \wedge l = l2) \vee (\exists l1', l1 = x :: l1').$

Proof using.

```
intros. destruct l1; rew_list in H; inverts H. { left¬. } { right×. }
```

Qed.

Lemma last_eq_middle_inv : $\forall x y l1 l2 l,$

$l \& x = l1 \& y ++ l2 \rightarrow$

$(l = l1 \wedge x = y \wedge l2 = \text{nil}) \vee (\exists l2', l2 = l2' \& x).$

Proof using.

```
introsv E. destruct (last_case l2) as [(z&t&K)].
```

```
{ subst. rew_list in *. lets (?&?): last_eq_last_inv E. left×. }
```

```
{ subst. repeat rewrite ← app_assoc in E. lets (?&?): last_eq_last_inv E.
```

subst. right×. }

Qed.

Lemma list_middle_inv : $\forall n l,$

$n < \text{length } l \rightarrow$

$\exists l1 x l2, l = l1 ++ x :: l2 \wedge \text{length } l1 = n.$

Proof using.

```
intros. gen l. induction n; introv N;
```

```
destruct l as [|x l']; rew_list in *; try solve [ false; math ].
```

```
{  $\exists \neg (@\text{nil} A) x l'$ . }
```

```
{ forwards (l1 & x' & l2 & E & M): IHn l'. math.
```

$\exists (x :: l1) x' l2. \text{subst } l'. \text{rew_list}\neg.$

Qed.

End ApplInversion.

Arguments last_eq_nil_inv [A] [a] [l].

Arguments nil_eq_last_inv [A] [a] [l].

```

Arguments app_eq_nil_inv [A] [l1] [l2].
Arguments nil_eq_app_inv [A] [l1] [l2].
Arguments nil_eq_middle_inv [A] [x] [l1] [l2].
Arguments cons_eq_middle_inv [A] [x] [y] [l1] [l2] [l].

```

`mem x l` asserts that `x` belongs to `l`. Remark: it could be also defined as `Exists (=x) l`.

```

Inductive mem A (x:A) : list A → Prop :=
| mem_here : ∀ l,
  mem x (x :: l)
| mem_next : ∀ y l,
  mem x l →
  mem x (y :: l).

```

Section Mem.

Variables (A : Type).

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

Hint Constructors `mem`.

Induction

```

Lemma mem_induct : ∀ (x : A) (P : list A → Prop),
  (∀ l : list A, P (x :: l)) →
  (∀ (y : A) (l : list A), mem x l → x ≠ y → P l → P (y :: l)) →
  (∀ l : list A, mem x l → P l).

```

Proof using.

```

intros HH HN M. induction l.
inverts M.
tests: (x = a). auto. inverts M; auto_false×.

```

Qed.

Rewriting

```

Lemma mem_nil_eq : ∀ x,
  mem x nil = False.

```

Proof using. intros. extens. iff H; inverts H. Qed.

```

Lemma mem_cons_eq : ∀ x y l,
  mem x (y :: l) = (x = y ∨ mem x l).

```

Proof using. intros. extens. iff H; inverts× H. Qed.

```

Lemma mem_cons_eq_cases : ∀ x y l,
  mem x (y :: l) = ((x = y) ∨ (x ≠ y ∧ mem x l)).

```

Proof using.

```

intros. extens. tests: (x = y).
{ autos×. }
{ iff H. { inverts¬ H. } { destruct H as [|(?&?)]; subst×. } }

```

Qed.

Lemma mem_one_eq : $\forall x y,$
 $\mathbf{mem} x (y :: \mathbf{nil}) = (x = y).$

Proof using. intros. extens. iff H; inverts \neg H. false_invert. Qed.

Lemma mem_app_eq : $\forall l1 l2 x,$
 $\mathbf{mem} x (l1 ++ l2) = (\mathbf{mem} x l1 \vee \mathbf{mem} x l2).$

Proof using.

```
intros. extens. induction l1; rew_list.  
{ split. { auto. } { introv [H?]. inverts H. auto. } }  
{ iff M.  
  { inverts $\neg$  M. rewrite IHl1 in H0. destruct $\times$  H0. }  
  { destruct M. inverts $\neg$  H. constructors. rewrite $\neg$  IHl1.  
    constructors. rewrite $\neg$  IHl1. } }
```

Qed.

Lemma mem_last_eq : $\forall x y l,$
 $\mathbf{mem} x (l & y) = (\mathbf{mem} x l \vee x = y).$

Proof using. intros. rewrite mem_app_eq. rewrite \neg mem_one_eq. Qed.

Lemma mem_last_eq_cases : $\forall x y l,$
 $\mathbf{mem} x (l & y) = ((x \neq y \wedge \mathbf{mem} x l) \vee (x = y)).$

Proof using.

```
intros. extens. induction l; rew_list.  
{ tests: (x = y). { autos $\times$ . }  
  { iff M.  
    { inverts $\neg$  M. }  
    { destruct M as [(?&H)]. { inverts $\neg$  H. } { subst $\times$ . } } } }  
{ tests: (x = y). { autos $\times$ . }  
  { iff M.  
    { inverts M as M'; auto. rewrite IHl in M'. destruct $\times$  M'. }  
    { destruct M as [(?&H)].  
      { inverts $\neg$  H. constructors. rewrite $\times$  IHl. }  
      { constructors. rewrite $\neg$  IHl. } } } }
```

Qed.

Backward

Lemma mem_cons : $\forall l x y,$
 $x = y \vee \mathbf{mem} x l \rightarrow$
 $\mathbf{mem} x (y :: l).$

Proof using. intros. rewrite \times mem_cons_eq. Qed.

Lemma mem_cons_l : $\forall l x,$
 $\mathbf{mem} x (x :: l).$

Proof using. intros. rewrite \times mem_cons_eq. Qed.

Lemma mem_cons_r : $\forall l x y,$

```

mem x l →
mem x (y::l).

Proof using. intros. rewrite× mem_cons_eq. Qed.

Lemma mem_app : ∀ l1 l2 x,
  mem x l1 ∨ mem x l2 →
  mem x (l1 ++ l2).

Proof using. intros. rewrite¬ mem_app_eq. Qed.

Lemma mem_app_l : ∀ l1 l2 x,
  mem x l1 →
  mem x (l1 ++ l2).

Proof using. intros. applys× mem_app. Qed.

Lemma mem_app_r : ∀ l1 l2 x,
  mem x l2 →
  mem x (l1 ++ l2).

Proof using. intros. applys× mem_app. Qed.

Lemma mem_last : ∀ l x y,
  mem x l ∨ x = y →
  mem x (l & y).

Proof using. intros. rewrite× mem_last_eq. Qed.

Lemma mem_last_r : ∀ l x,
  mem x (l & x).

Proof using. intros. rewrite× mem_last_eq. Qed.

Lemma mem_last_l : ∀ l x y,
  mem x l →
  mem x (l & y).

Proof using. intros. rewrite× mem_last_eq. Qed.

Inversion

Lemma mem_nil_inv : ∀ x,
  mem x nil →
  False.

Proof using. introv E. inverts E. Qed.

Lemma mem_cons_inv : ∀ l x y,
  mem x (y::l) →
  x = y ∨ mem x l.

Proof using. introv E. rewrite× mem_cons_eq in E. Qed.

Lemma not_mem_cons_inv : ∀ x y l,
  ¬ mem x (y::l) →
  x ≠ y ∧ ¬ mem x l.

Proof using.
  introv M. split.

```

$\{ \text{intro_subst. } \text{false} \times M. \}$
 $\{ \text{intros } N. \text{false} \times M. \}$

Qed.

Lemma mem_cons_inv_cases : $\forall l x y,$

mem $x (y :: l) \rightarrow$
 $x = y \vee (x \neq y \wedge \text{mem } x l).$

Proof using. *introv E. rewrite \times mem_cons_eq_cases in E.* Qed.

Lemma mem_app_inv : $\forall x l1 l2,$

mem $x (l1 ++ l2) \rightarrow$
mem $x l1 \vee \text{mem } x l2.$

Proof using. *introv E. rewrite \neg mem_app_eq in E.* Qed.

Lemma mem_last_inv : $\forall l x y,$

mem $x (l & y) \rightarrow$
 $(x \neq y \wedge \text{mem } x l) \vee x = y.$

Proof using. *introv E. rewrite \times mem_last_eq_cases in E.* Qed.

Lemma mem_inv_middle_first : $\forall l x,$

mem $x l \rightarrow$
 $\exists l1 l2, l = l1 ++ x :: l2 \wedge \neg \text{mem } x l1.$

Proof using.

introv M. induction M.
 $\{ \exists (@\text{nil } A) l. \text{rewrite \times mem_nil_eq. } \}$
 $\{ \text{tests } C: (x = y).$
 $\{ \exists (@\text{nil } A) l. \text{rewrite \times mem_nil_eq. } \}$
 $\{ \text{destruct } IHM \text{ as } (l1 \& l2 \& E \& N). \exists (y :: l1) l2.$
 $\text{subst. rew_list. rewrite \times mem_cons_eq. } \} \}$

Qed.

Lemma mem_inv_middle : $\forall l x,$

mem $x l \rightarrow$
 $\exists l1 l2, l = l1 ++ x :: l2.$

Proof using. *introv E. forwards \times (?&?&?&?): mem_inv_middle_first E.* Qed.

Lemma eq_list_of_not_mem : $\forall l,$

$(\forall x, \neg \text{mem } x l) \rightarrow$
 $l = \text{nil}.$

Proof using. *introv P. destruct \neg l. false P. applys mem_here.* Qed.

End Mem.

Hint Rewrite mem_nil_eq mem_cons_eq mem_app_eq mem_last_eq : *rew_listx.*

Nth n L x asserts that the n-th element of the list L exists and is exactly x

Inductive Nth A : **nat** \rightarrow **list** A \rightarrow A \rightarrow Prop :=
| Nth_zero : $\forall l x,$

```

Nth 0 (x::l) x
| Nth_succ : ∀ y n l x,
  Nth n l x →
  Nth (S n) (y::l) x.

```

Section Nth.

Variables (A : Type).

Implicit Types l : **list** A.

Implicit Types x : A.

Implicit Types n : **nat**.

Hint Constructors **mem** Nth.

Lemma Nth_cons_match : ∀ n l x y,

```

Nth n (y::l) x =
  match n with
  | O ⇒ x = y
  | S n' ⇒ Nth (S n') (y::l) x
end.

```

Proof using.

```

intros. extens. destruct n as [|n'].
{ iff M. inverts¬ M. subst¬. }
{ iff M. inverts¬ M. subst¬. }

```

Qed.

LATER: add nth_last lemma

Lemma Nth_functional : ∀ n l x1 x2,

```

Nth n l x1 →
Nth n l x2 →
x1 = x2.

```

Proof using. introv H1. induction H1; intro H2; inverts¬ H2. Qed.

Lemma Nth_mem : ∀ l x n,

```

Nth n l x →
mem x l.

```

Proof using. introv N. induction N; autos×. Qed.

Lemma mem_Nth : ∀ l x,

```

mem x l →
∃ n, Nth n l x.

```

Proof using.

```

introv H. induction l; rew_listx in *.
{ inverts H. }
{ destruct H. { subst×. } { forwards× (n&?): IHl. } }

```

Qed.

Lemma Nth_inbound : ∀ n l x,

```

Nth n l x →

```

$n < \text{length } l$.

Proof using.

```
intros n. induction n; introv H; inverts H; rew_list in *.  
{ math. }  
{ forwards*: IHn. math. }
```

Qed.

Lemma Nth_inbound_inv : $\forall n l,$

$n < \text{length } l \rightarrow$
 $\exists x, \mathbf{Nth} n l x.$

Proof using.

```
induction n; introv N; destruct l as [| a l']; rew_list in N; try solve [math].  
{ eauto. }  
{ simpls. forwards (x&Hx): IHn l'. math.  $\exists x. \text{apply} \times \text{Nth_succ.}$  }
```

Qed.

Lemma Nth_app_l : $\forall n x l1 l2,$

$\mathbf{Nth} n l1 x \rightarrow$
 $\mathbf{Nth} n (l1 ++ l2) x.$

Proof using. intros n. induction n; introv H; inverts H; rew_list \times . Qed.

Lemma Nth_app_r : $\forall n x l1 l2,$

$\mathbf{Nth} n l2 x \rightarrow$
 $\mathbf{Nth} (n + \text{length } l1)\%nat (l1 ++ l2) x.$

Proof using.

```
intros. gen n. induction l1; introv H; rew_list.  
{ applys_eq  $\neg$  H. }  
{ applys_eq  $\times$  Nth_succ. }
```

Qed.

Lemma Nth_app_r' : $\forall n m x l1 l2,$

$\mathbf{Nth} m l2 x \rightarrow$
 $n = (m + \text{length } l1)\%nat \rightarrow$
 $\mathbf{Nth} n (l1 ++ l2) x.$

Proof using. intros. subst. applys \times Nth_app_r. Qed.

Lemma Nth_nil_inv : $\forall n x,$

$\mathbf{Nth} n \text{ nil } x \rightarrow$
 $\mathbf{False}.$

Proof using. introv H. inverts H. Qed.

Lemma Nth_inv_neq_nil : $\forall n l x,$

$\mathbf{Nth} n l x \rightarrow$
 $l \neq \text{nil}.$

Proof using. introv H. inverts H; auto_false. Qed.

Lemma Nth_cons_inv : $\forall n x y q,$

$\mathbf{Nth} n (y :: q) x \rightarrow$

```

  ( $n = 0 \wedge x = y$ )
 $\vee (\exists m, n = m+1 \wedge \mathbf{Nth} m q x).$ 

```

Proof using.

```
intros H. inverts H. { left×. } { right.  $\exists n0. \text{splits} \neg. \text{math.}$  }
```

Qed.

Lemma Nth_app_inv : $\forall n x l1 l2,$

```

Nth  $n (l1++l2) x \rightarrow$ 
  (Nth  $n l1 x)$ 
 $\vee (\exists m, n = \text{length } l1 + m \wedge \mathbf{Nth} m l2 x).$ 

```

Proof using.

```

intros. gen n. induction l1; intros H; rew_list in H.
{ right. rew_list.  $\exists \neg n.$  }
{ inverts H.
  { left $\neg.$  }
  { forwards× M: IHl1. destruct M as [(m&?&?)].
    { left $\neg.$  }
    { right×. rew_list.  $\exists m. \text{split} \neg. \text{math.}$  } } }

```

Qed.

Lemma Nth_last_inv : $\forall n x y l,$

```

Nth  $n (l\&y) x \rightarrow$ 
  (Nth  $n l x)$ 
 $\vee (n = \text{length } l \wedge y = x).$ 

```

Proof using.

```

intros H. lets [(m&E&F)]: Nth_app_inv H.
{ left $\neg.$  }
{ right. inverts F as G. { splits $\neg.$  math. } { inverts G. } }

```

Qed.

End Nth.

19.4.1 nth_default as a partial function with a default

```

Fixpoint nth_default A (d:A) (n:nat) (l:list A) {struct l} : A :=
  match l with
  | nil ⇒ d
  | x::l' ⇒
    match n with
    | 0 ⇒ x
    | S n' ⇒ nth_default d n' l'
    end
  end.

```

Section NthDef.

```

Variables (A:Type).
Implicit Types n : nat.
Implicit Types d x : A.
Implicit Types l : list A.
Hint Constructors Nth.

Lemma nth_default_nil : ∀ n d,
  nth_default d n nil = d.
Proof using. introv. destruct n. Qed.

Lemma nth_default_zero : ∀ x l d,
  nth_default d 0 (x::l) = x.
Proof using. introv. reflexivity. Qed.

Lemma nth_default_succ : ∀ n x l d,
  nth_default d (S n) (x::l) = nth_default d n l.
Proof using. introv. reflexivity. Qed.

Definition nth_default_cons := nth_default_succ.

Lemma nth_default_gt_length : ∀ l d n,
  n ≥ length l →
  nth_default d n l = d.
Proof using.
  induction l as [|l']; introv N; rew_list in *.
  { auto. }
  { destruct n as [|n']. { false; math. }
    simpl. rewrite¬ IHl. math. }
Qed.

Lemma nth_default_of_Nth : ∀ n l x d,
  Nth n l x →
  nth_default d n l = x.
Proof using. introv H. induction¬ H. Qed.

Lemma Nth_of_nth_default : ∀ l d n x,
  nth_default d n l = x →
  n < length l →
  Nth n l x.
Proof using.
  intros l. induction l; rew_list; introv E N.
  { false. math. }
  { destruct n; simpls.
    { subst×. }
    { constructors. applys× IHl. math. } }
Qed.

End NthDef.

```

Arguments nth_default [A] : simpl never.

Hint Rewrite nth_default_nil nth_default_zero nth_default_succ : rew_listx.

19.4.2 nth as a partial function

Definition nth '{IA:**Inhab** A} :=
nth_default (arbitrary (A:=A)).

Section NthFunc.

Context (A:Type) {IA: **Inhab** A}.

Implicit Types n : **nat**.

Implicit Types x : A.

Implicit Types l : **list** A.

Lemma nth_lt_length : $\forall x l,$

$\text{nth } 0 (x :: l) = x.$

Proof using. intros. apply nth_default_zero. Qed.

Lemma nth_zero : $\forall x l,$

$\text{nth } 0 (x :: l) = x.$

Proof using. intros. apply nth_default_zero. Qed.

Lemma nth_succ : $\forall n x l,$

$\text{nth } (\textcolor{red}{S} n) (x :: l) = \text{nth } n l.$

Proof using. intros. apply nth_default_succ. Qed.

Definition nth_cons := nth_succ.

Lemma nth_cons_case : $\forall n x l,$

$\text{nth } n (x :: l) = (\text{If } n = 0 \text{ then } x \text{ else } \text{nth } (n - 1) l).$

Proof using.

intros. destruct n as [|n'].

{ case_if. rewrite ← nth_zero. }

{ case_if. rewrite ← nth_succ. f_equal; math. }

Qed.

Lemma nth_pos : $\forall n x l,$

$n > 0 \rightarrow$

$\text{nth } n (x :: l) = \text{nth } (n - 1) l.$

Proof using.

intros. destruct n.

{ false. math. }

{ rewrite nth_succ. f_equal. math. }

Qed.

Lemma nth_gt_length : $\forall l n,$

$n \geq \text{length } l \rightarrow$

$\text{nth } n l = \text{arbitrary}.$

Proof using. *intros* N . *applys* \neg *nth_default_gt_length*. Qed.

Lemma *nth_of_Nth* : $\forall n l x,$

Nth $n l x \rightarrow$

$\text{nth } n l = x.$

Proof using. *intros* H . *apply* \neg *nth_default_of_Nth*. Qed.

Lemma *Nth_of_nth* : $\forall l n x,$

$\text{nth } n l = x \rightarrow$

$n < \text{length } l \rightarrow$

Nth $n l x.$

Proof using. *intros*. *applys* \times *Nth_of_nth_default*. Qed.

Lemma *Nth_nth* : $\forall l n,$

$n < \text{length } l \rightarrow$

Nth $n l (\text{nth } n l).$

Proof using. *intros*. *applys* \times *Nth_of_nth*. Qed.

Lemma *mem_nth* : $\forall l x,$

mem $x l \rightarrow$

$\exists n, \text{nth } n l = x.$

Proof using.

intros. *forwards* [$n P$]: *mem_Nth* H . $\exists n$. *apply* \neg *nth_of_Nth*.

Qed.

Lemma *nth_mem* : $\forall n l x,$

$\text{nth } n l = x \rightarrow$

$n < \text{length } l \rightarrow$

mem $x l.$

Proof using. *intros* $E N$. *forwards* \neg H : *Nth_of_nth* E . *applys* \times *Nth_mem* H . Qed.

Lemma *nth_app_l* : $\forall n l1 l2,$

$n < \text{length } l1 \rightarrow$

$\text{nth } n (l1 ++ l2) = \text{nth } n l1.$

Proof using.

intros N . *applys* *nth_of_Nth*. *applys* *Nth_app_l*. *applys* \neg *Nth_nth*.

Qed.

Lemma *nth_app_r* : $\forall n l1 l2,$

$n \geq \text{length } l1 \rightarrow$

$\text{nth } n (l1 ++ l2) = \text{nth } (n - \text{length } l1) l2.$

Proof using.

unfold nth. *intros* $n l1$. *gen* n . *induction* $l1$; *intros* N ; *rew_list* in *.

 { *fequals*; *math*. }

 { *destruct* n as [| n']. { *false*; *math*. }}

unfold nth_default; *fold nth_default*. *simpl*.

rewrite IHl1; [|*math*].

 { *fequals*; *math*. }

Qed.

Lemma nth_app : $\forall n l1 l2$,
 $\text{nth } n (l1 ++ l2) = \text{If } (n < \text{length } l1) \text{ then } (\text{nth } n l1) \text{ else } (\text{nth } (n - \text{length } l1) l2)$.

Proof using.

```
intros. case_if. { applys¬ nth_app_l. } { applys nth_app_r; math. }
```

Qed.

Lemma nth_middle : $\forall n l1 x l2$,

$n = \text{length } l1 \rightarrow$

$\text{nth } n (l1 ++ x :: l2) = x$.

Proof using.

```
introv E. rewrite nth_app_r.
```

```
{ math_rewrite (n - length l1 = 0). rewrite¬ nth_zero. }
```

math.

Qed.

Lemma nth_last_case : $\forall n x l$,

$\text{nth } n (l & x) = (\text{If } n = \text{length } l \text{ then } x \text{ else } \text{nth } n l)$.

Proof using.

```
intros. case_if.
```

```
{ rewrite nth_app_r; [|math]. math_rewrite (n - length l = 0).
```

rewrite¬ nth_zero. }

```
{ tests C': (n < length l).
```

{ rewrite¬ nth_app_l. }

```
{ rewrite nth_gt_length; [|rew_list; math].
```

rewrite nth_gt_length; [|math]. auto. } }

Qed.

End NthFunc.

Section NthFuncAux.

Context ($A:\text{Type}$) {IA: **Inhab** A}.

Implicit Types $n : \text{nat}$.

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

Lemma eq_of_extens : $\forall l1 l2$,

$\text{length } l1 = \text{length } l2 \rightarrow$

$(\forall n, n < \text{length } l1 \rightarrow \text{nth } n l1 = \text{nth } n l2) \rightarrow$

$l1 = l2$.

Proof using IA.

```
intros l1. induction l1; intros l2; destruct l2; simpl; introv HL HN;
```

```
try solve [ eauto | false ]. rew_list in *. f_equal.
```

```
{ forwards M: (rm HN) 0. math. do 2 rewrite nth_zero in M. auto. }
```

```
{ applys IHl1. { math. }}
```

```
{ intros n L. forwards M: (rm HN) (S n). math.
```

```
do 2 rewrite nth_cons in M. auto. } }
```

Qed.

End NthFuncAux.

Arguments nth [A] {IA}.

Global Opaque nth.

Hint Rewrite nth_zero nth_succ : rew_listx.

19.4.3 Rev

Definition rev A (l:list A) : list A :=
 $\text{fold_left} (\text{fun } x \text{ acc} \Rightarrow x :: \text{acc}) (@\text{nil} A) l.$

Section Rev.

Variables (A : Type).

Implicit Types x : A.

Implicit Types l : list A.

Lemma rev_nil :

$\text{rev} (@\text{nil} A) = \text{nil}.$

Proof using. auto. Qed.

Lemma rev_app : $\forall l1 l2,$

$\text{rev} (l1 ++ l2) = \text{rev} l2 ++ \text{rev} l1.$

Proof using.

intros. unfold rev. asserts K1: $(\forall l \text{ accu},$

$\text{fold_left} (\text{fun } x \text{ acc} \Rightarrow x :: \text{acc}) \text{ accu } l =$

$\text{fold_left} (\text{fun } x \text{ acc} \Rightarrow x :: \text{acc}) \text{ nil } l ++ \text{accu}).$

{ induction l; intros.

{ auto. }

{ rew_listx. rewrite IHl. rewrite (@IHl (a::nil)). rew_list¬. } }

asserts K2: $(\forall \text{ accu},$

$\text{fold_left} (\text{fun } x \text{ acc} \Rightarrow x :: \text{acc}) \text{ accu } (l1 ++ l2) =$

$\text{fold_left} (\text{fun } x \text{ acc} \Rightarrow x :: \text{acc}) \text{ nil } l2 ++$

$\text{fold_left} (\text{fun } x \text{ acc} \Rightarrow x :: \text{acc}) \text{ nil } l1 ++ \text{accu}).$

{ induction l1; intros.

{ rew_list. apply K1. }

{ rew_listx. rewrite IHl1. rewrite (@K1 l1 (a::nil)). rew_list¬. } }

lets K3: $(@\text{K2 nil}). \text{ rewrite app_nil_r in K3. auto.}$

Qed.

Lemma rev_cons : $\forall x l,$

$\text{rev} (x :: l) = \text{rev} l & x.$

Proof using. intros. rewrite ← app_cons_one_r. rewrite¬ rev_app. Qed.

Lemma rev_one : $\forall x,$

```
rev (x::nil) = x::nil.
```

Proof using. intros. rewrite \neg rev_cons. Qed.

Lemma rev_last : $\forall x l,$
 $\text{rev}(l & x) = x :: (\text{rev } l).$

Proof using. intros. rewrite \neg rev_app. Qed.

Lemma rev_rev : $\forall l,$
 $\text{rev}(\text{rev } l) = l.$

Proof using.

```
intros. induction $\neg$  l. { rewrite rev_cons, rev_last. f_equal. }
```

Qed.

Lemma rev_inj : $\forall l1 l2,$
 $\text{rev } l1 = \text{rev } l2 \rightarrow$
 $l1 = l2.$

Proof using.

```
introv E. forwards E': f_equal (@rev A) (rm E).
```

```
do 2 rewrite $\neg$  rev_rev in E'.
```

Qed.

Lemma mem_rev : $\forall l x,$
 $\text{mem } x l \rightarrow$
 $\text{mem } x (\text{rev } l).$

Proof using.

```
introv H. induction H.
```

```
{ rewrite rev_cons. apply mem_last_r. }
{ rewrite rev_cons. apply $\neg$  mem_last_l. }
```

Qed.

Lemma mem_rev_eq : $\forall l x,$
 $\text{mem } x (\text{rev } l) = \text{mem } x l.$

Proof using.

```
extens. iff M.
```

```
{ lets H: mem_rev M. rewrite $\neg$  rev_rev in H. }
{ apply mem_rev. }
```

Qed.

Lemma length_rev : $\forall l,$
 $\text{length } (\text{rev } l) = \text{length } l.$

Proof using. intros. induction \neg l. { rewrite rev_cons. rew_list \neg . } Qed.

Lemma fold_right_rev : $\forall B (f : A \rightarrow B \rightarrow B) i l,$
 $\text{fold_right } f i (\text{rev } l) = \text{fold_left } f i l.$

Proof using.

```
introv. gen i. induction $\neg$  l.
{ introv. rewrite rev_cons. rew_listx $\neg$ . }
```

Qed.

Lemma Nth_rev : $\forall n x l, \text{Nth } n l x \rightarrow$

Nth ($\text{length } l - 1 - n$) ($\text{rev } l$) $x.$

Proof using.

```
introv. sets_eq N: (\text{length } l). gen x n l.  
induction N; introv E I; destruct l; tryfalse.  
{ inverts I. }  
{ rewrite rev_cons. inverts I as I.  
  { applys Nth_app_r' 0.  
    { constructors. }  
    { rewrite length_rev. rewrite length_cons in E. math. } } }  
{ rew_list in E. lets K: IHN I. { math. } }  
apply Nth_app_l. applys_eq K. math. } }
```

Qed.

Lemma Nth_rev' : $\forall n x l,$

$n < \text{length } l \rightarrow$

Nth ($\text{length } l - 1 - n$) $l x \rightarrow$

Nth $n (\text{rev } l) x.$

Proof using.

```
introv L M. applys_eq (» Nth_rev M). math.
```

Qed.

Lemma Nth_rev_inv : $\forall n x l,$

Nth $n (\text{rev } l) x \rightarrow$

Nth ($\text{length } l - 1 - n$) $l x.$

Proof using.

```
introv N. lets K: Nth_rev N. rewrite rev_rev, length_rev in K. auto.
```

Qed.

Lemma nth_rev : $\forall \{Inhab A\} n l,$

$n < \text{length } l \rightarrow$

$\text{nth } n (\text{rev } l) = \text{nth } (\text{length } l - 1 - n) l.$

Proof using.

```
introv L. applys nth_of_Nth. applys¬ Nth_rev'.
```

$\text{applys¬ Nth_of_nth. math.}$

Qed.

End Rev.

Global Opaque rev.

Hint Rewrite rev_nil rev_app rev_cons rev_last rev_rev length_rev : rew_list.

Hint Rewrite rev_nil rev_app rev_cons rev_last rev_rev length_rev : rew_listx.

19.4.4 Inversion for rev

Section RevInversion.

Variables ($A : \text{Type}$).

Implicit Types $l : \text{list } A$.

Lemma rev_eq_nil_inv : $\forall l,$

$\text{rev } l = \text{nil} \rightarrow$

$l = \text{nil}.$

Proof using.

intros $l.$ destruct $l;$ $\text{rew_list};$ intros.

{ auto. } { false \times last_eq_nil_inv. }

Qed.

Lemma nil_eq_rev_inv : $\forall l,$

$\text{nil} = \text{rev } l \rightarrow$

$l = \text{nil}.$

Proof using. introv $H.$ apply \neg rev_eq_nil_inv. Qed.

Lemma app_rev_eq_nil_inv : $\forall l1\ l2,$

$l1 ++ \text{rev } l2 = \text{nil} \rightarrow$

$l1 = \text{nil} \wedge l2 = \text{nil}.$

Proof using.

intros. lets $H1\ H2:$ app_eq_nil_inv $H.$

applys_to $H2$ rev_eq_nil_inv. autos \times .

Qed.

Lemma nil_eq_app_rev_inv : $\forall l1\ l2,$

$\text{nil} = l1 ++ \text{rev } l2 \rightarrow$

$l1 = \text{nil} \wedge l2 = \text{nil}.$

Proof using. intros. apply \times app_rev_eq_nil_inv. Qed.

End RevInversion.

Arguments rev_eq_nil_inv [A] [l].

Arguments nil_eq_rev_inv [A] [l].

Arguments app_rev_eq_nil_inv [A] [$l1$] [$l2$].

Arguments nil_eq_app_rev_inv [A] [$l1$] [$l2$].

19.5 Make

```
Fixpoint make  $A$  ( $n:\text{nat}$ ) ( $v:A$ ) :  $\text{list } A :=$ 
  match  $n$  with
  | 0  $\Rightarrow$  nil
  | S  $n' \Rightarrow v :: \text{make } n' v$ 
  end.
```

Section Make.

Context (A :Type) {IA:**Inhab** A }.

Implicit Types $i n$: **nat**.

Implicit Types v : A .

Implicit Types l : **list** A .

Lemma make_zero : $\forall v,$

$\text{make } 0 \ v = \text{nil}.$

Proof using. auto. Qed.

Lemma make_succ : $\forall n v,$

$\text{make } (\mathbf{S} \ n) \ v = v :: (\text{make } n \ v).$

Proof using. auto. Qed.

Lemma make_pos : $\forall n v,$

$n > 0 \rightarrow$

$\text{make } n \ v = v :: (\text{make } (n - 1) \ v).$

Proof using.

introv E. destruct n.

{ *math.* }

{ *rewrite make_succ. fequals_rec. math.* }

Qed.

Lemma length_make : $\forall n v,$

$\text{length } (\text{make } n \ v) = n.$

Proof using.

intros. induction n.

{ *auto.* }

{ *rewrite make_succ. rewrite length_cons. math.* }

Qed.

Lemma Nth_make : $\forall i n v,$

$i < n \rightarrow$

Nth i ($\text{make } n \ v$) $v.$

Proof using.

Hint Constructors **Nth**.

introv. gen n; induction i; introv E; destruct n; try solve [false; math].

{ *constructors.* }

{ *rewrite make_succ. applys Nth_succ. applys¬ IHi. math.* }

Qed.

Lemma nth_make : $\forall i n v,$

$i < n \rightarrow$

$\text{nth } i \ (\text{make } n \ v) = v.$

Proof using. *intros. applys nth_of_Nth. applys¬ Nth_make.* Qed.

End Make.

Global Opaque make.

Hint Rewrite make_zero make_succ length_make : rew_listx.

19.6 Update as a function

```
Fixpoint update A (n:nat) (v:A) (l:list A) { struct l } : list A :=
  match l with
  | nil => nil
  | x::l' =>
    match n with
    | 0 => v::l'
    | S n' => x::update n' v l'
  end
end.
```

Section Update.

Context (A:Type) {IA: Inhab A}.

Implicit Types n : nat.

Implicit Types x v : A.

Implicit Types l : list A.

Lemma update_nil : $\forall n v,$
 $\text{update } n v \text{ nil} = \text{nil}.$

Proof using. auto. Qed.

Lemma update_cons_match : $\forall n v x l,$
 $\text{update } n v (x::l) =$
match n with
| 0 => v::l
| S n' => x::(update n' v l)
end.

Proof using. auto. Qed.

Lemma update_zero : $\forall v x l,$
 $\text{update } 0 v (x::l) = v::l.$

Proof using. auto. Qed.

Lemma update_succ : $\forall n v x l,$
 $\text{update } (S n) v (x::l) = x::(\text{update } n v l).$

Proof using. auto. Qed.

Definition update_cons := update_succ.

Lemma update_cons_case : $\forall n v x l,$
 $\text{update } n v (x::l) = (\text{If } n = 0 \text{ then } v::l \text{ else } x::(\text{update } (n-1) v l)).$

Proof using.

```
intros. destruct n as [|n']; case_if.
{ applys update_zero. }
```

```
{ rewrite update_cons. f_equal. rec. math. }
Qed.
```

```
Lemma update_cons_pos : ∀ n v x l,
n > 0 →
update n v (x::l) = x::(update (n-1) v l).
```

Proof using.

```
introv N. rewrite update_cons_case. case_if.
{ false. math. }
{ auto. }
```

Qed.

```
Lemma update_app_l : ∀ l1 l2 n v,
n < length l1 →
update n v (l1 ++ l2) = update n v l1 ++ l2.
```

Proof using IA.

```
intros l1. induction l1 as [| x l1' ]; introv E; rew_list in *.
{ f_equal. math. }
{ destruct n as [|n']. { do 2 rewrite update_zero. rew_list. }
{ do 2 rewrite update_cons. rew_list.
f_equal. erewrite IHl1'. math. } }
```

Qed.

```
Lemma update_app_r : ∀ l1 l2 n v,
n ≥ length l1 →
update n v (l1 ++ l2) = l1 ++ update (n - length l1) v l2.
```

Proof using IA.

```
intros l1. induction l1 as [| x l1' ]; introv E; rew_list in *.
{ f_equal. math. }
{ destruct n as [|n']. { false. math. }
{ rewrite update_cons. f_equal. erewrite IHl1'. math. } }
```

Qed.

```
Lemma update_middle : ∀ l1 l2 n x v,
n = length l1 →
update n v (l1 ++ x :: l2) = l1 & v ++ l2.
```

Proof using IA.

```
intros. subst. rew_list. rewrites update_app_r.
{ math_rewrite (length l1 - length l1 = 0). rew_list. }
{ math. }
```

Qed.

```
Lemma update_ge_length : ∀ n v l,
n ≥ length l →
update n v l = l.
```

Proof using.

```
intros E. gen n. induction l; rew_list; intros.  
{ auto. }  
{ rewrite update_cons_pos; [|math]. fequals. applys IHl. math. }
```

Qed.

Lemma length_update : $\forall n v l$,
 $\text{length}(\text{update } n v l) = \text{length } l$.

Proof using.

```
intros. gen n. induction l; intros.  
{ auto. }  
{ destruct n as [|n'].  
 { rewrite update_zero. rew_list¬. }  
 { rewrite update_succ. rew_list. rewrite¬ IHl. } }
```

Qed.

Lemma nth_update_same : $\forall n l v$,
 $n < \text{length } l \rightarrow$
 $\text{nth } n (\text{update } n v l) = v$.

Proof using.

```
intros n l. gen n. induction l; introv N; rew_list in N.  
{ false. math. }  
{ destruct n as [|n'].  
 { rewrite update_zero. rew_listx¬. }  
 { rewrite update_cons. rew_listx. applys× IHl. math. } }
```

Qed.

Lemma nth_update_neq : $\forall n m l v$,
 $n \neq m \rightarrow$
 $\text{nth } n (\text{update } m v l) = \text{nth } n l$.

Proof using.

```
intros n m l. gen n m. induction l; introv N.  
{ rewrite¬ update_nil. }  
{ destruct m as [|m'].  
 { rewrite update_zero. do 2 (rewrite nth_pos; [|math]). auto. }  
 { rewrite update_succ. destruct n as [|n'].  
 { rew_listx¬. }  
 { rew_listx. applys¬ IHl. } }
```

Qed.

Lemma update_nth_same : $\forall n l$,
 $n < \text{length } l \rightarrow$
 $\text{update } n (\text{nth } n l) l = l$.

Proof using.

```
intros E. gen n. induction l as [|x l']; intros.
```

```

{ rewrite length_nil in E. false. math. }
{ rewrite length_cons in E. rewrite update_cons_case.
  destruct n as [|n'|]; case_if.
  { subst. rewrite× nth_zero. }
  { fequals. rewrite nth_cons. math_rewrite (S n' - 1 = n')%nat.
    rewrite× IHl'. math. } }

```

Qed.

End Update.

Global Opaque update.

Hint Rewrite update_nil update_zero update_succ : rew_listx.

19.6.1

Definition map A B (f:A→B) (l:list A) : list B :=
 fold_right (fun x acc ⇒ (f x)::acc) (@nil B) l.

Section Map.

Variable (A B : Type).

Implicit Types x : A.

Implicit Types l : list A.

Implicit Types f : A → B.

Lemma map_nil : ∀ f,
 map f nil = nil.

Proof using. auto. Qed.

Lemma map_cons : ∀ f x l,
 map f (x::l) = f x :: map f l.

Proof using. auto. Qed.

Lemma map_app : ∀ f l1 l2,
 map f (l1 ++ l2) = map f l1 ++ map f l2.

Proof using.

intros. unfold map.

assert (∀ accu,

```

  fold_right (fun x acc ⇒ f x :: acc) accu (l1 ++ l2) =
  fold_right (fun x acc ⇒ f x :: acc) nil l1 ++
  fold_right (fun x acc ⇒ f x :: acc) nil l2 ++ accu).

```

{ induction l1; intros; simpl.

{ rew_list. gen accu.

induction l2; intros.

{ auto. }

{ rew_listx. rewrite¬ IHl2. } }

{ rew_listx. fequals. } }

specializes H (@**nil** B). $\text{rew_list} \vdash$ in H .

Qed.

Lemma $\text{map_last} : \forall f x l,$
 $\text{map } f (l & x) = \text{map } f l & f x.$

Proof using. `intros. rewrite¬ map_app.` Qed.

Lemma $\text{map_rev} : \forall f l,$
 $\text{map } f (\text{rev } l) = \text{rev } (\text{map } f l).$

Proof using.

`intros. induction¬ l.`
`{ rewrite map_cons. rew_list . rewrite map_last. rewrite¬ IHl. }`

Qed.

Lemma $\text{length_map} : \forall f l,$
 $\text{length } (\text{map } f l) = \text{length } l.$

Proof using.

`intros. induction¬ l.`
`{ rewrite map_cons. do 2 rewrite length_cons. auto. }`

Qed.

Lemma $\text{map_make} : \forall f n v,$
 $\text{map } f (\text{make } n v) = \text{make } n (f v).$

Proof using.

`intros. induction n as [|n'].`
`{ do 2 rewrite make_zero. auto. }`
`{ do 2 rewrite make_succ. rewrite map_cons. fequals. }`

Qed.

Lemma $\text{map_update} : \forall n f l x,$
 $n < \text{length } l \rightarrow$
 $\text{map } f (\text{update } n x l) = \text{update } n (f x) (\text{map } f l).$

Proof using.

`introv I. gen n. induction l; intros; rew_list in *.`
`{ false; math. }`
`{ destruct n as [|n'].`
`{ rewrite¬ update_zero. }`
`{ rewrite map_cons. do 2 rewrite update_cons. rewrite map_cons.`
`fequals. applys IHl. math. } }`

Qed.

Lemma $\text{map_eq_nil_inv} : \forall f l,$
 $\text{map } f l = \text{nil} \rightarrow$
 $l = \text{nil}.$

Proof using.

`introv E. destruct¬ l. rewrite map_cons in E. false.`

Qed.

```

Lemma map_eq_cons_inv : ∀ f l1 (x2:B) t2,
  map f l1 = x2::t2 →
  ∃ x1 t1, l1 = x1::t1 ∧ x2 = f x1 ∧ t2 = map f t1.

```

Proof using.

```

  introv E. destruct l1; [false].
  rewrite map_cons in E. inverts× E.

```

Qed.

```

Lemma map_eq_app_inv : ∀ f l r1 r2,
  map f l = r1 ++ r2 →
  ∃ l1 l2, l = l1++l2 ∧ r1 = map f l1 ∧ r2 = map f l2.

```

Proof using.

```

  intros f l. induction l as [|x l']; introv E.
  { rewrite map_nil in E. lets (?&?): nil_eq_app_inv E. subst.
    ∃x (@nil A) (@nil A). }
  { rewrite map_cons in E. destruct r1 as [|y r1']; rew_list in E.
    { ∃x (@nil A) (x::l'). }
    { invert E ;=> N1 N2. forwards (l1'&l2&E1&E2&E3): IHl' N2.
      ∃ (x::l1') l2. subst×. } }

```

Qed.

```

Lemma map_eq_middle_inv : ∀ f l r1 y r2,
  map f l = r1 ++ y :: r2 →
  ∃ l1 x l2, l = l1++x::l2 ∧ r1 = map f l1 ∧ y = f x ∧ r2 = map f l2.

```

Proof using.

```

  introv E. lets (l1&l2'&E1&E2&E3): map_eq_app_inv (rm E).
  lets (x&l2&E4&E5&E6): map_eq_cons_inv (eq_sym E3).
  ∃ l1 x l2. subst×.

```

Qed.

```

Lemma map_inj : ∀ f l1 l2,
  (∀ x y, f x = f y → x = y) →
  map f l1 = map f l2 →
  l1 = l2.

```

Proof using.

```

  intros f l1. induction l1; introv I E.
  { rewrite map_nil in E. forwards*: map_eq_nil_inv. }
  { destruct l2 as [|b l2]; tryfalse. do 2 rewrite map_cons in E.
    inverts E. fequals×. }

```

Qed.

```

Lemma map_congr : ∀ f1 f2 l,
  (∀ x, mem x l → f1 x = f2 x) →
  map f1 l = map f2 l.

```

Proof using.

```

introv H. induction l. { auto. }
{ do 2 rewrite map_cons. fequals.
  { applys H. applys mem_here. }
  { applys IHl. intros x Hx. applys H. applys× mem_next. } }
Qed.

```

End Map.

Global Opaque map.

Hint Rewrite map_nil map_cons map_app map_last length_map : rew_listx.

Lemma map_id : $\forall A (l:\text{list } A)$,
map id $l = l$.

Proof using. introv. induction¬ l. rew_listx. fequals¬. Qed.

Lemma map_id_ext : $\forall A (l:\text{list } A) (f:A \rightarrow A)$,
 $(\forall x, f x = x) \rightarrow$
map f $l = l$.

Proof using. introv E. cuts_rewrite (f = id). apply map_id. extens¬. Qed.

Lemma mem_map : $\forall (A B:\text{Type}) (f:A \rightarrow B) (l:\text{list } A) x$,
mem x $l \rightarrow$
mem (f x) (map f l).

Proof using. introv M. induction M; rew_listx; auto. Qed.

Lemma mem_map' : $\forall A B (l:\text{list } A) (f:A \rightarrow B) (x:A) (y:B)$,
mem x $l \rightarrow$
 $y = f x \rightarrow$
mem y (map f l).

Proof using. intros. subst. applys× mem_map. Qed.

Lemma map_map : $\forall A B C (l:\text{list } A) (f:A \rightarrow B) (g:B \rightarrow C)$,
map g (map f l) = map (fun x ⇒ g (f x)) l.

Proof using.

```

intros. induction l as [|x l'].
{ auto. }
{ repeat rewrite map_cons. fequals. }

```

Qed.

Lemma Nth_map : $\forall (A B:\text{Type}) (f:A \rightarrow B) (l:\text{list } A) n (x:A)$,
Nth n l x →
Nth n (map f l) (f x).

Proof using.

Hint Constructors **Nth**.

introv M. induction M; rew_listx; auto.

Qed.

Lemma Nth_map_inv : $\forall (A B:\text{Type}) (f:A \rightarrow B) (l:\text{list } A) n (y:B)$,
Nth n (map f l) y →

$\exists x, y = f x \wedge \text{Nth } n l x.$

Proof using.

```
intros N. gen n. induction l; introv N.
{ inverts N. }
{ rewrite map_cons in N. inverts N as N.
  { autos×. }
  { lets (x&E&N'): IHl N. ∃× x. } }
```

Qed.

Lemma nth_map : $\forall \{IA:\mathbf{Inhab} A\} \{IB:\mathbf{Inhab} B\} (f:A \rightarrow B) (l:\mathbf{list} A) n,$
 $n < \text{length } l \rightarrow$
 $\text{nth } n (\text{map } f l) = f (\text{nth } n l).$

Proof using.

```
intros N. applys nth_of_Nth. applys Nth_map. applys¬ Nth_of_nth.
```

Qed.

19.6.2 Concat

Definition concat A (m:list (list A)) : list A :=
fold_right (@app A) (@nil A) m.

Section Concat.

Variables (A : Type).

Implicit Types x : A.

Implicit Types l : list A.

Implicit Types m : list (list A).

Lemma concat_nil :

```
concat (@nil (list A)) = nil.
```

Proof using. auto. Qed.

Lemma concat_cons : $\forall l m,$

```
concat (l::m) = l ++ concat m.
```

Proof using. auto. Qed.

Lemma concat_one : $\forall l,$

```
concat (l::nil) = l.
```

Proof using.

```
intros. rewrite concat_cons. rewrite concat_nil. rew_list¬.
```

Qed.

Lemma concat_app : $\forall m1 m2 : \mathbf{list} (list A),$
 $\text{concat } (m1 ++ m2) = \text{concat } m1 ++ \text{concat } m2.$

Proof using.

```
intros m1. induction m1; intros.
{ rewrite concat_nil. rew_list¬. }
{ rew_list. do 2 rewrite concat_cons. rew_list. fequals. }
```

Qed.

```
Lemma concat_last : ∀ l m,  
  concat (m & l) = concat m ++ l.
```

Proof using. intros. rewrite \neg concat_app. rewrite \neg concat_one. Qed.

```
Lemma mem_concat_eq : ∀ m x,  
  mem x (concat m)  
  = ∃ l, mem l m ∧ mem x l.
```

Proof using.

Hint Constructors mem.

introv. extens. induction m.

```
{ simpl. iff I. { inverts I. } { destruct I as (?&H&?). inverts H. } }  
{ rewrite concat_cons. rewrite mem_app_eq. iff I (l&M&N).  
{ destruct I as [I|I].  
{ ∃ a.  
{ rewrite IHm in I. destruct $\times$  I as (l&?&?). } }  
{ inverts $\times$  M. } }
```

Qed.

```
Lemma concat_eq_nil_mem_inv : ∀ l m,  
  concat m = nil →  
  mem l m →  
  l = nil.
```

Proof using.

```
introv E M. induction M as [|x].  
{ rewrite concat_cons in E. forwards*: app_eq_nil_inv E. }  
{ rewrite concat_cons in E. destruct x.  
{ rew_list in E. applys $\neg$  IHM. }  
{ rew_list in E. false. } }
```

Qed.

End Concat.

Global Opaque concat.

Hint Rewrite concat_nil concat_cons concat_app concat_last : rew_listx.

19.6.3 Filter

filter P $|$ produces a list l' that is the sublist of $|$ made exactly of the elements of $|$ that satisfy P .

```
Definition filter A (P:A $\rightarrow$ Prop) l :=  
  fold_right (fun x acc => If P x then x::acc else acc) (@nil A) l.
```

Section Filter.

Variables (A : Type).

```

Implicit Types  $x : A$ .
Implicit Types  $l : \text{list } A$ .
Implicit Types  $P : A \rightarrow \text{Prop}$ .
Hint Constructors mem.

Lemma filter_nil :  $\forall P,$ 
   $\text{filter } P \text{ nil} = \text{nil}$ .
Proof using. auto. Qed.

Lemma filter_cons :  $\forall x l P,$ 
   $\text{filter } P (x :: l) = (\text{If } P x \text{ then } x :: \text{filter } P l \text{ else filter } P l)$ .
Proof using. auto. Qed.

Lemma filter_app :  $\forall l1 l2 P,$ 
   $\text{filter } P (l1 ++ l2) = \text{filter } P l1 ++ \text{filter } P l2$ .
Proof using. intros. unfold filter.
  assert ( $\forall accu,$ 
     $\text{fold\_right } (\text{fun } x acc \Rightarrow \text{If } P x \text{ then } x :: acc \text{ else acc}) accu (l1 ++ l2) =$ 
     $\text{fold\_right } (\text{fun } x acc \Rightarrow \text{If } P x \text{ then } x :: acc \text{ else acc}) \text{nil } l1 ++$ 
     $\text{fold\_right } (\text{fun } x acc \Rightarrow \text{If } P x \text{ then } x :: acc \text{ else acc}) \text{nil } l2 ++ accu$ ).
  { induction l1; intros.
    { rew_list. gen accu. induction l2; intros.
      { auto. }
      { rew_listx. case_if. rew_list. fequals. rewrite IHl2. fequals. } }
      { rew_listx. case_if. rew_list. fequals. rewrite IHl1. fequals. } }
    specializes H (@nil A). rewrite¬ app_nil_r in H.
  }
Qed.

Lemma filter_last :  $\forall x l P,$ 
   $\text{filter } P (l & x) = \text{filter } P l ++ (\text{If } P x \text{ then } x :: \text{nil} \text{ else nil})$ .
Proof using. intros. rewrite¬ filter_app. Qed.

Lemma filter_rev :  $\forall l P,$ 
   $\text{filter } P (\text{rev } l) = \text{rev } (\text{filter } P l)$ .
Proof using.
  intros. induction l.
  - rewrite !rev_nil, filter_nil. auto.
  - rewrite rev_cons, filter_cons, filter_last, IHl.
    case_if~; rew_listx¬.
Qed.

Lemma filter_eq_self_of_mem_implies_P :  $\forall l P,$ 
   $(\forall x, \text{mem } x l \rightarrow P x) \rightarrow$ 
   $\text{filter } P l = l$ .
Proof using.
  induction l; introv M.
  { auto. }

```

```

{ rewrite filter_cons. case_if.
  { fequals. applys IHl. introv Mx. applys× M. }
  { false× M. } }

```

Qed.

See also *filter_of_Forall*

Lemma **mem_filter_eq** : $\forall x P l,$
 $\mathbf{mem} x (\text{filter } P l) = (\mathbf{mem} x l \wedge P x).$

Proof using.

```

intros. extens. induction l.
{ rewrite filter_nil. iff M (M&?); inverts M. }
{ rewrite filter_cons. case_if; rew_listx; rewrite IHl.
  { iff [M|M] N; subst×. }
  { iff M ([N|N]&K); subst×. } }

```

Qed.

Lemma **mem_filter** : $\forall x P l,$
 $\mathbf{mem} x l \rightarrow$
 $P x \rightarrow$
 $\mathbf{mem} x (\text{filter } P l).$

Proof using. intros. rewrite× mem_filter_eq. Qed.

Lemma **mem_filter_inv** : $\forall x P l,$
 $\mathbf{mem} x (\text{filter } P l) \rightarrow$
 $\mathbf{mem} x l \wedge P x.$

Proof using. introv E. rewrite× mem_filter_eq in E. Qed.

Lemma **length_filter** : $\forall l P,$
 $\text{length } (\text{filter } P l) \leq \text{length } l.$

Proof using.

```

intros. induction l.
{ rewrite filter_nil. math. }
{ rewrite filter_cons. case_if; rew_list; math. }

```

Qed.

Lemma **filter_length_two_disjoint** : $\forall (P Q : A \rightarrow \text{Prop}) l,$
 $(\forall x, \mathbf{mem} x l \rightarrow P x \rightarrow Q x \rightarrow \mathbf{False}) \rightarrow$
 $\text{length } (\text{filter } P l) + \text{length } (\text{filter } Q l) \leq \text{length } l.$

Proof using.

```

introv. induction l; introv H.
{ rew_list. nat_math. }
{ specializes IHl. { intros. applys× H x. }
  repeat rewrite filter_cons. do 2 case_if; rew_list.
  { false× H. } { nat_math. } { nat_math. } { nat_math. } }

```

Qed.

Lemma **filter_length_partition** : $\forall P l,$

```

length (filter (fun x => P x) l)
+ length (filter (fun x =>  $\neg$  P x) l)
= length l.

```

Proof using.

```

intros. induction l.
{ rew_list. nat_math. }
{ repeat rewrite filter_cons. repeat case_if; rew_list; math. }

```

Qed.

```

Lemma length_filter_eq_mem_ge_one :  $\forall x l,$ 
  mem x l  $\rightarrow$ 
  length (filter (= x) l)  $\geq 1$ .

```

Proof using.

```

intros M. induction l.
{ inverts M. }
{ rewrite filter_cons. case_if.
  { rew_list. nat_math. }
  { inverts M. false. applys $\neg$  IHl. } }

```

Qed.

End Filter.

Global Opaque filter.

```

Hint Rewrite filter_nil filter_cons filter_app filter_last filter_rev
      mem_filter_eq : rew_listx.

```

19.6.4 Remove

```

Definition remove A (a:A) (l:list A) :=
  filter ( $\neq$  a) l.

```

Section Remove.

Variables (A : Type).

Implicit Types a x : A.

Implicit Types l : list A.

```

Lemma remove_as_filter :  $\forall a l,$ 
  remove a l = filter ( $\neq$  a) l.

```

Proof using. auto. Qed.

```

Lemma remove_not_mem :  $\forall x l,$ 
   $\neg$  mem x l  $\rightarrow$ 
  remove x l = l.

```

Proof using.

```

intros. applys filter_eq_self_of_mem_implies_P.
intros y My  $\rightarrow$ . false.

```

Qed.

Lemma mem_remove_eq : $\forall x a l,$
 $\mathbf{mem} x (\text{remove } a l) = (\mathbf{mem} x l \wedge x \neq a).$

Proof using. intros. applys \times mem_filter_eq. Qed.

Lemma mem_remove : $\forall a x l,$
 $\mathbf{mem} x l \rightarrow$
 $x \neq a \rightarrow$
 $\mathbf{mem} x (\text{remove } a l).$

Proof using. intros. rewrite \times mem_remove_eq. Qed.

Lemma mem_remove_inv : $\forall x a l,$
 $\mathbf{mem} x (\text{remove } a l) \rightarrow$
 $\mathbf{mem} x l \wedge x \neq a.$

Proof using. introv E. rewrite \times mem_remove_eq in E. Qed.

Lemma mem_remove_same_inv : $\forall x l,$
 $\mathbf{mem} x (\text{remove } x l) \rightarrow$
False.

Proof using. introv E. lets: mem_remove_inv E. false \times . Qed.

Lemma length_remove : $\forall l a,$
 $\text{length} (\text{remove } a l) \leq \text{length } l.$

Proof using. intros. applys length_filter. Qed.

Lemma length_remove_mem : $\forall x l,$
 $\mathbf{mem} x l \rightarrow$
 $\text{length} (\text{remove } x l) < \text{length } l.$

Proof using.

```
unfold remove. intros x l. induction l; rew_listx; introv M.  
{ false. }  
{ case_if; rew_list.  
 { destruct M; tryfalse. forwards~: IHl. math. }  
 { lets: length_filter l ( $\neq$  x). math. } }
```

Qed.

End Remove.

Global Opaque remove.

noduplicates L asserts that L does not contain any duplicated item.

Inductive noduplicates A : list A \rightarrow Prop :=
| noduplicates_nil : noduplicates nil
| noduplicates_cons : $\forall x l,$
 $\neg (\mathbf{mem} x l) \rightarrow$
 noduplicates l \rightarrow
 noduplicates (x :: l).

Section Noduplicates.

Variables (A : Type).

Implicit Types l : list A .

Hint Constructors noduplicates.

Lemma noduplicates_one : $\forall (x:A),$

noduplicates $(x::\text{nil})$.

Proof using.

```
intros. applys noduplicates_cons. { intros M. false × mem_nil_inv. }
applys noduplicates_nil.
```

Qed.

Lemma noduplicates_two : $\forall (x y:A),$

$x \neq y \rightarrow$

noduplicates $(x::y::\text{nil})$.

Proof using.

```
intros. applys noduplicates_cons.
{ intros M. rewrite mem_one_eq in M. false. }
applys noduplicates_one.
```

Qed.

Lemma noduplicates_app : $\forall l1 l2,$

noduplicates $l1 \rightarrow$

noduplicates $l2 \rightarrow$

($\forall x, \text{mem } x l1 \rightarrow \text{mem } x l2 \rightarrow \text{False}$) \rightarrow

noduplicates $(l1 ++ l2)$.

Proof using.

Hint Constructors mem.

```
intros l1. induction l1; introv N1 N2 EQ; rew_list.
```

{ auto. }

{ inverts N1 as N N1'. constructors.

{ rewrite mem_app_eq. rew_logic. }

{ applys¬ IHl1. introv Mx1 Mx2. applys× EQ x. } }

Qed.

Lemma noduplicates_app_inv : $\forall l1 l2,$

noduplicates $(l1 ++ l2) \rightarrow$

noduplicates $l1$

\wedge noduplicates $l2$

$\wedge \neg (\exists x, \text{mem } x l1 \wedge \text{mem } x l2)$.

Proof using.

```
introv ND. splits.
```

{ induction l1.

{ constructors. }

{ rew_list in ND. inverts ND as ND1 ND2.

```

    rewrite mem_app_eq in ND1. rew_logic× in ND1. } }

{ induction l1; rew_list in ND.
  { auto. }
  { inverts¬ ND. } }

{ introv (x&I1&I2). induction I1; rew_list in ND.
  { inverts ND as ND1 ND2. false ND1. apply× mem_app. }
  { apply IHI1. inverts¬ ND. } }

```

Qed.

Lemma noduplicates_length_le : $\forall l1\ l2,$
noduplicates $l1 \rightarrow$
 $(\forall x, \mathbf{mem}\ x\ l1 \rightarrow \mathbf{mem}\ x\ l2) \rightarrow$
 $\text{length } l1 \leq \text{length } l2.$

Proof using.

Hint Constructors **mem**.

```

introv NL ML. gen l2. induction l1 as [|a l1']; intros; rew_list.
{ math. }
{ inverts NL as HM NL'. sets_eq l2': (remove a l2).
  forwards H: length_remove_mem a l2. { applys× ML. }
  rewrite ← EQl2' in H.
  forwards~: IHl1' l2'.
  { introv N. tests: (x = a). subst l2'. applys× mem_remove. }
  math. }

```

Qed.

Lemma noduplicates_length_eq : $\forall l1\ l2,$
noduplicates $l1 \rightarrow$
noduplicates $l2 \rightarrow$
 $(\forall x, \mathbf{mem}\ x\ l1 \leftrightarrow \mathbf{mem}\ x\ l2) \rightarrow$
 $\text{length } l1 = \text{length } l2.$

Proof using.

```

introv H1 H2 EQ.
forwards~: noduplicates_length_le l1 l2. { intros. rewrite¬ ← EQ. }
forwards~: noduplicates_length_le l2 l1. { intros. rewrite¬ EQ. }
math.

```

Qed.

Lemma noduplicates_Nth_same : $\forall l,$
 $(\forall n1\ n2\ x,$
Nth $n1\ l\ x \rightarrow$
Nth $n2\ l\ x \rightarrow$
 $n1 = n2) \rightarrow$
noduplicates $l.$

Proof using.

introv NL. induction l; constructors.

```

{ introv I. lets (n&N): mem_Nth (rm I).
  forwards × Ab: NL Nth_zero Nth_succ. inverts Ab. }
{ apply IHl. introv N1 N2. forwards G: NL.
  applics Nth_succ N1. applics Nth_succ N2. inverts¬ G. }

```

Qed.

Lemma noduplicates_Nth_same_inv : $\forall l n1 n2 x,$
noduplicates $l \rightarrow$
Nth $n1 l x \rightarrow$
Nth $n2 l x \rightarrow$
 $n1 = n2.$

Proof using.

```

introv NL. gen n1 n2. induction NL; introv N1 N2.
{ inverts N1. }
{ inverts N1 as N1; inverts N2 as N2; autos¬.
  { apply Nth_mem in N2. false×. }
  { apply Nth_mem in N1. false×. } }

```

Qed.

Lemma noduplicates_filter : $\forall P l,$
noduplicates $l \rightarrow$
noduplicates (filter $P l$).

Proof using.

```

introv H. induction H; rew_listx. { auto. }
{ case_if.
  { constructors×. rew_listx×. }
  { auto. } }

```

Qed.

Lemma noduplicates_remove : $\forall a l,$
noduplicates $l \rightarrow$
noduplicates (remove $a l$).

Proof using.

```
intros. rewrite remove_as_filter. applics× noduplicates_filter.
```

Qed.

Lemma noduplicates_rev : $\forall l,$
noduplicates $l \rightarrow$
noduplicates (rev l).

Proof using.

```

introv M. apply¬ noduplicates_Nth_same. introv N1 N2.
lets N1': Nth_rev_inv N1. lets N2': Nth_rev_inv N2.
lets I1: Nth_inbound N1. lets I2: Nth_inbound N2.
rewrite length_rev in *.
lets: noduplicates_Nth_same_inv M N1' N2'. math.

```

Qed.

End Noduplicates.

remove_duplicates | produces a list l' that is the sublist of | obtained by keeping only the first occurrence of every item.

Fixpoint remove_duplicates A (l:list A) :=

```
match l with
| nil => nil
| x::l' => x :: (remove x (remove_duplicates l'))
```

end.

Section Remove_duplicates.

Variables (A : Type).

Implicit Types l : list A.

Hint Constructors mem noduplicates.

Lemma remove_duplicates_spec : $\forall l l',$

$l' = \text{remove_duplicates } l \rightarrow$

noduplicates l'

$\wedge (\forall x, \text{mem } x l' \leftrightarrow \text{mem } x l)$

$\wedge \text{length } l' \leq \text{length } l.$

Proof using.

introv E.

asserts (R1&R2): (**noduplicates** $l' \wedge (\forall x, \text{mem } x l' \leftrightarrow \text{mem } x l)).$

{ gen l' E. induction l; introv E; simpls.

{ subst. splits \times . }

{ sets_eq l'': (**remove_duplicates** l). forwards \neg [E' M]: IHl. subst l'. splits.

{ constructors. applys \neg **mem_remove_same_inv**. applys \neg **noduplicates_remove**. }

{ intros x. lets (M1&M2): M x. iff N.

{ inverts N as R. auto. lets: **mem_remove_inv** R. constructors \times . }

{ lets [E|(H1&H2)]: **mem_cons_inv_cases** N. { subst \times . }

{ constructors. applys \times **mem_remove**. } } }

splits \neg . applys \neg **noduplicates_length_le**. introv Hx. rewrite \neg ← R2.

Qed.

Lemma noduplicates_remove_duplicates : $\forall l,$

noduplicates (**remove_duplicates** l).

Proof using. introv. forwards*: **remove_duplicates_spec** l. Qed.

Lemma **mem_remove_duplicates** : $\forall x l,$

mem x (**remove_duplicates** l) = **mem** x l.

Proof using.

introv. extens. repeat rewrite ← **Mem_mem**. apply \neg **remove_duplicates_spec**.

Qed.

Lemma **length_remove_duplicates** : $\forall l,$

```

length (remove_duplicates l) ≤ length l.
Proof using. introv. forwards*: remove_duplicates_spec l. Qed.
End Remove_duplicates.

Hint Rewrite mem_remove_duplicates : rew_listx.

```

19.6.5 Combine

```

Fixpoint combine A B (r:list A) (s:list B) : list (A×B) :=
  match r, s with
  | nil, nil ⇒ nil
  | a::r', b::s' ⇒ (a,b)::(combine r' s')
  | _, _ ⇒ arbitrary
  end.

```

Section Combine.

Variable (A B : Type).

Implicit Types r : list A.

Implicit Types s : list B.

Lemma combine_nil :

combine (@nil A) (@nil B) = nil.

Proof using. auto. Qed.

Lemma combine_cons : $\forall x r y s,$

combine (x::r) (y::s) = (x,y)::(combine r s).

Proof using. auto. Qed.

Lemma combine_app : $\forall r1 r2 s1 s2,$

length r1 = length s1 \rightarrow

combine (r1++r2) (s1++s2) = (combine r1 s1)++(combine r2 s2).

Proof using.

intros r1. induction r1; introv E; destruct s1; tryfalse.

{ auto. }

{ rew_list in *. do 2 rewrite combine_cons. rew_list. rewrite \neg IHr1. }

Qed.

Lemma combine_last : $\forall x r y s,$

length r = length s \rightarrow

combine (r&x) (s&y) = (combine r s)&(x,y).

Proof using. introv E. applys \neg combine_app. Qed.

Lemma combine_rev : $\forall r s,$

length r = length s \rightarrow

combine (rev r) (rev s) = rev (combine r s).

Proof using.

intros r. induction r; introv E; destruct s; tryfalse.

```

{ auto. }
{ rew_list in *. rewrite combine_last, combine_cons.
  { rewrite IHr. rew_list¬. math. }
  { rew_list. math. } }

```

Qed.

Lemma length_combine : $\forall r s,$
 $\text{length } r = \text{length } s \rightarrow$
 $\text{length } (\text{combine } r s) = \text{length } r.$

Proof using.

```

intros r. induction r; introv E; destruct s; tryfalse.
{ auto. }
{ rewrite combine_cons. rew_list¬. }

```

Qed.

End Combine.

Lemma Nth_combine : $\forall A B n (r:\text{list } A) (s:\text{list } B) x y,$
 $\text{Nth } n r x \rightarrow$
 $\text{Nth } n s y \rightarrow$
 $\text{Nth } n (\text{combine } r s) (x, y).$

Proof using.

```

introv N1. gen s. induction N1; introv N2; inverts N2.
{ constructors. }
{ rewrite combine_cons. constructors¬. }

```

Qed.

Lemma nth_combine : $\forall \{IA:\text{Inhab } A\} \{IB:\text{Inhab } B\} n (r:\text{list } A) (s:\text{list } B),$
 $n < \text{length } r \rightarrow$
 $\text{length } r = \text{length } s \rightarrow$
 $\text{nth } n (\text{combine } r s) = (\text{nth } n r, \text{nth } n s).$

Proof using.

```

introv N E. applys nth_of_Nth. applys× Nth_combine.
{ applys× Nth_nth. } { applys× Nth_nth. math. }

```

Qed.

Global Opaque combine.

Hint Rewrite combine_nil combine_cons : rew_listx.

19.6.6 Split

Fixpoint split A B (l:list(A×B)) : (list A × list B) :=
 $\text{match } l \text{ with}$
 $| \text{nil} \Rightarrow (\text{nil}, \text{nil})$
 $| (a, b)::l' \Rightarrow \text{let } (la, lb) := \text{split } l' \text{ in } (a::la, b::lb)$
 end.

```

Section Split.

Variable (A B : Type).
Implicit Types l : list (A × B).
Implicit Types x : A.
Implicit Types y : B.
Implicit Types r : list A.
Implicit Types s : list B.

Lemma split_nil :
  split (@nil (A × B)) = (nil, nil).
Proof using. auto. Qed.

Lemma split_cons_let : ∀ x y l,
  split ((x, y) :: l) = let '(r, s) := split l in (x :: r, y :: s).
Proof using. auto. Qed.

Lemma split_cons : ∀ x y l r s,
  (r, s) = split l →
  split ((x, y) :: l) = (x :: r, y :: s).
Proof using.
  introv H. rewrite split_cons_let. rewrite ↙ ← H.
Qed.

Lemma split_app : ∀ l1 l2 r1 r2 s1 s2,
  (r1, s1) = split l1 →
  (r2, s2) = split l2 →
  split (l1 ++ l2) = (r1 ++ r2, s1 ++ s2).
Proof using.
  intros l1. induction l1 as [[x y] l1']; introv H1 H2.
  { rewrite split_nil in H1. inverts ↗ H1. }
  { rewrite split_cons_let in H1. destruct (split l1') as [r1' s1'].
    inverts H1. rew_list. rewrite split_cons_let.
    erewrite ↗ (IHl1' l2). }
Qed.

Lemma split_last : ∀ x y l r s,
  (r, s) = split l →
  split (l & (x, y)) = (r & x, s & y).
Proof using. introv H. erewrite split_app; f_equal. Qed.

Lemma split_length : ∀ l r s,
  (r, s) = split l →
  length r = length l ∧ length s = length l.
Proof using.
  intros l. induction l as [[x y] l']; introv E.
  { rewrite split_nil in E. inverts ↗ E. }
  { rewrite split_cons_let in E. destruct (split l') as [r' s']. }

```

inverts E. rew_list. forwards \neg (?&?): IHl'. }

Qed.

Lemma split_length_l : $\forall l r s,$
 $(r, s) = \text{split } l \rightarrow$
 $\text{length } r = \text{length } l.$

Proof using. introv E. forwards*: split_length E. Qed.

Lemma split_length_r : $\forall l r s,$
 $(r, s) = \text{split } l \rightarrow$
 $\text{length } s = \text{length } l.$

Proof using. introv E. forwards*: split_length E. Qed.

Lemma Nth_split : $\forall n l x y r s,$
 $\mathbf{Nth} n l (x, y) \rightarrow$
 $(r, s) = \text{split } l \rightarrow$
 $\mathbf{Nth} n r x \wedge \mathbf{Nth} n s y.$

Proof using.

Hint Constructors **Nth**.

```
introv N E. gen_eq p: (x, y). gen r s x y.
induction N as [l' [x' y'][x' y'] n' l' [x'' y'']]; intros.
{ inverts EQp. rewrite split_cons_let in E.
  destruct (split l') as [r' s']. inverts× E. }
{ inverts EQp. rewrite split_cons_let in E.
  destruct (split l') as [r' s'].
  forwards*: IHN. inverts× E. }
```

Qed.

Lemma nth_split : $\forall \{IA:\mathbf{Inhab} A\} \{IB:\mathbf{Inhab} B\} n l (r:\text{list } A) (s:\text{list } B),$
 $(r, s) = \text{split } l \rightarrow$
 $n < \text{length } l \rightarrow$
 $\text{nth } n l = (\text{nth } n r, \text{nth } n s).$

Proof using.

```
introv E N. applys nth_of_Nth. lets ([x y]&M): Nth_inbound_inv N.
forwards (F1&F2): Nth_split M E.
rewrite (nth_of_Nth F1). rewrite¬ (nth_of_Nth F2).
```

Qed.

Lemma nth_split_l : $\forall \{IA:\mathbf{Inhab} A\} \{IB:\mathbf{Inhab} B\} n l (r:\text{list } A) (s:\text{list } B),$
 $(r, s) = \text{split } l \rightarrow$
 $n < \text{length } l \rightarrow$
 $\text{nth } n r = \text{fst } (\text{nth } n l).$

Proof using. introv E N. rewrites¬ (» nth_split E N). Qed.

Lemma nth_split_r : $\forall \{IA:\mathbf{Inhab} A\} \{IB:\mathbf{Inhab} B\} n l (r:\text{list } A) (s:\text{list } B),$
 $(r, s) = \text{split } l \rightarrow$
 $n < \text{length } l \rightarrow$

```

nth n s = snd (nth n l).
Proof using. introv E N. rewrites¬ (» nth_split E N). Qed.
End Split.

Global Opaque split.

Hint Rewrite split_nil : rew_listx.

```

19.6.7 Take

```

Fixpoint take A (n:nat) (l:list A) : list A :=
  match n with
  | 0 ⇒ nil
  | S n' ⇒ match l with
    | nil ⇒ nil
    | a::l' ⇒ a::(take n' l')
  end
end.

```

Section Take.

Variables (A : Type).

Implicit Types n : nat.

Implicit Types x : A.

Implicit Types l : list A.

Lemma take_nil : $\forall n,$
 $\text{take } n (@\text{nil } A) = \text{nil}.$

Proof using. intros. destruct n; auto. Qed.

Lemma take_zero : $\forall l,$
 $\text{take } 0 l = \text{nil}.$

Proof using. auto. Qed.

Lemma take_succ : $\forall x l n,$
 $\text{take } (S n) (x :: l) = x :: (\text{take } n l).$

Proof using. auto. Qed.

Definition take_cons := take_succ.

Lemma take_cons_pos : $\forall x l n,$
 $(n > 0) \rightarrow$
 $\text{take } n (x :: l) = x :: (\text{take } (n - 1) l).$

Proof using.

```

introv H. destruct n. false; math.
rewrite take_cons. fequals_rec. math.

```

Qed.

Lemma take_ge : $\forall n l,$

$(n \geq \text{length } l) \rightarrow$
 $\text{take } n \ l = l.$

Proof using.

```
induction n; destruct l; introv H; rew_list in *; auto.
{ false. math. }
{ rewrite take_cons. fequals. applys IHn. math. }
```

Qed.

Lemma take_is_prefix : $\forall n l,$
 $\exists q, l = \text{take } n \ l ++ q.$

Proof using.

```
intros n l. gen n. induction l; intros.
{  $\exists (@\text{nil} A).$  rewrite $\neg$  take_nil. }
{ destruct n as [|n']. 
  { rewrite take_zero.  $\exists \neg (a :: l).$  }
  { rewrite take_succ. forwards (q'&E): IHl n'.  $\exists q'.$ 
    rew_list. congruence. } }
```

Qed.

Lemma take_app_l : $\forall n l l',$
 $(n \leq \text{length } l) \rightarrow$
 $\text{take } n \ (l ++ l') = \text{take } n \ l.$

Proof using.

```
induction n; destruct l; introv H; rew_list in *; auto.
{ false. math. }
{ do 2 rewrite take_cons. fequals. applys IHn. math. }
```

Qed.

Lemma take_app_r : $\forall n l l',$
 $(n \geq \text{length } l) \rightarrow$
 $\text{take } n \ (l ++ l') = l ++ \text{take } (n - \text{length } l) \ l'.$

Proof using.

```
intros. gen n. induction l; introv H.
{ rewrite length_nil in *. do 2 rewrite app_nil_l.
  fequals. math. }
{ rewrite length_cons in *. destruct n as [|n']. 
  { false. math. }
  { rew_list. rewrite take_cons. fequals. applys IHl. math. } }
```

Qed.

Lemma take_prefix_length : $\forall l l',$
 $\text{take } (\text{length } l) \ (l ++ l') = l.$

Proof using.

```
intros. rewrite take_app_r; [|math].
math_rewrite ( $\forall a, a - a = 0$ ).
```

```

rewrite take_zero. rew_list $\vdash$ .
Qed.

Lemma take_full_length :  $\forall l,$ 
  take (length  $l$ )  $l = l$ .
Proof using.
  intros. lets  $H$ : (@take_prefix_length  $l$  nil). rew_list $\vdash$  in  $H$ .
Qed.

End Take.

```

```

Global Opaque take.

Hint Rewrite take_nil take_zero take_succ : rew_listx.

```

19.6.8 Drop

```

Fixpoint drop  $A$  ( $n:\text{nat}$ ) ( $l:\text{list } A$ ) :  $\text{list } A :=$ 
  match  $n$  with
  | 0  $\Rightarrow$   $l$ 
  |  $S n'$   $\Rightarrow$  match  $l$  with
    | nil  $\Rightarrow$  nil
    |  $a::l' \Rightarrow$  drop  $n' l'$ 
    end
  end.

```

```

Section Drop.

Variables ( $A$  : Type).
Implicit Types  $n$  : nat.
Implicit Types  $x$  :  $A$ .
Implicit Types  $l$  :  $\text{list } A$ .

```

```

Lemma drop_nil :  $\forall n,$ 
  drop  $n$  (@nil  $A$ ) = nil.
Proof using. intros. destruct  $n$ ; auto. Qed.

Lemma drop_zero :  $\forall l,$ 
  drop 0  $l = l$ .
Proof using. auto. Qed.

```

```

Lemma drop_succ :  $\forall x l n,$ 
  drop ( $S n$ ) ( $x::l$ ) = (drop  $n l$ ).
Proof using. auto. Qed.

```

```

Definition drop_cons := drop_succ.

Lemma drop_cons_pos :  $\forall x l n,$ 
  ( $n > 0$ )  $\rightarrow$ 
  drop  $n$  ( $x::l$ ) = drop ( $n-1$ )  $l$ .

```

Proof using.

```
intros H. destruct n. false; math.
  rewrite drop_cons. fequals_rec. math.
```

Qed.

Lemma drop_is_suffix : $\forall n l,$
 $\exists q, l = q ++ \text{drop } n l.$

Proof using.

```
intros n l. gen n. induction l; intros.
{  $\exists (@\text{nil } A).$  rewrite  $\neg$  drop_nil. }
{ destruct n as [|n'].  

  { rewrite drop_zero.  $\exists \neg (@\text{nil } A).$  }  

  { rewrite drop_succ. forwards (q'&E): IHl n'.  $\exists (a :: q').$   

    rew_list. congruence. } }
```

Qed.

Lemma drop_app_l : $\forall n l l',$
 $(n \leq \text{length } l) \rightarrow$
 $\text{drop } n (l ++ l') = \text{drop } n l ++ l'.$

Proof using.

```
induction n; destruct l; introv H; rew_list in *; auto.
{ false. math. }
{ do 2 rewrite drop_cons. fequals. applys IHn. math. }
```

Qed.

Lemma drop_app_r : $\forall n l l',$
 $(n \geq \text{length } l) \rightarrow$
 $\text{drop } n (l ++ l') = \text{drop } (n - \text{length } l) l'.$

Proof using.

```
induction n; destruct l; introv H; rew_list in *; auto.
{ false. math. }
{ rewrite drop_cons. rewrite IHn. fequals. math. }
```

Qed.

Lemma drop_app_length : $\forall l l',$
 $\text{drop}(\text{length } l) (l ++ l') = l'.$

Proof using.

```
intros. rewrite drop_app_r; [|math].
math_rewrite ( $\forall a, a - a = 0$ ).
rewrite drop_zero. rew_list  $\neg$ .
```

Qed.

Lemma drop_at_length : $\forall l,$
 $\text{drop}(\text{length } l) l = \text{nil}.$

Proof using.

```
intros. lets H: (@drop_app_length l nil). rew_list  $\neg$  in H.
```

Qed.

End Drop.

Global Opaque drop.

Hint Rewrite drop_nil drop_zero drop_succ : rew_listx.

19.6.9 Take and drop decomposition of a list

Section TakeAndDrop.

Variables (A : Type).

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

Lemma take_app_drop_spec : $\forall n l f r,$

$f = \text{take } n l \rightarrow$

$r = \text{drop } n l \rightarrow$

$n \leq \text{length } l \rightarrow$

$l = f ++ r$

$\wedge \text{length } f = n$

$\wedge \text{length } r = \text{length } l - n.$

Proof using.

intros n . induction n ; introv $F R L$.

{ subst. rew_listx. splits \neg . math. }

{ destruct l ; rew_listx in L .

{ rew_listx in L . false. math. }

{ forwards \neg ($F' \& R' \& L'$): ($\gg IHn l (\text{take } n l) r$). { math. }}

subst f . rew_listx. splits. { fequals. } { math. } { math. } }

Qed.

Lemma take_spec : $\forall n l,$

$n \leq \text{length } l \rightarrow$

$\exists l', \text{length } (\text{take } n l) = n$

$\wedge l = (\text{take } n l) ++ l'$.

Proof using. introv E . forwards \times ($E1 \& E2 \& E3$): take_app_drop_spec. Qed.

Lemma length_take : $\forall n l,$

$n \leq \text{length } l \rightarrow$

$\text{length } (\text{take } n l) = n.$

Proof using. introv E . forwards \neg ($l' \& N \& M$): take_spec $n l$. Qed.

Lemma drop_spec : $\forall n l,$

$n \leq \text{length } l \rightarrow$

$\exists l', \text{length } l' = n$

$\wedge l = l' ++ (\text{drop } n l).$

Proof using. introv E . forwards \times ($E1 \& E2 \& E3$): take_app_drop_spec. Qed.

```

Lemma length_drop : ∀ n l,
  n ≤ length l →
  length (drop n l) = length l - n.

```

Proof using.

```

  introv E. forwards¬ (l'&N&M): drop_spec n l.
  pattern l at 2. rewrite M. rew_list. math.

```

Qed.

```

Lemma list_eq_take_app_drop : ∀ n l,
  n ≤ length l →
  take n l ++ drop n l = l.

```

Proof using. introv H. forwards*: take_app_drop_spec n l. Qed.

End TakeAndDrop.

Arguments take_app_drop_spec [A].

Arguments take_spec [A].

Arguments drop_spec [A].

19.6.10 TakeDropLast

`take_drop_last` | returns a pair (`q,x`) such that `| = q & x`

```

Fixpoint take_drop_last ‘{IA:Inhab A} (l:list A) : (list A)*A :=
  match l with
  | nil ⇒ arbitrary
  | x::l' ⇒
    match l' with
    | nil ⇒ (nil, x)
    | _ ⇒ let (t,y) := take_drop_last l' in
      (x::t, y)
  end
end.

```

Section TakeDropLast.

Context (A:Type) {IA:Inhab A}.

Implicit Types `x` : `A`.

Implicit Types `l` : `list A`.

Lemma take_drop_last_cons : ∀ (x:A) (l: list A),

```

  take_drop_last (x::l) =
  match l with
  | nil ⇒ (nil, x)
  | _::_ ⇒ let (t, y) := take_drop_last l in (x :: t, y)
  end.

```

Proof using. auto. Qed.

```

Lemma take_drop_last_spec : ∀ (x:A) (l l': list A),
  (l', x) = take_drop_last l →
  l ≠ nil →
  l = l' & x.

```

Proof using.

```

induction l as [| a t]; introv E N. false.
rewrite take_drop_last_cons in E.
destruct (take_drop_last t) as [u r].
{ destruct t; inverts E. rewrite× app_nil_l.
  rew_list. fequals. applys IHt; auto_false×. }

```

Qed.

```

Lemma take_drop_last_length : ∀ l l' x,
  (l', x) = take_drop_last l →
  l ≠ nil →
  length l' = length l - 1.

```

Proof using.

```

introv E N. forwards: take_drop_last_spec E N.
subst l. rewrite length_last. math.

```

Qed.

End TakeDropLast.

Global Opaque take_drop_last.

Arguments take_drop_last [A] {IA}.

Arguments take_drop_last_spec [A] {IA}.

Arguments take_drop_last_length [A] {IA}.

Forall P | asserts that all the elements in the list | satisfy the predicate P .

```

Inductive Forall A (P:A→Prop) : list A → Prop :=
| Forall_nil :
  Forall P nil
| Forall_cons : ∀ l x,
  P x →
  Forall P l →
  Forall P (x::l).

```

Section ForallProp.

Variables A : Type.

Implicit Types l : list A.

Implicit Types P : A→Prop.

Hint Constructors Forall.

Lemma Forall_nil_eq : ∀ P,

Forall P nil = True.

Proof using. auto. Qed.

Lemma Forall_cons_eq : $\forall P l x,$
Forall $P (x :: l) = (P x \wedge \text{Forall } P l).$

Proof using.

intros. extens. iff M. { inverts \times M. } { constructors \times . }

Qed.

Lemma Forall_one_eq : $\forall P x,$
Forall $P (x :: \text{nil}) = (P x).$

Proof using.

intros. extens. rewrite Forall_cons_eq. rewrite \times Forall_nil_eq.

Qed.

Lemma Forall_app_eq : $\forall P l1 l2,$
Forall $P (l1 ++ l2) = (\text{Forall } P l1 \wedge \text{Forall } P l2).$

Proof using.

intros. extens. induction l1; rew_list.

{ autos \times . }

{ iff M (M1&M2). { inverts \times M. } { inverts \times M1. } }

Qed.

Lemma Forall_last_eq : $\forall P l x,$
Forall $P (l \& x) = (\text{Forall } P l \wedge P x).$

Proof using.

intros. extens. induction l; rew_list.

{ iff M (M1&M2). { inverts \times M. } { autos \times . } }

{ iff M (M1&M2). { inverts \times M. } { inverts \times M1. } }

Qed.

Lemma Forall_app : $\forall P l1 l2,$
Forall $P l1 \rightarrow$
Forall $P l2 \rightarrow$
Forall $P (l1 ++ l2).$

Proof using. intros. rewrite \times Forall_app_eq. Qed.

Lemma Forall_last : $\forall P l x,$
Forall $P l \rightarrow$
 $P x \rightarrow$
Forall $P (l \& x).$

Proof using. intros. rewrite \times Forall_last_eq. Qed.

Lemma Forall_cons_inv : $\forall P l x,$
Forall $P (x :: l) \rightarrow$
 $P x \wedge \text{Forall } P l.$

Proof using. introv H. rewrite \times Forall_cons_eq in H. Qed.

Lemma Forall_cons_inv_head : $\forall P l x,$
Forall $P (x :: l) \rightarrow$

$P\ x.$

Proof using. *introv H. rewrite \times Forall_cons_eq in H. Qed.*

Lemma Forall_cons_inv_tail : $\forall P\ l\ x,$

Forall $P\ (x::l) \rightarrow$

Forall $P\ l.$

Proof using. *introv H. rewrite \times Forall_cons_eq in H. Qed.*

Lemma Forall_app_inv : $\forall P\ l1\ l2,$

Forall $P\ (l1\ ++\ l2) \rightarrow$

Forall $P\ l1 \wedge \text{Forall } P\ l2.$

Proof using. *introv H. rewrite \times Forall_app_eq in H. Qed.*

Lemma Forall_last_inv : $\forall P\ l\ x,$

Forall $P\ (l\ &\ x) \rightarrow$

Forall $P\ l \wedge P\ x.$

Proof using. *introv H. rewrite \times Forall_last_eq in H. Qed.*

Lemma Forall_make : $\forall n\ v\ P,$

$P\ v \rightarrow$

Forall $P\ (\text{make } n\ v).$

Proof using.

introv N. induction n as [|n'].

{ *rewrite make_zero. applys \times Forall_nil.* }

{ *rewrite make_succ. applys \times Forall_cons.* }

Qed.

Lemma Forall_eq_forall_mem : $\forall P\ l,$

Forall $P\ l = (\forall x, \text{mem } x\ l \rightarrow P\ x).$

Proof using.

extens. introv. induction l; iff I.

{ *introv IN. inverts IN.* }

{ *auto.* }

{ *introv IN. rew_listx in IN. inverts I. destruct IN; subst \times .* }

{ *constructors.* }

{ *apply I. rew_listx \times .* }

{ *apply \neg IHl. introv IN. apply \neg I. rew_listx \times .* } }

Qed.

Lemma Forall_mem_inv : $\forall P\ l\ x,$

Forall $P\ l \rightarrow$

mem $x\ l \rightarrow$

$P\ x.$

Proof using. *introv F I. rewrite Forall_eq_forall_mem in F. apply \neg F. Qed.*

Lemma Forall_Nth_inv : $\forall P\ n\ l\ x,$

Forall $P\ l \rightarrow$

Nth $n l x \rightarrow P x.$

Proof using. *introv F N. applys \times Forall_mem_inv F. applys \times Nth_mem.* Qed.

Lemma Forall_nth_inv : $\forall \{IA:\mathbf{Inhab} A\} P n l,$

Forall $P l \rightarrow$
 $n < \text{length } l \rightarrow$
 $P (\text{nth } n l).$

Proof using. *introv F N. applys Forall_Nth_inv n F. applys \neg Nth_nth.* Qed.

Lemma Forall_rev : $\forall P l,$

Forall $P l \rightarrow$
Forall $P (\text{rev } l).$

Proof using.

introv E. induction l.
{ auto. }
{ *rew_list. rewrite Forall_last_eq. inverts \times E.* }

Qed.

Lemma Forall_rev_eq : $\forall P l,$

Forall $P (\text{rev } l) = \mathbf{Forall} P l.$

Proof using.

intros. extens. iff M.
{ *rewrite \leftarrow rev_rev. applys \neg Forall_rev.* }
{ *applys \neg Forall_rev.* }

Qed.

Lemma Forall_take : $\forall P n l,$

Forall $P l \rightarrow$
Forall $P (\text{take } n l).$

Proof using.

introv E. forwards (q&K): take_is_prefix n l.
rewrite K in E. rewrite \times Forall_app_eq in E.

Qed.

Lemma Forall_drop : $\forall P n l,$

Forall $P l \rightarrow$
Forall $P (\text{drop } n l).$

Proof using.

introv E. forwards (q&K): drop_is_suffix n l.
rewrite K in E. rewrite \times Forall_app_eq in E.

Qed.

Lemma Forall_pred_incl : $\forall P Q l,$

Forall $P l \rightarrow$
 $\text{pred_incl } P Q \rightarrow$
Forall $Q l.$

Proof using. *introv*. induction l ; *introv* $H L$; *inverts* \times H . Qed.

Lemma filter_eq_self_of_Forall : $\forall l P$,

Forall $P l \rightarrow$

filter $P l = l$.

Proof using.

introv M. rewrite Forall_eq_forall_mem in M.

applys \times filter_eq_self_of_mem_implies_P.

Qed.

Lemma Forall_filter_same : $\forall P l$,

Forall $P (\text{filter } P l)$.

Proof using.

introv. induction l.

{ *rewrite filter_nil. constructors* \neg . }

{ *rewrite filter_cons. cases_if* \neg . }

Qed.

Lemma Forall_filter_pred_incl : $\forall P Q l$,

pred_incl $P Q \rightarrow$

Forall $Q (\text{filter } P l)$.

Proof using.

introv E. applys \neg Forall_pred_incl P . *applys* Forall_filter_same.

Qed.

End ForallProp.

Hint Rewrite Forall_nil_eq Forall_cons_eq Forall_app_eq Forall_last_eq
Forall_rev_eq : *rew_listx*.

Forall2 $P L1 L2$ asserts that the lists $L1$ and $L2$ have the same length and that elements at corresponding indices are related by the binary relation P .

Inductive Forall2 A B ($P:A \rightarrow B \rightarrow \text{Prop}$) : list A \rightarrow list B \rightarrow Prop :=

| Forall2_nil :

Forall2 $P \text{ nil nil}$

| Forall2_cons : $\forall s r x y$,

$P x y \rightarrow$

Forall2 $P s r \rightarrow$

Forall2 $P (x::s) (y::r)$.

Section Forall2.

Variables A B : Type.

Implicit Types x : A.

Implicit Types y : B.

Implicit Types r : list A.

Implicit Types s : list B.

Implicit Types P : A \rightarrow B \rightarrow Prop.

Hint Constructors **Forall2**.

Lemma Forall2_inv_length : $\forall P r s,$

Forall2 $P r s \rightarrow$

$\text{length } r = \text{length } s.$

Proof using. *introv H. induction H; rew_list; math. Qed.*

Lemma Forall2_app : $\forall P r1 s1 r2 s2,$

Forall2 $P r1 s1 \rightarrow$

Forall2 $P r2 s2 \rightarrow$

Forall2 $P (r1 ++ r2) (s1 ++ s2).$

Proof using. *introv H H'. induction H; rew_list¬. Qed.*

Lemma Forall2_last : $\forall P r s x y,$

Forall2 $P r s \rightarrow$

$P x y \rightarrow$

Forall2 $P (r & x) (s & y).$

Proof using. *intros. apply¬ Forall2_app. Qed.*

Lemma Forall2_nil_eq : $\forall P,$

Forall2 $P \text{ nil nil} = \text{True}.$

Proof using. *intros. extens. iff M. { inverts× M. } { auto. } Qed.*

Lemma Forall2_cons_eq : $\forall P x y r s,$

Forall2 $P (x :: r) (y :: s) = (P x y \wedge \text{Forall2 } P r s).$

Proof using. *intros. extens. iff M (M1&M2). { inverts× M. } { auto. } Qed.*

Lemma Forall2_one_eq : $\forall P x y,$

Forall2 $P (x :: \text{nil}) (y :: \text{nil}) = P x y.$

Proof using. *intros. extens. rewrite Forall2_cons_eq. rewrite× Forall2_nil_eq. Qed.*

Lemma Forall2_app_eq : $\forall r1 r2 s1 s2 P,$

$\text{length } r1 = \text{length } s1 \rightarrow$

Forall2 $P (r1 ++ r2) (s1 ++ s2) = (\text{Forall2 } P r1 s1 \wedge \text{Forall2 } P r2 s2).$

Proof using.

intros r1. induction r1; introv E;

*destruct s1; tryfalse; rew_list in *; extens.*

{ autos×. }

{ iff M (M1&M2).

{ inverts M as N1 N2. rewrite× IHr1 in N2. }

{ inverts M1. constructors¬. rewrite× IHr1. } }

Qed.

Lemma Forall2_last_eq : $\forall P r s x y,$

Forall2 $P (r & x) (s & y) = (\text{Forall2 } P r s \wedge P x y).$

Proof using.

intros. extens. iff M (M1&M2).

$\{ \text{lets } E : \text{Forall2_inv_length } M. \text{rew_list} \text{ in } E.$
 $\quad \text{rewrite} \neg \text{Forall2_app_eq} \text{ in } M. \text{destruct } M \text{ as } (M1 \& M2). \text{inverts} \times M2. \}$
 $\quad \{ \text{applys} \neg \text{Forall2_last}. \}$
 Qed.

Lemma Forall2_cons_inv : $\forall P r s x y,$
 $\text{Forall2 } P (x :: r) (y :: s) \rightarrow$
 $P x y \wedge \text{Forall2 } P r s.$

Proof using. introv E. rewrite \neg Forall2_cons_eq in E. Qed.

Lemma Forall2_cons_l_inv : $\forall P r1 s x,$
 $\text{Forall2 } P (x :: r1) s \rightarrow$
 $\exists y s1, s = y :: s1 \wedge P x y \wedge \text{Forall2 } P r1 s1.$

Proof using. introv E. destruct s as [|y s1]; inverts E. autos \times . Qed.

Lemma Forall2_cons_r_inv : $\forall P r s1 y,$
 $\text{Forall2 } P r (y :: s1) \rightarrow$
 $\exists x r1, r = x :: r1 \wedge P x y \wedge \text{Forall2 } P r1 s1.$

Proof using. introv E. destruct r as [|x r1]; inverts E. autos \times . Qed.

Lemma Forall2_app_inv : $\forall P r1 s1 r2 s2,$
 $\text{Forall2 } P (r1 ++ r2) (s1 ++ s2) \rightarrow$
 $\text{length } r1 = \text{length } s1 \rightarrow$
 $\text{Forall2 } P r1 s1 \wedge \text{Forall2 } P r2 s2.$

Proof using. introv M E. rewrite \neg Forall2_app_eq in M. Qed.

Lemma Forall2_last_inv : $\forall P r s x y,$
 $\text{Forall2 } P (r \& x) (s \& y) \rightarrow$
 $\text{Forall2 } P r s \wedge P x y.$

Proof using. introv E. rewrite \neg Forall2_last_eq in E. Qed.

Lemma Forall2_last_l_inv : $\forall P r1 s x,$
 $\text{Forall2 } P (r1 \& x) s \rightarrow$
 $\exists s1 y, s = s1 \& y \wedge P x y \wedge \text{Forall2 } P r1 s1.$

Proof using.

$\text{introv } M. \text{forwards } (y \& s1 \& E); \text{list_neq_nil_inv_last } s.$
 $\{ \text{intro_subst. inverts } M. \text{applys} \times \text{nil_eq_last_inv}. \}$
 $\text{subst } s. \text{rewrite} \times \text{Forall2_last_eq} \text{ in } M.$

Qed.

Lemma Forall2_last_r_inv : $\forall P r s1 y,$
 $\text{Forall2 } P r (s1 \& y) \rightarrow$
 $\exists r1 x, r = r1 \& x \wedge P x y \wedge \text{Forall2 } P r1 s1.$

Proof using.

$\text{introv } M. \text{forwards } (x \& r1 \& E); \text{list_neq_nil_inv_last } r.$
 $\{ \text{intro_subst. inverts } M. \text{applys} \times \text{nil_eq_last_inv}. \}$
 $\text{subst } r. \text{rewrite} \times \text{Forall2_last_eq} \text{ in } M.$

Qed.

Lemma Forall2_swap : $\forall P r s,$

Forall2 $P r s \rightarrow$

Forall2 ($\text{fun } b a \Rightarrow P a b$) $s r.$

Proof using. *intros* F . *induction* $\neg F$. Qed.

Lemma Forall2_swap_inv : $\forall P r s,$

Forall2 ($\text{fun } b a \Rightarrow P a b$) $s r \rightarrow$

Forall2 $P r s.$

Proof using. *intros* F . *induction* $\neg F$. Qed.

Lemma Forall2_swap_eq : $\forall P r s,$

Forall2 $P r s = \text{Forall2}$ ($\text{fun } b a \Rightarrow P a b$) $s r.$

Proof using.

intros. *extens*. *iff* $M.$

{ *applys* \times Forall2_swap. }

{ *applys* \times Forall2_swap_inv. }

Qed.

Lemma Forall2_rel_le : $\forall P Q r s,$

Forall2 $P r s \rightarrow$

$\text{rel_incl } P Q \rightarrow$

Forall2 $Q r s.$

Proof using.

intros $F W$. *unfolds* rel_incl , pred_incl . *induction* F ; *constructors* \neg .

Qed.

Lemma Forall2_rev : $\forall P r s,$

Forall2 $P r s \rightarrow$

Forall2 $P (\text{rev } r) (\text{rev } s).$

Proof using.

Hint Resolve Forall2_last.

intros $P r$. *induction* r ; *intros* M ; *inverts* M ; *rew_listx* \neg .

Qed.

Lemma Forall2_rev_eq : $\forall P r s,$

Forall2 $P r s = \text{Forall2}$ $P (\text{rev } r) (\text{rev } s).$

Proof using.

intros. *extens*. *iff* $M.$

{ *applys* \neg Forall2_rev. }

{ *rewrite* $\leftarrow (\text{rev_rev } r)$. *rewrite* $\leftarrow (\text{rev_rev } s)$. *applys* \neg Forall2_rev. }

Qed.

Lemma Forall2_take : $\forall P n r s,$

Forall2 $P r s \rightarrow$

Forall2 $P (\text{take } n r) (\text{take } n s).$

Proof using. intros $P n$. induction n ; introv H ; inverts H ; rew_listx \neg . Qed.

Lemma Forall2_drop : $\forall P n r s$,

Forall2 $P r s \rightarrow$

Forall2 $P (\text{drop } n r) (\text{drop } n s)$.

Proof using. intros $P n$. induction n ; introv H ; inverts H ; rew_listx \neg . Qed.

Lemma Forall2_map_l : $\forall f P l$,

$(\forall y, P (f y) y) \rightarrow$

Forall2 $P (\text{map } f l) l$.

Proof using. introv I . induction l ; rew_listx \neg . Qed.

Lemma Forall2_map_r : $\forall f P l$,

$(\forall x, P x (f x)) \rightarrow$

Forall2 $P l (\text{map } f l)$.

Proof using. introv I . induction l ; rew_listx \neg . Qed.

Lemma Forall2_Nth_inv : $\forall P r s n x y$,

Forall2 $P r s \rightarrow$

Nth $n r x \rightarrow$

Nth $n s y \rightarrow$

$P x y$.

Proof using.

introv $F N1 N2$. gen n .

induction \neg F ; introv $N1 N2$; inverts $N1$; inverts \times $N2$.

Qed.

Lemma Forall2_nth_inv : $\forall \{IA:\mathbf{Inhab} A\} \{IB:\mathbf{Inhab} B\} P r s n$,

Forall2 $P r s \rightarrow$

$n < \text{length } r \rightarrow$

$P (\text{nth } n r) (\text{nth } n s)$.

Proof using.

introv $F N$. forwards: Forall2_inv_length F .

applys Forall2_Nth_inv $n F$; applys Nth_nth; math.

Qed.

Lemma Forall2_of_forall_nth : $\forall \{IA:\mathbf{Inhab} A\} \{IB:\mathbf{Inhab} B\} P r s$,

$(\forall n, n < \text{length } r \rightarrow P (\text{nth } n r) (\text{nth } n s)) \rightarrow$

$\text{length } r = \text{length } s \rightarrow$

Forall2 $P r s$.

Proof using.

introv $H E$. gen s . induction r ; intros; destruct s ; tryfalse.

{ auto. }

{ rew_list in E . constructors.

{ applys $H 0$. rew_list. math. }

{ applys \neg IHr . introv N . applys $H (\mathbf{S} n)$. rew_list. math. } }

Qed.

End Forall2.

Forall3 is similar to Forall2 except that it relates three lists.

Inductive **Forall3** $A\ B\ C$ ($P : A \rightarrow B \rightarrow C \rightarrow \text{Prop}$)

: **list** $A \rightarrow \text{list}\ B \rightarrow \text{list}\ C \rightarrow \text{Prop} :=$

| Forall3_nil :

Forall3 $P\ \text{nil}\ \text{nil}\ \text{nil}$

| Forall3_cons : $\forall l_1\ l_2\ l_3\ x_1\ x_2\ x_3,$

$P\ x_1\ x_2\ x_3 \rightarrow$

Forall3 $P\ l_1\ l_2\ l_3 \rightarrow$

Forall3 $P\ (x_1 :: l_1)\ (x_2 :: l_2)\ (x_3 :: l_3).$

Exists $P\ l$ asserts that there exists a value in the list l that satisfied the predicate P .

Inductive **Exists** A ($P : A \rightarrow \text{Prop}$) : **list** $A \rightarrow \text{Prop} :=$

| Exists_head : $\forall l\ x,$

$P\ x \rightarrow$

Exists $P\ (x :: l)$

| Exists_tail : $\forall l\ x,$

Exists $P\ l \rightarrow$

Exists $P\ (x :: l).$

Section Exists.

Variables $A : \text{Type}.$

Implicit Types $l : \text{list}\ A.$

Implicit Types $P : A \rightarrow \text{Prop}.$

Hint Constructors **Exists**.

Lemma **Exists_nil_eq** : $\forall P,$

Exists $P\ \text{nil} = \text{False}.$

Proof using. intros. extens. iff M. { invert \times M. } { false. } Qed.

Lemma **Exists_cons_eq** : $\forall P\ l\ x,$

Exists $P\ (x :: l) = (P\ x \vee \text{Exists}\ P\ l).$

Proof using. intros. extens. iff M. { inverts \times M. } { destruct \times M. } Qed.

Lemma **Exists_one_eq** : $\forall P\ x,$

Exists $P\ (x :: \text{nil}) = P\ x.$

Proof using. intros. extens. rewrite **Exists_cons_eq**. rewrite \times **Exists_nil_eq**. Qed.

Lemma **Exists_app_eq** : $\forall P\ l_1\ l_2,$

Exists $P\ (l_1 ++ l_2) = (\text{Exists}\ P\ l_1 \vee \text{Exists}\ P\ l_2).$

Proof using.

 intros. extens. induction l1; rew_list.

 { iff M [M|M]. { autos \times . } { inverts \neg M. } { autos \times . } }

 { iff M [M|M].

 { inverts \times M. }

$\{ \text{inverts } M. \{ \text{auto.} \} \{ \text{applys} \times \text{Exists_tail.} \} \}$
 $\{ \text{applys} \times \text{Exists_tail.} \} \}$

Qed.

Lemma `Exists_last_eq` : $\forall P l x,$
 $\text{Exists } P (l \& x) = (P x \vee \text{Exists } P l).$

Proof using.

`intros. extens. rewrite Exists_app_eq. rewrite Exists_cons_eq.`
`rewrite \times Exists_nil_eq.`

Qed.

Lemma `Exists_app_l` : $\forall P l1 l2,$
 $\text{Exists } P l1 \rightarrow$
 $\text{Exists } P (l1 ++ l2).$

Proof using. `intros. rewrite \times Exists_app_eq.` Qed.

Lemma `Exists_app_r` : $\forall P l1 l2,$
 $\text{Exists } P l2 \rightarrow$
 $\text{Exists } P (l1 ++ l2).$

Proof using. `intros. rewrite \times Exists_app_eq.` Qed.

Lemma `Exists_last` : $\forall P l x,$
 $\text{Exists } P l \rightarrow$
 $P x \rightarrow$
 $\text{Exists } P (l \& x).$

Proof using. `intros. rewrite \times Exists_last_eq.` Qed.

Lemma `Exists_nil_inv` : $\forall P,$
 $\text{Exists } P \text{ nil} \rightarrow$
 $\text{False}.$

Proof using. `introv H. rewrite \times Exists_nil_eq in H.` Qed.

Lemma `Exists_cons_inv` : $\forall P l x,$
 $\text{Exists } P (x :: l) \rightarrow$
 $P x \vee \text{Exists } P l.$

Proof using. `introv H. rewrite \times Exists_cons_eq in H.` Qed.

Lemma `Exists_one_inv` : $\forall P x,$
 $\text{Exists } P (x :: \text{nil}) \rightarrow$
 $P x.$

Proof using. `introv H. rewrite \times Exists_one_eq in H.` Qed.

Lemma `Exists_app_inv` : $\forall P l1 l2,$
 $\text{Exists } P (l1 ++ l2) \rightarrow$
 $\text{Exists } P l1 \vee \text{Exists } P l2.$

Proof using. `introv H. rewrite \times Exists_app_eq in H.` Qed.

Lemma `Exists_last_inv` : $\forall P l x,$

Exists $P(l \& x) \rightarrow$

$P x \vee \text{Exists } P l.$

Proof using. *introv H. rewrite \times Exists_last_eq in H. Qed.*

Lemma Exists_eq_exists_mem : $\forall P l,$

Exists $P l = (\exists x, \text{mem } x l \wedge P x).$

Proof using.

Hint Constructors **mem**.

introv. extens. induction l as [|y l'].

{ iff $M(x \& M \& H)$. { inverts M . } { inverts M . } }

{ iff $M(x \& M \& H)$.

{ inverts M as N .

{ $\exists x. y$. }

{ forwards $(x \& M \& H)$: (**proj1** IHl') $N. \exists x. x$. }

{ destruct (mem_cons_inv M) as [N|N].

{ subst x . }

{ applys \times Exists_tail. }

Qed.

Lemma mem_Exists : $\forall P l x,$

mem $x l \rightarrow$

$P x \rightarrow$

Exists $P l.$

Proof using. *introv M H. rewrite \times Exists_eq_exists_mem. Qed.*

Lemma mem_Exists_eq : $\forall P l x,$

mem $x l = \text{Exists } (= x) l.$

Proof using.

intros. extens. rewrite \times Exists_eq_exists_mem. iff $M(y \& ? \& ?)$; subst x .

Qed.

Lemma Nth_Exists : $\forall P n l x,$

Nth $n l x \rightarrow$

$P x \rightarrow$

Exists $P l.$

Proof using. *introv M H. applys \times mem_Exists. applys \times Nth_mem. Qed.*

Lemma nth_Exists : $\forall \{IA:\text{Inhab } A\} P n l,$

$P(\text{nth } n l) \rightarrow$

$n < \text{length } l \rightarrow$

Exists $P l.$

Proof using. *introv H N. applys \times Nth_Exists n. applys \neg Nth_nth. Qed.*

Lemma Exists_rev : $\forall P l,$

Exists $P l \rightarrow$

Exists $P(\text{rev } l).$

Proof using.

```

introv E. induction l.
{ auto. }
{ rew_list. rewrite Exists_last_eq. inverts× E. }
Qed.

```

Lemma $\text{Exists_rev_eq} : \forall P l,$
Exists $P (\text{rev } l) \equiv \text{Exists } P l.$

Proof using.

```

intros. extens. iff M.
{ rewrite ← rev_rev. applys¬ Exists_rev. }
{ applys¬ Exists_rev. }

```

Qed.

Lemma $\text{Exists_pred_incl} : \forall P Q l,$
Exists $P l \rightarrow$
 $\text{pred_incl } P Q \rightarrow$
Exists $Q l.$

Proof using. *introv. induction l; introv H L; inverts× H.* Qed.

Lemma $\text{Exists_filter_pred_incl} : \forall P Q l,$
Exists $P l \rightarrow$
 $\text{pred_incl } P Q \rightarrow$
Exists $P (\text{filter } Q l).$

Proof using.

```

introv M N. induction M; rew_listx.
{ forwards*: N. case_if×. }
{ case_if¬. }

```

Qed.

Lemma $\text{Exists_filter_same} : \forall P l,$
Exists $P l \rightarrow$
Exists $P (\text{filter } P l).$

Proof using. *introv M. applys× Exists_filter_pred_incl. applys pred_incl_refl.* Qed.

Lemma $\text{Exists_take_inv} : \forall P n l,$
Exists $P (\text{take } n l) \rightarrow$
Exists $P l.$

Proof using.

```

introv E. forwards (q&K): take_is_prefix n l.
rewrite K. rewrite× Exists_app_eq.

```

Qed.

Lemma $\text{Exists_drop_inv} : \forall P n l,$
Exists $P (\text{drop } n l) \rightarrow$
Exists $P l.$

Proof using.

```

introv E. forwards (q&K): drop_is_suffix n l.

```

```
rewrite K. rewrite× Exists_app_eq.
```

Qed.

Lemma Exists_inv_middle_first : $\forall P l,$

Exists $P l \rightarrow$

$\exists l1 x l2,$

$l = l1 ++ x :: l2$

$\wedge \text{Forall } (\text{fun } x \Rightarrow \neg P x) l1$

$\wedge P x.$

Proof using.

introv E. induction E.

$\exists (@\text{nil} A) x l. \text{splits} \sim. \text{constructors} \sim.$

lets (l1&x'&l2&E1&F&HP): (rm IHE). tests Px: (P x).

$\exists (@\text{nil} A) x l. \text{splits} \sim. \text{constructors} \sim.$

substs. $\exists (x :: l1) x' l2. \text{splits} \sim. \text{constructors} \sim.$

Qed.

End Exists.

Hint Rewrite Exists_nil_eq Exists_cons_eq Exists_one_eq Exists_app_eq Exists_last_eq : *rew_listx.*

Exists2 $P L1 L2$ asserts that there exists an index n such that the n -th element of $L1$ and the n -th element of $L2$ are related by the binary relation P .

Inductive **Exists2** $A1 A2 (P : A1 \rightarrow A2 \rightarrow \text{Prop})$

$: \text{list } A1 \rightarrow \text{list } A2 \rightarrow \text{Prop} :=$

$| \text{Exists2_here} : \forall l1 l2 x1 x2,$

$P x1 x2 \rightarrow$

Exists2 $P (x1 :: l1) (x2 :: l2)$

$| \text{Exists2_next} : \forall l1 l2 x1 x2,$

Exists2 $P l1 l2 \rightarrow$

Exists2 $P (x1 :: l1) (x2 :: l2).$

count $P l$ returns the number of elements of l satisfying the predicate P .

Definition count $A (P : A \rightarrow \text{Prop}) (l : \text{list } A) : \text{nat} :=$

$\text{length} (\text{filter } P l).$

Section Count.

Variables $A : \text{Type}.$

Implicit Types $l : \text{list } A.$

Implicit Types $P : A \rightarrow \text{Prop}.$

Lemma count_eq_length_filter : $\forall P l,$

$\text{count } P l = \text{length} (\text{filter } P l).$

Proof using. auto. Qed.

Lemma count_nil : $\forall P,$

```

count P nil = 0.
Proof using. auto. Qed.

Lemma count_one : ∀ P x,
  count P (x::nil) = If P x then 1 else 0.
Proof using. intros. unfold count. rew_listx. case_if¬. Qed.

Lemma count_app : ∀ P l1 l2,
  count P (l1++l2) = count P l1 + count P l2.
Proof using. intros. unfold count. rew_listx¬. Qed.

Lemma count_cons : ∀ P l x,
  count P (x::l) = count P l + If P x then 1 else 0.
Proof using.
  intros. rewrite ← app_cons_one_r, count_app, count_one. math.
Qed.

Lemma count_last : ∀ P l x,
  count P (l&x) = count P l + If P x then 1 else 0.
Proof using. intros. rewrite count_app, count_one. math. Qed.

Lemma count_rev : ∀ P l,
  count P (rev l) = count P l.
Proof using. intros. unfold count. rew_listx¬. Qed.

End Count.

Hint Rewrite count_nil count_cons count_one count_app count_rev : rew_listx.

Section Count2.

Variables A : Type.
Implicit Types l : list A.
Implicit Types P : A → Prop.

Lemma count_le_length : ∀ P l,
  count P l ≤ length l.
Proof using. intros. apply length_filter. Qed.

Lemma Forall_eq_count_eq_length : ∀ P l,
  Forall P l = (count P l = length l).
Proof using.
  intros. induction l; rew_listx.
  { extens ×. }
  { rewrite IHl. extens. iff (M1&M2) M; case_if.
    { math. }
    { split¬. math. }
    { false. forwards~: count_le_length P l. math. } }
Qed.

Lemma count_Forall : ∀ P l,

```

Forall $P l \rightarrow$

count $P l = \text{length } l$.

Proof using. *introv. rewrite \neg Forall_eq_count_eq_length.* Qed.

Lemma Forall_of_count_eq_length : $\forall P l,$

count $P l = \text{length } l \rightarrow$

Forall $P l.$

Proof using. *introv. rewrite \neg Forall_eq_count_eq_length.* Qed.

Lemma Forall_not_eq_count_eq_zero : $\forall P l,$

Forall (fun $x \Rightarrow \neg P x) l = (\text{count } P l = 0).$

Proof using.

intros. induction l; rew_listx.

{ extens \times . }

{ rewrite IHl. extens. iff (M1&M2) M; case_if.

{ math. }

{ false. math. }

{ split \neg . math. } }

Qed.

Lemma Forall_not_of_count_eq_zero : $\forall P l,$

count $P l = 0 \rightarrow$

Forall (fun $x \Rightarrow \neg P x) l.$

Proof using. *introv. rewrite \neg Forall_not_eq_count_eq_zero.* Qed.

Lemma count_eq_zero_of_Forall_not : $\forall P l,$

Forall (fun $x \Rightarrow \neg P x) l \rightarrow$

count $P l = 0.$

Proof using. *introv. rewrite \neg Forall_not_eq_count_eq_zero.* Qed.

Lemma Exists_eq_count_pos : $\forall P l,$

Exists $P l = (\text{count } P l > 0).$

Proof using.

intros. induction l; rew_listx.

{ extens. iff M; false; math. }

{ rewrite IHl. extens. iff [M|M] M; case_if; try math.

{ left \neg . }

{ right. math. } }

Qed.

Lemma Exists_of_count_pos : $\forall P l,$

count $P l > 0 \rightarrow$

Exists $P l.$

Proof using. *introv. rewrite \neg Exists_eq_count_pos.* Qed.

Lemma count_pos_of_Exists : $\forall P l,$

Exists $P l \rightarrow$

```
count P l > 0.
Proof using. introv. rewrite¬ Exists_eq_count_pos. Qed.
```

```
Lemma count_pos_eq_exists_mem : ∀ P l,
  (count P l > 0) = (exists x, mem x l ∧ P x).
```

```
Proof using.
  intros. rewrite ← Exists_eq_count_pos, Exists_eq_exists_mem. auto.
Qed.
```

```
Lemma count_pos_of_mem : ∀ x P l,
  mem x l →
  P x →
  count P l > 0.
```

```
Proof using. introv. rewrite× count_pos_eq_exists_mem. Qed.
```

```
Lemma exists_mem_of_count_pos : ∀ P l,
  count P l > 0 →
  exists x, mem x l ∧ P x.
```

```
Proof using. introv. rewrite× count_pos_eq_exists_mem. Qed.
```

```
End Count2.
```

```
Global Opaque count.
```

19.7 Nat seq

nat_seq i n generates a list of variables x1;x2;..;xn with x1=i and xn=i+n-1. Such lists are useful for generic programming.

```
Fixpoint nat_seq (start:nat) (nb:nat) :=
  match nb with
  | 0 ⇒ nil
  | S nb' ⇒ start :: nat_seq (S start) nb'
  end.
```

```
Lemma length_nat_seq : ∀ start nb,
  length (nat_seq start nb) = nb.
```

```
Proof using.
  intros. gen start. induction nb; simpl; intros.
  { auto. } { rew_list. rewrite¬ IHnb. }
Qed.
```

19.8 Fold

WARNING: EXPERIMENTAL SECTION

```
Definition fold A B (m:monoid_op B) (f:A→B) (L:list A) : B :=
```

```

fold_right (fun x acc => monoid_oper m (f x) acc) (monoid_neutral m) L.

Section Fold.

Variables (A B:Type).

Implicit Types m : monoid_op B.
Implicit Types l : list A.
Implicit Types f g : A → B.

Lemma fold_nil : ∀ m f,
  fold m f nil = monoid_neutral m.

Proof using. auto. Qed.

Lemma fold_cons : ∀ m f x l,
  fold m f (x::l) = monoid_oper m (f x) (fold m f l).

Proof using. auto. Qed.

Lemma fold_one : ∀ m f x,
  Monoid m →
  fold m f (x::nil) = f x.

Proof using.
  intros. rewrite fold_cons, fold_nil. rewrite¬ monoid_neutral_r.
Qed.

Lemma fold_app : ∀ m f l1 l2,
  Monoid m →
  fold m f (l1 ++ l2) = monoid_oper m (fold m f l1) (fold m f l2).

Proof using.
  unfold fold. intros. rewrite fold_right_app. gen l2.
  induction l1; intros.
  repeat rewrite fold_right_nil. rewrite¬ monoid_neutral_l.
  repeat rewrite fold_right_cons. rewrite ← monoid_assoc. f_equal.
Qed.

Lemma fold_last : ∀ m f x l,
  Monoid m →
  fold m f (l & x) = monoid_oper m (fold m f l) (f x).

Proof using.
  intros. rewrite¬ fold_app. rewrite fold_cons.
  rewrite fold_nil. rewrite monoid_neutral_r. auto.
Qed.

Lemma fold_congruence : ∀ m f g l,
  (∀ x, mem x l → f x = g x) →
  fold m f l = fold m g l.

Proof using.
  Hint Constructors mem.
  intros. unfold fold.
  induction l as [| x l' ]; intros; simpl.

```

```
{ eauto. }
{ rew_listx. fequals. applys¬ H. eauto. }
```

Qed.

Lemma fold_equiv_step : $\forall m f l a,$
Comm_monoid $m \rightarrow$
noduplicates $l \rightarrow$
mem $a l \rightarrow$
 $\exists l',$
 $\text{fold } m f l = \text{fold } m f (a :: l')$
 $\wedge (\forall x, \text{mem } x l \leftrightarrow \text{mem } x (a :: l'))$
 $\wedge \text{noduplicates } (a :: l').$

Proof using. *introv Hm. induction l as [| b t]; introv DL La. inverts La.*

tests: $(a = b).$

$\exists t. \text{splits} \times.$

inverts La. false. inverts DL as DLb DT. forwards¬ (L' & EL' & EQ & DL'): IHt.

$\exists (b :: L'). \text{splits}.$

```
do 3 rewrite fold_cons. rewrite EL'.
rewrite fold_cons. do 2 rewrite monoid_assoc.
rewrite¬ (monoid_comm (f b)).
intros x. specializes EQ x. rewrite mem_cons_eq in EQ.
do 3 rewrite mem_cons_eq. autos×.
inverts DL'. constructors.
```

introv N. inverts N. false. false.

constructors¬. introv N. applys DLb. rewrite EQ. constructors¬.

Qed.

Lemma fold_equiv : $\forall m f l1 l2,$
Comm_monoid $m \rightarrow$
noduplicates $l1 \rightarrow$
noduplicates $l2 \rightarrow$
 $(\forall x, \text{mem } x l1 \leftrightarrow \text{mem } x l2) \rightarrow$
 $\text{fold } m f l1 = \text{fold } m f l2.$

Proof using. *intros m f l1. induction l1; introv HM D1 D2 EQ.*

cuts_rewrite (l2 = nil). rewrite¬ fold_nil.

destruct l2; auto. forwards¬ M: (proj2 (EQ a)). inverts M.

inverts D1. asserts L2a: (mem a l2). rewrite¬ ← EQ.

forwards× (l2' & V2' & EQ' & D2'): fold_equiv_step f L2a.

rewrite V2'. do 2 rewrite fold_cons.

inverts D2'.

rewrite¬ (IHl1 l2'). intros.

tests: $(x = a).$

iff; auto_false ×.

asserts_rewrite (mem x l1 = mem x (a :: l1)).

```

extens. iff  $\neg M$ . inverts  $\neg M$ . false.
asserts_rewrite (mem x l2' = mem x (a :: l2')).  

extens. iff  $\neg M$ . inverts  $\neg M$ . false.
rewrite EQ. rewrite $\times$  EQ'.

```

Qed.

End Fold.

Lemma fold_pointwise : $\forall B m (leB : B \rightarrow B \rightarrow \text{Prop})$,

```

Monoid m →
refl leB →
Proper (leB  $\leftrightarrow$  leB  $\leftrightarrow$  leB) (monoid_oper m) →
 $\forall A (l : \text{list } A)$ ,
 $\forall (f f' : A \rightarrow B)$ ,
( $\forall x, \text{mem } x l \rightarrow leB (f x) (f' x)$ ) →
leB (fold m f l) (fold m f' l).

```

Proof using. Hint Constructors **mem**.

```

intros HM HR HP. induction l; introv HL.
do 2 rewrite fold_nil. applys HR.
do 2 rewrite fold_cons. apply HP. applys $\neg$  HL. applys $\neg$  IHl.

```

Qed.

Global Opaque fold.

Hint Rewrite fold_nil fold_cons fold_app : rew_listx.

Section ListSub.

Variable (A:Type).

Sub-list well-founded order

```

Inductive list_sub : list A → list A → Prop :=
| list_sub_cons :  $\forall x l,$ 
  list_sub l (x :: l)
| list_sub_tail :  $\forall x l1 l2,$ 
  list_sub l1 l2 →
  list_sub l1 (x :: l2).

```

Hint Constructors **list_sub**.

Lemma list_sub_wf : LibWf.wf **list_sub**.

Proof using.

```

intros l. induction l; apply Acc_intro; introv H.
{ inverts $\neg$  H. }
{ inverts $\neg$  H. applys $\neg$  IHl. }

```

Qed.

End ListSub.

Arguments **list_sub** [A].

Hint Constructors **list_sub**.

Hint Resolve *list_sub_wf* : wf.

Induction on all but last item

Lemma *list_ind_last* : $\forall (A : \text{Type}) (P : \text{list } A \rightarrow \text{Prop}),$

$P \text{ nil} \rightarrow$
 $(\forall (a : A) (l : \text{list } A), P l \rightarrow P (l & a)) \rightarrow$
 $\forall l : \text{list } A, P l.$

Proof using.

introv H1 H2. intros. induction_wf IH: (wf_measure (@length A)) l.

lets [E|(x&l'&E)]: (last_case l); subst. auto.

unfolds measure. rewrite length_last in IH. auto with maths.

Qed.

Lemma *list2_ind* : $\forall A B (P : \text{list } A \rightarrow \text{list } B \rightarrow \text{Prop}) l1 l2,$

$\text{length } l1 = \text{length } l2 \rightarrow$

$P \text{ nil nil} \rightarrow$

$(\forall x1 xs1 x2 xs2,$

$\text{length } xs1 = \text{length } xs2 \rightarrow P xs1 xs2 \rightarrow P (x1 :: xs1) (x2 :: xs2)) \rightarrow$

$P l1 l2.$

Proof using.

introv E M1 M2. gen l2. induction l1 as [| x1 l1]; intros;

destruct l2 as [| x2 l2]; try solve [false; math]; auto.

Qed.

Tactic Notation "list2_ind" constr(*l1*) constr(*l2*) :=

pattern *l2*; pattern *l1*;

match goal with $\vdash (\text{fun } a \Rightarrow (\text{fun } b \Rightarrow @?P a b) _) _ \Rightarrow$

$\text{let } X := \text{fresh "P" in set } (X := P); \text{ applys list2_ind } X; \text{ unfold } X; \text{ try clear } X$
end.

Tactic Notation "list2_ind" " \sim " constr(*l1*) constr(*l2*) :=

list2_ind l1 l2; auto_tilde.

Tactic Notation "list2_ind" "*" constr(*l1*) constr(*l2*) :=

list2_ind l1 l2; auto_star.

Tactic Notation "list2_ind" constr(*E*) :=

match type of *E* with $\text{length } ?l1 = \text{length } ?l2 \Rightarrow$

list2_ind l1 l2; [apply E ||] end.

Lemma *list2_ind_last* : $\forall A B (P : \text{list } A \rightarrow \text{list } B \rightarrow \text{Prop}) l1 l2,$

$\text{length } l1 = \text{length } l2 \rightarrow$

$P \text{ nil nil} \rightarrow$

$(\forall x1 xs1 x2 xs2,$

$\text{length } xs1 = \text{length } xs2 \rightarrow P xs1 xs2 \rightarrow P (xs1 & x1) (xs2 & x2)) \rightarrow$

$P \ l1 \ l2.$

Proof using.

```
introv E M1 M2. gen l2. induction l1 using list_ind_last;
  [| rename a into x1, l1 into l1'|]; intros;
  destruct (last_case l2) as [|(x2&l2'&E2)]; subst; rew_list in *;
  try solve [false; math]; auto.
```

Qed.

```
Tactic Notation "list2_ind_last" constr(l1) constr(l2) :=
  pattern l2; pattern l1;
  match goal with ⊢ (fun a ⇒ (fun b ⇒ @?P a b) _) _ ⇒
```

```
  let X := fresh "P" in set (X := P); applys list2_ind_last X; unfold X; try clear X
end.
```

```
Tactic Notation "list2_ind_last" "˜" constr(l1) constr(l2) :=
  list2_ind_last l1 l2; auto_tilde.
```

```
Tactic Notation "list2_ind_last" "*" constr(l1) constr(l2) :=
  list2_ind_last l1 l2; auto_star.
```

```
Tactic Notation "list2_ind_last" constr(E) :=
  match type of E with length ?l1 = length ?l2 ⇒
    list2_ind_last l1 l2; [ apply E || ] end.
```

Chapter 20

Library SLF.LibListExec

```
Set Implicit Arguments.  
Generalizable Variables A B.  
From SLF Require Import LibTactics LibReflect.  
From SLF Require Export LibList.
```

```
Module REWLISTEXEC.
```

```
Tactic Notation "rew_list_exec" :=  
  autorewrite with rew_list_exec.  
Tactic Notation "rew_list_exec" "in" "*" :=  
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_list_exec).  
Tactic Notation "rew_list_exec" "in" hyp(H) :=  
  autorewrite with rew_list_exec in H.
```

```
End REWLISTEXEC.
```

```
Definition is_nil A (l:list A) : bool :=  
  match l with  
  | nil => true  
  | _ => false  
  end.
```

```
Lemma is_nil_eq : ∀ A (l:list A),  
  is_nil l = isTrue (l = nil).
```

```
Proof using. intros. destruct l; simpl; rew_bool_eq; auto_false. Qed.
```

```
Definition is_not_nil A (l:list A) : bool :=  
  match l with  
  | nil => false  
  | _ => true  
  end.
```

```
Lemma is_not_nil_eq : ∀ A (l:list A),  
  is_not_nil l = isTrue (l ≠ nil).
```

Proof.

```
intros. destruct l; simpl; rew_bool_eq; auto_false.  
Qed.
```

Hint Rewrite is_nil_eq is_not_nil_eq : rew_list_exec.

Definition length : $\forall A, \text{list } A \rightarrow \text{nat} :=$
`List.length.`

Lemma length_eq :

```
length = LibList.length.
```

Proof using. extens ;=> A l. induction l; simpl; rew_list; auto. Qed.

Hint Rewrite length_eq : rew_list_exec.

Definition app : $\forall A, \text{list } A \rightarrow \text{list } A \rightarrow \text{list } A :=$
`List.app.`

Lemma app_eq :

```
app = LibList.app.
```

Proof using.

```
extens ;=> A L1 L2. induction L1; simpl; rew_list; congruence.
```

Qed.

Hint Rewrite app_eq : rew_list_exec.

Definition rev : $\forall A, \text{list } A \rightarrow \text{list } A :=$
`List.rev.`

Lemma rev_eq : $\forall A,$

```
@List.rev A = @LibList.rev A.
```

Proof using.

```
extens ;=> L. induction L; simpl; rew_list. { auto. }  
{ rewrite IHL. rewrite ← app_eq. unfold app. fequals. }
```

Qed.

Hint Rewrite rev_eq : rew_list_exec.

Definition fold_right : $\forall A B, (B \rightarrow A \rightarrow A) \rightarrow A \rightarrow \text{list } B \rightarrow A :=$
`List.fold_right.`

Lemma fold_right_eq : $\forall A B (f:B \rightarrow A \rightarrow A) (a:A) (l:\text{list } B),$
 $\text{fold_right } f a l = \text{LibList.fold_right } f a l.$

Proof using. intros. induction l; simpl; rew_listx; fequals. Qed.

Hint Rewrite fold_right_eq : rew_list_exec.

Definition map : $\forall A B, (A \rightarrow B) \rightarrow \text{list } A \rightarrow \text{list } B :=$
`List.map.`

Lemma map_eq :

```
map = LibList.map.
```

Proof using.

```
extens ;=> A B f L. induction L; simpl; rew_listx; congruence.  
Qed.
```

Hint Rewrite map_eq : rew_list_exec.

```
Definition combine : ∀ A B, list A → list B → list (A × B) :=  
List.combine.
```

```
Lemma combine_eq : ∀ A B (L1:list A) (L2:list B),  
LibList.length L1 = LibList.length L2 →  
combine L1 L2 = LibList.combine L1 L2.
```

Proof using. introv E. gen L2.

```
induction L1 as [|x1 L1']; intros; destruct L2 as [|x2 L2']; tryfalse.  
{ auto. }  
{ rew_list in E. rew_listx. simpl. fequals¬. }
```

Qed.

Hint Rewrite combine_eq : rew_list_exec.

```
Fixpoint mem A (cmp:A→A→bool) (x:A) (l:list A) : bool :=  
match l with  
| nil ⇒ false  
| y::l' ⇒ cmp x y || mem cmp x l'  
end.
```

```
Lemma mem_exec_eq : ∀ A (cmp:A→A→bool) x l,  
is_beq cmp →  
mem cmp x l = isTrue (LibList.mem x l).
```

Proof using.

```
introv M. induction l as [|y l']; simpl; rew_listx; rew_isTrue; fequals.  
Qed.
```

Chapter 21

Library `SLF.LibListZ`

```
Set Implicit Arguments.  
Generalizable Variables A B.  
From SLF Require Import LibTactics LibLogic LibOperation LibReflect  
    LibProd LibNat LibInt LibOption LibWf.  
From SLF Require Export LibList LibNat.  
From SLF Require Import LibInt.  
From SLF Require Export LibContainer.  
  
Open Scope Int_scope.  
Local Open Scope comp_scope.  
Ltac auto_tilde ::= eauto with maths.
```

21.1 List operations using indices in Z

21.2 Length, with result as *int*

Defined using a coercion from `nat` to `int`

```
Definition length A (l:list A) : int :=  
    LibList.length l.
```

Section Length.

Variables A : Type.

Implicit Types x : A.

Implicit Types l : list A.

Implicit Types i : int.

```
Lemma length_eq : ∀ l,  
    length l = LibList.length l.
```

Proof using. auto. Qed.

```

Lemma abs_length : ∀ i l,
  i = length l →
  abs i = LibList.length l.

```

Proof using.

```

  introv E. unfold length in E.
  applys eq_nat_of_eq_int.
  rewrite abs_nonneg; math.

```

Qed.

```

Lemma length_nonneg : ∀ (l: list A),
  0 ≤ length l.

```

Proof using. intros. rewrite length_eq. math. Qed.

Lemma length_nil :

```

  length (@nil A) = 0.

```

Proof using. auto. Qed.

Lemma length_cons : ∀ x l,

```

  length (x :: l) = 1 + length l.

```

Proof using. intros. unfold length. rew_list¬. Qed.

Lemma length_one: ∀ x,

```

  length (x :: nil) = 1.

```

Proof using. reflexivity. Qed.

Lemma length_app : ∀ l1 l2,

```

  length (l1 ++ l2) = length l1 + length l2.

```

Proof using. intros. unfold length. rew_list¬. Qed.

Lemma length_last : ∀ x l,

```

  length (l & x) = 1 + length l.

```

Proof using. intros. unfold length. rew_list¬. Qed.

End Length.

```

Hint Rewrite length_nil length_cons length_app
length_last : rew_list.

```

```

Hint Rewrite length_nil length_cons length_app
length_last : rew_listx.

```

Automation for *math*, to unfold *length*

```

Hint Rewrite length_eq : rew_maths.

```

Demo of automation with maths

Goal ∀ A (l : list A), 0 ≤ length l.

Proof using. intros. math. Qed.

Goal ∀ (l : list Z) (s n : int),

```

  s ≤ n →

```

```

  s = length l →

```

$n \geq 0$.

Proof using. intros. math. Qed.

21.3 Inversion lemmas for structural composition

Section ApplInversion.

Variables $A : \text{Type}$.

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

Lemma length_zero_inv : $\forall l,$
 $\text{length } l = 0 \rightarrow$
 $l = \text{nil}.$

Proof using. intros. unfolds length. applys \neg LibList.length_zero_inv. Qed.

Lemma length_zero_eq_eq_nil : $\forall l,$
 $(\text{length } l = 0) = (l = \text{nil}).$

Proof using.

intros. unfolds length. rewrite \leftarrow LibList.length_zero_eq_eq_nil. math.

Qed.

Lemma length_neq_inv : $\forall l_1 l_2,$
 $\text{length } l_1 \neq \text{length } l_2 \rightarrow$
 $(l_1 \neq l_2).$

Proof using. introv N E. subst \times . Qed.

Lemma length_pos_inv_cons : $\forall l,$
 $(\text{length } l > 0) \rightarrow$
 $\exists x l', l = x :: l'.$

Proof using.

intros. unfolds length. applys \neg LibList.length_pos_inv_cons.

Qed.

Lemma length_pos_inv_last : $\forall l,$
 $(\text{length } l > 0) \rightarrow$
 $\exists x l', l = l' & x.$

Proof using.

intros. unfolds length. applys \neg LibList.length_pos_inv_last.

Qed.

End ApplInversion.

21.4 index, with length as *int*, as typeclass

Definition index_impl $A (l:\text{list } A) (i:\text{int}) : \text{Prop} :=$

```

index (LibList.length l : int) i.

Instance index_inst : ∀ A, BagIndex int (list A).
Proof using. constructor. rapply (@index_impl A). Defined.

Section Index.
Variables (A : Type).
Implicit Types l : list A.
Implicit Types n i : int.

Lemma index_eq_inbound : ∀ l i,
  index l i = (0 ≤ i < length l).
Proof using. auto. Qed.

Lemma index_of_inbound : ∀ l i,
  0 ≤ i < length l →
  index l i.
Proof using. intros. rewrite¬ index_eq_inbound. Qed.

Lemma index_eq_index_length : ∀ l i,
  index l i = index (length l) i.
Proof using. auto. Qed.

Lemma index_of_index_length : ∀ l i,
  index (length l) i →
  index l i.
Proof using. introv H. rewrite× index_eq_index_length. Qed.

Reformulation of above, helpful for automation

Lemma index_of_index_length' : ∀ l n i,
  index n i →
  n = length l →
  index l i.
Proof using. intros. subst. rewrite¬ index_eq_index_length. Qed.

End Index.

Global Opaque index_inst.

```

21.5 read, with length as *int*, as typeclass

```

Definition read_impl '{Inhab A} (l:list A) (i:int) : A :=
  If i < 0 then arbitrary else nth (abs i) l.

Instance read_inst : ∀ '{Inhab A}, BagRead int A (list A).
Proof using. constructor. rapply (@read_impl A H). Defined.

Section Read.
Context (A : Type) '{Inhab A}.

```

```

Implicit Types  $x : A$ .
Implicit Types  $l : \text{list } A$ .
Implicit Types  $n i : \text{int}$ .

Lemma read_cons_case :  $\forall l i v, (v :: l)[i] = (\text{If } i = 0 \text{ then } v \text{ else } l[i - 1])$ .
Proof using.
  introv. simpl. unfold read_impl. case_if.
  { case_if. math. case_if~. math. }
  { case_if~.
    { subst. rewrite abs_0. rew_listx~. }
    { rewrite~ abs_eq_succ_abs_minus_one. rew_listx.
      case_if. math. auto. } }
Qed.

Lemma read_zero :  $\forall x l, (x :: l)[0] = x$ .
Proof using.
  intros. rewrite read_cons_case. case_if. auto.
Qed.

Lemma read_succ :  $\forall x l i, 0 \leq i < \text{length } l \rightarrow (x :: l)[i + 1] = l[i]$ .
Proof using.
  introv M. rewrite read_cons_case. case_if. { math. }
  fequals. math.
Qed.

Lemma read_last_case :  $\forall l i v, (l & v)[i] = (\text{If } i = \text{length } l \text{ then } v \text{ else } l[i])$ .
Proof using.
  introv. simpl. unfold read_impl. case_if.
  { case_if~; math. }
  { rewrite nth_last_case. rewrite~ abs_eq_nat_eq. }
Qed.

Lemma read_middle :  $\forall i l1 l2 x, i = \text{length } l1 \rightarrow (l1 ++ x :: l2)[i] = x$ .
Proof.
  introv M. rewrite length_eq in M. unfold read, read_inst, read_impl.
  case_if. { false; math. }
  rewrite~ nth_middle.
Qed.

Lemma read_app :  $\forall i l1 l2,$ 

```

$(l1 ++ l2)[i] = (\text{If } i < \text{length } l1 \text{ then } l1[i] \text{ else } l2[i - \text{length } l1]).$

Proof using.

```
intros. rewrite length_eq. unfold read, read_inst, read_impl. case_if.
{ case_if. { auto. } { false; math. } }
case_if as C'.
{ applys nth_app_l. applys lt_nat_of_lt_int. rewrite abs_nonneg; math. }
case_if. { false; math. }
rewrite abs_gt_minus_nat; [|math]. applys nth_app_r.
{ applys ge_nat_of_ge_int. rewrite abs_nonneg; math. }
```

Qed.

21.6 Equality between two lists from equality of length and

extensional equality of reads.

```
Lemma eq_of_extens_range : ∀ l1 l2,
  length l1 = length l2 →
  (∀ i, 0 ≤ i < length l1 → l1[i] = l2[i]) →
  l1 = l2.
```

Proof using.

```
introv HL HR. do 2 rewrite length_eq in HL.
unfold read, read_inst, read_impl in HR.
applys¬ LibList.eq_of_extens l1 l2.
{ intros n L. forwards M: (rm HR) (nat_to_Z n). math.
  case_if. false; math. rewrite¬ abs_nat in M. }
```

Qed.

```
Lemma eq_of_extens : ∀ l1 l2,
  length l1 = length l2 →
  (∀ i, index l1 i → l1[i] = l2[i]) →
  l1 = l2.
```

Proof using. intros. applys¬ eq_of_extens_range. Qed.

End Read.

Global Opaque read_inst.

21.7 update, with index as *int*, as typeclass

```
Definition update_impl A (l:list A) (i:int) (v:A) : list A :=
  If i < 0 then l else LibList.update (abs i) v l.
```

Instance update_inst : ∀ A, BagUpdate int A (list A).

Proof using. constructor. *rapply (@update_impl A)*. Defined.

Section Update.

Transparent index_inst read_inst update_inst.

Context (A : Type) ‘{IA:**Inhab** A }.

Implicit Types $x v w : A$.

Implicit Types l : **list** A .

Implicit Types $i j$: int.

Lemma length_update : $\forall l i v$,
 $\text{length } (l[i:=v]) = \text{length } l$.

Proof using.

intros. unfold update_inst, update_impl, length, update. simpl.
case_if. math. rewrite¬ length_update.

Qed.

Lemma index_update_eq : $\forall l i j v$,
 $\text{index } (l[j:=v]) i = \text{index } l i$.

Proof using. intros. rewrite index_eq_index_length in *. rewrite¬ length_update. Qed.

Lemma index_update : $\forall l i j v$,
 $\text{index } l i \rightarrow$
 $\text{index } (l[j:=v]) i$.

Proof using. intros. rewrite¬ index_update_eq. Qed.

Lemma read_update_case : $\forall l i j v$,
 $\text{index } l j \rightarrow$
 $l[i:=v][j] = (\text{If } i = j \text{ then } v \text{ else } l[j])$.

Proof using.

intros. unfold index_inst, index_impl, update_inst, update_impl, update,
read_inst, read_impl. simpl. introv N . rewrite int_index_eq in N .
case_if. math.
case_if. case_if. auto. case_if.
subst. rewrite¬ nth_update_same.
rewrite¬ nth_update_neq.

Qed.

Lemma read_update_same : $\forall l i v$,
 $\text{index } l i \rightarrow$
 $(l[i:=v])[i] = v$.

Proof using. introv N . rewrite¬ read_update_case. case_if¬. Qed.

Lemma read_update_neq : $\forall l i j v$,
 $\text{index } l j \rightarrow$
 $(i \neq j) \rightarrow$
 $(l[i:=v])[j] = l[j]$.

Proof using. introv N . rewrite¬ read_update_case. case_if; auto_false¬. Qed.

End Update.

```

Section UpdateNoInhab.

Transparent index_inst read_inst update_inst.

Context (A : Type).

Implicit Types x v w : A.
Implicit Types l : list A.
Implicit Types i j : int.

Lemma update_zero : ∀ v x l,
  (x :: l) [0 := v] = v :: l.

Proof using.
  intros. unfold update, update_inst, updateImpl.
  case_if. false; math. rewrite¬ update_zero.

Qed.

Lemma update_cons_pos : ∀ n v x l,
  n > 0 →
  (x :: l) [n := v] = x :: (l [(n - 1) := v]).
```

Proof using.

```

  introv N. unfold update, update_inst, updateImpl.
  do 2 (case_if; try solve [ false; math ]).
  rewrite¬ update_cons_pos.
  { f_equal_rec. rewrite ← abs_gt_minus_nat. f_equal. math. }
```

Qed.

```

Lemma update_update_same : ∀ l i v w,
  index l i →
  l [i := v] [i := w] = l [i := w].
```

Proof using.

```

  intros. asserts IA: (Inhab A). typeclass.
  eapply eq_of_extens; repeat rewrite length_update. { auto. }
  intros k Hk. repeat rewrite index_update_eq in Hk.
  repeat rewrite read_update_case by eauto using index_update.
  case_if¬.
```

Qed.

```

Lemma update_update_neq : ∀ l i j v w,
  index l i →
  index l j →
  i ≠ j →
  l [i := v] [j := w] = l [j := w] [i := v].
```

Proof using.

```

  intros. asserts IA: (Inhab A). typeclass.
  applys eq_of_extens; repeat rewrite length_update. { auto. }
  intros k Hk. repeat rewrite index_update_eq in Hk.
  repeat rewrite read_update_case by eauto using index_update.
```

```
repeat case_if¬.
```

Qed.

```
Lemma update_app_l : ∀ l1 i l2 v,  
  0 ≤ i < length l1 →  
  (l1 ++ l2) [i:=v] = (l1 [i:=v]) ++ l2.
```

Proof using.

```
  introv N asserts IA: (Inhab A). typeclass.  
  unfold LibContainer.update, update_inst, update_impl.  
  rewrite length_eq in N. case_if¬. rewrite¬ update_app_l.
```

Qed.

```
Lemma update_app_r : ∀ l2 j l1 i ij v,  
  i = length l1 →  
  0 ≤ j →  
  ij = i + j →  
  (l1 ++ l2) [ij:=v] = l1 ++ (l2 [j:=v]).
```

Proof using.

```
  intros. asserts IA: (Inhab A). typeclass. subst ij.  
  unfold LibContainer.update, update_inst, update_impl.  
  do 2 (case_if; [ math | ]).  
  rewrite Zabs2Nat.inj_add; try math. subst i.  
  rewrite length_eq. rewrite abs_nat.  
  rewrite¬ update_app_r. fequals_rec. math.
```

Qed.

```
Lemma update_middle : ∀ i l1 l2 v w,  
  i = length l1 →  
  (l1 ++ w :: l2) [i := v] = l1 & v ++ l2.
```

Proof using.

```
  introv E. rewrites¬ (» update_app_r 0 i).  
  rewrite update_zero. rew_list¬.
```

Qed.

End UpdateNoInhab.

Global Opaque update_inst.

21.8 make, with length as *int*

```
Definition make A (n:int) (v:A) : list A :=  
  If n < 0 then arbitrary else make (abs n) v.
```

Section Make.

Transparent index_inst read_inst.

Context (A : Type) {IA:Inhab A}.

```
Implicit Types  $x v : A$ .
```

```
Implicit Types  $l : \text{list } A$ .
```

```
Implicit Types  $n i : \text{int}$ .
```

```
Lemma length_make :  $\forall n v,$ 
```

```
   $n \geq 0 \rightarrow$ 
```

```
  length (make  $n v$ ) =  $n$ .
```

Proof using.

```
  introv  $N$ . unfold make. case_if. math.
```

```
  unfold length. rewrite LibList.length_make.
```

```
  rewrite¬ abs_nonneg.
```

Qed.

```
Lemma index_make :  $\forall n i v,$ 
```

```
  index  $n i \rightarrow$ 
```

```
  index (make  $n v$ )  $i$ .
```

Proof using.

```
  introv  $H$ . rewrite index_eq_index_length.
```

```
  rewrite int_index_eq in  $H$ .
```

```
  rewrite¬ length_make.
```

Qed.

```
Lemma read_make :  $\forall i n v,$ 
```

```
  index  $n i \rightarrow$ 
```

```
  (make  $n v$ ) [ $i$ ] =  $v$ .
```

Proof using.

```
  introv  $N$ . rewrite int_index_eq in  $N$ .
```

```
  unfold make, read_inst, read_impl.
```

```
  case_if. math. simpl. case_if. math.
```

```
  applys nth_make. forwards: lt_abs_abs  $i n$ ; try math.
```

Qed.

```
Lemma make_zero :  $\forall v,$ 
```

```
  make 0  $v$  = nil.
```

Proof using.

```
  intros. unfold make. case_if. { false; math. }
```

```
  asserts_rewrite (abs 0 = 0%nat).
```

```
  { applys eq_nat_of_eq_int. rewrite abs_nonneg; math. }
```

```
  auto.
```

Qed.

End Make.

```
Section MakeNolhab.
```

```
Transparent index_inst read_inst.
```

```
Context ( $A : \text{Type}$ ).
```

```
Implicit Types  $x v : A$ .
```

```

Implicit Types  $l$  : list  $A$ .
Implicit Types  $n i$  : int.

Lemma make_succ_l :  $\forall n v,$ 
   $n \geq 0 \rightarrow$ 
   $\text{make } (n+1) v = v :: \text{make } n v.$ 
Proof using.
  introv  $N$ . unfold make. case_if; case_if; try solve [false;math].
  rewrite ← LibList.make_succ. fequal.
  { rewrite succ_abs_eq_abs_one_plus. fequal. math. math. }
Qed.

Lemma make_succ_r :  $\forall n v,$ 
   $n \geq 0 \rightarrow$ 
   $\text{make } (n+1) v = \text{make } n v & v.$ 
Proof using.
  intros. asserts IA: (Inhab  $A$ ). applys Inhab_of_val  $v$ .
  applys eq_of_extens_range.
  { rewrite length_make. rew_list. rewrite length_make.
    math. math. math. }
  { intros  $i Ei$ . rewrite length_make in  $Ei$ ; [] math].
    rewrite read_make; [] rewrite int_index_eq; math].
    rewrite read_app. rewrite length_make; []math].
    case_if as  $C$ .
    { rewrite read_make. auto. rewrite int_index_eq; math. }
    { math_rewrite ( $i - n = 0$ ). rewrite¬ read_zero. } }
Qed.

Lemma make_eq_cons_make_pred :  $\forall n v,$ 
   $0 < n \rightarrow$ 
   $\text{make } n v = v :: (\text{make } (n-1) v).$ 
Proof using.
  intros. math_rewrite ( $n = (n-1)+1$ ). rewrite make_succ_l.
  fequals_rec. math. math.
Qed.

End MakeNoInhab.

Global Opaque make.

```

21.9 *LibList.rev* interactions with **LibListZ** operations

```

Section Rev.
Variables ( $A$  : Type).
Implicit Types  $x : A$ .

```

```

Implicit Types  $l : \text{list } A$ .
Lemma length_rev :  $\forall l$ ,
   $\text{length} (\text{rev } l) = \text{length } l$ .
Proof using. intros. unfold length.  $\text{rewrite } \text{rew\_list} \leftarrow$ . Qed.
End Rev.

Hint Rewrite length_rev :  $\text{rew\_list}$ .
Hint Rewrite length_rev :  $\text{rew\_list}x$ .

```

21.10 *LibList.map* interactions with LibListZ operations

Section Map.

Transparent index_inst read_inst update_inst.

Implicit Types $n i : \text{int}$.

Lemma length_map : $\forall A B (l:\text{list } A) (f:A \rightarrow B)$,
 $\text{length} (\text{map } f l) = \text{length } l$.

Proof using. intros. unfold length. $\text{rewrite } \text{length} \leftarrow \text{length_map}$. Qed.

Lemma index_map_eq : $\forall A B (l:\text{list } A) i (f:A \rightarrow B)$,
 $\text{index} (\text{map } f l) i = \text{index } l i$.

Proof using. intros. $\text{rewrite } \text{index_eq_inbound} \text{ in } ^*$. $\text{rewrite } \text{length} \leftarrow \text{length_map}$. Qed.

Lemma index_map : $\forall A B (l:\text{list } A) i (f:A \rightarrow B)$,
 $\text{index } l i \rightarrow \text{index} (\text{map } f l) i$.

Proof using. intros. $\text{rewrite } \text{index_map_eq}$. Qed.

Lemma map_make : $\forall A B (f:A \rightarrow B) (n:\text{int}) (v:A)$,
 $n \geq 0 \rightarrow$
 $\text{map } f (\text{make } n v) = \text{make } n (f v)$.

Proof using.

Transparent make.

intros. unfold make. $\text{case_if. } \{ \text{false}; \text{math.} \}$

applys map_make.

Qed.

Lemma map_update : $\forall A B (l:\text{list } A) (i:\text{int}) (x:A) (f:A \rightarrow B)$,
 $\text{index } l i \rightarrow$
 $\text{map } f (l[i := x]) = (\text{map } f l)[i := f x]$.

Proof using.

introv H. rewrite index_eq_inbound in H.

unfold update_inst, update_impl, update. simpl.

case_if. { false; math. }

{ applys LibList.map_update.

{ applys lt_nat_of_lt_int. rewrite abs_nonneg; math. } }

Qed.

```
Lemma read_map : ∀ {IA:Inhab A} {IB:Inhab B} (l:list A) (i:int) (f:A→B),  
  index l i →  
  (map f l)[i] = f (l[i]).
```

Proof using.

```
  intros H. rewrite index_eq_inbound in H.  
  unfold read_inst, read_impl. simpl. case_if.  
  { false; math. }  
  { rewrite nth_map. auto.  
    applys lt_nat_of_lt_int. rewrite abs_nonneg; math. }
```

Qed.

End Map.

```
Hint Rewrite length_map index_map_eq : rew_listx.
```

21.11 *LibList.filter* interactions with **LibListZ** operations

Section Filter.

Variables (A : Type).

Implicit Types x : A.

Implicit Types l : **list** A.

Implicit Types P : A → Prop.

```
Lemma length_filter : ∀ l P,  
  length (filter P l) ≤ length l.
```

Proof using.

```
  intros. unfolds length. forwards~: LibList.length_filter l P.
```

Qed.

```
Lemma filter_length_two_disjoint : ∀ (P Q : A→ Prop) l,  
  (∀ x, mem x l → P x → Q x → False) →  
  length (filter P l) + length (filter Q l) ≤ length l.
```

Proof using.

```
  intros. unfolds length.  
  forwards~: LibList.filter_length_two_disjoint.
```

Qed.

```
Lemma filter_length_partition : ∀ P l,  
  length (filter (fun x ⇒ P x) l)  
  + length (filter (fun x ⇒ ¬ P x) l)  
  ≤ length l.
```

Proof using.

```
  intros. unfolds length. forwards~: LibList.filter_length_partition P l.
```

Qed.

```

Lemma length_filter_eq_mem_ge_one : ∀ x l,
  mem x l →
  length (filter (= x) l) ≥ 1.
Proof using.
  intros. unfolds length. forwards~: LibList.length_filter_eq_mem_ge_one.
Qed.
End Filter.

```

21.12 *LibList.remove* interactions with **LibListZ** operations

Section Remove.

Variables ($A : \text{Type}$).

Implicit Types $a x : A$.

Implicit Types $l : \text{list } A$.

```

Lemma length_remove : ∀ l a,
  length (LibList.remove a l) ≤ length l.

```

Proof using. intros. rewrite remove_as_filter. applys length_filter. Qed.

```

Lemma length_remove_mem : ∀ x l,
  mem x l →
  length (LibList.remove x l) < length l.

```

Proof using.

intros. unfolds length. forwards~: LibList.length_remove_mem.

Qed.

End Remove.

21.12.1 Take, with an *int* as the number of elements

```

Definition take A (n:int) (l:list A) : list A :=
  LibList.take (to_nat n) l.

```

Section Take.

Variables ($A : \text{Type}$).

Implicit Types $n : \text{int}$.

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

```

Lemma take_nil : ∀ n,
  take n (@nil A) = nil.

```

Proof using. intros. unfold take. apply LibList.take_nil. Qed.

Lemma take_zero : ∀ l,

```

take 0 l = nil.
Proof using. auto. Qed.

Lemma take_succ : ∀ x l n,
  0 ≤ n →
  take (n+1) (x::l) = x :: (take n l).

```

Proof using.

```

intros. unfold take. rew_to_nat_nonneg¬.
rewrite Nat.add_1_r. apply LibList.take_succ.

```

Qed.

Definition take_cons := take_succ.

```

Lemma take_cons_pos : ∀ x l n,
  (n > 0) →
  take n (x::l) = x :: (take (n-1) l).

```

Proof using.

```

intros. unfold take. rew_to_nat_nonneg¬.
rewrite¬ LibList.take_cons_pos.

```

Qed.

```

Lemma take_neg : ∀ n l,
  n ≤ 0 →
  take n l = nil.

```

Proof using. intros. unfold take. rewrite¬ to_nat_neg. Qed.

```

Lemma take_ge : ∀ n l,
  (n ≥ length l) →
  take n l = l.

```

Proof using.

```

intros. unfold take, length in *. applys¬ LibList.take_ge.

```

Qed.

```

Lemma take_is_prefix : ∀ n l,
  ∃ q, l = take n l ++ q.

```

Proof using.

```

intros. unfold take. applys¬ LibList.take_is_prefix.

```

Qed.

```

Lemma take_app_l : ∀ n l l',
  (n ≤ length l) →
  take n (l ++ l') = take n l.

```

Proof using.

```

intros. tests: (0 ≤ n).
{ unfold take, length in *.
  applys¬ LibList.take_app_l. }
{ rewrite !take_neg; auto; math. }

```

Qed.

```

Lemma take_app_r : ∀ n l l',
  (n ≥ length l) →
  take n (l ++ l') = l ++ take (n - length l) l'.

```

Proof using.

```

intros. unfold take, length in *. rew_to_nat_nonneg¬.
applys¬ LibList.take_app_r.

```

Qed.

```

Lemma take_prefix_length : ∀ l l',
  take (length l) (l ++ l') = l.

```

Proof using.

```

intros. unfold take, length. rew_to_nat_nonneg¬.
applys¬ LibList.take_prefix_length.

```

Qed.

```

Lemma take_full_length : ∀ l,
  take (length l) l = l.

```

Proof using.

```

intros. unfold take, length. rew_to_nat_nonneg¬.
apply LibList.take_full_length.

```

Qed.

End Take.

```
Hint Rewrite take_nil take_zero take_succ : rew_listx.
```

21.12.2 Drop, with an *int* as the number of elements.

```

Definition drop A (n:int) (l:list A) : list A :=
  LibList.drop (to_nat n) l.

```

Section Drop.

Variables (A : Type).

Implicit Types n : int.

Implicit Types x : A.

Implicit Types l : list A.

```

Lemma drop_nil : ∀ n,
  drop n (@nil A) = nil.

```

Proof using. intros. unfold drop. apply LibList.drop_nil. Qed.

```

Lemma drop_zero : ∀ l,
  drop 0 l = l.

```

Proof using. auto. Qed.

```

Lemma drop_succ : ∀ x l n,
  0 ≤ n →

```

```
drop (n+1) (x::l) = (drop n l).
```

Proof using.

```
intros. unfold drop. rew_to_nat_nonneg¬.  
rewrite Nat.add_1_r. apply LibList.drop_succ.
```

Qed.

Definition drop_cons := drop_succ.

Lemma drop_neg : $\forall n l,$

$n \leq 0 \rightarrow$
 $\text{drop } n l = l.$

Proof using. intros. unfold drop. rewrite¬ to_nat_neg. Qed.

Lemma drop_cons_pos : $\forall x l n,$

$(n > 0) \rightarrow$
 $\text{drop } n (x::l) = \text{drop } (n-1) l.$

Proof using.

```
intros. unfold drop. rew_to_nat_nonneg¬.  
apply¬ LibList.drop_cons_pos.
```

Qed.

Lemma drop_is_suffix : $\forall n l,$

$\exists q, l = q ++ \text{drop } n l.$

Proof using.

```
intros. unfold drop. apply LibList.drop_is_suffix.
```

Qed.

Lemma drop_app_l : $\forall n l l',$

$(n \leq \text{length } l) \rightarrow$
 $\text{drop } n (l ++ l') = \text{drop } n l ++ l'.$

Proof using.

```
intros. tests: (0 ≤ n).  
{ unfold drop, length in *.  
  apply LibList.drop_app_l. rewrites¬ to_nat_le_nat_le. }  
{ rewrite !drop_neg; auto; math. }
```

Qed.

Lemma drop_app_r : $\forall n l l',$

$(n \geq \text{length } l) \rightarrow$
 $\text{drop } n (l ++ l') = \text{drop } (n - \text{length } l) l'.$

Proof using.

```
intros. unfold drop, length in *. rew_to_nat_nonneg¬.  
apply LibList.drop_app_r. rewrites¬ to_nat_ge_nat_ge.
```

Qed.

Lemma drop_app_length : $\forall l l',$

$\text{drop } (\text{length } l) (l ++ l') = l'.$

Proof using.

```

intros. unfold drop, length. rew_to_nat_nonneg¬.
apply LibList.drop_app_length.

```

Qed.

```

Lemma drop_at_length : ∀ l,
  drop (length l) l = nil.

```

Proof using.

```

intros. unfold drop, length. rew_to_nat_nonneg¬.
apply LibList.drop_at_length.

```

Qed.

End Drop.

```
Hint Rewrite drop_nil drop_zero drop_succ : rew_listx.
```

21.12.3 Take and drop decomposition of a list

Section TakeAndDrop.

Variables (A : Type).

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

```

Lemma take_app_drop_spec : ∀ n l f r,
  f = take n l →
  r = drop n l →
  (0 ≤ n ≤ length l →
    l = f ++ r
   ∧ length f = n
   ∧ length r = length l - n) ∧
  (n ≤ 0 →
    f = nil ∧ r = l).

```

Proof using.

```

introv ? ?. split.
{ intros (? & Hn). unfold take, drop, length in *.
  forwards¬ (? & ? & Hlenrest): @LibList.take_app_drop_spec (to_nat n) l f r. }
{ intros. rewrites¬ take_neg in *. rewrites¬ drop_neg in *. }

```

Qed.

```

Lemma take_spec : ∀ n l,
  0 ≤ n ≤ length l →
  ∃ l', length (take n l) = n
  ∧ l = (take n l) ++ l'.

```

Proof using. introv E. forwards× (E1&_): take_app_drop_spec. forwards*: E1. Qed.

```

Lemma length_take_nonneg : ∀ n l,
  0 ≤ n ≤ length l →

```

$\text{length}(\text{take } n \ l) = n$.

Proof using. *intros* E . $\text{forwards} \neg (l' \& N \& M)$: $\text{take_spec } n \ l$. Qed.

Lemma $\text{length_take} : \forall n \ l,$

$n \leq \text{length } l \rightarrow$

$\text{length}(\text{take } n \ l) = \text{Z.max } 0 \ n$.

Proof using.

intros. *tests*: $(0 \leq n)$.

{ $\text{rewrite} \neg \text{length_take_nonneg}$. }

{ $\text{rewrite} \neg \text{take_neg}$. $\text{rewrite} \neg \text{Z.max_l}$. }

Qed.

Lemma $\text{drop_spec} : \forall n \ l,$

$0 \leq n \leq \text{length } l \rightarrow$

$\exists l', \text{length } l' = n$

$\wedge l = l' ++ (\text{drop } n \ l)$.

Proof using. *intros* E . $\text{forwards} \times (E1 \& _)$: $\text{take_app_drop_spec}$. $\text{forwards}^*: E1$. Qed.

Lemma $\text{length_drop_nonneg} : \forall n \ l,$

$0 \leq n \leq \text{length } l \rightarrow$

$\text{length}(\text{drop } n \ l) = \text{length } l - n$.

Proof using.

intros E . $\text{forwards} \neg (l' \& N \& M)$: $\text{drop_spec } n \ l$.

pattern l at 2. *rewrite* M . *rew_list*. *math*.

Qed.

Lemma $\text{length_drop} : \forall n \ l,$

$n \leq \text{length } l \rightarrow$

$\text{length}(\text{drop } n \ l) = \text{Z.min}(\text{length } l) (\text{length } l - n)$.

Proof using.

intros. *tests*: $(0 \leq n)$.

{ $\text{rewrite} \neg \text{length_drop_nonneg}$. }

{ $\text{rewrite} \neg \text{drop_neg}$. }

Qed.

Lemma $\text{list_eq_take_app_drop} : \forall n \ l,$

$n \leq \text{length } l \rightarrow$

$\text{take } n \ l ++ \text{drop } n \ l = l$.

Proof using.

intros H . *tests*: $(0 \leq n)$.

{ $\text{forwards}^*: \text{take_app_drop_spec } n \ l$. }

{ $\text{rewrite} \neg \text{take_neg}$. $\text{rewrite} \neg \text{drop_neg}$. }

Qed.

End TakeAndDrop.

Arguments $\text{take_app_drop_spec} [A]$.

Arguments $\text{take_spec} [A]$.

Arguments drop_spec [A].

21.12.4 count, returning an int

Definition count A (P: A → Prop) (l: list A): int :=
abs (LibList.count P l).

Section Count.

Variables A : Type.

Implicit Types l : list A.

Implicit Types P : A → Prop.

Ltac gowith L :=

```
let H := fresh in
lets H: L;
repeat (let x := fresh in intro x; specializes H x);
unfold count in *; repeat rewrites¬ abs_nonneg in *;
try solve [
  first [ rewrites¬ H in * | forwards¬: H ];
  repeat case_if¬
].
```

Lemma count_nil : ∀ P,

count P nil = 0.

Proof using. auto. Qed.

Lemma count_one : ∀ P x,

count P (x::nil) = If P x then 1 else 0.

Proof using. gowith LibList.count_one. Qed.

Lemma count_cons : ∀ P l x,

count P (x::l) = count P l + If P x then 1 else 0.

Proof using. gowith LibList.count_cons. Qed.

Lemma count_app : ∀ P l1 l2,

count P (l1++l2) = count P l1 + count P l2.

Proof using. gowith LibList.count_app. Qed.

Lemma count_last : ∀ P l x,

count P (l&x) = count P l + If P x then 1 else 0.

Proof using. gowith LibList.count_last. Qed.

Lemma count_rev : ∀ P l,

count P (rev l) = count P l.

Proof using. gowith LibList.count_rev. Qed.

Lemma count_eq_length_filter : ∀ P l,

count P l = length (filter P l).

Proof using. *gowith* LibList.count_eq_length_filter. Qed.

Lemma count_nonneg : $\forall P l,$
 $0 \leq \text{count } P l.$

Proof using. intros. unfold count. math. Qed.

Lemma count_le_length : $\forall P l,$
 $\text{count } P l \leq \text{length } l.$

Proof using. *gowith* LibList.count_le_length. Qed.

Lemma Forall_eq_count_eq_length : $\forall P l,$
Forall $P l = (\text{count } P l = \text{length } l).$

Proof using.

intros. rewrite LibList.Forall_eq_count_eq_length, length_eq.
unfold count. rewrite abs_nonneg; math.

Qed.

Lemma count_Forall : $\forall P l,$
Forall $P l \rightarrow$
 $\text{count } P l = \text{length } l.$

Proof using. introv. rewrite \lhd Forall_eq_count_eq_length. Qed.

Lemma Forall_of_count_eq_length : $\forall P l,$
 $\text{count } P l = \text{length } l \rightarrow$
Forall $P l.$

Proof using. introv. rewrite \lhd Forall_eq_count_eq_length. Qed.

Lemma Forall_not_eq_count_eq_zero : $\forall P l,$
Forall $(\text{fun } x \Rightarrow \neg P x) l = (\text{count } P l = 0).$

Proof using.

intros. rewrite LibList.Forall_not_eq_count_eq_zero.
unfold count. rewrite abs_nonneg; math.

Qed.

Lemma Forall_not_of_count_eq_zero : $\forall P l,$
 $\text{count } P l = 0 \rightarrow$
Forall $(\text{fun } x \Rightarrow \neg P x) l.$

Proof using. introv. rewrite \lhd Forall_not_eq_count_eq_zero. Qed.

Lemma count_eq_zero_of_Forall_not : $\forall P l,$
Forall $(\text{fun } x \Rightarrow \neg P x) l \rightarrow$
 $\text{count } P l = 0.$

Proof using. introv. rewrite \lhd Forall_not_eq_count_eq_zero. Qed.

Lemma Exists_eq_count_pos : $\forall P l,$
Exists $P l = (\text{count } P l > 0).$

Proof using.

intros. rewrite LibList.Exists_eq_count_pos.

```

unfold count. rewrite abs_nonneg; math.
Qed.

Lemma Exists_of_count_pos : ∀ P l,
  count P l > 0 →
  Exists P l.

Proof using. introv. rewrite¬ Exists_eq_count_pos. Qed.

Lemma count_pos_of_Exists : ∀ P l,
  Exists P l →
  count P l > 0.

Proof using. introv. rewrite¬ Exists_eq_count_pos. Qed.

Lemma count_pos_eq_exists_mem : ∀ P l,
  (count P l > 0) = (exists x, mem x l ∧ P x).

Proof using.
  intros. rewrite ← LibList.count_pos_eq_exists_mem.
  unfold count. rewrite abs_nonneg; math.
Qed.

Lemma count_pos_of_mem : ∀ x P l,
  mem x l →
  P x →
  count P l > 0.

Proof using. introv. rewrite× count_pos_eq_exists_mem. Qed.

Lemma exists_mem_of_count_pos : ∀ P l,
  count P l > 0 →
  exists x, mem x l ∧ P x.

Proof using. introv. rewrite× count_pos_eq_exists_mem. Qed.

End Count.

Opaque count.

Hint Rewrite count_nil count_cons count_app count_last count_rev : rew_listx.

```

21.13 card, with result as **nat**, as typeclass

Note that `card` produces a **nat**, whereas `length` produces an *int*. Currently, in practice, we use `LibList.length` rather than `card`.

```

Definition card_impl A (l:list A) : nat :=
  LibList.length l.

Instance card_inst : ∀ A, BagCard (list A).
Proof using. constructor. rapply (@card_impl A). Defined.

Global Opaque card_inst.

```

21.14 Normalization tactics

rew_array_nocase is a light normalization tactic for array

```
Hint Rewrite @read_make @length_make @length_update @read_update_same
  : rew_array_nocase.

Tactic Notation "rew_array_nocase" :=
  autorewrite with rew_array_nocase.

Tactic Notation "rew_array_nocase" "in" hyp(H) :=
  autorewrite with rew_array_nocase in H.

Tactic Notation "rew_array_nocase" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_array_nocase).

Tactic Notation "rew_array_nocase" "~" :=
  rew_array_nocase; auto_tilde.

Tactic Notation "rew_array_nocase" "*" :=
  rew_array_nocase; auto_star.

Tactic Notation "rew_array_nocase" "~" "in" hyp(H) :=
  rew_array_nocase in H; auto_tilde.

Tactic Notation "rew_array_nocase" "*" "in" hyp(H) :=
  rew_array_nocase in H; auto_star.

Tactic Notation "rew_array_nocase" "~" "in" "*" :=
  rew_array_nocase in *; auto_tilde.

Tactic Notation "rew_array_nocase" "*" "in" "*" :=
  rew_array_nocase in *; auto_star.
```

rew_array is a normalization tactic for array, which introduces case analyses for all read-on-update operations.

```
Hint Rewrite @read_make @length_make @length_update @read_update_same
  @read_update_case @read_cons_case @read_last_case : rew_array.

Tactic Notation "rew_array" :=
  autorewrite with rew_array.

Tactic Notation "rew_array" "in" hyp(H) :=
  autorewrite with rew_array in H.

Tactic Notation "rew_array" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_array).

Tactic Notation "rew_array" "~" :=
  rew_array; auto_tilde.

Tactic Notation "rew_array" "*" :=
  rew_array; auto_star.

Tactic Notation "rew_array" "~" "in" hyp(H) :=
  rew_array in H; auto_tilde.

Tactic Notation "rew_array" "*" "in" hyp(H) :=
  rew_array in H; auto_star.
```

```
Tactic Notation "rew_array" "˜" "in" "*" :=  
  rew_array in *; auto_tilde.  
Tactic Notation "rew_array" "*" "in" "*" :=  
  rew_array in *; auto_star.  
  
Ltac auto_tilde ::= auto_tilde_default.
```

21.15 binds, with length as *int*, as typeclass

Chapter 22

Library SLF.LibMin

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibReflect LibOperation
LibRelation LibOrder LibEpsilon.

Generalizable Variables A .

Definition lower_bound A ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) ($x:A$) :=
 $\forall y, P y \rightarrow le x y$.

Definition min_element A ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) ($x:A$) :=
 $P x \wedge \text{lower_bound } le P x$.

Definition mmin ‘{Inhab A }’ ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) :=
epsilon (min_element $le P$).

Definition upper_bound A ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) ($x:A$) :=
 $\forall y, P y \rightarrow le y x$.

Definition max_element A ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) ($x:A$) :=
 $P x \wedge \text{upper_bound } le P x$.

Definition mmax ‘{Inhab A }’ ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) :=
epsilon (max_element $le P$).

Definition lub A ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) ($x:A$) :=
min_element $le (\text{upper_bound } le P) x$.

Definition glb A ($le:\text{binary } A$) ($P:A \rightarrow \text{Prop}$) ($x:A$) :=
max_element $le (\text{lower_bound } le P) x$.

Lemma upper_bound_inverse : $\forall A$ ($le:\text{binary } A$),
 $\text{upper_bound } le = \text{lower_bound } (\text{inverse } le)$.

Proof using.

extens. intros P x . unfolds lower_bound, upper_bound. iff \times .

Qed.

Lemma max_element_inverse : $\forall A (le:\text{binary } A) (P:A \rightarrow \text{Prop}) (x:A)$,

$\text{max_element } le P x = \text{min_element} (\text{inverse } le) P x$.

Proof using.

extens. unfold max_element, min_element. rewrite \times upper_bound_inverse.

Qed.

Lemma mmax_inverse : $\forall \{Inhab\ A\} (le:\text{binary } A) (P:A \rightarrow \text{Prop})$,

$\text{mmax } le P = \text{mmin} (\text{inverse } le) P$.

Proof using.

intros. applys epsilon_eq. intros x . rewrite \times max_element_inverse.

Qed.

Definition bounded_has_minimal $A (le:\text{binary } A) :=$

```
   $\forall P,$ 
  ex  $P \rightarrow$ 
  ex ( $\text{lower\_bound } le P \rightarrow$ 
  ex ( $\text{min\_element } le P$ )).
```

Lemma mmin_spec : $\forall \{Inhab\ A\} (le:\text{binary } A) (P:A \rightarrow \text{Prop}) m$,

$m = \text{mmin } le P \rightarrow$

ex $P \rightarrow$

ex ($\text{lower_bound } le P \rightarrow$

$\text{bounded_has_minimal } le \rightarrow$

$\text{min_element } le P m$.

Proof using.

intros. subst. unfold mmin. epsilon \times m .

Qed.

From SLF Require Import LibNat.

Lemma increment_lower_bound_nat : $\forall (P : \text{nat} \rightarrow \text{Prop}) x$,

$\text{lower_bound } le P x \rightarrow$

$\neg P x \rightarrow$

$\text{lower_bound } le P (x + 1) \% \text{nat}$.

Proof using.

introv hlo ?. intros y ?.

destruct (eq_nat_dec x y).

{ subst. tauto. }

{ forwards: hlo; eauto. nat_math. }

Qed.

```

Lemma bounded_has_minimal_nat :
  @bounded_has_minimal nat le.

Proof using.
  intros P [ y ? ].
  cut (
     $\forall (k x : \text{nat}), (y + 1 - x) \% \text{nat} = k \rightarrow$ 
    lower_bound le P x  $\rightarrow$ 
     $\exists z, \text{min\_element } \text{le } P z$ 
  ). { intros ? [ x ? ]. eauto. }
  induction k; introv ? hlo.
  { false. forwards: hlo; eauto. nat_math. }

destruct (prop_inv (P x)).
{  $\exists x. \text{split}; \text{eauto.}$  }
{ eapply (IHk (x + 1) \% nat). nat_math. eauto using increment_lower_bound_nat. }
Qed.

```

Hint Resolve bounded_has_minimal_nat : bounded_has_minimal.

```

Lemma admits_lower_bound_nat :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$ 
  ex (lower_bound le P).

```

Proof using.

```
 $\exists 0 \% \text{nat}. \text{unfold lower_bound. nat_math.}$ 
```

Qed.

Hint Resolve admits_lower_bound_nat : admits_lower_bound.

```

Lemma bounded_has_maximal_nat :
  @bounded_has_minimal nat (inverse le).

```

Proof using.

```

  intros P [ y ? ] [ x h ].
  assert ( $y \leq x$ ).
  { forwards: h. eauto. eauto. }

assert (self_inverse:  $\forall i, i \leq x \rightarrow (x - (x - i)) \% \text{nat} = i$ .
  intros. nat_math.

```

```
forwards [ z [ ? hz ]]: (@bounded_has_minimal_nat (fun i => P (x - i) \% nat)).
```

```
{  $\exists (x - y) \% \text{nat}. \text{rewrite self\_inverse by eauto. eauto.}$  }
{ eauto using admits_lower_bound_nat. }
```

```
 $\exists (x - z) \% \text{nat}.$ 
```

```
clear dependent y.
```

```
split; [ assumption | ].
```

```
intros y ?.
```

```
assert ( $y \leq x$ ).
```

```

{ forwards: h. eauto. eauto. }
forwards: hz (x - y)%nat.
{ rewrite self_inverse by eauto. eauto. }
unfold inverse. nat_math.

```

Qed.

Hint Resolve bounded_has_maximal_nat : bounded_has_minimal.

Lemma mmin_spec_nat:

```

 $\forall (P:\text{nat} \rightarrow \text{Prop}) m,$ 
 $m = \text{mmin le } P \rightarrow$ 
 $\text{ex } P \rightarrow$ 
 $P m \wedge (\forall x, P x \rightarrow m \leq x).$ 

```

Proof using.

```

introv E Q. applys (@mmin_spec _ _ _ P m E Q).
applys admits_lower_bound_nat.
applys bounded_has_minimal_nat.

```

Qed.

Definition MMin ‘{Inhab A} ‘{Le A} := mmin le.

Definition MMax ‘{Inhab A} ‘{Le A} := mmax le.

Chapter 23

Library SLF.LibSet

```
Set Implicit Arguments.
Generalizable Variables A B.
Require Import Coq.Classes.Morphisms. From SLF Require Import LibTactics LibLogic Li-
bReflect LibList
  LibOperation LibMonoid LibInt LibNat
  LibEpsilon LibRelation LibMin.
From SLF Require Export LibContainer.
```

23.1 Construction of sets as predicates

23.1.1 Basic definitions

```
Definition set (A : Type) := A → Prop.
```

```
Section Operations.
```

```
Variables (A B : Type).
```

```
Implicit Types x : A.
```

```
Implicit Types E F G : set A.
```

```
Definition set_st (P:A→Prop) : set A :=
  P.
```

```
Definition empty_impl : set A :=
  (fun _ => False).
```

```
Definition full_impl : set A :=
  (fun _ => True).
```

```
Definition single_impl x :=
  (= x).
```

```
Definition in_impl x E :=
```

$E\ x.$

Definition compl_impl : set A \rightarrow set A :=
 $\text{@pred_not } A.$

Definition union_impl : set A \rightarrow set A \rightarrow set A :=
 $\text{@pred_or } A.$

Definition inter_impl : set A \rightarrow set A \rightarrow set A :=
 $\text{@pred_and } A.$

Definition remove_impl : set A \rightarrow set A \rightarrow set A :=
 $\text{fun } E\ F\ x \Rightarrow E\ x \wedge \neg F\ x.$

Definition incl_impl : set A \rightarrow set A \rightarrow Prop :=
 $\text{@pred_incl } A.$

Definition disjoint_impl : set A \rightarrow set A \rightarrow Prop :=
 $\text{fun } E\ F : \text{set } A \Rightarrow \text{inter_impl } E\ F = \text{empty_impl}.$

Definition list_repr_impl (E:set A) (l:list A) :=
 $\text{nодuplicates } l \wedge \forall x, \text{mem } x\ l \leftrightarrow E\ x.$

Definition to_list (E:set A) :=
 $\text{epsilon } (\text{list_repr_impl } E).$

Definition to_set (xs : list A) : set A :=
 $\text{set_st } (\text{fun } x \Rightarrow \text{mem } x\ xs).$

Definition list_covers_impl (E:set A) L :=
 $\forall x, E\ x \rightarrow \text{mem } x\ L.$

Definition finite (E:set A) :=
 $\exists L, \text{list_covers_impl } E\ L.$

Definition card_impl (E:set A) : nat :=
 $\text{mmin le } (\text{fun } n \Rightarrow \exists L, \text{list_covers_impl } E\ L \wedge n = \text{length } L).$

Definition fold_impl (m:monoid_op B) (f:A \rightarrow B) (E:set A) :=
 $\text{LibList.fold_right } (\text{fun } x\ acc \Rightarrow \text{monoid_oper } m\ (f\ x)\ acc)$
 $(\text{monoid_neutral } m) (\text{to_list } E).$

End Operations.

23.1.2 Notations to help the typechecker

Notation "x \indom E" := (x \in (dom E : set _))
(at level 39) : container_scope.

Notation "x \notindom E" := (x \notin ((dom E) : set _))
(at level 39) : container_scope.

23.1.3 Inhabited

Instance **Inhab_set** : $\forall A, \mathbf{Inhab}(set\ A)$.
Proof using. intros. apply (**Inhab_of_val** (@empty_impl A)). Qed.

23.1.4 Notation through typeclasses

Lemma **in_inst** : $\forall A, \mathbf{BagIn}\ A$ (set A).

Proof using. constructor. exact (@in_impl A). Defined.

Hint Extern 1 (**BagIn** _ (set _)) \Rightarrow apply **in_inst** : *typeclass_instances*.

Instance **empty_inst** : $\forall A, \mathbf{BagEmpty}\ (set\ A)$.
constructor. apply (@empty_impl A). Defined.

Instance **single_inst** : $\forall A, \mathbf{BagSingle}\ A$ (set A).
constructor. rapply (@single_impl A). Defined.

Instance **union_inst** : $\forall A, \mathbf{BagUnion}\ (set\ A)$.
constructor. rapply (@union_impl A). Defined.

Instance **inter_inst** : $\forall A, \mathbf{BagInter}\ (set\ A)$.
constructor. rapply (@inter_impl A). Defined.

Instance **remove_inst** : $\forall A, \mathbf{BagRemove}\ (set\ A)$ (set A).
constructor. rapply (@remove_impl A). Defined.

Instance **incl_inst** : $\forall A, \mathbf{BagIncl}\ (set\ A)$.
constructor. rapply (@incl_impl A). Defined.

Instance **disjoint_inst** : $\forall A, \mathbf{BagDisjoint}\ (set\ A)$.
constructor. rapply (@disjoint_impl A). Defined.

Instance **fold_inst** : $\forall A\ B, \mathbf{BagFold}\ B\ (A \rightarrow B)$ (set A).
constructor. rapply (@fold_impl $A\ B$). Defined.

Instance **card_inst** : $\forall A, \mathbf{BagCard}\ (set\ A)$.
constructor. rapply (@card_impl A). Defined.

Global Opaque set finite in_inst empty_inst single_inst union_inst inter_inst
remove_inst incl_inst disjoint_inst card_inst fold_inst.

list_repr $E\ L$ asserts that elements of E are exactly the elements from the list L .

Definition **list_repr** $A\ (E:\text{set } A)\ (L:\text{list } A) :=$
noduplicates $L \wedge (\forall x, \mathbf{mem}\ x\ L \leftrightarrow x \setminus \in E)$.

list_covers $E\ L$ asserts that all elements of E all belong to the list L .

Definition **list_covers** $A\ (E:\text{set } A)\ (L:\text{list } A) :=$
 $\forall x, x \setminus \in E \rightarrow \mathbf{mem}\ x\ L$.

23.1.5 Notations for building sets

DISCLAIMER: these definitions are experimental, they'll probably change

Declare Scope set_scope.

```
Notation "\set{ x | P }" := (@set_st _ (fun x => P))
  (at level 0, x ident, P at level 200) : set_scope.
Notation "\set{ x : A | P }" := (@set_st A (fun x => P))
  (at level 0, x ident, P at level 200) : set_scope.
Notation "\set{ x '\in' E | P }" := (@set_st _ (fun x => x \in E ∧ P))
  (at level 0, x ident, P at level 200) : set_scope.
Notation "\set{= e | x '\in' E }" :=
  (@set_st _ (fun a => exists_ x \in E, a = e ))
  (at level 0, x ident, E at level 200) : set_scope.
Notation "\set{= e | x '\in' E , y '\in' F }" :=
  (@set_st _ (fun a => exists_ x \in E, exists_ y \in F, a = e ))
  (at level 0, x ident, F at level 200) : set_scope.
Notation "\set{= e | x y '\in' E }" :=
  (@set_st _ (fun a => exists_ x y \in E, a = e ))
  (at level 0, x ident, y ident, E at level 200) : set_scope.
```

23.2 Properties of sets

Section Instances.

Variables ($A:\text{Type}$).

Implicit Types $E F : \text{set } A$.

```
Transparent set finite empty_inst single_inst singleImpl in_inst
  incl_inst inter_inst union_inst card_inst fold_inst remove_inst
  disjoint_inst.
```

Hint Constructors mem.

Local tactic to help unfolding all intermediate definitions

```
Ltac set_unf := unfold finite,
  card_inst, card_impl, card,
  to_list,
  disjoint_impl, disjoint_inst, disjoint,
  incl_inst, incl_impl,
  empty_inst, empty_impl, empty,
  single_inst, single_impl, single,
  in_inst, in_impl, is_in,
  incl_inst, incl_impl, incl,
  compl_impl, pred_not,
```

```

inter_inst, interImpl, inter, pred_and,
union_inst, unionImpl, union, pred_or,
remove_inst, removeImpl, remove,
fold_inst, foldImpl, fold in *.

```

```

Lemma disjoint_eq_inter_empty : ∀ E F,
  (E \# F) = (E \n F = \{}).

```

Proof using. auto. Qed.

```

Lemma in_set_st_eq : ∀ (P:A→Prop) x,
  x \in set_st P = P x.

```

Proof using. intros. apply× prop_ext. Qed.

```

Lemma set_ext_eq : ∀ E F,
  (E = F) = (∀ (x:A), x \in E ↔ x \in F).

```

Proof using.

```

  intros. apply prop_ext. iff H. subst×. extens×.

```

Qed.

```

Lemma set_ext : ∀ E F,
  (∀ (x:A), x \in E ↔ x \in F) →
  E = F.

```

Proof using. intros. rewrite¬ set_ext_eq. Qed.

```

Lemma set_st_eq : ∀ A (P Q : A → Prop),
  (∀ (x:A), P x ↔ Q x) →
  set_st P = set_st Q.

```

Proof using. intros. asserts_rewrite¬ (P = Q). extens¬. Qed.

Global Instance in_extens_inst : **In_extens** (A:=A) (T:=set A).

Proof using. constructor. intros. rewrite× set_ext_eq. Qed.

Global Instance in_empty_eq_inst : **In_empty_eq** (A:=A) (T:=set A).

Proof using. constructor. intros. apply× prop_ext. Qed.

Global Instance in_single_eq_inst : **In_single_eq** (A:=A) (T:=set A).

Proof using. constructor. intros. apply× prop_ext. Qed.

Global Instance in_union_eq_inst : **In_union_eq** (A:=A) (T:=set A).

Proof using. constructor. intros. set_unf. simpl. apply× prop_ext. Qed.

Global Instance in_inter_eq_inst : **In_inter_eq** (A:=A) (T:=set A).

Proof using. constructor. intros. set_unf. apply× prop_ext. Qed.

Global Instance in_remove_eq_inst : **In_remove_eq** (A:=A) (T:=set A).

Proof using. constructor. intros. set_unf. apply× prop_ext. Qed.

Global Instance incl_in_eq_inst : **Incl_in_eq** (A:=A) (T:=set A).

Proof using. constructor. intros. set_unf. autos×. Qed.

Global Instance disjoint_eq_inst : **Disjoint_eq** (T:=set A).

Proof using.

```
constructor. intros. rewrite disjoint_eq_inter_empty.  
set_unf. applys prop_ext. iff M.  
intros x. rewrite $\leftarrow$  (@fun_eq_1 _ _ x _ _ M).  
extens $\times$ .
```

Qed.

Lemma eq_union_single_remove_one : $\forall E x,$
 $x \in E \rightarrow$
 $E = \{x\} \cup (E \setminus x).$

Proof using.

```
introv H. set_unf. extens. intros y. iff M.  
simpls. tests*: (y = x).  
destruct M. subst $\times$ . autos $\times$ .
```

Qed.

Lemma set_remove_one_add_same : $\forall E x,$
 $x \notin E \rightarrow$
 $E = (E \setminus \{x\}) \cup x.$

Proof using.

```
introv Hx. set_unf. extens. iff.  
{ split. eauto. intro. subst $\times$ . }  
{ tauto. }
```

Qed.

Lemma list_covers_of_list_repr : $\forall E L,$
list_repr E L \rightarrow
list_covers E L.

Proof using. introv (ND&EQ). introv Hx. rewrite \neg EQ. Qed.

Lemma list_repr_disjoint_union : $\forall E F LE LF,$
 $E \# F \rightarrow$
list_repr E LE \rightarrow
list_repr F LF \rightarrow
list_repr (E \cup F) (LE ++ LF).

Proof using.

```
introv D (HE&QE) (HF&QF). split.  
applys $\neg$  noduplicates_app.  
intros x ? ?. applys $\times$  @disjoint_inv x.  
typeclass. rewrite $\neg$   $\leftarrow$  QE. rewrite $\neg$   $\leftarrow$  QF.  
intros x. rewrite mem_app_eq. rewrite in_union_eq.  
rewrite  $\leftarrow$  QE. rewrite $\times$   $\leftarrow$  QF.
```

Qed.

Lemma noduplicates_of_list_repr : $\forall E xs,$

`list_repr E xs →
noduplicates xs.`

Proof using. unfold `list_repr`. `tauto`. Qed.

Lemma `ex_list_repr_impl_of_ex_list_covers_impl` : $\forall E,$
`ex (list_covers_impl E) →`
`ex (list_repr_impl E).`

Proof using.

`introv (L&M). sets_eq L1 EQL1: (remove_duplicates L).`

`forwards¬ (HN&HM&_): remove_duplicates_spec EQL1.`

`sets L2: (filter (fun x ⇒ x \in E) L1).`

$\exists L2.$ `split.`

`applys× noduplicates_filter.`

`intros x. specializes M x. rewrite ← HM in M. set_unf. iff N.`

`subst L2. forwards*: mem_filter_inv N.`

`applys× mem_filter.`

Qed.

Lemma `list_repr_to_list_of_finite` : $\forall E,$
`finite E →`
`list_repr E (to_list E).`

Proof using.

`introv FE. unfolds to_list, finite, list_repr_impl.`

`epsilon¬ L'.`

`applys¬ ex_list_repr_impl_of_ex_list_covers_impl.`

Qed.

Lemma `eq_to_list_inv` : $\forall E L,$
`L = to_list E →`
`finite E →`
`list_repr E L.`

Proof.

`introv EQ HE. unfolds. subst. forwards× (?&?): list_repr_to_list_of_finite HE.`

Qed.

Lemma `finite_eq_in_iff_mem_to_list` : $\forall E,$
`finite E = ($\forall x, x \in E \leftrightarrow \text{mem } x (\text{to_list } E)$).`

Proof.

`intros. applys prop_ext. iff M.`

$\{ \text{forwards} \times (N1 \& N2): \text{eq_to_list_inv } E. \text{intros } x. \text{specializes } N2 x. \text{autos} \times. \}$

$\{ \exists (\text{to_list } E). \text{intros } x Ex. \text{rewrite} \leftarrow M. \}$

Qed.

Lemma `to_list_empty` :
`to_list ({}:set A) = nil.`

Proof using.

```

set_unf. epsilon l.
{ ∃ (@nil A). split. { constructor. } { intros. rewrite× mem_nil_eq. } }
  intros Hl. inverts Hl. simpls. destruct¬ l. false. rewrite ← H0. simple¬.
Qed.

```

```

Lemma to_list_single : ∀ (x:A),
  to_list (\{x\}) = x::nil.

```

Proof using.

```

intros. unfold to_list. epsilon l.
{ ∃ (x::nil). split.
  { applys noduplicates_one. }
  { unfold single_inst, single_impl. simple¬.
    intros. rewrite× mem_one_eq. } }
introv Hl. unfolds single_inst, single_impl. simpls¬.
inverts Hl as H H0. destruct (H0 x). specializes¬ H2.
destruct l.
{ inverts H2. }
{ tests E: (x = a).
  { fequals. destruct l. auto. forwards~: (proj1 (H0 a0)).
    subst. inverts H as M1 M2. false× M1. }
  { inverts H2. false. forwards~: (proj1 (H0 a)). false. } }

```

Qed.

```

Lemma finite_of_list_covers : ∀ (E:set A) L,
  list_covers E L →
  finite E.

```

Proof using. introv H. ∃× L. Qed.

```

Lemma finite_of_list_repr : ∀ (E:set A) L,
  list_repr E L →
  finite E.

```

Proof using. introv (ND&EQ). ∃¬ L. introv Hx. rewrite¬ EQ. Qed.

```

Lemma finite_of_ex_list_covers : ∀ (E:set A),
  ex (list_covers E) →
  finite E.

```

Proof using. introv (L&H). applys× finite_of_list_covers. Qed.

```

Definition finite_inv_list_covers_and_card : ∀ (E:set A),
  finite E →
  ∃ L, list_covers E L ∧ card E = length L.

```

Proof.

```

introv (L&H). sets m: (card E).
forwards× (R&P): mmin_spec_nat m.

```

Qed.

Lemma finite_inv_list_covers : $\forall (E:\text{set } A)$,
 finite $E \rightarrow$
 $\exists L, \text{list_covers } E \text{ } L.$
Proof using. *introv* ($L\&HN$). $\exists L.$ *intros*. *applys* \times HN . **Qed.**

Lemma finite_empty :
 finite ($\{\} : \text{set } A$).
Proof using.
intros. *apply* finite_of_ex_list_covers. *set_unf*.
 $\exists (@\text{nil } A).$ *introv* $M.$ *inverts* $M.$
Qed.

Lemma finite_single : $\forall (a : A)$,
 finite $\{a\}.$
Proof using.
intros. *apply* finite_of_ex_list_covers. *set_unf*.
 $\exists (a::@\text{nil}).$ *introv* $M.$ *hnf in* $M.$ *subst* \times .
Qed.

Lemma finite_union : $\forall E \text{ } F,$
 finite $E \rightarrow$
 finite $F \rightarrow$
 finite $(E \cup F).$
Proof using.
introv $H1 \text{ } H2.$ *apply* finite_of_ex_list_covers.
lets ($L1\&E1$): finite_inv_list_covers $H1.$
lets ($L2\&E2$): finite_inv_list_covers $H2.$
 $\exists (L1++L2).$ *unfolds* list_covers.
introv $M.$
rewrite @in_union_eq in $M;$ *try typeclass.*
rewrite \times mem_app_eq.
Qed.

Lemma finite_inter : $\forall E \text{ } F,$
 finite $E \vee \text{finite } F \rightarrow$
 finite $(E \cap F).$
Proof using.
introv $H.$ *apply* finite_of_ex_list_covers. *destruct* $H.$
lets ($L\&EQ$): finite_inv_list_covers $H.$ $\exists L.$ *unfold* list_covers. *set_unf.* *autos* \times .
lets ($L\&EQ$): finite_inv_list_covers $H.$ $\exists L.$ *unfold* list_covers. *set_unf.* *autos* \times .
Qed.

Lemma finite_incl : $\forall E \text{ } F,$
 $E \subset F \rightarrow$
 finite $F \rightarrow$
 finite $E.$

Proof using.

```
introv HI HF. apply finite_of_ex_list_covers.  
lets (L&EQ): finite_inv_list_covers HF. unfold list_covers.  
set_unf.  $\exists x L$ . introv Ex. applys EQ. applys  $\neg$  HI.
```

Qed.

Lemma finite_remove : $\forall E F$,
finite $E \rightarrow$
finite $(E \setminus F)$.

Proof using.

```
introv HE. apply finite_of_ex_list_covers.  
lets (L&EQ): finite_inv_list_covers HE. unfold list_covers. set_unf.  $\exists x L$ .
```

Qed.

Section Finite_remove_inv.

Local Opaque remove_inst single_inst.

Lemma finite_remove_inv : $\forall E F$,
finite $(E \setminus F) \rightarrow$
finite $F \rightarrow$
finite E .

Proof using.

```
introv H1 H2. lets (L1&R1): finite_inv_list_covers H1.  
lets (L2&R2): finite_inv_list_covers H2.  
applys finite_of_list_covers (L1 ++ L2).  
intros y Hy. rewrite  $\neg$  mem_app_eq. tests:  $(y \in F)$ .  
autos  $\neg$ .  
forwards  $\neg M: R1 y$ . rewrite  $\neg$  @in_remove_eq. typeclass.
```

Qed.

End Finite_remove_inv.

Lemma finite_remove_one_inv : $\forall E x$,
finite $(E \setminus x) \rightarrow$
finite E .

Proof using.

```
introv H. applys finite_remove_inv H. applys finite_single.
```

Qed.

Lemma list_repr_nil:

```
list_repr  $\{\}$  (@nil A).
```

Proof using.

```
rewrite  $\leftarrow$  to_list_empty.  
eapply eq_to_list_inv; eauto using finite_empty.
```

Qed.

Definition list_covers_inv_card : $\forall (E:\text{set } A) L$,

`list_covers E L →
 (card E ≤ length L)%nat.`

Proof using.

`introv H. sets m: (card E). set_unf.
 forwards× (R&P): mmin_spec_nat m.
 simpls. applics P. ∃ L. splits¬.`

Qed.

Definition finite_inv_list_repr_and_card : ∀ (E:set A),
 $\text{finite } E \rightarrow$
 $\exists L, \text{list_repr } E L \wedge \text{card } E = \text{length } L.$

Proof.

`introv H. forwards (L1&HL1&EL1): finite_inv_list_covers_and_card H.
 sets L2: (remove_duplicates L1).
 forwards¬ (ND&EQ&LE): remove_duplicates_spec L1 L2.
 sets L3: (filter (fun x ⇒ x \in E) L2).
 asserts: (length L3 ≤ length L2)%nat. applics length_filter.
 asserts R3: (list_repr E L3).
 split.
 applics¬ noduplicates_filter.
 intros x. iff M.
 unfold L3 in M. lets¬ (_&?): mem_filter_inv M.
 applics¬ mem_filter. rewrite¬ EQ.
 forwards C3: list_covers_of_list_repr R3.
 ∃ L3. splits×.
 forwards: list_covers_inv_card C3. math.`

Qed.

Lemma list_repr_inv_card : ∀ (E:set A) (L:list A),
`list_repr E L →
 card E = length L.`

Proof using.

`introv HR. lets (ND&EQ): HR.
 forwards¬ (L'&(ND'&HR')&EQ'): finite_inv_list_repr_and_card E.
 applics× finite_of_list_repr.
 unfold card. simpl. rewrite EQ'.
 applics¬ noduplicates_length_eq.
 intros x. rewrite EQ. rewrite× HR'.`

Qed.

Definition finite_inv_card_ge : ∀ (E:set A) n,
 $\text{finite } E \rightarrow$
 $(\forall L, \text{list_covers } E L \rightarrow (\text{length } L ≥ n)%nat) \rightarrow$
 $(\text{card } E ≥ n)%nat.$

Proof using.

intros H . *sets* m : (*card* E).
forwards \times ($R\&P$): *mmin_spec_nat* m .
 lets ($L\&EL$): *finite_inv_list_covers* H . $\exists \neg (\text{length } L) L$.
 simpls. *intros* HL . *destruct* R as ($L\&CR\&ER$).
 forwards \sim : $HL L$. *math*.

Qed.

Definition *list_covers_inv_card_eq* : $\forall (E:\text{set } A) L$,
 list_covers $E L \rightarrow$
 $(\forall L', \text{list_covers } E L' \rightarrow (\text{length } L' \geq \text{length } L) \% \text{nat}) \rightarrow$
 card $E = \text{length } L$.

Proof using.

intros $HC HG$.
forwards \sim : *list_covers_inv_card* HC .
forwards \sim : *finite_inv_card_ge* HG .
 applys \times *finite_of_list_covers*.
 math.

Qed.

Lemma *card_eq_length_to_list* : $\forall (E:\text{set } A)$,
 finite $E \rightarrow$
 card $E = \text{length} (\text{to_list } E)$.

Proof using.

intros FE . *applys* *list_repr_inv_card*. *applys* \neg *eq_to_list_inv*.

Qed.

Global Instance *card_empty_inst* : **Card_empty** ($T := \text{set } A$).

Proof using.

constructor. *rewrite* *card_eq_length_to_list*.
lets E : *to_list_empty*. *set_unf*. *rewrite* E . *rew_list* \neg .
 applys *finite_empty*.

Qed.

Global Instance *card_single_inst* : **Card_single** ($T := \text{set } A$).

Proof using.

constructor. *intros* a . *rewrite* *card_eq_length_to_list*.
lets E : *to_list_single* a . *set_unf*. *rewrite* E . *rew_list* \neg .
 applys *finite_single*.

Qed.

End Instances.

Hint Resolve *finite_empty* *finite_single* *finite_union*
finite_inter *finite_incl* *finite_remove* : *finite*.

23.3 Tactics for proving set equalities and set inclusions

The tactic `set_prove` aims at proving set equality by testing double inclusion using a boolean tautology decision procedure.

Section Autorewrite.

Variables $(A : \text{Type})$.

Implicit Types $x y : A$.

Implicit Types $E F : \text{set } A$.

Lemma `set_in_empty_eq` : $\forall x,$
 $x \in (\{\} : \text{set } A) = \text{False}$.

Proof using. apply `in_empty_eq`. Qed.

Lemma `set_in_single_eq` : $\forall x y,$
 $x \in (\{y\} : \text{set } A) = (x = y)$.

Proof using. apply `in_single_eq`. Qed.

Lemma `set_in_inter_eq` : $\forall x E F,$
 $x \in (E \cap F) = (x \in E \wedge x \in F)$.

Proof using. apply `in_inter_eq`. Qed.

Lemma `set_in_union_eq` : $\forall x E F,$
 $x \in (E \cup F) = (x \in E \vee x \in F)$.

Proof using. apply `in_union_eq`. Qed.

Lemma `set_in_remove_eq` : $\forall x E F,$
 $x \in (E \setminus F) = (x \in E \wedge \neg x \in F)$.

Proof using. apply `in_remove_eq`. Qed.

Lemma `set_in_extens_eq` : $\forall E F,$
 $(E = F) = (\forall x, x \in E \leftrightarrow x \in F)$.

Proof using.

extens. iff M.

subst x.

applys @`in_extens_eq`. typeclass. intros. extens x.

Qed.

Lemma `set_incl_in_eq` : $\forall E F,$
 $(E \subset F) = (\forall x, x \in E \rightarrow x \in F)$.

Proof using. apply `incl_in_eq`. Qed.

Lemma `set_disjoint_eq` : $\forall E F,$
 $(E \# F) = (\forall x, x \in E \rightarrow x \in F \rightarrow \text{False})$.

Proof using. apply `disjoint_eq`. Qed.

End Autorewrite.

Hint Rewrite `in_set_st_eq` `set_in_empty_eq` `set_in_single_eq`
`set_in_inter_eq` `set_in_union_eq` `set_in_remove_eq` `set_in_extens_eq`

```

set_incl_in_eq set_disjoint_eq : rew_set.

Ltac rew_set_tactic tt :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_set).

Ltac set_specialize_hyps A x :=
  repeat match goal with H: ∀ _:?A, _ ⊢ _ ⇒
    specializes H x
  end.

Ltac set_specialize_classic x :=
  repeat match goal with E: set _ ⊢ _ ⇒
    match goal with
      | H: x \in E ∨ ¬ x \in E ⊢ _ => fail 1
      | _ => let s: (prop_inv (x \in E))
    end
  end.

Ltac set_specialize_use_classic :=
  match goal with ⊢ ∀ _:?A, _ ⇒
    let x := fresh "x" in intros x;
    set_specialize_hyps A x;
    match use_classic with
      | true => set_specialize_classic x
      | false => idtac
    end
  end.

Ltac set_prove_setup use_classic :=
  intros;
  rew_set_tactic tt;
  try set_specialize use_classic;
  rew_set_tactic tt.

Ltac set_prove_conclude :=
  solve [ intros; subst; tauto ].

Ltac set_prove :=
  set_prove_setup false; set_prove_conclude.

Ltac set_prove_classic :=
  set_prove_setup true; set_prove_conclude.

```

23.4 More properties

Lemma card_le_of_incl : ∀ A (E F:set A),
 finite F →

$E \setminus c F \rightarrow$
 $(\text{card } E \leq \text{card } F) \% nat.$

Proof using.

```
intros FF CF. lets FE: finite_incl CF FF.
lets (LF&RF&QF): finite_inv_list_covers_and_card FF.
rewrite QF. applys list_covers_inv_card. introv Ex.
applys RF. applys × @incl_inv. typeclass.
```

Qed.

Lemma card_union_le : $\forall A (E F:\text{set } A),$
 $\text{finite } E \rightarrow$
 $\text{finite } F \rightarrow$
 $\text{card } (E \setminus u F) \leq (\text{card } E + \text{card } F) \% nat.$

Proof using.

```
intros FE FF.
lets (LE&RE&QE): finite_inv_list_covers_and_card FE.
lets (LF&RF&QF): finite_inv_list_covers_and_card FF.
lets H: list_covers_inv_card (E \setminus u F) (LE++LF) _ _.
unfold list_covers. intros. apply mem_app.
rewrite in_union_eq in H. autos ×.
rew_list in H. math.
```

Qed.

Lemma card_disjoint_union : $\forall A (E F:\text{set } A),$
 $\text{finite } E \rightarrow$
 $\text{finite } F \rightarrow$
 $E \setminus \# F \rightarrow$
 $\text{card } (E \setminus u F) = (\text{card } E + \text{card } F) \% nat.$

Proof using.

```
intros FE FF EF.
forwards: card_union_le FE FF.
cuts: (card (E \setminus u F) ≥ (card E + card F) \% nat). math. clear H.
forwards (L&LC&LL): finite_inv_list_covers_and_card (E \setminus u F). applys ⊨ finite_union.
rewrite LL. clear LL.
sets PE: (fun x ⇒ x \in E). sets LE: (filter PE L).
sets PF: (fun x ⇒ x \in F). sets LF: (filter PF L).
forwards: list_covers_inv_card E LE.
unfold LE, PE. introv Ex. forwards: LC x. set_prove. applys ⊨ mem_filter.
forwards: list_covers_inv_card F LF.
unfold LF, PF. introv Fx. forwards: LC x. set_prove. applys ⊨ mem_filter.
forwards LEF: filter_length_two_disjoint PE PF L.
intros _ HEx HFx. unfolds PE, PF. applys × @disjoint_inv. typeclass.
subst LE LF. math.
```

Qed.

Lemma card_inter_le_l : $\forall A (E F:\text{set } A)$,
 $\text{finite } E \rightarrow$
 $\text{card } (E \setminus F) \leq \text{card } E.$

Proof using.

intros. applys \neg card_le_of_incl. set_prove.

Qed.

Lemma card_inter_le_r : $\forall A (E F:\text{set } A)$,
 $\text{finite } F \rightarrow$
 $\text{card } (E \setminus F) \leq \text{card } F.$

Proof using.

intros. rewrite inter_comm. applys \neg card_inter_le_l.

Qed.

Lemma card_ge_one : $\forall A (E:\text{set } A) x$,
 $x \in E \rightarrow$
 $\text{finite } E \rightarrow$
 $1\%nat \leq \text{card } E.$

Proof using.

intros.
rewrite \leftarrow (card_single x).
applys \neg card_le_of_incl.
set_prove.

Qed.

Lemma card_disjoint_union_single : $\forall A (E:\text{set } A) x$,
 $\text{finite } E \rightarrow$
 $x \notin E \rightarrow$
 $(\text{card } (E \setminus \{x\}) = \text{card } E + 1)\%nat.$

Proof using.

intros.
replace $1\%nat$ with $(\text{card } \{x\})$ by eauto using card_single.
applys \neg card_disjoint_union. applys finite_single.
rewrite disjoint_single_r_eq. auto.

Qed.

Lemma card_diff_single : $\forall A (E:\text{set } A) x$,
 $\text{finite } E \rightarrow$
 $x \in E \rightarrow$
 $(\text{card } (E \setminus x) = \text{card } E - 1)\%nat.$

Proof using.

intros.
assert (h1: $(E \setminus x) \cup \{x\} = E$).
{ rewrite union_comm. rewrite eq_union_single_remove_one by eauto. reflexivity. }
forwards h2: card_disjoint_union_single $(E \setminus x) x$.

```

{ eauto with finite. }
{ unfold notin. rewrite set_in_remove_eq.
  rew_logic. right.
  eapply in_single_self. }
rewrite h1 in h2.
math.

Qed.

```

Lemma fold_eq_fold_to_list : $\forall A B (m:\mathbf{monoid_op} B) (f:A \rightarrow B) (E: \text{set } A)$,
 $\text{fold } m f E = \text{LibList.fold } m f (\text{to_list } E)$.

Proof using. reflexivity. Qed.

Lemma fold_eq_fold_list_repr : $\forall A B (m:\mathbf{monoid_op} B) (f:A \rightarrow B) (E: \text{set } A) L$,
 $\mathbf{Comm_monoid} m \rightarrow$
 $\text{list_repr } E L \rightarrow$
 $\text{fold } m f E = \text{LibList.fold } m f L$.

Proof using.

```

introsv HM EL. rewrite fold_eq_fold_to_list.
forwards  $\neg (N \& EQ2)$ : eq_to_list_inv E. applys  $\times$  finite_of_list_repr.
destruct EL as (ND & EQ1).
applys  $\neg$  LibList.fold_equiv. intros. rewrite EQ2. rewrite  $\times$  EQ1.

```

Qed.

Lemma fold_induction:

```

 $\forall A B (m : \mathbf{monoid\_op} B) (f : A \rightarrow B) (P : B \rightarrow \text{Prop})$ ,
 $\mathbf{Comm\_monoid} m \rightarrow$ 
 $P (\mathbf{monoid\_neutral} m) \rightarrow$ 
 $(\forall x a, P x \rightarrow P (\mathbf{monoid\_oper} m (f a) x)) \rightarrow$ 
 $\forall E,$ 
 $\text{finite } E \rightarrow$ 
 $P (\text{fold } m f E)$ .

```

Proof using. introv Hm Hbase Hstep Hfinite.

```

assert ( $\forall xs, P (\text{LibList.fold } m f xs)$ ).
{ induction xs; rew_listx; eauto. }
forwards: list_repr_to_list_of_finite Hfinite.
erewrite fold_eq_fold_list_repr by eauto.
eauto.

```

Qed.

Lemma fold_congruence : $\forall A B (m:\mathbf{monoid_op} B) (f g:A \rightarrow B) (E:\text{set } A)$,
 $\mathbf{Comm_monoid} m \rightarrow$
 $\text{finite } E \rightarrow$
 $(\forall x, x \setminus \text{in } E \rightarrow f x = g x) \rightarrow$
 $\text{fold } m f E = \text{fold } m g E$.

Proof using. introv Hm HE h. do 2 rewrite fold_eq_fold_to_list.

```

eapply LibList.fold_congruence. intros.
eapply h. rewrite finite_eq_in_iff_mem_to_list in HE. rewrite× HE.
Qed.

```

```

Lemma fold_empty : ∀ A B (m:monoid_op B) (f:A→B),
fold m f ({}:set A) = monoid_neutral m.

```

Proof using.

```

intros. rewrite fold_eq_fold_to_list.
rewrite to_list_empty. rewrite¬ LibList.fold_nil.

```

Qed.

```

Lemma fold_single : ∀ A B (m:monoid_op B) (f:A→B) (x:A),
Monoid m →
fold m f {x} = f x.

```

Proof using.

```

intros. rewrite fold_eq_fold_to_list.
rewrite to_list_single. rewrite¬ fold_cons.
rewrite fold_nil. rewrite¬ monoid_neutral_r.

```

Qed.

```

Lemma fold_union : ∀ A B (m:monoid_op B) (f:A→B) (E F:set A),
Comm_monoid m →
finite E →
finite F →
E \# F →
fold m f (E ∪ F) = monoid_oper m (fold m f E) (fold m f F).

```

Proof using.

```

introv HM HE HF HD.
rewrites (» fold_eq_fold_to_list E).
rewrites (» fold_eq_fold_to_list F).
hint list_repr_to_list_of_finite.
forwards¬ HR: list_repr_disjoint_union HD.
rewrites¬ (» fold_eq_fold_list_repr HR).
rewrite¬ LibList.fold_app. typeclass.

```

Qed.

```

Lemma fold_isolate : ∀ A (E:set A) x,
finite E →
x \in E →
∀ B (m : monoid_op B),
Comm_monoid m →
∀ (f : A → B),
fold m f E = monoid_oper m (f x) (fold m f (E \- {x})).

```

Proof using. intros.

```

rewrite (eq_union_single_remove_one E x) at 1 by eauto.

```

```

erewrite ← (fold_single f x) by typeclass.
eapply fold_union; eauto using remove_disjoint with finite.
Qed.

```

23.4.1 Structural properties

Rewriting tactics *rew_set*

```

Hint Rewrite in_set_st_eq : rew_set.

Tactic Notation "rew_set" :=
  autorewrite with rew_set.

Tactic Notation "rew_set" "in" hyp(H) :=
  autorewrite with rew_set in H.

Tactic Notation "rew_set" "in" "*" :=
  autorewrite with rew_set in *.

```

23.5 MORE

23.5.1 TEMPORARY Foreach

TODO: these lemmas should be instead derived as typeclasses in a generic way, in LibContainer.

TODO: add a paragraphe of the definition: foreach P E = (forall x, x \in E -> P x)

Section ForeachProp.

Variables (A : Type).

Implicit Types P Q : A → Prop.

Implicit Types E F : set A.

Lemma foreach_empty : ∀ P,

 @foreach A (set A) _ P \{ \}.

Proof using. intros_all. false. Qed.

Lemma foreach_single : ∀ P X,

 P X →

 @foreach A (set A) _ P (\{ X \}).

Proof using. intros_all. rewrite in_single_eq in H0. subst×. Qed.

Lemma foreach_union : ∀ P E F,

 foreach P E →

 foreach P F →

 foreach P (E \u F).

Proof using. intros_all. destruct¬ (in_union_inv H1). Qed.

Lemma foreach_union_inv : ∀ P E F,

 foreach P (E \u F) →

foreach $P E \wedge$ foreach $P F$.

Proof using.

```
introv H. split; introv K.  
apply H. rewrite¬ @in_union_eq. typeclass.  
apply H. rewrite¬ @in_union_eq. typeclass.
```

Qed.

Lemma foreach_union_eq : $\forall P E F,$
 $\text{foreach } P (E \cup F) = (\text{foreach } P E \wedge \text{foreach } P F)$.

Proof using.

```
intros. extens. iff.  
apply¬ foreach_union_inv.  
intuition eauto using foreach_union.
```

Qed.

Lemma foreach_single_eq : $\forall P X,$
 $\text{foreach } P (\set{X}:set A) = P X$.

Proof using.

```
intros. extens. iff.  
apply H. apply in_single_self.  
apply¬ foreach_single.
```

Qed.

Lemma foreach_of_pred_incl: $\forall P Q E,$
 $\text{foreach } P E \rightarrow$
 $\text{pred_incl } P Q \rightarrow$
 $\text{foreach } Q E$.

Proof using. introv H L K. apply¬ L. Qed.

Lemma foreach_remove_of_foreach_all : $\forall P E F,$
 $\text{foreach } P E \rightarrow$
 $\text{foreach } P (E \setminus F)$.

Proof using. introv M H. applys M. rewrite in_remove_eq in H. autos×. Qed.

Lemma foreach_remove : $\forall P E F,$
 $(\forall x, x \in E \rightarrow x \notin F \rightarrow P x) \rightarrow$
 $\text{foreach } P (E \setminus F)$.

Proof using. introv M Px. rewrite in_remove_eq in Px. applys× M. Qed.

Lemma notin_of_foreach_not : $\forall P x E,$
 $\text{foreach } P E \rightarrow$
 $\neg P x \rightarrow$
 $x \notin E$.

Proof using. introv M N I. applys N. applys¬ M. Qed.

End FforeachProp.

Hint Resolve foreach_empty foreach_single foreach_union.

```

Hint Rewrite foreach_union_eq foreach_single_eq : rew_FOREACH.

Tactic Notation "rew_FOREACH" :=
  autorewrite with rew_FOREACH.

Tactic Notation "rew_FOREACH" "in" hyp(H) :=
  autorewrite with rew_FOREACH in H.

Tactic Notation "rew_FOREACH" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_FOREACH).

Tactic Notation "rew_FOREACH" "~" :=
  rew_FOREACH; auto_tilde.

Tactic Notation "rew_FOREACH" "*" :=
  rew_FOREACH; auto_star.

Tactic Notation "rew_FOREACH" "~" "in" constr(H) :=
  rew_FOREACH in H; auto_tilde.

Tactic Notation "rew_FOREACH" "*" "in" constr(H) :=
  rew_FOREACH in H; auto_star.

Tactic Notation "rew_FOREACH" "~" "in" "*" :=
  rew_FOREACH in *; auto_tilde.

Tactic Notation "rew_FOREACH" "*" "in" "*" :=
  rew_FOREACH in *; auto_star.

```

Chapter 24

Library SLF.LibChoice

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibEpsilon LibRelation.

Generalizable Variables A B .

This files includes several versions of the axiom of choice. This “axiom” is actually proved in terms of indefinite description. Remark: the choice results contained in this file are not very useful in practice since it is usually more convenient to use the epsilon operator directly.

24.1 Functional choice

24.1.1 Functional choice

This result can be used to build a function from a relation that maps every input to at least one output.

```
Lemma functional_choice : ∀ A B (R:A→B→Prop),  
  (∀ x, ∃ y, R x y) →  
  (∃ f, ∀ x, R x (f x)).
```

Proof using.

```
  intros. ∃ (fun x ⇒ sig_val (indefinite_description (H x))).  
  intro x. apply (sig_proof (indefinite_description (H x))).
```

Qed.

24.1.2 Dependent functional choice

It is a generalization of functional choice to dependent functions.

```
Scheme and_indd := Induction for and Sort Prop.  
Scheme eq_indd := Induction for eq Sort Prop.
```

Lemma dependent_functional_choice :

$$\begin{aligned} \forall A \ (B:\text{Type}) \ (R:\forall x, B x \rightarrow \text{Prop}), \\ (\forall x, \exists y, R x y) \rightarrow \\ (\exists f, \forall x, R x (f x)). \end{aligned}$$

Proof using.

```

intros H.
pose (B' := { x:A & B x }).
pose (R' := fun (x:A) (y:B') => projT1 y = x ∧ R (projT1 y) (projT2 y)).
destruct (functional_choice R') as (f,Hf).
intros x. destruct (H x) as (y,Hy).
exists (existT (fun x => B x) x y). split.
sets proj1_transparent: (fun P Q (p:P ∧ Q) => let (a,b) := p in a).
exists (fun x => eq_rect _ _ (projT2 (f x)) _ (proj1_transparent _ _ (Hf x))).
intros x. destruct (Hf x) as (Heq,HR) using and_indd.
destruct (f x). simpls. destruct Heq using eq_indd. apply HR.

```

Qed.

Arguments dependent_functional_choice [A] [B].

24.1.3 Guarded functional choice

Similar to functional choice, except that it targets partial functions

$$\begin{aligned} \text{Lemma guarded_functional_choice : } \forall A \ ' \{ \text{Inhab } B \} \ (P : A \rightarrow \text{Prop}) \ (R : A \rightarrow B \rightarrow \text{Prop}), \\ (\forall x, P x \rightarrow \exists y, R x y) \rightarrow \\ (\exists f, \forall x, P x \rightarrow R x (f x)). \end{aligned}$$

Proof using.

```

intros. apply (functional_choice (fun x y => P x → R x y)).
intros. applys × indep_general_premises.

```

Qed.

24.1.4 Omniscient functional choice

Similar to functional choice except that the proof of functionality of the relation is given after the fact, for each argument.

$$\begin{aligned} \text{Lemma omniscient_functional_choice : } \forall A \ ' \{ \text{Inhab } B \} \ (R : A \rightarrow B \rightarrow \text{Prop}), \\ \exists f, \forall x, (\exists y, R x y) \rightarrow R x (f x). \end{aligned}$$

Proof using. intros. applys guarded_functional_choice. Qed.

24.2 Functional unique choice

This section provides a similar set of results excepts that it is specialized for the case where each argument has a unique image.

24.2.1 Functional unique choice

`Lemma functional_unique_choice : ∀ A B (R:A→B→Prop),
 (∀ x , ∃! y , R x y) →
 (∃! f , ∀ x , R x (f x)).`

Proof using.

```
intros. destruct (functional_choice R) as [f Hf].  
intros. apply (ex_of_ex_unique (H x)).  
exists f. split. auto.  
intros g Hg. apply fun_ext_1. intros y.  
apply¬ (at_most_one_of_ex_unique (H y)).
```

Qed.

24.2.2 Dependent functional unique choice

`Theorem dependent_functional_unique_choice :
 ∀ (A:Type) (B:A→Type) (R:∀ (x:A), B x → Prop),
 (∀ (x:A), ∃! y : B x , R x y) →
 (∃! f : (∀ (x:A), B x) , ∀ (x:A), R x (f x)).`

Proof using.

```
intros. destruct (dependent_functional_choice R) as [f Hf].  
intros. apply (ex_of_ex_unique (H x)).  
exists f. split. auto.  
intros g Hg. extens. intros y.  
apply¬ (at_most_one_of_ex_unique (H y)).
```

Qed.

Arguments dependent_functional_unique_choice [A] [B].

24.2.3 Guarded functional unique choice

`Lemma guarded_functional_unique_choice :
 ∀ A ‘{Inhab B} (P : A→Prop) (R : A→B→Prop),
 (∀ x, P x → ∃! y , R x y) →
 (∃! f , ∀ x, P x → R x (f x)).`

Proof using.

```
introtv I M. apply (functional_choice (fun x y ⇒ P x → R x y)).  
intros. apply× indep_general_premises.  
introtv H. destruct× (M _ H) as (y&Hy&_).  
Qed.
```

24.2.4 Omniscient functional unique choice

Lemma omniscient_functional_unique_choice :

$$\begin{aligned} & \forall A \{Inhab\ B\} (R : A \rightarrow B \rightarrow \text{Prop}), \\ & \exists f, \forall x, (\exists! y, R x y) \rightarrow R x (f x). \end{aligned}$$

Proof using.

```
intros. destruct (omniscient_functional_choice R) as [f F].
exists f. introv (y&Hy&Uy). autos ×.
```

Qed.

24.3 Relational choice

Relational choice can be used to extract from a relation a subrelation that describes a function, by mapping every argument to a unique image.

24.3.1 Relational choice

Lemma rel_choice : $\forall A B (R : A \rightarrow B \rightarrow \text{Prop}),$
 $(\forall x, \exists y, R x y) \rightarrow$
 $(\exists R', \text{rel_incl } R' R$
 $\wedge \forall x, \exists! y, R' x y).$

Proof using.

```
introv H. destruct (functional_choice R) as [f Hf].
exists f. introv E. simpls. subst ⊥.
intros x. exists (f x).
```

Qed.

24.3.2 Guarded relational choice

Lemma guarded_rel_choice : $\forall A B (P : A \rightarrow \text{Prop}) (R : A \rightarrow B \rightarrow \text{Prop}),$
 $(\forall x, P x \rightarrow \exists y, R x y) \rightarrow$
 $(\exists R', \text{rel_incl } R' R$
 $\wedge \forall x, P x \rightarrow \exists! y, R' x y).$

Proof using.

```
intros. destruct (rel_choice (fun (x:sig P) (y:B) => R (sig_val x y))
as (R',(HR'R,M)).
intros (x,HPx). destruct (H _ HPx) as (y,HRxy). exists y.
set (R'') := fun (x:A) (y:B) => ∃ (H : P x), R' (exist P x H) y.
exists R''. split.
intros x y (HPx,HR'xy). apply (HR'R _ _ HR'xy).
```

```

intros x HPx. destruct (M (exist P x HPx)) as (y,(HR'xy,Uniq)).
   $\exists y.$  split.
     $\exists \neg HPx.$ 
      intros y' (H'Px,HR'xy'). apply Uniq.
      rewrite $\neg$  (proof_irrelevance HPx H'Px).
Qed.

```

24.3.3 Omniscient relation choice

Lemma omniscient_rel_choice : $\forall A B (R : A \rightarrow B \rightarrow \text{Prop}),$
 $\exists R', \text{rel_incl } R' R$
 $\wedge \forall x, (\exists y, R x y) \rightarrow (\exists! y, R' x y).$
Proof using. intros. apply \neg guarded_rel_choice. Qed.

Chapter 25

Library SLF.LibUnit

```
Set Implicit Arguments.
```

```
From SLF Require Import LibTactics LibLogic LibReflect.
```

25.1 Unit type

25.1.1 Definition

```
From the Prelude.
```

```
Inductive unit : Type := | tt : unit.
```

```
Add Printing If bool. Delimit Scope bool_scope with bool.
```

25.1.2 Inhabited

```
Instance Inhab_unit : Inhab unit.
```

```
Proof using. intros. apply (Inhab_of_val tt). Qed.
```

25.2 Properties

25.2.1 Uniqueness

```
Lemma unit_eq : ∀ (tt1 tt2 : unit),  
  tt1 = tt2.
```

```
Proof using. intros. destruct tt1. destruct tt2. Qed.
```

Chapter 26

Library SLF.LibFun

DEPRECATED? \o notation for composition is used in LibFixDemos

26.1

26.2

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibContainer LibSet.

From *SLF* Require LibList.

Generalizable Variables A .

26.2.1 Identity function

```
Definition id {A} (x : A) :=  
  x.
```

```
Definition const {A B} (v : B) : A → B :=  
  fun _ => v.
```

```
Definition const1 :=  
  @const.
```

```
Definition const2 {A1 A2 B} (v:B) : A1 → A2 → B :=  
  fun _ _ => v.
```

```
Definition const3 {A1 A2 A3 B} (v:B) : A1 → A2 → A3 → B :=  
  fun _ _ _ => v.
```

```
Definition const4 {A1 A2 A3 A4 B} (v:B) : A1 → A2 → A3 → A4 → B :=  
  fun _ _ _ _ => v.
```

```
Definition const5 {A1 A2 A3 A4 A5 B} (v:B) : A1 → A2 → A3 → A4 → A5 → B :=
```

```

fun _ _ _ _  $\Rightarrow$  v.

Definition apply {A B} (f : A  $\rightarrow$  B) (x : A) :=
  f x.

Definition apply_to (A : Type) (x : A) (B : Type) (f : A  $\rightarrow$  B) :=
  f x.

Definition compose {A B C} (g : B  $\rightarrow$  C) (f : A  $\rightarrow$  B) :=
  fun x  $\Rightarrow$  g (f x).

Declare Scope fun_scope.

Notation "f1  $\circ$  f2" := (compose f1 f2)
  (at level 49, right associativity) : fun_scope.

Section Combinators.

Open Scope fun_scope.

Variables (A B C D : Type).

Lemma compose_id_l :  $\forall (f:A \rightarrow B),$ 
  id  $\circ$  f = f.
Proof using. intros. apply fun_ext_1. Qed.

Lemma compose_id_r :  $\forall (f:A \rightarrow B),$ 
  f  $\circ$  id = f.
Proof using. intros. apply fun_ext_1. Qed.

Lemma compose_assoc :  $\forall (f:C \rightarrow D) (g:B \rightarrow C) (h:A \rightarrow B),$ 
  (f  $\circ$  g)  $\circ$  h = f  $\circ$  (g  $\circ$  h).
Proof using. intros. apply fun_ext_1. Qed.

Lemma compose_eq_l :  $\forall (f:B \rightarrow C) (g1\ g2:A \rightarrow B),$ 
  g1 = g2  $\rightarrow$ 
  f  $\circ$  g1 = f  $\circ$  g2.
Proof using. intros. subst. Qed.

Lemma compose_eq_r :  $\forall (f:A \rightarrow B) (g1\ g2:B \rightarrow C),$ 
  g1 = g2  $\rightarrow$ 
  g1  $\circ$  f = g2  $\circ$  f.
Proof using. intros. subst. Qed.

Composition of LibList.map behaves well. Import LibList.

Lemma list_map_compose :  $\forall A B C (f : A \rightarrow B) (g : B \rightarrow C) l,$ 
  LibList.map g (LibList.map f l) = LibList.map (g  $\circ$  f) l.
Proof using.
  introv. induction l.
  reflexivity.
  rew_listx. fequals.
Qed.

```

End Combinators.

Tactic for simplifying function compositions

```
Hint Rewrite compose_id_l compose_id_r compose_assoc : rew_compose.
Tactic Notation "rew_compose" :=
  autorewrite with rew_compose.
Tactic Notation "rew_compose" "in" "*" :=
  autorewrite with rew_compose in *.
Tactic Notation "rew_compose" "in" hyp(H) :=
  autorewrite with rew_compose in H.
```

26.2.2 Function update

fupdate f a b x is like f except that it returns b for input a

```
Definition fupdate A B (f : A → B) (a : A) (b : B) : A → B :=
  fun x => If (x = a) then b else f x.
```

```
Lemma fupdate_eq : ∀ A B (f:A→B) a b x,
  fupdate f a b x = If (x = a) then b else f x.
```

Proof using. auto. Qed.

```
Lemma fupdate_same : ∀ A B (f:A→B) a b,
  fupdate f a b a = b.
```

Proof using. intros. unfold fupdate. case_if×. Qed.

```
Lemma fupdate_neq : ∀ A B (f:A→B) a b x,
  x ≠ a →
  fupdate f a b x = f x.
```

Proof using. intros. unfold fupdate. case_if×. Qed.

26.2.3 Function image

Section FunctionImage.

Open Scope set_scope.

Import LibList.

```
Definition image A B (f : A → B) (E : set A) : set B :=
  \set{y | exists_ x \in E, y = f x}.
```

```
Lemma in_image_prove_eq : ∀ A B x (f : A → B) (E : set A),
  x \in E → f x \in image f E.
```

Proof using. introv N. unfold image. rew_set. ∃× x. Qed.

```
Lemma in_image_prove : ∀ A B x y (f : A → B) (E : set A),
  x \in E → y = f x → y \in image f E.
```

Proof using. intros. subst. applys× in_image_prove_eq. Qed.

Lemma `in_image_inv` : $\forall A B y (f : A \rightarrow B) (E : \text{set } A)$,

$y \in \text{image } f E \rightarrow \exists x, x \in E \wedge y = f x.$

Proof using. `introv N. unfolds image. rew_set in N. auto.` Qed.

Lemma `finite_image` : $\forall A B (f : A \rightarrow B) (E : \text{set } A)$,

$\text{finite } E \rightarrow$

$\text{finite } (\text{image } f E).$

Proof using.

`introv M. lets (L&H): finite_inv_list_covers M.`

`applys finite_of_list_covers (LibList.map f L). introv N.`

`lets (y&Hy&Ey): in_image_inv (rm N). subst x. applys× mem_map.`

Qed.

Lemma `image_covariant` : $\forall A B (f : A \rightarrow B) (E F : \text{set } A)$,

$E \subset F \rightarrow$

$\text{image } f E \subset \text{image } f F.$

Proof using.

`introv. do 2 rewrite incl_in_eq. introv M N.`

`lets (y&Hy&Ey): in_image_inv (rm N). applys× in_image_prove.`

Qed.

Lemma `image_union` : $\forall A B (f : A \rightarrow B) (E F : \text{set } A)$,

$\text{image } f (E \cup F) = \text{image } f E \cup \text{image } f F.$

Proof using.

`Hint Resolve in_image_prove.`

`introv. apply in_extens. intros x. iff N.`

`lets (y&Hy&Ey): in_image_inv (rm N). rewrite in_union_eq in Hy.`

`rewrite in_union_eq. destruct× Hy.`

`rewrite in_union_eq in N. destruct N as [N|N].`

`lets (y&Hy&Ey): in_image_inv (rm N). applys× in_image_prove.`

`rewrite in_union_eq. eauto.`

`lets (y&Hy&Ey): in_image_inv (rm N). applys× in_image_prove.`

`rewrite in_union_eq. eauto.`

Qed.

Lemma `image_singleton` : $\forall A B (f : A \rightarrow B) (x : A)$,

$\text{image } f \setminus \{x\} = \setminus \{f x\}.$

Proof using.

`intros. apply in_extens. intros z. rewrite in_single_eq. iff N.`

`lets (y&Hy&Ey): in_image_inv (rm N). rewrite in_single_eq in Hy. subst¬.`

`applys× in_image_prove. rewrite¬ @in_single_eq. typeclass.`

Qed.

End FunctionImage.

Hint Resolve finite_image : finite.

26.2.4 Function preimage

Section FunctionPreimage.

Open Scope set_scope.

Definition preimage $A B (f : A \rightarrow B) (E : \text{set } B) : \text{set } A := \set{x \mid \exists y \in E, y = f x}$.

End FunctionPreimage.

26.2.5 Function iteration

Fixpoint applyn $A n (f : A \rightarrow A) x :=$
 $\text{match } n \text{ with}$
 $| 0 \Rightarrow x$
 $| S n' \Rightarrow f (\text{applyn } n' f x)$
 end.

Lemma applyn_fix : $\forall A n f (x : A),$
 $\text{applyn } (S n) f x = \text{applyn } n f (f x).$

Proof using. introv. induction n. simpls. rewrite IHn. Qed.

Lemma applyn_comp : $\forall A n m f (x : A),$
 $\text{applyn } n f (\text{applyn } m f x) = \text{applyn } (n + m) f x.$

Proof using.

introv. gen m; induction n; introv; simpls.
 $\text{rewrite } IHn.$

Qed.

Lemma applyn_nested : $\forall A n m f (x : A),$
 $\text{applyn } n (\text{applyn } m f) x = \text{applyn } (n \times m) f x.$

Proof using.

introv. gen m. induction n; introv; simpls.
 $\text{rewrite } IHn. \text{ rewrite } \text{applyn_comp}.$

Qed.

Lemma applyn_altern : $\forall A B (f : A \rightarrow B) (g : B \rightarrow A) x n,$
 $\text{applyn } n (\text{fun } x \Rightarrow f (g x)) (f x) =$
 $f (\text{applyn } n (\text{fun } x \Rightarrow g (f x)) x).$

Proof using. introv. gen x. induction n. introv. repeat rewrite applyn_fix. autos.
 $\text{repeat } \text{rewrite } \text{applyn_fix}.$

Qed.

Lemma applyn_ind : $\forall A (P : A \rightarrow \text{Prop}) (f : A \rightarrow A) x n,$
 $(\forall x, P x \rightarrow P (f x)) \rightarrow$
 $P x \rightarrow$
 $P (\text{applyn } n f x).$

Proof using. introv I. induction n; introv Hx; autos.
 $\text{repeat } \text{rewrite } \text{applyn_fix}.$ Qed.

Chapter 27

Library **SLF.LibString**

```
Set Implicit Arguments.  
From SLF Require Import LibTactics LibReflect.  
Require Export Coq.Strings.String.
```

27.1 Inhabited

```
Instance Inhab_string : Inhab string.  
Proof using apply (Inhab_of_val EmptyString). Qed.
```

Chapter 28

Library `SLF.LibMultiset`

DISCLAIMER: this file is under construction and still contains a couple admits.

28.1

28.2

Set Implicit Arguments.

Generalizable Variables $A\ B$.

From *SLF* Require Import LibTactics LibLogic LibReflect

LibRelation LibList LibInt LibNat LibOperation

LibEpsilon LibSet LibMonoid.

From *SLF* Require Export LibContainer.

Open Scope nat_scope .

28.3 Construction of sets as predicates

28.3.1 Basic definitions

Definition multiset ($A : \text{Type}$) := $A \rightarrow \text{nat}$.

Section Operations.

Variables ($A\ B : \text{Type}$).

Implicit Types $x : A$.

Implicit Types $E\ F\ G : \text{multiset}\ A$.

Definition empty_impl : multiset $A := \text{fun } _ \Rightarrow 0$.

Definition single_impl $x := \text{fun } y \Rightarrow \text{If } x = y \text{ then } 1 \text{ else } 0$.

```

Definition in_implementation x E := E x > 0.

Definition union_implementation E F := fun x => (E x + F x)%nat.

Definition incl_implementation E F := ∀ x, E x ≤ F x.

Definition dom_implementation E := set_st (fun x => E x > 0).

Definition list_repr_implementation (E:multiset A) (l:list (A×nat)) :=
  noduplicates (LibList.map (@fst _ _) l)
  ∧ ∀ n x, mem (x,n) l ↔ (n = E x ∧ n > 0).

Definition to_list_implementation (E:multiset A) := epsilon (list_repr_implementation E).

Definition fold_implementation (m:monoid_op B) (f:A→nat→B) (E:multiset A) :=
  LibList.fold_right (fun p acc => let (x,n) := p : A×nat in monoid_oper m (f x n) acc)
  (monoid_neutral m) (to_list_implementation E).

End Operations.

Definition card_implementation A (E:multiset A) :=
  fold_implementation (monoid_make plus 0) (fun _ n => n) E.

```

28.3.2 Notation through typeclasses

Lemma in_inst : ∀ A, **BagIn** A (multiset A).
 Proof using. constructor. exact (@in_implementation A). Defined.

Hint Extern 1 (**BagIn** _ (multiset _)) ⇒ apply in_inst
 : typeclass_instances.

Instance empty_inst : ∀ A, **BagEmpty** (multiset A).
 constructor. rapply (@empty_implementation A). Defined.

Instance single_inst : ∀ A, **BagSingle** A (multiset A) .
 constructor. rapply (@single_implementation A). Defined.

Instance union_inst : ∀ A, **BagUnion** (multiset A).
 constructor. rapply (@union_implementation A). Defined.

Instance incl_inst : ∀ A, **BagIncl** (multiset A).
 constructor. rapply (@incl_implementation A). Defined.

Instance fold_inst : ∀ A B, **BagFold** B (A→nat→B) (multiset A).
 constructor. rapply (@fold_implementation A B). Defined.

Instance card_inst : ∀ A, **BagCard** (multiset A).
 constructor. rapply (@card_implementation A). Defined.

Global Opaque multiset empty_inst single_inst in_inst
 union_inst incl_inst fold_inst card_inst.

Instance Inhab_multiset : ∀ A, **Inhab** (multiset A).
 Proof using. intros. apply (Inhab_of_val (@empty_implementation A)). Qed.

28.4 Properties of multisets

28.4.1 Structural properties

Section Instances.

Variables ($A:\text{Type}$).

Hint Extern 1 **False** \Rightarrow *math*.

Hint Extern 1 $(_ > _)$ \Rightarrow solve [*math* | *false*].

Hint Extern 1 $(_ = _)$ \Rightarrow *math*.

Transparent multiset empty_inst single_inst in_inst
union_inst incl_inst fold_inst card_inst.

Global Instance in_empty_eq_inst : **In_empty_eq** ($A:=A$) ($T:=\text{multiset } A$).

Proof using.

```
constructor. intros. extens. simpl.  
unfold emptyImpl, inImpl. autos×.
```

Qed.

Global Instance in_single_eq_inst : **In_single_eq** ($A:=A$) ($T:=\text{multiset } A$).

Proof using.

```
constructor. intros. extens. simpl.  
unfold singleImpl, inImpl. case_if×.
```

Qed.

Global Instance in_union_eq_inst : **In_union_eq** ($A:=A$) ($T:=\text{multiset } A$).

Proof using.

```
constructor. intros. extens. simpl.  
unfold singleImpl, unionImpl, inImpl.  
iff. tests: (E x = 0). right¬. left¬. destruct H; math.
```

Qed.

Global Instance incl_inv_inst : **Incl_inv** ($A:=A$) ($T:=\text{multiset } A$).

Proof using.

```
constructor. introv x. introv M N.  
unfold inclInst, inclImpl, incl, inInst, inImpl, isIn in *.  
specializes M x. math.
```

Qed.

Global Instance union_empty_inv_inst : **Union_empty_inv** ($T:=\text{multiset } A$).

Proof using.

```
constructor. introv.  
unfold unionInst, unionImpl, union, emptyImpl,  
emptyInst, empty, emptyImpl, multiset. introv N.  
split; extens¬.  
intros x. lets: funEq_1 x N. math.  
intros x. lets: funEq_1 x N. math.
```

Qed.

Global Instance union_empty_l_eq_inst : **Union_empty_l** ($T:=\text{multiset } A$).

Proof using.

constructor. *intros_all simpl*.

unfold unionImpl, emptyImpl, multiset. *simpl extens¬*.

Qed.

Global Instance union_comm_eq_inst : **Union_comm** ($T:=\text{multiset } A$).

Proof using.

constructor. *intros_all simpl*.

unfold unionImpl, multiset. *simpl extens¬*.

Qed.

Global Instance union_assoc_eq_inst : **Union_assoc** ($T:=\text{multiset } A$).

Proof using.

constructor. *intros_all simpl*.

unfold unionImpl, multiset. *simpl extens¬*.

Qed.

Global Instance empty_incl_inst : **Empty_incl** ($T:=\text{multiset } A$).

Proof using.

constructor. *intros_all simpl*.

unfold emptyImpl, multiset. *math*.

Qed.

Global Instance card_empty_inst : **Card_empty** ($T:=\text{multiset } A$).

Proof using. *admit. Admitted*.

Global Instance card_single_inst : **Card_single** ($A:=A$) ($T:=\text{multiset } A$).

Proof using. *admit. Admitted*.

Global Instance card_union_inst : **Card_union** ($T:=\text{multiset } A$).

Proof using. *admit. Admitted*.

End Instances.

28.5 Additional predicates

28.5.1 Foreach

TODO: foreach properties should be shared in LibContainer

Section ForeachProp.

Variables ($A : \text{Type}$).

Implicit Types $P Q : A \rightarrow \text{Prop}$.

Implicit Types $E F : \text{multiset } A$.

Lemma foreach_empty : $\forall P,$

$\text{@foreach } A \text{ (multiset } A) \subseteq P \setminus \{\}.$

Proof using. *intros_all*. *rewrite in_empty_eq in H*. *false*. Qed.

Lemma *foreach_single* : $\forall P X,$

$P X \rightarrow$

$\text{@foreach } A \text{ (multiset } A) \subseteq P (\setminus\{X\}).$

Proof using. *intros_all*. *rewrite in_single_eq in H0*. *subst* \times . Qed.

Lemma *foreach_union* : $\forall P E F,$

foreach P E \rightarrow *foreach P F* \rightarrow *foreach P (E \cup F)*.

Proof using. *intros_all*. *destruct* \neg (*in_union_inv H1*). Qed.

Hint Resolve *foreach_empty foreach_single foreach_union*.

Lemma *foreach_union_inv* : $\forall P E F,$

foreach P (E \cup F) \rightarrow

foreach P E \wedge *foreach P F*.

Proof using.

introv H. *split*; *introv K*.

apply H. *rewrite* \neg *@in_union_eq*. *typeclass*.

apply H. *rewrite* \neg *@in_union_eq*. *typeclass*.

Qed.

Lemma *foreach_union_eq* : $\forall P E F,$

foreach P (E \cup F) $=$ (*foreach P E* \wedge *foreach P F*).

Proof using.

intros. *extens iff*.

apply \neg *foreach_union_inv*. *apply* \times *foreach_union*.

Qed.

Lemma *foreach_single_eq* : $\forall P X,$

foreach P (\setminus\{X\}:multiset A) $=$ *P X*.

Proof using.

intros. *extens iff*.

apply H. *apply in_single_self*.

apply \neg *foreach_single*.

Qed.

Lemma *foreach_weaken* : $\forall P Q E,$

foreach P E \rightarrow

pred_incl P Q \rightarrow

foreach Q E.

Proof using. *introv H L K*. *apply* \neg *L*. Qed.

End *ForeachProp*.

28.6 Tactics

EXPERIMENTAL tactics – TODO: add documentation

28.6.1 Tactics to prove equalities on unions

Lemma for_multiset_union_assoc : $\forall A, \text{assoc}(\text{union}(T:=\text{multiset } A))$.

Proof using. intros. apply union_assoc. Qed.

Lemma for_multiset_union_comm : $\forall A, \text{comm}(\text{union}(T:=\text{multiset } A))$.

Proof using. intros. apply union_comm. Qed.

Lemma for_multiset_union_empty_l : $\forall A (E:\text{multiset } A), \{\} \setminus E = E$.

Proof using. intros. apply union_empty_l. Qed.

Lemma for_multiset_union_empty_r : $\forall A (E:\text{multiset } A), E \setminus \{\} = E$.

Proof using. intros. apply union_empty_r. Qed.

Lemma for_multiset_empty_incl : $\forall A (E:\text{multiset } A), \{\} \subset E$.

Proof using. intros. apply empty_incl. Qed.

Hint Rewrite \leftarrow for_multiset_union_assoc : *rew_permut_simpl*.

Hint Rewrite for_multiset_union_empty_l for_multiset_union_empty_r : *rew_permut_simpl*.

Ltac *rew_permut_simpl* :=

 autorewrite with *rew_permut_simpl*.

Ltac *rews_permut_simpl* :=

 autorewrite with *rew_permut_simpl* in *.

Section PermutationTactic.

Context ($A:\text{Type}$).

Implicit Types $l : \text{multiset } A$.

Lemma permut_get_1 : $\forall l1 l2,$

$(l1 \setminus u l2) = (l1 \setminus u l2)$.

Proof using. intros. auto. Qed.

Lemma permut_get_2 : $\forall l1 l2 l3,$

$(l1 \setminus u l2 \setminus u l3) = (l2 \setminus u l1 \setminus u l3)$.

Proof using. intros. apply union_comm_assoc. Qed.

Lemma permut_get_3 : $\forall l1 l2 l3 l4,$

$(l1 \setminus u l2 \setminus u l3 \setminus u l4) = (l2 \setminus u l3 \setminus u l1 \setminus u l4)$.

Proof using.

 intros. do 2 rewrite (union_assoc $l2$). apply permut_get_2.

Qed.

Lemma permut_get_4 : $\forall l1 l2 l3 l4 l5,$

$(l1 \setminus u l2 \setminus u l3 \setminus u l4 \setminus u l5)$

$= (l2 \setminus u l3 \setminus u l4 \setminus u l1 \setminus u l5)$.

Proof using.

```
intros. do 2 rewrite (union_assoc l2). apply permut_get_3.
Qed.
```

```
Lemma permut_get_5 : ∀ l1 l2 l3 l4 l5 l6,
  (l1 \u l2 \u l3 \u l4 \u l5 \u l6)
  = (l2 \u l3 \u l4 \u l5 \u l1 \u l6).
```

Proof using.

```
intros. do 2 rewrite (union_assoc l2). apply permut_get_4.
Qed.
```

```
Lemma permut_get_6 : ∀ l1 l2 l3 l4 l5 l6 l7,
  (l1 \u l2 \u l3 \u l4 \u l5 \u l6 \u l7)
  = (l2 \u l3 \u l4 \u l5 \u l6 \u l1 \u l7).
```

Proof using.

```
intros. do 2 rewrite (union_assoc l2). apply permut_get_5.
Qed.
```

```
Lemma permut_get_7 : ∀ l1 l2 l3 l4 l5 l6 l7 l8,
  (l1 \u l2 \u l3 \u l4 \u l5 \u l6 \u l7 \u l8)
  = (l2 \u l3 \u l4 \u l5 \u l6 \u l7 \u l1 \u l8).
```

Proof using.

```
intros. do 2 rewrite (union_assoc l2). apply permut_get_6.
Qed.
```

```
Lemma permut_get_8 : ∀ l1 l2 l3 l4 l5 l6 l7 l8 l9,
  (l1 \u l2 \u l3 \u l4 \u l5 \u l6 \u l7 \u l8 \u l9)
  = (l2 \u l3 \u l4 \u l5 \u l6 \u l7 \u l8 \u l1 \u l9).
```

Proof using.

```
intros. do 2 rewrite (union_assoc l2). apply permut_get_7.
Qed.
```

```
Lemma permut_tactic_setup : ∀ C l1 l2,
  C (\{} \u l1 \u \{}) (l2 \u \{}) → C l1 l2.
```

Proof using. intros. *rews_permut_simpl*. *eauto*. Qed.

```
Lemma permut_tactic_keep : ∀ C l1 l2 l3 l4,
  C ((l1 \u l2) \u l3) l4 →
  C (l1 \u (l2 \u l3)) l4.
```

Proof using. intros. *rews_permut_simpl*. *eauto*. Qed.

```
Lemma permut_tactic_trans : ∀ C l1 l2 l3,
  l3 = l2 → C l1 l3 → C l1 l2.
```

Proof using. intros. *subst¬*. Qed.

End PermutationTactic.

permut_lemma_get n returns the lemma *permut_get_n* for the given value of *n*

```
Ltac permut_lemma_get n :=
  match number_to_nat n with
```

```

| 1%nat ⇒ constr:(permut_get_1)
| 2%nat ⇒ constr:(permut_get_2)
| 3%nat ⇒ constr:(permut_get_3)
| 4%nat ⇒ constr:(permut_get_4)
| 5%nat ⇒ constr:(permut_get_5)
| 6%nat ⇒ constr:(permut_get_6)
| 7%nat ⇒ constr:(permut_get_7)
| 8%nat ⇒ constr:(permut_get_8)
end.
```

permut_prepare applies to a goal of the form $\text{permut} \mid l'$ and sets \mid and l' in the form $\mid 1 \backslash u \mid 2 \backslash u \dots \backslash u \backslash \{\}$, (some of the lists li are put in the form $x::\{\}$).

```

Ltac permut_simpl_prepare :=
  rew_permut_simpl;
  apply permut_tactic_setup;
  repeat rewrite ← union_assoc.
```

```

Ltac permut_index_of l lcontainer :=
  match constr:(lcontainer) with
  | l \u _ ⇒ constr:(1)
  | _ \u l \u _ ⇒ constr:(2)
  | _ \u _ \u l \u _ ⇒ constr:(3)
  | _ \u _ \u _ \u l \u _ ⇒ constr:(4)
  | _ \u _ \u _ \u _ \u l \u _ ⇒ constr:(5)
  | _ \u _ \u _ \u _ \u _ \u l \u _ ⇒ constr:(6)
  | _ \u _ \u _ \u _ \u _ \u _ \u l \u _ ⇒ constr:(7)
  | _ \u l \u _ ⇒ constr:(8)
  | _ ⇒ constr:(0)
  end.
```

permut_simplify simplifies a goal of the form $\text{permut} \mid l'$ where \mid and l' are lists built with concatenation and consing

```

Ltac permut_simpl_once_for permut_tactic_simpl :=
  let go l0 l' :=
    match l0 with
    | _ \u \{} ⇒ fail 1
    | _ \u (?l \u ?lr) ⇒
      match permut_index_of l l' with
      | 0 ⇒ apply permut_tactic_keep
      | ?n ⇒ let F := permut_lemma_get n in
        eapply permut_tactic_trans;
        [ eapply F; try typeclass
        | apply permut_tactic_simpl ]
    end
  end
```

```

    end in
  match goal with
  | ⊢ _ ?x ?y ⇒ go x y
  | ⊢ _ _ ?x ?y ⇒ go x y
  | ⊢ _ _ _ ?x ?y ⇒ go x y
  | ⊢ _ _ _ _ ?x ?y ⇒ go x y
  end.

Ltac permut_conclude :=
  first [ apply refl_equal
    | apply for_multiset_empty_incl ].

Ltac permut_simpl_for permut_tactic_simpl :=
  permut_simpl_prepare;
  repeat permut_simpl_once_for permut_tactic_simpl;
  rew_permut_simpl;
  try permut_conclude.

Lemma permut_tactic_simpl_eq : ∀ A (l1 l2 l3 l4:multiset A),
  (l1 \u l3) = l4 →
  (l1 \u (l2 \u l3)) = (l2 \u l4).

Proof using. intros. subst. apply permut_get_2. Qed.

Lemma permut_tactic_simpl_incl : ∀ A (l1 l2 l3 l4:multiset A),
  (l1 \u l3) \c l4 →
  (l1 \u (l2 \u l3)) \c (l2 \u l4).

Admitted.

Ltac get_premut_tactic_simpl tt :=
  match goal with
  | ⊢ ?x = ?y ⇒ constr:(permut_tactic_simpl_eq)
  | ⊢ ?x \c ?y ⇒ constr:(permut_tactic_simpl_incl)
  end.

Ltac permut_simpl_once :=
  let L := get_premut_tactic_simpl tt in permut_simpl_once_for L.

Ltac permut_simpl :=
  let L := get_premut_tactic_simpl tt in permut_simpl_for L.

Tactic Notation "multiset_eq" :=
  check_noevar_goal; permut_simpl.

Section DemoSetUnion.
Variables (A:Type).
Implicit Types l : multiset A.

Lemma demo_multiset_union_permut_simpl_1 :
  ∀ l1 l2 l3 : multiset A,
  (l1 \u l2 \u l3) = (l3 \u l2 \u l1).

```

Proof using.

```
intros.  
permut_simpl_prepare.  
permut_simpl_once.  
permut_simpl_once.  
permut_simpl_once.  
rew_permut_simpl.  
permut_conclude.
```

Qed.

Lemma demo_multiset_union_permut_simpl_2 :

```
forall  
(x:A) l1 l2 l3 l4,  
(l1 \u \{x\} \u l3 \u l2) \c (l1 \u l2 \u l4 \u (\{x\} \u l3)).
```

Proof using.

```
intros.  
permut_simpl_prepare.  
permut_simpl_once.  
permut_simpl_once.  
permut_simpl_once.  
permut_simpl_once.  
permut_simpl_once.  
rew_permut_simpl.  
permut_conclude.
```

Qed.

Lemma demo_multiset_union_permut_simpl_3 : $\forall (x y:A) l1 l1' l2 l3 l4,$

```
l1 \c l4 \u l1' →  
(l1 \u (\{x\} \u l2) \u \{x\} \u (\{y\} \u l3)) \c  
(\{y\} \u \{x\} \u (l1' \u l2 \u l4) \u (\{x\} \u l3)).
```

Proof using.

```
intros. dup.  
permut_simpl_prepare.  
permut_simpl_once.  
permut_simpl_once.  
permut_simpl_once.  
permut_simpl_once.  
permut_simpl_once.  
try permut_simpl_once.  
rew_permut_simpl.  
equates 1. apply H.  
permut_simpl_prepare.  
permut_simpl_once.  
rew_permut_simpl.
```

```

permut_conclude.
permut_simpl. applys_eq H. permut_simpl.
Qed.
End DemoSetUnion.

```

28.6.2 Tactics to prove membership

Section InUnionGet.

Variables (A :Type).

Implicit Types l : multiset A .

Lemma in_union_get_1 : $\forall x l1 l2,$
 $x \in l1 \rightarrow x \in (l1 \cup l2).$

Proof using. intros. apply in_union_l. auto. Qed.

Lemma in_union_get_2 : $\forall x l1 l2 l3,$
 $x \in l2 \rightarrow x \in (l1 \cup l2 \cup l3).$

Proof using. intros. apply in_union_r. apply in_union_get_1. Qed.

Lemma in_union_get_3 : $\forall x l1 l2 l3 l4,$
 $x \in l3 \rightarrow x \in (l1 \cup l2 \cup l3 \cup l4).$

Proof using. intros. apply in_union_r. apply in_union_get_2. Qed.

Lemma in_union_get_4 : $\forall x l1 l2 l3 l4 l5,$
 $x \in l4 \rightarrow x \in (l1 \cup l2 \cup l3 \cup l4 \cup l5).$

Proof using. intros. apply in_union_r. apply in_union_get_3. Qed.

Lemma in_union_get_5 : $\forall x l1 l2 l3 l4 l5 l6,$
 $x \in l5 \rightarrow x \in (l1 \cup l2 \cup l3 \cup l4 \cup l5 \cup l6).$

Proof using. intros. apply in_union_r. apply in_union_get_4. Qed.

End InUnionGet.

Arguments in_union_get_1 [A] [x] [l1] [l2].

Arguments in_union_get_2 [A] [x] [l1] [l2] [l3].

Arguments in_union_get_3 [A] [x] [l1] [l2] [l3].

Arguments in_union_get_4 [A] [x] [l1] [l2] [l3] [l4].

Arguments in_union_get_5 [A] [x] [l1] [l2] [l3] [l4] [l5].

in_union_get solves a goal of the form $x \in F1 \cup \dots \cup FN$ when there exists an hypothesis of the form $x \in Fi$ for some i .

```

Ltac in_union_get :=
  match goal with H: ?x \in ?A ⊢ ?x \in ?B ⇒
    match B with context [A] ⇒
      let go tt := first
        [ apply (in_union_get_1 H)
        | apply (in_union_get_2 H)

```

```

| apply (in_union_get_3 H)
| apply (in_union_get_4 H)
| apply (in_union_get_5 H) ] in
first [ go tt
| rewrite ← (for_multiset_union_empty_r B);
repeat rewrite ← for_multiset_union_assoc;
go tt ]
end end.

```

Section InUnionExtract.

Variables (A:Type).

Implicit Types l : multiset A.

Lemma in_union_extract_1 : $\forall x l1, x \in (\{x\} \cup l1)$.

Proof using. intros. apply in_union_get_1. apply in_single_self. Qed.

Lemma in_union_extract_2 : $\forall x l1 l2, x \in (l1 \cup \{x\} \cup l2)$.

Proof using. intros. apply in_union_get_2. apply in_single_self. Qed.

Lemma in_union_extract_3 : $\forall x l1 l2 l3, x \in (l1 \cup l2 \cup \{x\} \cup l3)$.

Proof using. intros. apply in_union_get_3. apply in_single_self. Qed.

Lemma in_union_extract_4 : $\forall x l1 l2 l3 l4, x \in (l1 \cup l2 \cup l3 \cup \{x\} \cup l4)$.

Proof using. intros. apply in_union_get_4. apply in_single_self. Qed.

Lemma in_union_extract_5 : $\forall x l1 l2 l3 l4 l5, x \in (l1 \cup l2 \cup l3 \cup l4 \cup \{x\} \cup l5)$.

Proof using. intros. apply in_union_get_5. apply in_single_self. Qed.

End InUnionExtract.

in_union_extract solves a goal of the form $x \in F1 \cup \dots \cup \{x\} \cup \dots \cup FN$.

Ltac *in_union_extract* :=

```

match goal with ⊢ ?x ∈ ?A ⇒
match A with context [ \{x\}] ⇒
let go tt := first
[ apply (in_union_extract_1)
| apply (in_union_extract_2)
| apply (in_union_extract_3)
| apply (in_union_extract_4)
| apply (in_union_extract_5) ] in
first [ go tt
| rewrite ← (for_multiset_union_empty_r A);
repeat rewrite ← for_multiset_union_assoc;

```

```

    go tt ]
end end.

Tactic Notation "multiset_in" :=
  first [ in_union_extract | in_union_get ].
```

28.6.3 Tactics to invert a membership hypothesis

Section InversionsTactic.

Context ($A:\text{Type}$).

Implicit Types $l : \text{multiset } A$.

Implicit Types $x : A$.

Lemma empty_eq_single_inv_1 : $\forall x l1 l2,$
 $l1 = l2 \rightarrow x \setminus \text{notin } l1 \rightarrow x \setminus \text{in } l2 \rightarrow \text{False}.$

Proof using. intros. subst x . Qed.

Lemma empty_eq_single_inv_2 : $\forall x l1 l2,$
 $l1 = l2 \rightarrow x \setminus \text{notin } l2 \rightarrow x \setminus \text{in } l1 \rightarrow \text{False}.$

Proof using. intros. subst x . Qed.

Lemma notin_empty : $\forall x,$
 $x \setminus \text{notin } (\{\}: \text{multiset } A).$

Proof using. intros. unfold notin. rewrite in_empty_eq. auto. Qed.

End InversionsTactic.

Hint Resolve notin_empty.

Ltac in_union_meta :=
 match goal with
 | $\vdash _ \setminus \text{in } \{_ \} \Rightarrow \text{apply in_single_self}$
 | $\vdash _ \setminus \text{in } \{_ \} _u _ \Rightarrow \text{apply in_union_l}; \text{apply in_single_self}$
 | $\vdash _ \setminus \text{in } _ _u _ \Rightarrow \text{apply in_union_r}; \text{in_union_meta}$
 end.

Ltac fseq_inv_core $H :=$
 let go $L :=$
 $false L; [\text{apply } H$
 $| \text{try apply notin_empty}$
 $| \text{instantiate; try in_union_meta}] \text{ in}$
 match type of H with
 | $\{\} = _ \Rightarrow \text{go empty_eq_single_inv_1}$
 | $_ = \{\} \Rightarrow \text{go empty_eq_single_inv_2}$
 | $_ = _ \Rightarrow \text{go empty_eq_single_inv_1}$
 end.

$\text{multiset_inv } H$ solves a goal by contraction if there exists an hypothesis of the form $\{\} = E1 \setminus u \dots \setminus u \setminus \{x\} \setminus u \dots \setminus u EN$ (or its symmetric). multiset_inv speculates on the

hypothesis H to be used.

```
Tactic Notation "multiset_inv" constr(H) :=
  fseq_inv_core H.
```

```
Tactic Notation "multiset_inv" :=
  match goal with
  | ⊢ {} ≠ _ ⇒ let H := fresh in intro H; multiset_inv H
  | ⊢ _ ≠ {} ⇒ let H := fresh in intro H; multiset_inv H
  | H: {} = _ ⊢ _ ⇒ multiset_inv H
  | H: _ = {} ⊢ _ ⇒ multiset_inv H
  end.
```

```
Section InUnionInv.
```

```
Variables (A:Type).
```

```
Implicit Types l : multiset A.
```

```
Lemma in_empty_inv : ∀ x,
  x \in ({}:multiset A) → False.
```

```
Proof using. introv. apply notin_empty. Qed.
```

```
Lemma in_single_inv : ∀ x y : A,
  x \in ({}:multiset A) → x = y.
```

```
Proof using. intros. rewrite @in_single_eq in H. auto. typeclass. Qed.
```

```
Lemma in_union_inv : ∀ x l1 l2,
  x \in (l1 ∪ l2) → x \in l1 ∨ x \in l2.
```

```
Proof using. introv H. rewrite @in_union_eq in H. auto. typeclass. Qed.
```

```
End InUnionInv.
```

```
Arguments in_single_inv [A] [x] [y].
```

```
Arguments in_union_inv [A] [x] [l1] [l2].
```

```
Ltac multiset_in_inv_base H M :=
  match type of H with
  | _ \in {} ⇒ false; apply (@in_empty_inv _ _ H)
  | _ \in {} ⇒
    generalize (in_single_inv H); try clear H; try intro_subst_hyp
  | _ \in _ ∪ _ ⇒
    let H' := fresh "TEMP" in
    destruct (in_union_inv H) as [H'|H'];
    try clear H; multiset_in_inv_base H' M
  | _ ⇒ rename H into M
  end.
```

```
Tactic Notation "multiset_in" constr(H) "as" ident(M) :=
  multiset_in_inv_base H M.
```

```
Tactic Notation "multiset_in" constr(H) :=
  let M := fresh "H" in multiset_in H as M.
```

```

Lemma union_empty_inv_multiset : ∀ A (l1 l2:multiset A),
  l1 ∩ l2 = {} → l1 = {} ∧ l2 = {}.
Proof using. intros. eapply union_empty_inv_inst. eauto. Qed.

Arguments union_empty_inv_multiset [A] [l1] [l2].
```

Ltac multiset_empty_core H :=

```

  match type of H with
  | _ ∩ _ = {} ⇒
    let H1 := fresh "M" in let H2 := fresh "M" in
    destruct (union_empty_inv_multiset H) as [H1 H2]; clear H;
    multiset_empty_core H1; multiset_empty_core H2
  | ?E = {} ⇒ try subst E; try solve [ false ]
  | {} = _ ∩ _ ⇒ symmetry in H; multiset_empty_core H
  | {} = ?E ⇒ symmetry in H; multiset_empty_core H
  end.
```

Tactic Notation "multiset_empty" "in" hyp(H) :=
 multiset_empty_core H.

Tactic Notation "multiset_empty" :=
 let H := fresh "M" in intro H; multiset_empty in H.

Tactic Notation "multiset_empty" constr(E) :=
 let H := fresh "M" in lets H:E; multiset_empty in H.

Chapter 29

Library **SLF.LibCore**

From *SLF* Require Export LibTactics LibLogic LibOperation LibReflect.

From *SLF* Require Export LibUnit LibProd LibSum LibOption LibNat LibInt LibList.

From *SLF* Require Export LibRelation LibOrder LibWf.

Export *LibTacticsCompatibility*.

Open Scope *Z_scope*.

Open Scope *Int_scope*.

Open Scope *comp_scope*.

Chapter 30

Library `SLF.LibSepTLCbuffer`

30.1 LibSepTLCbuffer: Appendix - Temporary Additions to TLC

This file contains temporary definitions that will eventually get merged into the various files from the TLC library.

Set Implicit Arguments.

From *SLF* Require Import LibInt.

Generalizable Variables *A B*.

Global Opaque `Z.mul`.

Global Opaque `Z.add`.

Chapter 31

Library `SLF.LibSepFmap`

31.1 LibSepFmap: Appendix - Finite Maps

Set Implicit Arguments.

From *SLF* Require Import LibCore.

31.2 Representation of Finite Maps

This file provides a representation of finite maps, which may be used to represent the memory state of a program.

- It implements basic operations such as creation of a singleton map, union of maps, read and update operations.
- It includes predicates to assert disjointness of two maps (predicate `disjoint`), and coherence of two maps on the intersection of their domain (predicate `agree`).
- It comes with a tactic for `fmap_eq` proving equalities modulo commutativity and associativity of map union.

The definition of the type `fmap` is slightly technical in that it involves a dependent pair to pack the partial function of type $A \rightarrow \text{option } B$ that represents the map, together with a proof of finiteness of the domain of this map. One useful lemma established in this file is the existence of fresh keys: for any finite map whose keys are natural numbers, there exists a natural number that does not already belong to the domain of that map.

31.2.1 Representation of Potentially-Infinite Maps as Partial Functions

Representation

Type of partial functions from A to B

```
Definition map (A B : Type) : Type :=
  A → option B.
```

Operations

Disjoint union of two partial functions

```
Definition map_union (A B : Type) (f1 f2 : map A B) : map A B :=
  fun (x:A) => match f1 x with
    | Some y => Some y
    | None => f2 x
  end.
```

Removal from a partial functions

```
Definition map_remove (A B : Type) (f : map A B) (k:A) : map A B :=
  fun (x:A) => If x = k then None else f x.
```

Finite domain of a partial function

```
Definition map_finite (A B : Type) (f : map A B) :=
  ∃ (L : list A), ∀ (x:A), f x ≠ None → mem x L.
```

Disjointness of domain of two partial functions

```
Definition map_disjoint (A B : Type) (f1 f2 : map A B) :=
  ∀ (x:A), f1 x = None ∨ f2 x = None.
```

Compatibility of two partial functions on the intersection of their domains (only for Separation Logic with RO-permissions)

```
Definition map_agree (A B : Type) (f1 f2 : map A B) :=
  ∀ x v1 v2,
  f1 x = Some v1 →
  f2 x = Some v2 →
  v1 = v2.
```

Domain of a map (as a predicate)

```
Definition map_indom (A B : Type) (f1 : map A B) : (A → Prop) :=
  fun (x:A) => f1 x ≠ None.
```

Filter the bindings of a map

```
Definition map_filter A B (F:A → B → Prop) (f:map A B) : map A B :=
  fun (x:A) => match f x with
    | None => None
    | Some y => If F x y then Some y else None
  end.
```

Map a function on the values of a map

```
Definition map_map A B1 B2 (F:A → B1 → B2) (f:map A B1) : map A B2 :=
```

```

fun (x:A) => match f x with
| None => None
| Some y => Some (F x y)
end.

```

Properties

Section MapOps.

Variables (A B : Type).

Implicit Types f : map A B.

Symmetry of disjointness

Lemma map_disjoint_sym :

sym (@map_disjoint A B).

Proof using.

introv H. unfolds map_disjoint. intros z. specializes H z. intuition.

Qed.

Commutativity of disjoint union

Lemma map_union_comm : $\forall f1 f2,$

$\text{map_disjoint } f1 f2 \rightarrow$

$\text{map_union } f1 f2 = \text{map_union } f2 f1.$

Proof using.

introv H. unfold map.

extens. intros x. unfolds map_disjoint, map_union.

specializes H x. cases (f1 x); cases (f2 x); auto. destruct H; false.

Qed.

Finiteness of union

Lemma map_union_finite : $\forall f1 f2,$

$\text{map_finite } f1 \rightarrow$

$\text{map_finite } f2 \rightarrow$

$\text{map_finite } (\text{map_union } f1 f2).$

Proof using.

introv [L1 F1] [L2 F2]. $\exists (L1 ++ L2).$ intros x M.

specializes F1 x. specializes F2 x. unfold map_union in M.

apply mem_app. destruct $\neg (f1 x).$

Qed.

Finiteness of removal

Definition map_remove_finite : $\forall x f,$

$\text{map_finite } f \rightarrow$

$\text{map_finite } (\text{map_remove } f x).$

Proof using.

*intros [L F]. $\exists L$. intros $x' M$.
specializes $F x'$. unfold map_remove in M . case_if \neg .
Qed.*

Finiteness of filter

Definition `map_filter_finite` : $\forall (F:A \rightarrow B \rightarrow \text{Prop}) f$,
 $\text{map_finite } f \rightarrow$
 $\text{map_finite} (\text{map_filter } F f)$.

Proof using.

*intros [L N]. $\exists L$. intros $x' M$.
specializes $N x'$. unfold map_filter in M .
destruct ($f x'$); tryfalse. case_if. applys N ; auto_false.*

Qed.

Finiteness of map

Definition `map_map_finite` : $\forall C (F:A \rightarrow B \rightarrow C) f$,
 $\text{map_finite } f \rightarrow$
 $\text{map_finite} (\text{map_map } F f)$.

Proof using.

*intros [L N]. $\exists L$. intros $x' M$.
specializes $N x'$. unfold map_map in M .
destruct ($f x'$); tryfalse. applys N ; auto_false.*

Qed.

End MapOps.

31.2.2 Representation of Finite Maps as Dependent Pairs

Representation

Inductive `fmap` ($A B : \text{Type}$) : $\text{Type} := \text{make} \{$
 $\text{fmap_data} :> \text{map } A B$;
 $\text{fmap_finite} : \text{map_finite } \text{fmap_data} \}$.

Arguments make [A] [B].

Operations

Declare Scope fmap_scope.

Domain of a fmap (as a predicate)

Definition `indom` ($A B : \text{Type}$) ($h:\text{fmap } A B$) : $(A \rightarrow \text{Prop}) :=$
 $\text{map_indom } h$.

Empty fmap

Program Definition `empty` ($A B : \text{Type}$) : `fmap A B` :=

```
make (fun l => None) _.  
Next Obligation.  $\exists (@\text{nil} A). \text{auto\_false}$ . Qed.
```

Arguments empty {A} {B}.

Singleton fmap

```
Program Definition single A B (x:A) (v:B) : fmap A B :=  
  make (fun x' => If x = x' then Some v else None) _.
```

Next Obligation.

```
   $\exists (x::\text{nil}). \text{intros. case\_if. subst} \neg$ .
```

Qed.

Union of fmaps

```
Program Definition union A B (h1 h2:fmap A B) : fmap A B :=  
  make (map_union h1 h2) _.
```

Next Obligation. destruct h1. destruct h2. apply \neg map_union_finite. Qed.

```
Notation "h1 \+ h2" := (union h1 h2)  
  (at level 51, right associativity) : fmap_scope.
```

Open Scope *fmap_scope*.

Update of a fmap

```
Definition update A B (h:fmap A B) (x:A) (v:B) : fmap A B :=  
  union (single x v) h.
```

Read in a fmap

```
Definition read (A B : Type) {IB:Inhab B} (h:fmap A B) (x:A) : B :=  
  match fmap_data h x with  
  | Some y => y  
  | None => arbitrary  
  end.
```

Removal from a fmap

```
Program Definition remove A B (h:fmap A B) (x:A) : fmap A B :=  
  make (map_remove h x) _.
```

Next Obligation. destruct h. apply \neg map_remove_finite. Qed.

Filter from a fmap

```
Program Definition filter A B (F: $A \rightarrow B \rightarrow \text{Prop}$ ) (h:fmap A B) : fmap A B :=  
  make (map_filter F h) _.
```

Next Obligation. destruct h. apply \neg map_filter_finite. Qed.

Map a function onto the keys of a fmap

```
Program Definition map_ A B1 B2 (F: $A \rightarrow B1 \rightarrow B2$ ) (h:fmap A B1) : fmap A B2 :=  
  make (map_map F h) _.
```

Next Obligation. destruct h. apply \neg map_map_finite. Qed.

Inhabited type **fmap**

```
Global Instance Inhab_fmap A B : Inhab (fmap A B).
Proof using. intros. applys Inhab_of_val (@empty A B). Qed.
```

31.2.3 Predicates on Finite Maps

Compatible fmmaps (only for Separation Logic with RO-permissions)

```
Definition agree A B (h1 h2:fmap A B) :=
  map_agree h1 h2.
```

Disjoint fmmaps

```
Definition disjoint A B (h1 h2 : fmap A B) : Prop :=
  map_disjoint h1 h2.
```

Three disjoint fmmaps (not needed for basic separation logic)

```
Definition disjoint_3 A B (h1 h2 h3 : fmap A B) :=
  disjoint h1 h2
   $\wedge$  disjoint h2 h3
   $\wedge$  disjoint h1 h3.
```

Notation for disjointness

```
Module NOTATIONFORFMAPDISJOINT.
```

```
Notation "\# h1 h2" := (disjoint h1 h2)
  (at level 40, h1 at level 0, h2 at level 0, no associativity) : fmap_scope.
```

```
Notation "\# h1 h2 h3" := (disjoint_3 h1 h2 h3)
  (at level 40, h1 at level 0, h2 at level 0, h3 at level 0, no associativity)
  : fmap_scope.
```

```
End NOTATIONFORFMAPDISJOINT.
```

```
Import NotationForFmapDisjoint.
```

31.3 Properties of Operations on Finite Maps

```
Section FmapProp.
```

```
Variables (A B : Type).
```

```
Implicit Types f g h : fmap A B.
```

```
Implicit Types x : A.
```

```
Implicit Types v : B.
```

31.3.1 Equality

```
Lemma make_eq :  $\forall (f1 f2:\text{map } A B) F1 F2,$ 
```

```
( $\forall x, f1\ x = f2\ x$ )  $\rightarrow$ 
  make  $f1\ F1$   $=$  make  $f2\ F2$ .
```

Proof using.

```
intros H. asserts: ( $f1 = f2$ ). { unfold map. extens $\neg$ . }
  subst. fequals. Qed.
```

```
Lemma eq_inv_fmap_data_eq :  $\forall h1\ h2,$ 
   $h1 = h2 \rightarrow$ 
```

```
 $\forall x, \text{fmap\_data } h1\ x = \text{fmap\_data } h2\ x.$ 
```

Proof using. intros. fequals. Qed.

```
Lemma fmap_extens :  $\forall h1\ h2,$ 
  ( $\forall x, \text{fmap\_data } h1\ x = \text{fmap\_data } h2\ x$ )  $\rightarrow$ 
   $h1 = h2.$ 
```

Proof using. intros [f1 F1] [f2 F2] M. simpls. applys \neg make_eq. Qed.

31.3.2 Disjointness

```
Lemma disjoint_sym :  $\forall h1\ h2,$ 
   $\setminus\# h1\ h2 \rightarrow$ 
   $\setminus\# h2\ h1.$ 
```

Proof using. intros [f1 F1] [f2 F2]. apply map_disjoint_sym. Qed.

```
Lemma disjoint_comm :  $\forall h1\ h2,$ 
   $\setminus\# h1\ h2 = \setminus\# h2\ h1.$ 
```

Proof using. lets: disjoint_sym. extens \times . Qed.

```
Lemma disjoint_empty_l :  $\forall h,$ 
   $\setminus\# \text{empty } h.$ 
```

Proof using. intros [f F] x. simple \neg . Qed.

```
Lemma disjoint_empty_r :  $\forall h,$ 
   $\setminus\# h \text{ empty}.$ 
```

Proof using. intros [f F] x. simple \neg . Qed.

Hint Resolve disjoint_sym disjoint_empty_l disjoint_empty_r.

```
Lemma disjoint_union_eq_r :  $\forall h1\ h2\ h3,$ 
   $\setminus\# h1\ (h2 \setminus\# h3) =$ 
   $(\setminus\# h1\ h2 \wedge \setminus\# h1\ h3).$ 
```

Proof using.

```
intros [f1 F1] [f2 F2] [f3 F3].
  unfolds disjoint, union. simpls.
  unfolds map_disjoint, map_union. extens. iff M [M1 M2].
  split; intros x; specializes M x;
  destruct (f2 x); intuition; tryfalse.
  intros x. specializes M1 x. specializes M2 x.
```

```
destruct (f2 x); intuition.
```

Qed.

```
Lemma disjoint_union_eq_l : ∀ h1 h2 h3,  
  \# (h2 \+ h3) h1 =  
  (\# h1 h2 ∧ \# h1 h3).
```

Proof using.

```
  intros. rewrite disjoint_comm.  
  apply disjoint_union_eq_r.
```

Qed.

```
Lemma disjoint_single_single : ∀ (x1 x2:A) (v1 v2:B),  
  x1 ≠ x2 →  
  \# (single x1 v1) (single x2 v2).
```

Proof using.

```
  introv N. intros x. simpls. do 2 case_if; auto.
```

Qed.

```
Lemma disjoint_single_single_same_inv : ∀ (x:A) (v1 v2:B),  
  \# (single x v1) (single x v2) →  
  False.
```

Proof using.

```
  introv D. specializes D x. simpls. case_if. destruct D; tryfalse.
```

Qed.

```
Lemma disjoint_3_unfold : ∀ h1 h2 h3,  
  \# h1 h2 h3 = (\# h1 h2 ∧ \# h2 h3 ∧ \# h1 h3).
```

Proof using. auto. Qed.

```
Lemma disjoint_single_set : ∀ h l v1 v2,  
  \# (single l v1) h →  
  \# (single l v2) h.
```

Proof using.

```
  introv M. unfolds disjoint, single, map_disjoint; simpls.  
  intros l'. specializes M l'. case_if¬. destruct M; auto_false.
```

Qed.

31.3.3 Union

```
Lemma union_self : ∀ h,  
  h \+ h = h.
```

Proof using.

```
  intros [f F]. apply¬ make_eq. simpl. intros x.  
  unfold map_union. cases¬ (f x).
```

Qed.

```
Lemma union_empty_l : ∀ h,
```

`empty \+ h = h.`

Proof using.

```
intros [f F]. unfold union, map_union, empty. simpl.  
apply make_eq.
```

Qed.

`Lemma union_empty_r : ∀ h,
h \+ empty = h.`

Proof using.

```
intros [f F]. unfold union, map_union, empty. simpl.  
apply make_eq. intros x. destruct (f x).
```

Qed.

`Lemma union_eq_empty_inv_l : ∀ h1 h2,
h1 \+ h2 = empty →
h1 = empty.`

Proof using.

```
intros (f1&F1) (f2&F2) M. inverts M as M.  
applys make_eq. intros l.  
unfolds map_union.  
lets: fun_eq_1 l M.  
cases (f1 l); auto_false.
```

Qed.

`Lemma union_eq_empty_inv_r : ∀ h1 h2,
h1 \+ h2 = empty →
h2 = empty.`

Proof using.

```
intros (f1&F1) (f2&F2) M. inverts M as M.  
applys make_eq. intros l.  
unfolds map_union.  
lets: fun_eq_1 l M.  
cases (f1 l); auto_false.
```

Qed.

`Lemma union_comm_of_disjoint : ∀ h1 h2,
\# h1 h2 →
h1 \+ h2 = h2 \+ h1.`

Proof using.

```
intros [f1 F1] [f2 F2] H. simpls. apply make_eq. simpl.  
intros. rewrite map_union_comm.
```

Qed.

`Lemma union_comm_of_agree : ∀ h1 h2,
agree h1 h2 →
h1 \+ h2 = h2 \+ h1.`

Proof using.

```
intros [f1 F1] [f2 F2] H. simpls. apply make_eq. simpl.  
intros l. specializes H l. unfolds map_union. simpls.  
cases (f1 l); cases (f2 l); auto. fequals. applys× H.
```

Qed.

Lemma union_assoc : $\forall h1 h2 h3,$
 $(h1 \+ h2) \+ h3 = h1 \+ (h2 \+ h3).$

Proof using.

```
intros [f1 F1] [f2 F2] [f3 F3]. unfolds union. simpls.  
apply make_eq. intros x. unfold map_union. destruct¬ (f1 x).
```

Qed.

Lemma union_eq_inv_of_disjoint : $\forall h2 h1 h1',$
 $\# h1 h2 \rightarrow$
 $\# h1' h2 \rightarrow$
 $h1 \+ h2 = h1' \+ h2 \rightarrow$
 $h1 = h1'.$

Proof using.

```
intros [f2 F2] [f1' F1'] [f1 F1] D D' E.  
applys make_eq. intros x. specializes D x. specializes D' x.  
lets E': eq_inv fmap_data_eq (rm E) x. simpls.  
unfolds map_union.  
cases (f1' x); cases (f1 x);  
destruct D; try congruence;  
destruct D'; try congruence.
```

Qed.

31.3.4 Compatibility

Lemma agree_refl : $\forall h,$
agree h h.

Proof using.

```
intros h. introv M1 M2. congruence.
```

Qed.

Lemma agree_sym : $\forall f1 f2,$
agree f1 f2 \rightarrow
agree f2 f1.

Proof using.

```
introv M. intros l v1 v2 E1 E2.  
specializes M l E1.
```

Qed.

Lemma agree_of_disjoint : $\forall h1 h2,$

`disjoint h1 h2 →
agree h1 h2.`

Proof using.

`introv HD. intros l v1 v2 M1 M2. destruct (HD l); false.`

Qed.

`Lemma agree_empty_l : ∀ h,
agree empty h.`

Proof using. `intros h l v1 v2 E1 E2. simpls. false.` **Qed.**

`Lemma agree_empty_r : ∀ h,
agree h empty.`

Proof using.

`hint agree_sym, agree_empty_l. eauto.`

Qed.

`Lemma agree_union_l : ∀ f1 f2 f3,
agree f1 f3 →
agree f2 f3 →
agree (f1 \+ f2) f3.`

Proof using.

`introv M1 M2. intros l v1 v2 E1 E2.
specializes M1 l. specializes M2 l.
simpls. unfolds map_union. cases (fmap_data f1 l).
{ inverts E1. applys× M1. }
{ applys× M2. }`

Qed.

`Lemma agree_union_r : ∀ f1 f2 f3,
agree f1 f2 →
agree f1 f3 →
agree f1 (f2 \+ f3).`

Proof using.

`hint agree_sym, agree_union_l. eauto.`

Qed.

`Lemma agree_union_lr : ∀ f1 g1 f2 g2,
agree g1 g2 →
\# f1 f2 (g1 \+ g2) →
agree (f1 \+ g1) (f2 \+ g2).`

Proof using.

`introv M1 (M2a&M2b&M2c).
rewrite disjoint_union_eq_r in *.
applys agree_union_l; applys agree_union_r;
try solve [applys× agree_of_disjoint].
auto.`

Qed.

Lemma agree_union_ll_inv : $\forall f1 f2 f3,$
 $\text{agree } (f1 \setminus+ f2) f3 \rightarrow$
 $\text{agree } f1 f3.$

Proof using.

 introv M. intros l v1 v2 E1 E2.
 specializes M l. simpls. unfolds map_union.
 rewrite E1 in M. applys \times M.

Qed.

Lemma agree_union_rl_inv : $\forall f1 f2 f3,$
 $\text{agree } f1 (f2 \setminus+ f3) \rightarrow$
 $\text{agree } f1 f2.$

Proof using.

 hint agree_union_ll_inv, agree_sym. eauto.

Qed.

Lemma agree_union_lr_inv_agree_agree : $\forall f1 f2 f3,$
 $\text{agree } (f1 \setminus+ f2) f3 \rightarrow$
 $\text{agree } f1 f2 \rightarrow$
 $\text{agree } f2 f3.$

Proof using.

 introv M D. rewrite \neg (@union_comm_of_agree f1 f2) in M.
 applys \times agree_union_ll_inv.

Qed.

Lemma agree_union_rr_inv_agree : $\forall f1 f2 f3,$
 $\text{agree } f1 (f2 \setminus+ f3) \rightarrow$
 $\text{agree } f2 f3 \rightarrow$
 $\text{agree } f1 f3.$

Proof using.

 hint agree_union_lr_inv_agree_agree, agree_sym. eauto.

Qed.

Lemma agree_union_l_inv : $\forall f1 f2 f3,$
 $\text{agree } (f1 \setminus+ f2) f3 \rightarrow$
 $\text{agree } f1 f2 \rightarrow$
 $\text{agree } f1 f3$
 $\wedge \text{agree } f2 f3.$

Proof using.

 introv M1 M2. split.
 { applys \times agree_union_ll_inv. }
 { applys \times agree_union_lr_inv_agree_agree. }

Qed.

Lemma agree_union_r_inv : $\forall f1 f2 f3,$

```

agree f1 (f2 \+ f3) →
agree f2 f3 →
  agree f1 f2
  △ agree f1 f3.

```

Proof using.

hint agree_sym.

intros. forwards¬ (M1&M2): agree_union_l_inv f2 f3 f1.

Qed.

31.3.5 Domain

Lemma indom_single : $\forall x v,$
 $\text{indom} (\text{single } x v) x.$

Proof using.

intros. hnf. simpl. case_if. auto_false.

Qed.

Lemma indom_union_l : $\forall h1 h2 x,$
 $\text{indom } h1 x \rightarrow$
 $\text{indom} (\text{union } h1 h2) x.$

Proof using.

intros. hnf. unfold union, map_union. simpl.
case_eq (fmap_data h1 x); auto_false.

Qed.

Lemma indom_union_r : $\forall h1 h2 x,$
 $\text{indom } h2 x \rightarrow$
 $\text{indom} (\text{union } h1 h2) x.$

Proof using.

intros. hnf. unfold union, map_union. simpl.
case_eq (fmap_data h1 x); auto_false.

Qed.

31.3.6 Disjoint and Domain

Lemma disjoint_eq_not_indom_both : $\forall h1 h2,$
 $\text{disjoint } h1 h2 = (\forall x, \text{indom } h1 x \rightarrow \text{indom } h2 x \rightarrow \text{False}).$

Proof using.

extens. iff D E.
{ introv M1 M2. destruct (D x); false×. }
{ intros x. specializes E x. unfolds indom, map_indom.
 applys not_not_inv. intros N. rew_logic in N. false×. }

Qed.

```
Lemma disjoint_of_not_indom_both : ∀ h1 h2,
  (forall x, indom h1 x → indom h2 x → False) →
  disjoint h1 h2.
```

Proof using. *introv M. rewrite¬ disjoint_eq_not_indom_both. Qed.*

```
Lemma disjoint_inv_not_indom_both : ∀ h1 h2 x,
  disjoint h1 h2 →
  indom h1 x →
  indom h2 x →
  False.
```

Proof using. *introv. rewrite× disjoint_eq_not_indom_both. Qed.*

```
Lemma disjoint_single_of_not_indom : ∀ h x v,
  ~ indom h x →
  disjoint (single x v) h.
```

Proof using.

*introv N. unfolds disjoint, map_disjoint. unfolds single, indom, map_indom.
simpl. rew_logic in N. intros l'. case_if; subst; autos×.*

Qed.

Note that the reciprocal of the above lemma is a special instance of `disjoint_inv_not_indom_both`

31.3.7 Read

```
Lemma read_single : ∀ {IB:Inhab B} x v,
  read (single x v) x = v.
```

Proof using.

intros. unfold read, single. simpl. case_if¬.

Qed.

```
Lemma read_union_l : ∀ {IB:Inhab B} h1 h2 x,
  indom h1 x →
  read (union h1 h2) x = read h1 x.
```

Proof using.

*intros. unfold read, union, map_union. simpl.
case_eq (fmap_data h1 x); auto_false.*

Qed.

```
Lemma read_union_r : ∀ {IB:Inhab B} h1 h2 x,
  ~ indom h1 x →
  read (union h1 h2) x = read h2 x.
```

Proof using.

*intros. unfold read, union, map_union. simpl.
case_eq (fmap_data h1 x).
{ intros v Hv. false H. auto_false. }*

$\{ \text{auto_false.} \}$
 Qed.

31.3.8 Update

Lemma `update_eq_union_single` : $\forall h x v,$
 $\text{update } h x v = \text{union} (\text{single } x v) h.$

Proof using. `auto.` `Qed.`

Lemma `update_single` : $\forall x v w,$
 $\text{update} (\text{single } x v) x w = \text{single } x w.$

Proof using.

```

intros. rewrite update_eq_union_single.
applys make_eq. intros y.
unfold map_union, single. simpl. case_if  $\neg$ .

```

`Qed.`

Lemma `update_union_l` : $\forall h1 h2 x v,$
 $\text{indom } h1 x \rightarrow$
 $\text{update} (\text{union } h1 h2) x v = \text{union} (\text{update } h1 x v) h2.$

Proof using.

```

intros. do 2 rewrite update_eq_union_single.
applys make_eq. intros y.
unfold map_union, union, map_union. simpl. case_if  $\neg$ .

```

`Qed.`

Lemma `update_union_r` : $\forall h1 h2 x v,$
 $\neg \text{indom } h1 x \rightarrow$
 $\text{update} (\text{union } h1 h2) x v = \text{union } h1 (\text{update } h2 x v).$

Proof using.

```

introv M. asserts IB: (Inhab B). { applys Inhab_of_val v. }
do 2 rewrite update_eq_union_single.
applys make_eq. intros y.
unfold map_union, union, map_union. simpl. case_if  $\neg$ .
{ subst. case_eq (fmap_data h1 y); auto_false.
  { intros w Hw. false M. auto_false. } }

```

`Qed.`

31.3.9 Removal

Lemma `remove_union_single_l` : $\forall h x v,$
 $\neg \text{indom } h x \rightarrow$
 $\text{remove} (\text{union} (\text{single } x v) h) x = h.$

Proof using.

```

intros M. applys fmap_extens. intros y.
unfold remove, map_remove, union, map_union, single. simpls. case_if.
{ destruct h as [f F]. unfolds indom, map_indom. simpls. subst. rew_logic in M. }
{ case_if. }
Qed.

```

End FmapProp.

Fixing arguments

```

Arguments union_assoc [A] [B].
Arguments union_comm_of_disjoint [A] [B].
Arguments union_comm_of_agree [A] [B].

```

31.4 Tactics for Finite Maps

31.4.1 Tactic disjoint for proving disjointness results

`disjoint` proves goals of the form $\# h1 h2$ and $\# h1 h2 h3$ by expanding all hypotheses into binary forms $\# h1 h2$ and then exploiting symmetry and disjointness with `empty`.

Module Export DISJOINTHINTS.

Hint Resolve disjoint_sym disjoint_empty_l disjoint_empty_r.

End DISJOINTHINTS.

```

Hint Rewrite
  disjoint_union_eq_l
  disjoint_union_eq_r
  disjoint_3_unfold : rew_disjoint.

```

```

Tactic Notation "rew_disjoint" :=
  autorewrite with rew_disjoint in *.

```

```

Tactic Notation "rew_disjoint" "*" :=
  rew_disjoint; auto_star.

```

```

Ltac fmap_disjoint_pre tt :=
  subst; rew_disjoint; jauto_set.

```

```

Tactic Notation "fmap_disjoint" :=
  solve [ fmap_disjoint_pre tt; auto ].

```

```

Lemma disjoint_demo : ∀ A B (h1 h2 h3 h4 h5:fmap A B),
  h1 = h2 \+ h3 →
  \# h2 h3 →
  \# h1 h4 h5 →
  \# h3 h2 h5 ∧ \# h4 h5.

```

Proof using.

intros. dup 2.

```

{ subst. rew_disjoint. jauto_set. auto. auto. auto. auto. }
{ fmap_disjoint. }

Qed.

```

31.4.2 Tactic *rew_map* for Normalizing Expressions

```

Hint Rewrite
  union_assoc
  union_empty_l
  union_empty_r : rew_fmap rew_fmap_for_fmap_eq.

```

```

Tactic Notation "rew_fmap" :=
  autorewrite with rew_fmap in *.

```

```

Tactic Notation "rew_fmap" "˜" :=
  rew_fmap; auto_tilde.

```

```

Tactic Notation "rew_fmap" "*" :=
  rew_fmap; auto_star.

```

31.4.3 Tactic *fmap_eq* for Proving Equalities

```
Section StateEq.
```

```
Variables (A B : Type).
```

```
Implicit Types h : fmap A B.
```

eq proves equalities between unions of fmmaps, of the form $h1 \setminus+ h2 \setminus+ h3 \setminus+ h4 = h1' \setminus+ h2' \setminus+ h3' \setminus+ h4'$. It attempts to discharge the disjointness side-conditions. Disclaimer: it cancels heaps at depth up to 4, but no more.

```

Lemma union_eq_cancel_1 : ∀ h1 h2 h2',
  h2 = h2' →
  h1 \+ h2 = h1 \+ h2'.

```

```
Proof using. intros. subst. auto. Qed.
```

```

Lemma union_eq_cancel_1' : ∀ h1,
  h1 = h1.

```

```
Proof using. intros. auto. Qed.
```

```

Lemma union_eq_cancel_2 : ∀ h1 h1' h2 h2',
  \# h1 h1' →
  h2 = h1' \+ h2' →
  h1 \+ h2 = h1' \+ h1 \+ h2'.

```

```
Proof using.
```

```

  intros. subst. rewrite ← union_assoc.
  rewrite (union_comm_of_disjoint h1 h1').
  rewrite¬ union_assoc. auto.

```

Qed.

Lemma union_eq_cancel_2' : $\forall h1\ h1'\ h2,$
 $\quad \text{\# } h1\ h1' \rightarrow$
 $\quad h2 = h1' \rightarrow$
 $\quad h1 \setminus+ h2 = h1' \setminus+ h1.$

Proof using.

intros. subst. apply \neg union_comm_of_disjoint.

Qed.

Lemma union_eq_cancel_3 : $\forall h1\ h1'\ h2\ h2'\ h3,$
 $\quad \text{\# } h1\ (h1' \setminus+ h2') \rightarrow$
 $\quad h2 = h1' \setminus+ h2' \setminus+ h3' \rightarrow$
 $\quad h1 \setminus+ h2 = h1' \setminus+ h2' \setminus+ h1 \setminus+ h3'.$

Proof using.

intros. subst.
rewrite \leftarrow (union_assoc $h1'\ h2'\ h3'$).
rewrite \leftarrow (union_assoc $h1'\ h2'\ (h1 \setminus+ h3')$).
apply \neg union_eq_cancel_2.

Qed.

Lemma union_eq_cancel_3' : $\forall h1\ h1'\ h2\ h2',$
 $\quad \text{\# } h1\ (h1' \setminus+ h2') \rightarrow$
 $\quad h2 = h1' \setminus+ h2' \rightarrow$
 $\quad h1 \setminus+ h2 = h1' \setminus+ h2' \setminus+ h1.$

Proof using.

intros. subst.
rewrite \leftarrow (union_assoc $h1'\ h2'\ h1$).
apply \neg union_eq_cancel_2'.

Qed.

Lemma union_eq_cancel_4 : $\forall h1\ h1'\ h2\ h2'\ h3'\ h4,$
 $\quad \text{\# } h1\ ((h1' \setminus+ h2') \setminus+ h3') \rightarrow$
 $\quad h2 = h1' \setminus+ h2' \setminus+ h3' \setminus+ h4' \rightarrow$
 $\quad h1 \setminus+ h2 = h1' \setminus+ h2' \setminus+ h3' \setminus+ h1 \setminus+ h4'.$

Proof using.

intros. subst.
rewrite \leftarrow (union_assoc $h1'\ h2'\ (h3' \setminus+ h4')$).
rewrite \leftarrow (union_assoc $h1'\ h2'\ (h3' \setminus+ h1 \setminus+ h4')$).
apply \neg union_eq_cancel_3.

Qed.

Lemma union_eq_cancel_4' : $\forall h1\ h1'\ h2\ h2'\ h3,$
 $\quad \text{\# } h1\ (h1' \setminus+ h2' \setminus+ h3') \rightarrow$
 $\quad h2 = h1' \setminus+ h2' \setminus+ h3' \rightarrow$
 $\quad h1 \setminus+ h2 = h1' \setminus+ h2' \setminus+ h3' \setminus+ h1.$

Proof using.

```
intros. subst.  
rewrite ← (union_assoc h2' h3' h1).  
apply¬ union_eq_cancel_3'.  
Qed.
```

End StateEq.

```
Ltac fmap_eq_step tt :=  
  match goal with | ⊢ ?G ⇒ match G with  
    | ?h1 = ?h1 ⇒ apply union_eq_cancel_1'  
    | ?h1 \+ ?h2 = ?h1 \+ ?h2' ⇒ apply union_eq_cancel_1'  
    | ?h1 \+ ?h2 = ?h1' \+ ?h1 ⇒ apply union_eq_cancel_2'  
    | ?h1 \+ ?h2 = ?h1' \+ ?h1 \+ ?h2' ⇒ apply union_eq_cancel_2'  
    | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h1 ⇒ apply union_eq_cancel_3'  
    | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h3' \+ ?h1 ⇒ apply union_eq_cancel_3'  
    | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h3' \+ ?h1 ⇒ apply union_eq_cancel_4'  
    | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h3' \+ ?h1 \+ ?h4' ⇒ apply union_eq_cancel_4'  
  end end.
```

```
Tactic Notation "fmap_eq" :=  
  subst;  
  autorewrite with rew_fmap_for_fmap_eq;  
  repeat (fmap_eq_step tt);  
  try match goal with  
    | ⊢ \# _ _ ⇒ fmap_disjoint  
    | ⊢ \# _ _ _ ⇒ fmap_disjoint  
  end.
```

```
Tactic Notation "fmap_eq" "˜" :=  
  fmap_eq; auto_tilde.
```

```
Tactic Notation "fmap_eq" "*" :=  
  fmap_eq; auto_star.
```

```
Lemma fmap_eq_demo : ∀ A B (h1 h2 h3 h4 h5:fmap A B),  
  \# h1 h2 h3 →  
  \# (h1 \+ h2 \+ h3) h4 h5 →  
  h1 = h2 \+ h3 →  
  h4 \+ h1 \+ h5 = h2 \+ h5 \+ h4 \+ h3.
```

Proof using.

```
intros. dup 2.  
{ subst. rew_fmap.  
  fmap_eq_step tt. fmap_disjoint.  
  fmap_eq_step tt.  
  fmap_eq_step tt. fmap_disjoint. auto. }  
{ fmap_eq. }
```

Qed.

31.5 Existence of Fresh Locations

31.5.1 Consecutive Locations

The notion of “consecutive locations” is useful for reasoning about records and arrays.

```
Fixpoint conseq (B:Type) (vs:list B) (l:nat) : fmap nat B :=
  match vs with
  | nil => empty
  | v::vs' => (single l v) \+ (conseq vs' (S l))
  end.
```

```
Lemma conseq_nil : ∀ B (l:nat),
  conseq (@nil B) l = empty.
```

Proof using. auto. Qed.

```
Lemma conseq_cons : ∀ B (l:nat) (v:B) (vs:list B),
  conseq (v::vs) l = (single l v) \+ (conseq vs (S l)).
```

Proof using. auto. Qed.

```
Lemma conseq_cons' : ∀ B (l:nat) (v:B) (vs:list B),
  conseq (v::vs) l = (single l v) \+ (conseq vs (l+1)).
```

Proof using. intros. math_rewrite (l+1 = S l)%nat. applys conseq_cons. Qed.

Global Opaque conseq.

Hint Rewrite conseq_nil conseq_cons : rew_listx.

31.5.2 Existence of Fresh Locations

```
Definition loc_fresh_gen (L : list nat) :=
  (1 + fold_right plus O L)%nat.
```

```
Lemma loc_fresh_ind : ∀ l L,
  mem l L →
  (l < loc_fresh_gen L)%nat.
```

Proof using.

```
  intros l L. unfold loc_fresh_gen.
  induction L; introv M; inverts M; rew_listx.
  { math. }
  { forwards~: IHL. math. }
```

Qed.

```
Lemma loc_fresh_nat_ge : ∀ (L:list nat),
  ∃ (l:nat), ∀ (i:nat), ¬ mem (l+i)%nat L.
```

Proof using.

```
intros L.  $\exists$  (loc_fresh_gen L). intros i M.  
lets: loc_fresh_ind M. math.
```

Qed.

For any finite list of locations (implemented as **nat**), there exists one location not in that list.

Lemma loc_fresh_nat : $\forall (L:\text{list nat})$,
 $\exists (l:\text{nat}), \neg \text{mem } l L$.

Proof using.

```
intros L. forwards (l&P): loc_fresh_nat_ge L.  
 $\exists l$ . intros M. applys (P 0%nat). applys_eq M. math.
```

Qed.

Section FmapFresh.

Variables (B : Type).

Implicit Types h : **fmap** **nat** B.

For any heap, there exists one (non-null) location not already in the domain of that heap.

Lemma exists_fresh : $\forall \text{null } h$,
 $\exists l, \neg \text{indom } h l \wedge l \neq \text{null}$.

Proof using.

```
intros null (m&(L&M)). unfold indom, map_indom. simpl.  
lets (l&F): (loc_fresh_nat (null::L)).  
 $\exists l$ . split.  
{ simpl. intros l'. forwards  $\neg E: M l$ . }  
{ intro_subst. applys  $\neg F$ . }
```

Qed.

For any heap **h**, there exists one (non-null) location **l** such that the singleton map that binds **l** to a value **v** is disjoint from **h**.

Lemma single_fresh : $\forall \text{null } h v$,
 $\exists l, \text{single } l v \text{ } h \wedge l \neq \text{null}$.

Proof using.

```
intros. forwards (l&F&N): exists_fresh null h.  
 $\exists l$ . split  $\neg$ . applys  $\times$  disjoint_single_of_not_indom.
```

Qed.

Lemma conseq_fresh : $\forall \text{null } h vs$,
 $\exists l, \text{conseq } vs l \text{ } h \wedge l \neq \text{null}$.

Proof using.

```
intros null (m&(L&M)) vs.  
unfold disjoint, map_disjoint. simpl.  
lets (l&F): (loc_fresh_nat_ge (null::L)).
```

```

 $\exists l. \text{split}.$ 
{ intros l'. gen l. induction vs as [|v vs'|]; intros.
  { simple $\neg$ . }
  { rewrite conseq_cons. destruct (IHvs' (S l)%nat) as [E|?].
    { intros i N. applys F (S i). applys_eq N. math. }
    { simpl. unfold map_union. case_if $\neg$ .
      { subst. right. applys not_not_inv. intros H. applys F 0%nat.
        constructor. math_rewrite (l'+0 = l')%nat. applys $\neg$  M. } }
    { auto. } }
  { intro_subst. applys $\neg$  F 0%nat. rew_nat. auto. }
Qed.

```

Lemma disjoint_single_conseq : $\forall B l l' L (v:B),$
 $(l < l')\%nat \vee (l \geq l' + \text{length } L)\%nat \rightarrow$
 $\text{single } l v \text{ (conseq } L l').$

Proof using.

```

intros N. gen l'. induction L as [|L']; intros.
{ rewrite $\neg$  conseq_nil. }
{ rew_list in N. rewrite conseq_cons. rew_disjoint. split.
  { applys disjoint_single_single. destruct N; math. }
  { applys IHL. destruct N. { left. math. } { right. math. } } }

```

Qed.

31.5.3 Existence of a Smallest Fresh Locations

The notion of smallest fresh location is useful for setting up deterministic semantics.

Definition fresh (null:nat) (h:fmap nat B) (l:nat) : Prop :=
 $\neg \text{indom } h l \wedge l \neq \text{null}.$

Definition smallest_fresh (null:nat) (h:fmap nat B) (l:nat) : Prop :=
 $\text{fresh null } h l \wedge (\forall l', l' < l \rightarrow \neg \text{fresh null } h l').$

Lemma exists_smallest_fresh : $\forall \text{null } h,$
 $\exists l, \text{smallest_fresh null } h l.$

Proof using.

```

intros.
cuts M: ( $\forall l_0, \text{fresh null } h l_0 \rightarrow$ 
 $\exists l, \text{fresh null } h l$ 
 $\wedge (\forall l', l' < l \rightarrow \neg \text{fresh null } h l')).$ 
{ lets (l0&F&N): exists_fresh null h. applys M l0. split $\times$ . }
intros l0. induction_wf IH: wf_lt l0. intros F.
tests C: ( $\forall l', l' < l_0 \rightarrow \neg \text{fresh null } h l')$ .
{  $\exists \times l_0.$  }
{ destruct C as (l0'&K). rew_logic in K. destruct K as (L&F').
  applys $\times$  IH l0'. }

```

Qed.

31.5.4 Existence of Nonempty Heaps

The existence of nonempty (nat-indexed) finite maps is useful for constructing counterexamples.

Lemma exists_not_empty : $\forall \{IB:\mathbf{Inhab} B\},$
 $\exists (h:\mathbf{fmap} \mathbf{nat} B), h \neq \text{empty}.$

Proof using.

```
intros. sets l: 0%nat. sets h: (single l (arbitrary (A:=B))).  
   $\exists h.$  intros N.  
  sets h': (empty:fmap nat B).  
  asserts M1: (indom h l).  
  { applys indom_single. }  
  asserts M2: ( $\neg$  indom h' l).  
  { unfolds @indom, map_indom, @empty. simpls. auto_false. }  
  rewrite N in M1. false $\times$ .
```

Qed.

End FmapFresh.

Chapter 32

Library **SLF.LibSepVar**

32.1 LibSepVar: Appendix - Program Variables

Set Implicit Arguments.

From *SLF* Require LibListExec.

From *SLF* Require Export LibString LibList LibCore.

From *SLF* Require Import LibSepFmap LibSepTLCbuffer.

Open Scope *string_scope*.

Generalizable Variables *A*.

32.2 Representation of Program Variables

This file contains definitions, lemmas, tactics and notations for manipulating program variables and list of program variables.

32.2.1 Representation of Variables

Variables are represented as strings

Definition var : Type := **string**.

The boolean function var_eq *s1 s2* compares two variables.

Definition var_eq (*s1:var*) (*s2:var*) : **bool** :=
if **String.string_dec** *s1 s2* then **true** else **false**.

The boolean function var_eq *s1 s2* returns **true** iff the equality *v1 = v2* holds.

Lemma var_eq_spec : $\forall s1\ s2,$
 $\text{var_eq}\ s1\ s2 = \text{isTrue}\ (s1 = s2).$

Proof using.

intros. unfold var_eq. case_if; rew_bool_eq¬.

Qed.

```
Global Opaque var.
```

32.2.2 Tactic *case_var*

The tactic *case_var* performs case analysis on expressions of the form `if var_eq x y then .. else ..` that appear in the goal.

```
Tactic Notation "case_var" :=
  repeat rewrite var_eq_spec in *; repeat case_if.
```

```
Tactic Notation "case_var" "˜" :=
  case_var; auto_tilde.
```

```
Tactic Notation "case_var" "*" :=
  case_var; auto_star.
```

32.3 Representation of List of Variables

32.3.1 Definition of Distinct Variables

`vars` is the type of a list of variables

```
Definition vars : Type := list var.
```

`var_fresh y xs` asserts that `y` does not belong to the list `xs`

```
Fixpoint var_fresh (y:var) (xs:vars) : bool :=
  match xs with
  | nil => true
  | x::xs' => if var_eq x y then false else var_fresh y xs'
  end.
```

`var_distinct xs` asserts that `xs` consists of a list of pairwise distinct variables, i.e., a list of variables distinct from each other.

```
Fixpoint var_distinct (xs:vars) : Prop :=
  match xs with
  | nil => True
  | x::xs' => var_fresh x xs' ∧ var_distinct xs'
  end.
```

`var_distinct_exec xs` is a boolean function that decides whether a list of variables contains only distinct variables, that is, whether the proposition `var_distinct xs` is satisfied.

```
Fixpoint var_distinct_exec (xs:vars) : bool :=
  match xs with
  | nil => true
  | x::xs' => var_fresh x xs' && var_distinct_exec xs'
  end.
```

```
Lemma var_distinct_exec_eq : ∀ xs,
  var_distinct_exec xs = isTrue (var_distinct xs).
```

Proof using.

```
intros. induction xs as [|x xs']; simpl; rew_isTrue.
{ auto. } { rewrite¬ IHxs'.
```

Qed.

The following lemma asserts that if x is a variable in the list xs , and y is fresh from this list xs , then y is not equal to x .

```
Lemma var_fresh_mem_inv : ∀ y x xs,
  var_fresh x xs →
  mem y xs →
  x ≠ y.
```

Proof using.

```
introv H M N. subst. induction xs as [|x xs'].
{ inverts M. }
{ simpls. case_var. inverts¬ M. }
```

Qed.

32.3.2 Definition of n Distinct Variables

`var_funs xs n` asserts that xs consists of n distinct variables, where n is asserted to be a positive number.

```
Definition var_funs (xs:vars) (n:nat) : Prop :=
  var_distinct xs
  ∧ length xs = n
  ∧ xs ≠ nil.
```

`var_funs xs n` is a boolean function that decides whether the proposition `var_funs xs n` is satisfied

```
Definition var_funs_exec (xs:vars) (n:nat) : bool :=
  LibNat.beq n (LibListExec.length xs)
  && LibListExec.is_not_nil xs
  && var_distinct_exec xs.
```

```
Lemma var_funs_exec_eq : ∀ (n:nat) xs,
  var_funs_exec xs n = isTrue (var_funs xs n).
```

Proof using.

```
intros. unfold var_funs_exec, var_funs.
rewrite LibNat.beq_eq.
rewrite LibListExec.is_not_nil_eq.
rewrite LibListExec.length_eq.
rewrite var_distinct_exec_eq.
extens. rew_istrue. iff ×.
```

Qed.

32.3.3 Generation of n Distinct Variables

`nat_to_var n` converts `nat` values into distinct *name* values.

Definition `dummy_char :=`
`Ascii.ascii_of_nat 0%nat.`

Fixpoint `nat_to_var (n:nat) : var :=`
 `match n with`
 | `O` \Rightarrow `String.EmptyString`
 | `S n'` \Rightarrow `String.String dummy_char (nat_to_var n')`
 `end.`

Lemma `injective_nat_to_var :`
 `injective nat_to_var.`

Proof using.

```
intros n. induction n as [|n']; intros m E; destruct m as [|m']; tryfalse.  
{ auto. }  
{ inverts E. fequals¬. }
```

Qed.

`var_seq i n` generates a list of variables $x_1; x_2; \dots; x_n$ with $x_1 = i$ and $x_n = i + n - 1$. The ability to start at a given offset is sometimes useful.

Fixpoint `var_seq (start:nat) (nb:nat) : vars :=`
 `match nb with`
 | `O` \Rightarrow `nil`
 | `S nb'` \Rightarrow `(nat_to_var start) :: var_seq (S start) nb'`
 `end.`

The properties of `var_seq` are stated next. They assert that this function produce the expected number of variables, that the variables are pairwise distinct

Section `Var_seq.`

Implicit Types `start nb : nat.`

Lemma `var_fresh_var_seq_lt` : $\forall (x:\text{nat}) \text{ start nb},$
 $(x < \text{start})\%nat \rightarrow$
`var_fresh (nat_to_var x) (var_seq start nb).`

Proof using.

```
intros. gen start. induction nb; intros.  
{ auto. }  
{ simpl. case_var.  
{ lets: injective_nat_to_var C. math. }  
{ applys IHnb. math. } }
```

Qed.

```

Lemma var_fresh_var_seq_ge : ∀ (x:nat) start nb,
  (x ≥ start+nb)%nat →
  var_fresh (nat_to_var x) (var_seq start nb).

```

Proof using.

```

intros. gen start. induction nb; intros.
{ auto. }
{ simpl. case_var.
  { lets: injective_nat_to_var C. math. }
  { applys IHnb. math. } }

```

Qed.

```

Lemma var_distinct_var_seq : ∀ start nb,
  var_distinct (var_seq start nb).

```

Proof using.

```

intros. gen start. induction nb; intros.
{ simple¬. }
{ split.
  { applys var_fresh_var_seq_lt. math. }
  { auto. } }

```

Qed.

```

Lemma length_var_seq : ∀ start nb,
  length (var_seq start nb) = nb.

```

Proof using.

```

intros. gen start. induction nb; simpl; intros.
{ auto. } { rew_list. rewrite¬ IHnb. }

```

Qed.

```

Lemma var_funcs_var_seq : ∀ start nb,
  (nb > 0%nat)%nat →
  var_funcs (var_seq start nb) nb.

```

Proof using.

```

introv E. splits.
{ applys var_distinct_var_seq. }
{ applys length_var_seq. }
{ destruct nb. { false. math. } { simpl. auto_false. } }

```

Qed.

End Var_seq.

32.4 Notation for Program Variables

When writing deeply-embedded programs, it is nice to write identifiers (program variables) without quotes. In the future, the “Custom Entry” mechanism might allow for this in a generic manner. In the meantime, we need one definition or one notation for each identifier

32.4.1 Program Variables Expressed Using Definitions

These definitions allowing to use the string notation "x" without any double quote or quotes. However, the scope of these definitions should be wrapped in a module to avoid clashes with local variables. E.g. `Module Export Foo. Import DEFINITIONSFORVARIABLES. (...) End Foo.`

`Module DEFINITIONSFORVARIABLES.`

```
Definition a := ("a":var).
Definition b := ("b":var).
Definition c := ("c":var).
Definition d := ("d":var).
Definition e := ("e":var).
Definition f := ("f":var).
Definition g := ("g":var).
Definition h := ("h":var).
Definition i := ("i":var).
Definition j := ("j":var).
Definition k := ("k":var).
Definition l := ("l":var).
Definition m := ("m":var).
Definition n := ("n":var).
Definition o := ("o":var).
Definition p := ("p":var).
Definition q := ("q":var).
Definition r := ("r":var).
Definition s := ("s":var).
Definition t := ("t":var).
Definition u := ("u":var).
Definition v := ("v":var).
Definition w := ("w":var).
Definition x := ("x":var).
Definition y := ("y":var).
Definition z := ("z":var).

Definition a1 := ("a1":var).
Definition b1 := ("b1":var).
Definition c1 := ("c1":var).
Definition d1 := ("d1":var).
Definition e1 := ("e1":var).
Definition f1 := ("f1":var).
Definition g1 := ("g1":var).
Definition h1 := ("h1":var).
Definition i1 := ("i1":var).
```

```

Definition j1 := ("j1":var).
Definition k1 := ("k1":var).
Definition l1 := ("l1":var).
Definition m1 := ("m1":var).
Definition n1 := ("n1":var).
Definition o1 := ("o1":var).
Definition p1 := ("p1":var).
Definition q1 := ("q1":var).
Definition r1 := ("r1":var).
Definition s1 := ("s1":var).
Definition t1 := ("t1":var).
Definition u1 := ("u1":var).
Definition v1 := ("v1":var).
Definition w1 := ("w1":var).
Definition x1 := ("x1":var).
Definition y1 := ("y1":var).
Definition z1 := ("z1":var).

Definition a2 := ("a2":var).
Definition b2 := ("b2":var).
Definition c2 := ("c2":var).
Definition d2 := ("d2":var).
Definition e2 := ("e2":var).
Definition f2 := ("f2":var).
Definition g2 := ("g2":var).
Definition h2 := ("h2":var).
Definition i2 := ("i2":var).
Definition j2 := ("j2":var).
Definition k2 := ("k2":var).
Definition l2 := ("l2":var).
Definition m2 := ("m2":var).
Definition n2 := ("n2":var).
Definition o2 := ("o2":var).
Definition p2 := ("p2":var).
Definition q2 := ("q2":var).
Definition r2 := ("r2":var).
Definition s2 := ("s2":var).
Definition t2 := ("t2":var).
Definition u2 := ("u2":var).
Definition v2 := ("v2":var).
Definition w2 := ("w2":var).
Definition x2 := ("x2":var).
Definition y2 := ("y2":var).

```

```

Definition z2 := ("z2":var).
Definition a3 := ("a3":var).
Definition b3 := ("b3":var).
Definition c3 := ("c3":var).
Definition d3 := ("d3":var).
Definition e3 := ("e3":var).
Definition f3 := ("f3":var).
Definition g3 := ("g3":var).
Definition h3 := ("h3":var).
Definition i3 := ("i3":var).
Definition j3 := ("j3":var).
Definition k3 := ("k3":var).
Definition l3 := ("l3":var).
Definition m3 := ("m3":var).
Definition n3 := ("n3":var).
Definition o3 := ("o3":var).
Definition p3 := ("p3":var).
Definition q3 := ("q3":var).
Definition r3 := ("r3":var).
Definition s3 := ("s3":var).
Definition t3 := ("t3":var).
Definition u3 := ("u3":var).
Definition v3 := ("v3":var).
Definition w3 := ("w3":var).
Definition x3 := ("x3":var).
Definition y3 := ("y3":var).
Definition z3 := ("z3":var).

```

Tactic to unfold these definitions. Useful to avoid the definitions to get in the way of `simpl`.

```

Ltac libsepvar_unfold :=
  unfold
  a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,
  a1, b1, c1, d1, e1, f1, g1, h1, i1, j1, k1, l1, m1, n1, o1, p1, q1, r1, s1, t1, u1, v1, w1, x1, y1,
  z1,
  a2, b2, c2, d2, e2, f2, g2, h2, i2, j2, k2, l2, m2, n2, o2, p2, q2, r2, s2, t2, u2, v2, w2, x2, y2,
  z2,
  a3, b3, c3, d3, e3, f3, g3, h3, i3, j3, k3, l3, m3, n3, o3, p3, q3, r3, s3, t3, u3, v3, w3, x3, y3,
  z3.

```

End DEFINITIONSFORVARIABLES.

32.4.2 Program Variables Expressed Using Notation, with a Quote Symbol

To avoid using the string notation "`x`" for referring to a variable called `x`, one can use the notation '`x`', available by importing the following module.

Declare Custom Entry trm.

Module NOTATIONFORVARIABLES.

Declare Scope trm_scope.

```
Notation ""a"" := ("a":var) (in custom trm at level 0) : trm_scope.
Notation ""b"" := ("b":var) (in custom trm at level 0) : trm_scope.
Notation ""c"" := ("c":var) (in custom trm at level 0) : trm_scope.
Notation ""d"" := ("d":var) (in custom trm at level 0) : trm_scope.
Notation ""e"" := ("e":var) (in custom trm at level 0) : trm_scope.
Notation ""f"" := ("f":var) (in custom trm at level 0) : trm_scope.
Notation ""g"" := ("g":var) (in custom trm at level 0) : trm_scope.
Notation ""h"" := ("h":var) (in custom trm at level 0) : trm_scope.
Notation ""i"" := ("i":var) (in custom trm at level 0) : trm_scope.
Notation ""j"" := ("j":var) (in custom trm at level 0) : trm_scope.
Notation ""k"" := ("k":var) (in custom trm at level 0) : trm_scope.
Notation ""l"" := ("l":var) (in custom trm at level 0) : trm_scope.
Notation ""m"" := ("m":var) (in custom trm at level 0) : trm_scope.
Notation ""n"" := ("n":var) (in custom trm at level 0) : trm_scope.
Notation ""o"" := ("o":var) (in custom trm at level 0) : trm_scope.
Notation ""p"" := ("p":var) (in custom trm at level 0) : trm_scope.
Notation ""q"" := ("q":var) (in custom trm at level 0) : trm_scope.
Notation ""r"" := ("r":var) (in custom trm at level 0) : trm_scope.
Notation ""s"" := ("s":var) (in custom trm at level 0) : trm_scope.
Notation ""t"" := ("t":var) (in custom trm at level 0) : trm_scope.
Notation ""u"" := ("u":var) (in custom trm at level 0) : trm_scope.
Notation ""v"" := ("v":var) (in custom trm at level 0) : trm_scope.
Notation ""w"" := ("w":var) (in custom trm at level 0) : trm_scope.
Notation ""x"" := ("x":var) (in custom trm at level 0) : trm_scope.
Notation ""y"" := ("y":var) (in custom trm at level 0) : trm_scope.
Notation ""z"" := ("z":var) (in custom trm at level 0) : trm_scope.

Notation ""a1"" := ("a1":var) (in custom trm at level 0) : trm_scope.
Notation ""b1"" := ("b1":var) (in custom trm at level 0) : trm_scope.
Notation ""c1"" := ("c1":var) (in custom trm at level 0) : trm_scope.
Notation ""d1"" := ("d1":var) (in custom trm at level 0) : trm_scope.
Notation ""e1"" := ("e1":var) (in custom trm at level 0) : trm_scope.
Notation ""f1"" := ("f1":var) (in custom trm at level 0) : trm_scope.
Notation ""g1"" := ("g1":var) (in custom trm at level 0) : trm_scope.
```



```

Notation ""x2"" := ("x2":var) (in custom trm at level 0) : trm_scope.
Notation ""y2"" := ("y2":var) (in custom trm at level 0) : trm_scope.
Notation ""z2"" := ("z2":var) (in custom trm at level 0) : trm_scope.

Notation ""a3"" := ("a3":var) (in custom trm at level 0) : trm_scope.
Notation ""b3"" := ("b3":var) (in custom trm at level 0) : trm_scope.
Notation ""c3"" := ("c3":var) (in custom trm at level 0) : trm_scope.
Notation ""d3"" := ("d3":var) (in custom trm at level 0) : trm_scope.
Notation ""e3"" := ("e3":var) (in custom trm at level 0) : trm_scope.
Notation ""f3"" := ("f3":var) (in custom trm at level 0) : trm_scope.
Notation ""g3"" := ("g3":var) (in custom trm at level 0) : trm_scope.
Notation ""h3"" := ("h3":var) (in custom trm at level 0) : trm_scope.
Notation ""i3"" := ("i3":var) (in custom trm at level 0) : trm_scope.
Notation ""j3"" := ("j3":var) (in custom trm at level 0) : trm_scope.
Notation ""k3"" := ("k3":var) (in custom trm at level 0) : trm_scope.
Notation ""l3"" := ("l3":var) (in custom trm at level 0) : trm_scope.
Notation ""m3"" := ("m3":var) (in custom trm at level 0) : trm_scope.
Notation ""n3"" := ("n3":var) (in custom trm at level 0) : trm_scope.
Notation ""o3"" := ("o3":var) (in custom trm at level 0) : trm_scope.
Notation ""p3"" := ("p3":var) (in custom trm at level 0) : trm_scope.
Notation ""q3"" := ("q3":var) (in custom trm at level 0) : trm_scope.
Notation ""r3"" := ("r3":var) (in custom trm at level 0) : trm_scope.
Notation ""s3"" := ("s3":var) (in custom trm at level 0) : trm_scope.
Notation ""t3"" := ("t3":var) (in custom trm at level 0) : trm_scope.
Notation ""u3"" := ("u3":var) (in custom trm at level 0) : trm_scope.
Notation ""v3"" := ("v3":var) (in custom trm at level 0) : trm_scope.
Notation ""w3"" := ("w3":var) (in custom trm at level 0) : trm_scope.
Notation ""x3"" := ("x3":var) (in custom trm at level 0) : trm_scope.
Notation ""y3"" := ("y3":var) (in custom trm at level 0) : trm_scope.
Notation ""z3"" := ("z3":var) (in custom trm at level 0) : trm_scope.

```

End NOTATIONFORVARIABLES.

32.5 Optional Material

32.5.1 Bonuse: the Tactic *var_neq*

This tactic is not used in the SLF volume.

The tactic *var_neq* helps prove calls of the form $x \neq y$, where x and y are two concrete program variables. This tactic is registered as hint for *auto*

```

Ltac var_neq :=
  match goal with ⊢ ?x ≠ ?y :> var ⇒
    solve [ let E := fresh in

```

```
destruct (String.string_dec x y) as [E|E];
[ false | apply E ] ] end.
```

Hint Extern 1 (?x \neq ?y) \Rightarrow var_neq.

Chapter 33

Library `SLF.LibSepSimpl`

33.1 LibSepSimpl: Appendix - Simplification of Entailments

Set Implicit Arguments.

From *SLF* Require Export `LibCore`.

From *SLF* Require Export `LibSepTLCbuffer`.

33.2 A Functor for Separation Logic Entailment

This file consists of a functor that provides a tactic for simplifying entailment relations. This tactic is somewhat generic in that it can be used for several variants of Separation Logic, hence the use of a functor to implement the tactic with respect to abstract definitions of heap predicates.

The file provides a number of lemmas that hold in any variant of Separation Logic satisfying the requirements of the functor. It also provides the following key tactics:

- *xsimpl* simplifies heap entailments.
- *xpull* is a restricted version of *xsimpl* that only act over the left-hand side of the entailment, leaving the right-hand side unmodified.
- *xchange* performs transitivity steps in entailments, it enables replacing a subset of the heap predicates on the right-hand side with another set of heap predicates entailed by the former.

Bonus: the tactic *rew_heap* normalizes heap predicate expressions; it is not used in the course.

33.3 Assumptions of the functor

Module Type XSIMPLPARAMS.

33.3.1 Operators

The notion of heap predicate and entailment must be provided.

Parameter $hprop : \text{Type}$.

Parameter $himpl : hprop \rightarrow hprop \rightarrow \text{Prop}$.

Definition $\text{qimpl } A (Q1\ Q2:A \rightarrow hprop) := \forall r, himpl (Q1\ r) (Q2\ r)$.

The core operators of Separation Logic must be provided.

Parameter $hempty : hprop$.

Parameter $hstar : hprop \rightarrow hprop \rightarrow hprop$.

Parameter $hpure : \text{Prop} \rightarrow hprop$.

Parameter $htop : hprop$.

Parameter $hgc : hprop$.

Parameter $hwand : hprop \rightarrow hprop \rightarrow hprop$.

Parameter $qwand : \forall A, (A \rightarrow hprop) \rightarrow (A \rightarrow hprop) \rightarrow hprop$.

Parameter $hexists : \forall A, (A \rightarrow hprop) \rightarrow hprop$.

Parameter $hforall : \forall A, (A \rightarrow hprop) \rightarrow hprop$.

The predicate $haffine$ must be provided. For a fully linear logic, use the always-false predicate. For a fully affine logic, use the always-true predicate.

Parameter $haffine : hprop \rightarrow \text{Prop}$.

33.3.2 Notation

The following notation is used for stating the required properties.

Declare Scope $heap_scope$.

Notation " $H1 ==> H2$ " := ($himpl\ H1\ H2$)
(at level 55) : $heap_scope$.

Notation " $Q1 ==> Q2$ " := ($\text{qimpl}\ Q1\ Q2$)
(at level 55) : $heap_scope$.

Notation " $\set{\cdot}$ " := ($hempty$)
(at level 0) : $heap_scope$.

Notation " \set{P} " := ($hpure\ P$)

```

(at level 0, format "[ P ]") : heap_scope.

Notation "\Top" := (htop) : heap_scope.

Notation "\GC" := (hgc) : heap_scope.

Notation "H1 `*` H2" := (hstar H1 H2)
  (at level 41, right associativity) : heap_scope.

Notation "Q `*+` H" := (fun x => hstar (Q x) H)
  (at level 40) : heap_scope.

Notation "H1 `-*` H2" := (hwand H1 H2)
  (at level 43, right associativity) : heap_scope.

Notation "Q1 `-*` Q2" := (qwand Q1 Q2)
  (at level 43) : heap_scope.

Notation "'\exists' x1 .. xn , H" :=
  (hexists (fun x1 => .. (hexists (fun xn => H) ..)))
  (at level 39, x1 binder, H at level 50, right associativity,
   format "[ '\exists' / ' x1 .. xn , / ' H ]") : heap_scope.

Notation "'\forall' x1 .. xn , H" :=
  (hforall (fun x1 => .. (hforall (fun xn => H) ..)))
  (at level 39, x1 binder, H at level 50, right associativity,
   format "[ '\forall' / ' x1 .. xn , / ' H ]") : heap_scope.

Local Open Scope heap_scope.

```

33.3.3 Properties Assumed by the Functor

The following properties must be satisfied.

Implicit Types $P : \text{Prop}$.
 Implicit Types $H : \text{hprop}$.

Entailment must be an order.

Parameter $\text{himpl_refl} : \forall H,$
 $H ==> H$.

Parameter $\text{himpl_trans} : \forall H2 H1 H3,$
 $(H1 ==> H2) \rightarrow$
 $(H2 ==> H3) \rightarrow$
 $(H1 ==> H3)$.

Parameter $\text{himpl_antisym} : \forall H1 H2,$
 $(H1 ==> H2) \rightarrow$
 $(H2 ==> H1) \rightarrow$
 $(H1 = H2)$.

The star and the empty heap predicate must form a commutative monoid.

```

Parameter hstar_hempty_l : ∀ H,
  [] ∗ H = H.

Parameter hstar_hempty_r : ∀ H,
  H ∗ [] = H.

Parameter hstar_comm : ∀ H1 H2,
  H1 ∗ H2 = H2 ∗ H1.

Parameter hstar_assoc : ∀ H1 H2 H3,
  (H1 ∗ H2) ∗ H3 = H1 ∗ (H2 ∗ H3).

```

The frame property for entailment must hold.

```

Parameter himpl_frame_lr : ∀ H1 H1' H2 H2',
  H1 ==> H1' →
  H2 ==> H2' →
  (H1 ∗ H2) ==> (H1' ∗ H2').

```

```

Parameter himpl_hstar_trans_l : ∀ H1 H2 H3 H4,
  H1 ==> H2 →
  H2 ∗ H3 ==> H4 →
  H1 ∗ H3 ==> H4.

```

Characterization of `hpure`

```

Parameter himpl_hempty_hpure : ∀ P,
  P →
  [] ==> \[P].

```

```

Parameter himpl_hstar_hpure_l : ∀ P H H',
  (P → H ==> H') →
  (\[P] ∗ H) ==> H'.

```

Characterization of `hexists`

```

Parameter himpl_hexists_l : ∀ A H (J:A→hprop),
  (∀ x, J x ==> H) →
  (hexists J) ==> H.

```

```

Parameter himpl_hexists_r : ∀ A (x:A) H J,
  (H ==> J x) →
  H ==> (hexists J).

```

```

Parameter hstar_hexists : ∀ A (J:A→hprop) H,
  (hexists J) ∗ H = hexists (fun x => (J x) ∗ H).

```

Characterization of `hforall`

```

Parameter himpl_hforall_r : ∀ A (J:A→hprop) H,
  (∀ x, H ==> J x) →
  H ==> (hforall J).

```

```

Parameter hstar_hforall : ∀ H A (J:A→hprop),

```

$(\text{hforall } J) \setminus* H ==> \text{hforall } (J \setminus*+ H).$

Characterization of `hwand`

Parameter `hwand_equiv` : $\forall H_0 H_1 H_2,$
 $(H_0 ==> H_1 \setminus* H_2) \leftrightarrow (H_1 \setminus* H_0 ==> H_2).$

Parameter `hwand_curry_eq` : $\forall H_1 H_2 H_3,$
 $(H_1 \setminus* H_2) \setminus* H_3 = H_1 \setminus* (H_2 \setminus* H_3).$

Parameter `hwand_hempty_l` : $\forall H,$
 $(\setminus[] \setminus* H) = H.$

Characterization of `qwand`

Parameter `qwand_equiv` : $\forall H A (Q_1 Q_2 : A \rightarrow \text{hprop}),$
 $H ==> (Q_1 \setminus* Q_2) \leftrightarrow (Q_1 \setminus*+ H) ==> Q_2.$

Parameter `hwand_cancel` : $\forall H_1 H_2,$
 $H_1 \setminus* (H_1 \setminus* H_2) ==> H_2.$

Parameter `qwand_specialize` : $\forall A (x : A) (Q_1 Q_2 : A \rightarrow \text{hprop}),$
 $(Q_1 \setminus* Q_2) ==> (Q_1 x \setminus* Q_2 x).$

Characterization of `htop`

Parameter `himpl_htop_r` : $\forall H,$
 $H ==> \setminus\text{Top}.$

Parameter `hstar_htop_htop` :
 $\setminus\text{Top} \setminus* \setminus\text{Top} = \setminus\text{Top}.$

Characterization of `hgc`

Parameter `haffine_hempty` :
 $\text{haffine} \setminus[].$

Parameter `himpl_hgc_r` : $\forall H,$
 $\text{haffine } H \rightarrow$
 $H ==> \setminus\text{GC}.$

Parameter `hstar_hgc_hgc` :
 $\setminus\text{GC} \setminus* \setminus\text{GC} = \setminus\text{GC}.$

End XSIMPLPARAMS.

33.4 Body of the Functor

When all the assumptions of the functor are statisfied, all the lemmas stated below hold, and all the tactics defined below may be used.

Module XSIMPLSETUP ($HP : \text{XSIMPLPARAMS}$).
Import HP .

Local Open Scope `heap_scope`.

```
Implicit Types  $H : hprop$ .
```

```
Implicit Types  $P : \text{Prop}$ .
```

```
Hint Resolve  $\text{himpl\_refl}$ .
```

33.4.1 Properties of himpl

```
Lemma  $\text{himpl\_of\_eq} : \forall H1 H2,$ 
```

```
 $H1 = H2 \rightarrow$ 
```

```
 $H1 ==> H2$ .
```

```
Proof. intros. subst.  $\text{applys} \neg \text{himpl\_refl}$ . Qed.
```

```
Lemma  $\text{himpl\_of\_eq\_sym} : \forall H1 H2,$ 
```

```
 $H1 = H2 \rightarrow$ 
```

```
 $H2 ==> H1$ .
```

```
Proof. intros. subst.  $\text{applys} \neg \text{himpl\_refl}$ . Qed.
```

33.4.2 Properties of qimpl

```
Lemma  $\text{qimpl\_refl} : \forall A (Q:A \rightarrow hprop),$ 
```

```
 $Q ==> Q$ .
```

```
Proof using. intros.  $\text{hnfs} \times$ . Qed.
```

```
Hint Resolve  $\text{qimpl\_refl}$ .
```

```
Lemma  $\text{qimpl\_trans} : \forall A (Q2 Q1 Q3:A \rightarrow hprop),$ 
```

```
 $(Q1 ==> Q2) \rightarrow$ 
```

```
 $(Q2 ==> Q3) \rightarrow$ 
```

```
 $(Q1 ==> Q3)$ .
```

```
Proof using.  $\text{introv} M1 M2$ .  $\text{intros} v$ .  $\text{applys} \times \text{himpl\_trans}$ . Qed.
```

```
Lemma  $\text{qimpl\_antisym} : \forall A (Q1 Q2:A \rightarrow hprop),$ 
```

```
 $(Q1 ==> Q2) \rightarrow$ 
```

```
 $(Q2 ==> Q1) \rightarrow$ 
```

```
 $(Q1 = Q2)$ .
```

```
Proof using.  $\text{introv} M1 M2$ .  $\text{apply fun\_ext\_1}$ .  $\text{intros} v$ .  $\text{applys} \times \text{himpl\_antisym}$ . Qed.
```

33.4.3 Properties of hstar

```
Lemma  $\text{hstar\_comm\_assoc} : \forall H1 H2 H3,$ 
```

```
 $H1 \ast H2 \ast H3 = H2 \ast H1 \ast H3$ .
```

```
Proof using.
```

```
 $\text{intros. rewrite} \leftarrow \text{hstar\_assoc}$ .
```

```
 $\text{rewrite} (@\text{hstar\_comm} H1 H2). \text{rewrite} \neg \text{hstar\_assoc}$ .
```

```
Qed.
```

33.4.4 Representation Predicates

The arrow notation typically takes the form $x \sim > R x$, to indicate that X is the logical value that describes the heap structure x , according to the representation predicate R . It is just a notation for the heap predicate $R X x$.

```
Definition repr (A:Type) (S:A→hprop) (x:A) : hprop :=  
  S x.
```

```
Notation "x 'sim >' S" := (repr S x)  
  (at level 33, no associativity) : heap_scope.
```

```
Lemma repr_eq : ∀ (A:Type) (S:A→hprop) (x:A),  
  (x ~> S) = (S x).
```

Proof using. auto. Qed.

$x \sim > \text{Id } X$ holds when x is equal to X in the empty heap. Id is called the identity representation predicate.

```
Definition Id {A:Type} (X:A) (x:A) :=  
  \[ X = x ].
```

$x\text{repr_clean}$ simplifies instances of $p \sim > (\text{fun } _ \Rightarrow _)$ by unfolding the arrow, but only when the body does not capture locally bound variables. This tactic should normally not be used directly

```
Ltac xrepr_clean_core tt :=  
  repeat match goal with  
    | ⊢ context C [?p ~> ?E] ⇒  
      match E with (fun _ ⇒ _) ⇒  
        let E' := eval cbv beta in (E p) in  
        let G' := context C [E'] in  
        let G := match goal with  
          | ⊢ ?G ⇒ G end in  
          change G with G' end end.
```

```
Tactic Notation "xrepr_clean" :=  
  xrepr_clean_core tt.
```

```
Lemma repr_id : ∀ A (x X:A),  
  (x ~> \[X = x]).
```

Proof using. intros. unfold Id. xrepr_clean. auto. Qed.

33.4.5 *rew_heap* Tactic to Normalize Expressions with **hstar**

rew_heap removes empty heap predicates, and normalizes w.r.t. associativity of star.

rew_heap_assoc only normalizes w.r.t. associativity. (It should only be used internally for tactic implementation.)

```
Lemma star_post_empty : ∀ B (Q:B→hprop),  
  Q \*+ \[] = Q.
```

Proof using. extens. intros. rewrite× hstar_hempty_r. Qed.

```

Hint Rewrite hstar_hempty_l hstar_hempty_r
    hstar_assoc star_post_empty hwand_hempty_l : rew_heap.

Tactic Notation "rew_heap" :=
    autorewrite with rew_heap.

Tactic Notation "rew_heap" "in" "*" :=
    autorewrite with rew_heap in *.

Tactic Notation "rew_heap" "in" hyp(H) :=
    autorewrite with rew_heap in H.

Tactic Notation "rew_heap" "˜" :=
    rew_heap; auto_tilde.

Tactic Notation "rew_heap" "˜" "in" "*" :=
    rew_heap in *; auto_tilde.

Tactic Notation "rew_heap" "˜" "in" hyp(H) :=
    rew_heap in H; auto_tilde.

Tactic Notation "rew_heap" "*" :=
    rew_heap; auto_star.

Tactic Notation "rew_heap" "*" "in" "*" :=
    rew_heap in *; auto_star.

Tactic Notation "rew_heap" "*" "in" hyp(H) :=
    rew_heap in H; auto_star.

Hint Rewrite hstar_assoc : rew_heap_assoc.

Tactic Notation "rew_heap_assoc" :=
    autorewrite with rew_heap_assoc.

```

33.4.6 Auxiliary Tactics Used by *xpull* and *xsimpl*

```

Ltac remove_empty_heaps_from H :=
    match H with context[ ?H1 \* \[] ] =>
        match is_evar_as_bool H1 with
            | false => rewrite (@hstar_hempty_r H1)
            | true => let X := fresh in
                set (X := H1);
                rewrite (@hstar_hempty_r X);
                subst X
        end end.

Ltac remove_empty_heaps_haffine tt :=
    repeat match goal with ⊢ haffine ?H => remove_empty_heaps_from H end.

Ltac remove_empty_heaps_left tt :=
    repeat match goal with ⊢ ?H1 ==> _ => remove_empty_heaps_from H1 end.

Ltac remove_empty_heaps_right tt :=

```

```
repeat match goal with  $\vdash \_ ==> ?H2$   $\Rightarrow$  remove_empty_heaps_from  $H2$  end.
```

33.4.7 Tactics *ximpl* and *xpull* for Heap Entailments

The implementation of the tactics is fairly involved. The high-level specification of the tactic appears in the last appendix of: http://www.chargueraud.org/research/2020/seq_seplogic/seq_seplogic.pdf

xaffine placeholder

```
Ltac xaffine_core tt :=
  try solve [ assumption | apply haffine_hempty ].
```

```
Tactic Notation "xaffine" :=
  xaffine_core tt.
```

Hints for tactics such as *ximpl*

```
Inductive Ximpl_hint : list Boxer  $\rightarrow$  Type :=
| ximpl_hint :  $\forall (L:\text{list Boxer}), \text{Ximpl\_hint } L$ .
```

```
Ltac ximpl_hint_put L :=
  let H := fresh "Hint" in
  generalize (ximpl_hint L); intros H.
```

```
Ltac ximpl_hint_next cont :=
  match goal with H: Ximpl_hint ((boxer ?x)::?L)  $\vdash \_ \Rightarrow$ 
    clear H; ximpl_hint_put L; cont x end.
```

```
Ltac ximpl_hint_remove tt :=
  match goal with H: Ximpl_hint  $\_ \vdash \_ \Rightarrow$  clear H end.
```

Lemmas *hstars_reorder_...* to flip an iterated *hstar*.

hstars_flip tt applies to a goal of the form $H1 \ast \dots \ast Hn \ast _ \equiv ?H$ and instantiates *H* with $Hn \ast \dots \ast H1 \ast _ \equiv _$. If $n > 9$, the maximum arity supported, the tactic unifies *H* with the original LHS.

```
Lemma hstars_flip_0 :
```

```
   $\_ = \_$ .
```

```
Proof using. auto. Qed.
```

```
Lemma hstars_flip_1 :  $\forall H1,$ 
   $H1 \ast \_ = H1 \ast \_$ .
```

```
Proof using. auto. Qed.
```

```
Lemma hstars_flip_2 :  $\forall H1 H2,$ 
   $H1 \ast H2 \ast \_ = H2 \ast H1 \ast \_$ .
```

Proof using. intros. *rew_heap*. *rewrite* (*hstar_comm H2*). *rew_heap*¬. Qed.

Lemma hstars_flip_3 : $\forall H1 H2 H3,$

$$H1 \ast H2 \ast H3 \ast [] = H3 \ast H2 \ast H1 \ast [].$$

Proof using. intros. *rewrite* \leftarrow (*hstars_flip_2 H1*). *rew_heap*. *rewrite* (*hstar_comm H3*). *rew_heap*¬. Qed.

Lemma hstars_flip_4 : $\forall H1 H2 H3 H4,$

$$H1 \ast H2 \ast H3 \ast H4 \ast [] = H4 \ast H3 \ast H2 \ast H1 \ast [].$$

Proof using. intros. *rewrite* \leftarrow (*hstars_flip_3 H1*). *rew_heap*. *rewrite* (*hstar_comm H4*). *rew_heap*¬. Qed.

Lemma hstars_flip_5 : $\forall H1 H2 H3 H4 H5,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast [] = H5 \ast H4 \ast H3 \ast H2 \ast H1 \ast [].$$

Proof using. intros. *rewrite* \leftarrow (*hstars_flip_4 H1*). *rew_heap*. *rewrite* (*hstar_comm H5*). *rew_heap*¬. Qed.

Lemma hstars_flip_6 : $\forall H1 H2 H3 H4 H5 H6,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast []$$

$$= H6 \ast H5 \ast H4 \ast H3 \ast H2 \ast H1 \ast [].$$

Proof using. intros. *rewrite* \leftarrow (*hstars_flip_5 H1*). *rew_heap*. *rewrite* (*hstar_comm H6*). *rew_heap*¬. Qed.

Lemma hstars_flip_7 : $\forall H1 H2 H3 H4 H5 H6 H7,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast []$$

$$= H7 \ast H6 \ast H5 \ast H4 \ast H3 \ast H2 \ast H1 \ast [].$$

Proof using. intros. *rewrite* \leftarrow (*hstars_flip_6 H1*). *rew_heap*. *rewrite* (*hstar_comm H7*). *rew_heap*¬. Qed.

Lemma hstars_flip_8 : $\forall H1 H2 H3 H4 H5 H6 H7 H8,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8 \ast []$$

$$= H8 \ast H7 \ast H6 \ast H5 \ast H4 \ast H3 \ast H2 \ast H1 \ast [].$$

Proof using. intros. *rewrite* \leftarrow (*hstars_flip_7 H1*). *rew_heap*. *rewrite* (*hstar_comm H8*). *rew_heap*¬. Qed.

Lemma hstars_flip_9 : $\forall H1 H2 H3 H4 H5 H6 H7 H8 H9,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8 \ast H9 \ast []$$

$$= H9 \ast H8 \ast H7 \ast H6 \ast H5 \ast H4 \ast H3 \ast H2 \ast H1 \ast [].$$

Proof using. intros. *rewrite* \leftarrow (*hstars_flip_8 H1*). *rew_heap*. *rewrite* (*hstar_comm H9*). *rew_heap*¬. Qed.

Ltac *hstars_flip_lemma i* :=

match *number_to_nat i* with

| 0%nat \Rightarrow constr:(*hstars_flip_0*)

| 1%nat \Rightarrow constr:(*hstars_flip_1*)

| 2%nat \Rightarrow constr:(*hstars_flip_2*)

| 3%nat \Rightarrow constr:(*hstars_flip_3*)

| 4%nat \Rightarrow constr:(*hstars_flip_4*)

```

| 5%nat ⇒ constr:(hstars_flip_5)
| 6%nat ⇒ constr:(hstars_flip_6)
| 7%nat ⇒ constr:(hstars_flip_7)
| 8%nat ⇒ constr:(hstars_flip_8)
| 9%nat ⇒ constr:(hstars_flip_9)
| _ ⇒ constr:(hstars_flip_1)
end.

Ltac hstars_arity i Hs :=
  match Hs with
  | [] ⇒ constr:(i)
  | ?H1 ∗ ?H2 ⇒ hstars_arity (S i) H2
  end.

Ltac hstars_flip_arity tt :=
  match goal with ⊢ ?HL = ?HR ⇒ hstars_arity 0%nat HL end.

Ltac hstars_flip tt :=
  let i := hstars_arity tt in
  let L := hstars_flip_lemma i in
  eapply L.

```

Lemmas *hstars_pick_...* to extract hyps in depth.

hstars_search Hs test applies to an expression *Hs* of the form either $H_1 ∗ … ∗ H_n ∗ []$ or $H_1 ∗ … ∗ H_n$. It invokes the function *test i Hi* for each of the *Hi* in turn until the tactic succeeds. In the particular case of invoking *test n Hn* when there is no trailing $[]$, the call is of the form *test (hstars_last n) Hn* where *hstars_last* is an identity tag.

Definition *hstars_last* (*A*:Type) (*X*:*A*) := *X*.

```

Ltac hstars_search Hs test :=
  let rec aux i Hs :=
    first [ match Hs with ?H ∗ _ ⇒ test i H end
           | match Hs with _ ∗ ?H' ⇒ aux (S i) H' end
           | match Hs with ?H ⇒ test (hstars_last i) H end ] in
  aux 1%nat Hs.

```

hstars_pick_lemma i returns one of the lemma below, which enables reordering in iterated stars, by extracting the *i*-th item to bring it to the front.

Lemma *hstars_pick_1* : $\forall H_1 H,$

$$H_1 ∗ H = H_1 ∗ H.$$

Proof using. auto. Qed.

Lemma *hstars_pick_2* : $\forall H_1 H_2 H,$

$$H_1 ∗ H_2 ∗ H = H_2 ∗ H_1 ∗ H.$$

Proof using. intros. rewrite ← (hstar_comm_assoc *H1*). Qed.

Lemma hstars_pick_3 : $\forall H1 H2 H3 H,$

$$H1 \ast H2 \ast H3 \ast H = H3 \ast H1 \ast H2 \ast H.$$

Proof using. intros. rewrite \neg (hstar_comm_assoc H2). *applys* hstars_pick_2. Qed.

Lemma hstars_pick_4 : $\forall H1 H2 H3 H4 H,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H = H4 \ast H1 \ast H2 \ast H3 \ast H.$$

Proof using. intros. rewrite \neg (hstar_comm_assoc H3). *applys* hstars_pick_3. Qed.

Lemma hstars_pick_5 : $\forall H1 H2 H3 H4 H5 H,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H = H5 \ast H1 \ast H2 \ast H3 \ast H4 \ast H.$$

Proof using. intros. rewrite \neg (hstar_comm_assoc H4). *applys* hstars_pick_4. Qed.

Lemma hstars_pick_6 : $\forall H1 H2 H3 H4 H5 H6 H,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H$$

$$= H6 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H.$$

Proof using. intros. rewrite \neg (hstar_comm_assoc H5). *applys* hstars_pick_5. Qed.

Lemma hstars_pick_7 : $\forall H1 H2 H3 H4 H5 H6 H7 H,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H$$

$$= H7 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H.$$

Proof using. intros. rewrite \neg (hstar_comm_assoc H6). *applys* hstars_pick_6. Qed.

Lemma hstars_pick_8 : $\forall H1 H2 H3 H4 H5 H6 H7 H8 H,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8 \ast H$$

$$= H8 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H.$$

Proof using. intros. rewrite \neg (hstar_comm_assoc H7). *applys* hstars_pick_7. Qed.

Lemma hstars_pick_9 : $\forall H1 H2 H3 H4 H5 H6 H7 H8 H9 H,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8 \ast H9 \ast H$$

$$= H9 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8 \ast H.$$

Proof using. intros. rewrite \neg (hstar_comm_assoc H8). *applys* hstars_pick_8. Qed.

Lemma hstars_pick_last_1 : $\forall H1,$

$$H1 = H1.$$

Proof using. auto. Qed.

Lemma hstars_pick_last_2 : $\forall H1 H2,$

$$H1 \ast H2 = H2 \ast H1.$$

Proof using. intros. rewrite \neg (hstar_comm). Qed.

Lemma hstars_pick_last_3 : $\forall H1 H2 H3,$

$$H1 \ast H2 \ast H3 = H3 \ast H1 \ast H2.$$

Proof using. intros. rewrite \neg (hstar_comm H2). *applys* hstars_pick_2. Qed.

Lemma hstars_pick_last_4 : $\forall H1 H2 H3 H4,$

$$H1 \ast H2 \ast H3 \ast H4 = H4 \ast H1 \ast H2 \ast H3.$$

Proof using. intros. rewrite \neg (hstar_comm H3). *applys* hstars_pick_3. Qed.

Lemma hstars_pick_last_5 : $\forall H1 H2 H3 H4 H5,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 = H5 \ast H1 \ast H2 \ast H3 \ast H4.$$

Proof using. intros. rewrite \neg (*hstar_comm H4*). applys *hstars_pick_4*. Qed.

Lemma *hstars_pick_last_6* : $\forall H1 H2 H3 H4 H5 H6,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6$$

$$= H6 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5.$$

Proof using. intros. rewrite \neg (*hstar_comm H5*). applys *hstars_pick_5*. Qed.

Lemma *hstars_pick_last_7* : $\forall H1 H2 H3 H4 H5 H6 H7,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7$$

$$= H7 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6.$$

Proof using. intros. rewrite \neg (*hstar_comm H6*). applys *hstars_pick_6*. Qed.

Lemma *hstars_pick_last_8* : $\forall H1 H2 H3 H4 H5 H6 H7 H8,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8$$

$$= H8 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7.$$

Proof using. intros. rewrite \neg (*hstar_comm H7*). applys *hstars_pick_7*. Qed.

Lemma *hstars_pick_last_9* : $\forall H1 H2 H3 H4 H5 H6 H7 H8 H9,$

$$H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8 \ast H9$$

$$= H9 \ast H1 \ast H2 \ast H3 \ast H4 \ast H5 \ast H6 \ast H7 \ast H8.$$

Proof using. intros. rewrite \neg (*hstar_comm H8*). applys *hstars_pick_8*. Qed.

Ltac *hstars_pick_lemma i* :=

```
let unsupported tt := fail 100 "hstars_pick supports only arity up to 9" in
match i with
| hstars_last ?j => match number_to_nat j with
  | 1%nat => constr:(hstars_pick_last_1)
  | 2%nat => constr:(hstars_pick_last_2)
  | 3%nat => constr:(hstars_pick_last_3)
  | 4%nat => constr:(hstars_pick_last_4)
  | 5%nat => constr:(hstars_pick_last_5)
  | 6%nat => constr:(hstars_pick_last_6)
  | 7%nat => constr:(hstars_pick_last_7)
  | 8%nat => constr:(hstars_pick_last_8)
  | 9%nat => constr:(hstars_pick_last_9)
  | _ => unsupported tt
end
| ?j => match number_to_nat j with
  | 1%nat => constr:(hstars_pick_1)
  | 2%nat => constr:(hstars_pick_2)
  | 3%nat => constr:(hstars_pick_3)
  | 4%nat => constr:(hstars_pick_4)
  | 5%nat => constr:(hstars_pick_5)
  | 6%nat => constr:(hstars_pick_6)
  | 7%nat => constr:(hstars_pick_7)
  | 8%nat => constr:(hstars_pick_8)
```

```

| 9%nat ⇒ constr:(hstars_pick_9)
| _ ⇒ unsupported tt
end
end.

```

Documentation for the Tactic *xsimpl*

... doc for *xsimpl* to update

At the end, there remains a heap entailment with a simplified LHS and a simplified RHS, with items not cancelled out. At this point, if the goal is of the form $H ==> \GC$ or $H ==> \Top$ or $H ==> ?H'$ for some evar H' , then *xsimpl* solves the goal. Else, it leaves whatever remains.

For the cancellation part, *xsimpl* cancels out H from the LHS with H' from the RHS if either H' is syntactically equal to H , or if H and H' both have the form $x \sim > ...$ for the same x . Note that, in case of a mismatch with $x \sim > R X$ on the LHS and $x \sim > R' X'$ on the RHS, *xsimpl* will produce a goal of the form $(x \sim > R X) = (x \sim > R' X')$ which will likely be unsolvable. It is the user's responsibility to perform the appropriate conversion steps prior to calling *xsimpl*.

Remark: the reason for the special treatment of $x \sim > ...$ is that it is very useful to be able to automatically cancel out $x \sim > R X$ from the LHS with $x \sim > R ?Y$, for some evar $?Y$ which typically is introduced from an existential, e.g. $\exists Y, x \sim > R Y$.

Remark: importantly, *xsimpl* does not attempt any simplification on a representation predicate of the form $?x \sim > ...$, when the $?x$ is an uninstantiated evar. Such situation may arise for example with the following RHS: $\exists p, (r \sim > Ref p) \ast (p \sim > Ref 3)$.

As a special feature, *xsimpl* may be provided optional arguments for instantiating the existentials (instead of introducing evars). These optional arguments need to be given in left-right order, and are used on a first-match basis: the head value is used if its type matches the type expected by the existential, else an evar is introduced for that existential.

$\text{Xsimpl } (Hla, Hlw, Hlt) (Hra, Hrg, Hrt)$ is interepreted as the entailment $Hla \ast Hlw \ast Hlt ==> Hra \ast Hrg \ast Hrt$ where

- Hla denotes “cleaned up” items from the left hand side
- Hlw denotes the $H1 \dashv\ast H2$ and $Q1 \dashv\ast Q2$ items from the left hand side
- Hlt denotes the remaining items to process items from the left hand side
- Hra denotes “cleaned up” items from the right hand side
- Hrg denotes the \GC and \Top items from the right hand side
- Hrt denotes the remaining items to process from the right hand side

Note: we assume that all items consist of iterated hstars, and are always terminated by an empty heap.

```

Definition Xsimpl (HL HR:hprop×hprop×hprop) :=
  let '(Hla,Hlw,Hlt) := HL in
  let '(Hra,Hrg,Hrt) := HR in
  Hla \* Hlw \* Hlt ==> Hra \* Hrg \* Hrt.

```

protect X is use to prevent $xsimpl$ from investigating inside X

```
Definition protect (A:Type) (X:A) : A := X.
```

Auxiliary lemmas to prove lemmas for $xsimpl$ implementation.

```

Lemma Xsimpl_trans_l : ∀ Hla1 Hlw1 Hlt1 Hla2 Hlw2 Hlt2 HR,
  Xsimpl (Hla2 , Hlw2 , Hlt2) HR →
  Hla1 \* Hlw1 \* Hlt1 ==> Hla2 \* Hlw2 \* Hlt2 →
  Xsimpl (Hla1 , Hlw1 , Hlt1) HR.

```

Proof using.

*introv M1 E. destruct HR as [[Hra Hrg] Hrt]. unfolds Xsimpl.
applys× himpl_trans M1.*

Qed.

```

Lemma Xsimpl_trans_r : ∀ Hra1 Hrg1 Hrt1 Hra2 Hrg2 Hrt2 HL,
  Xsimpl HL (Hra2 , Hrg2 , Hrt2) →
  Hra2 \* Hrg2 \* Hrt2 ==> Hra1 \* Hrg1 \* Hrt1 →
  Xsimpl HL (Hra1 , Hrg1 , Hrt1).

```

Proof using.

*introv M1 E. destruct HL as [[Hla Hlw] Hlt]. unfolds Xsimpl.
applys× himpl_trans M1.*

Qed.

```

Lemma Xsimpl_trans : ∀ Hla1 Hlw1 Hlt1 Hla2 Hlw2 Hlt2 Hra1 Hrg1 Hrt1 Hra2 Hrg2 Hrt2 ,
  Xsimpl (Hla2 , Hlw2 , Hlt2) (Hra2 , Hrg2 , Hrt2) →
  (Hla2 \* Hlw2 \* Hlt2 ==> Hra2 \* Hrg2 \* Hrt2 →
   Hla1 \* Hlw1 \* Hlt1 ==> Hra1 \* Hrg1 \* Hrt1) →
  Xsimpl (Hla1 , Hlw1 , Hlt1) (Hra1 , Hrg1 , Hrt1).

```

Proof using. *introv M1 E. unfolds Xsimpl. eauto. Qed.*

Basic cancellation tactic used to establish lemmas used by $xsimpl$

```

Lemma hstars_simpl_start : ∀ H1 H2,
  H1 \* [] ==> [] \* H2 \* [] →
  H1 ==> H2.

```

Proof using. *introv M. rew_heap¬ in *. Qed.*

```

Lemma hstars_simpl_keep : ∀ H1 Ha H Ht,
  H1 ==> (H \* Ha) \* Ht →
  H1 ==> Ha \* H \* Ht.

```

Proof using. *introv M. rew_heap in *. rewrite¬ hstar_comm_assoc. Qed.*

```

Lemma hstars_simpl_cancel : ∀ H1 Ha H Ht,
  H1 ==> Ha \* Ht →
  H \* H1 ==> Ha \* H \* Ht.
Proof using. introv M. rewrite hstar_comm_assoc. applys¬ himpl_frame_lr. Qed.

Lemma hstars_simpl_pick_lemma : ∀ H1 H1' H2,
  H1 = H1' →
  H1' ==> H2 →
  H1 ==> H2.
Proof using. introv M. subst¬. Qed.

Ltac hstars_simpl_pick i :=
  let L := hstars_pick_lemma i in
  eapply hstars_simpl_pick_lemma; [ apply L | ].

Ltac hstars_simpl_start tt :=
  match goal with ⊢ ?H1 ==> ?H2 ⇒ idtac end;
  applys hstars_simpl_start;
  rew_heap_assoc.

Ltac hstars_simpl_step tt :=
  match goal with ⊢ ?Hl ==> ?Ha \* ?H \* ?H2 ⇒
    first [
      hstars_search Hl ltac:(fun i H' ⇒
        match H' with H ⇒ hstars_simpl_pick i end);
      apply hstars_simpl_cancel
      | apply hstars_simpl_keep ]
    end.
  end.

Ltac hstars_simpl_post tt :=
  rew_heap; try apply himpl_refl.

Ltac hstars_simpl_core tt :=
  hstars_simpl_start tt;
  repeat (hstars_simpl_step tt);
  hstars_simpl_post tt.

Tactic Notation "hstars_simpl" :=
  hstars_simpl_core tt.

```

Transition lemmas

Transition lemmas to start the process

```

Lemma xpull_protect : ∀ H1 H2,
  H1 ==> protect H2 →
  H1 ==> H2.
Proof using. auto. Qed.

```

Lemma xsimpl_start : $\forall H1 H2,$
 $\text{Xsimpl}(\text{\textbackslash}[], \text{\textbackslash}[], (H1 \ast \text{\textbackslash}[])) (\text{\textbackslash}[], \text{\textbackslash}[], (H2 \ast \text{\textbackslash}[])) \rightarrow$
 $H1 ==> H2.$

Proof using. *introv M. unfolds Xsimpl. rew_heap¬ in *.* Qed.

Transition lemmas for LHS extraction operations

Ltac xsimpl_l_start M :=
introv M;
match goal with $HR: hprop \times hprop \times hprop \vdash _ \Rightarrow$
destruct HR as $[[Hra\ Hrg]\ Hrt];$ *unfolds Xsimpl end.*

Ltac xsimpl_l_start' M :=
 $xsimpl_l_start\ M;$ *applys himpl_trans (rm M); hstars_simpl.*

Lemma xsimpl_l_hempty : $\forall Hla\ Hlw\ Hlt\ HR,$
 $\text{Xsimpl}(Hla, Hlw, Hlt) HR \rightarrow$
 $\text{Xsimpl}(Hla, Hlw, (\text{\textbackslash}[] \ast Hlt)) HR.$

Proof using. $xsimpl_l_start'\ M.$ Qed.

Lemma xsimpl_l_hpure : $\forall P\ Hla\ Hlw\ Hlt\ HR,$
 $(P \rightarrow \text{Xsimpl}(Hla, Hlw, Hlt) HR) \rightarrow$
 $\text{Xsimpl}(Hla, Hlw, (\text{\textbackslash}[P] \ast Hlt)) HR.$

Proof using.

$xsimpl_l_start\ M.$ *rewrite hstars_pick_3. applys× himpl_hstar_hpure_l.*

Qed.

Lemma xsimpl_l_hexists : $\forall A (J:A \rightarrow hprop)\ Hla\ Hlw\ Hlt\ HR,$
 $(\forall x, \text{Xsimpl}(Hla, Hlw, (J x \ast Hlt)) HR) \rightarrow$
 $\text{Xsimpl}(Hla, Hlw, (\text{hexists } J \ast Hlt)) HR.$

Proof using.

$xsimpl_l_start\ M.$ *rewrite hstars_pick_3. rewrite hstar_hexists.*
applys× himpl_hexists_l. intros. rewrite¬ ← hstars_pick_3.

Qed.

Lemma xsimpl_l_acc_wand : $\forall H\ Hla\ Hlw\ Hlt\ HR,$
 $\text{Xsimpl}(Hla, (H \ast Hlw), Hlt) HR \rightarrow$
 $\text{Xsimpl}(Hla, Hlw, (H \ast Hlt)) HR.$

Proof using. $xsimpl_l_start'\ M.$ Qed.

Lemma xsimpl_l_acc_other : $\forall H\ Hla\ Hlw\ Hlt\ HR,$
 $\text{Xsimpl}((H \ast Hla), Hlw, Hlt) HR \rightarrow$
 $\text{Xsimpl}(Hla, Hlw, (H \ast Hlt)) HR.$

Proof using. $xsimpl_l_start'\ M.$ Qed.

Transition lemmas for LHS cancellation operations Hlt is meant to be empty there

Lemma xsimpl_l_cancel_hwand_hempty : $\forall H2\ Hla\ Hlw\ Hlt\ HR,$
 $\text{Xsimpl}(Hla, Hlw, (H2 \ast Hlt)) HR \rightarrow$
 $\text{Xsimpl}(Hla, ((\text{\textbackslash}[] \ast H2) \ast Hlw), Hlt) HR.$

Proof using. *xsimpl_l_start'* M. Qed.

Lemma *xsimpl_l_cancel_hwand* : $\forall H1 H2 Hla Hlw Hlt HR,$
 $\text{Xsimpl } (\text{[]}, Hlw, (Hla \ast H2 \ast Hlt)) HR \rightarrow$
 $\text{Xsimpl } ((H1 \ast Hla), ((H1 \dashv\ast H2) \ast Hlw), Hlt) HR.$

Proof using. *xsimpl_l_start'* M. *applys* *hwand_cancel*. Qed.

Lemma *xsimpl_l_cancel_qwand* : $\forall A (x:A) (Q1 Q2:A \rightarrow hprop) Hla Hlw Hlt HR,$
 $\text{Xsimpl } (\text{[]}, Hlw, (Hla \ast Q2 x \ast Hlt)) HR \rightarrow$
 $\text{Xsimpl } ((Q1 x \ast Hla), ((Q1 \dashv\ast Q2) \ast Hlw), Hlt) HR.$

Proof using.

xsimpl_l_start' M. *rewrite hstar_comm*. *applys himpl_hstar_trans_l*.
applys qwand_specialize x. rewrite hstar_comm. *applys hwand_cancel*.

Qed.

Lemma *xsimpl_l_keep_wand* : $\forall H Hla Hlw Hlt HR,$
 $\text{Xsimpl } ((H \ast Hla), Hlw, Hlt) HR \rightarrow$
 $\text{Xsimpl } (Hla, (H \ast Hlw), Hlt) HR.$

Proof using. *xsimpl_l_start'* M. Qed.

Lemma *xsimpl_l_hwand_reorder* : $\forall H1 H1' H2 Hla Hlw Hlt HR,$
 $H1 = H1' \rightarrow$
 $\text{Xsimpl } (Hla, ((H1' \dashv\ast H2) \ast Hlw), Hlt) HR \rightarrow$
 $\text{Xsimpl } (Hla, ((H1 \dashv\ast H2) \ast Hlw), Hlt) HR.$

Proof using. *intros. subst x*. Qed.

Lemma *xsimpl_l_cancel_hwand_hstar* : $\forall H1 H2 H3 Hla Hlw Hlt HR,$
 $\text{Xsimpl } (Hla, Hlw, ((H2 \dashv\ast H3) \ast Hlt)) HR \rightarrow$
 $\text{Xsimpl } ((H1 \ast Hla), (((H1 \ast H2) \dashv\ast H3) \ast Hlw), Hlt) HR.$

Proof using.

xsimpl_l_start' M. *rewrite hwand_curry_eq*. *applys hwand_cancel*.

Qed.

Lemma *xsimpl_l_cancel_hwand_hstar_hempty* : $\forall H2 H3 Hla Hlw Hlt HR,$
 $\text{Xsimpl } (Hla, Hlw, ((H2 \dashv\ast H3) \ast Hlt)) HR \rightarrow$
 $\text{Xsimpl } (Hla, (((\text{[]} \ast H2) \dashv\ast H3) \ast Hlw), Hlt) HR.$

Proof using. *xsimpl_l_start'* M. Qed.

Transition lemmas for RHS extraction operations

Ltac *xsimpl_r_start* M :=
introv M;
match goal with HL: hprop × hprop × hprop ⊢ _ ⇒
destruct HL as [[Hla Hlw] Hlt]; unfolds Xsimpl end.

Ltac *xsimpl_r_start'* M :=
xsimpl_r_start M; applys himpl_trans (rm M); hstars_simpl.

Lemma *xsimpl_r_hempty* : $\forall Hra Hrg Hrt HL,$
 $\text{Xsimpl } HL (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, (\lambda [] \ * Hrt)).$

Proof using. $\text{xsimpl_r_start}' M$. Qed.

Lemma $\text{xsimpl_r_hwand_same} : \forall H \ Hra \ Hrg \ Hrt \ HL,$

$\text{Xsimpl } HL \ (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, ((H \ \setminus\ast H) \ * Hrt)).$

Proof using. $\text{xsimpl_r_start}' M$. rewrite hwand_equiv . $\text{rew_heap}\neg$. Qed.

Lemma $\text{xsimpl_r_hpure} : \forall P \ Hra \ Hrg \ Hrt \ HL,$

$P \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, (\lambda [P] \ * Hrt)).$

Proof using.

$\text{introv } HP. \text{ xsimpl_r_start}' M. \text{ applys} \times \text{ himpl_hempty_hpure}.$

Qed.

Lemma $\text{xsimpl_r_hexists} : \forall A \ (x:A) \ (J:A \rightarrow \text{hprop}) \ Hra \ Hrg \ Hrt \ HL,$

$\text{Xsimpl } HL \ (Hra, Hrg, (J x \ * Hrt)) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, (\text{hexists } J \ * Hrt)).$

Proof using. $\text{xsimpl_r_start}' M. \text{ applys} \times \text{ himpl_hexists_r}$. Qed.

Lemma $\text{xsimpl_r_id} : \forall A \ (x X:A) \ Hra \ Hrg \ Hrt \ HL,$

$(X = x) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, (x \simgt \text{Id } X \ * Hrt)).$

Proof using.

$\text{introv } \rightarrow. \text{ xsimpl_r_start}' M. \text{ rewrite repr_id}.$

$\text{applys} \times \text{ himpl_hempty_hpure}.$

Qed.

Lemma $\text{xsimpl_r_id_unify} : \forall A \ (x:A) \ Hra \ Hrg \ Hrt \ HL,$

$\text{Xsimpl } HL \ (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, (x \simgt \text{Id } x \ * Hrt)).$

Proof using. $\text{introv } M. \text{ applys} \neg \text{ xsimpl_r_id}$. Qed.

Lemma $\text{xsimpl_r_keep} : \forall H \ Hra \ Hrg \ Hrt \ HL,$

$\text{Xsimpl } HL \ ((H \ * Hra), Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, (H \ * Hrt)).$

Proof using. $\text{xsimpl_r_start}' M$. Qed.

Transition lemmas for $\setminus Top$ and $\setminus GC$ cancellation

Lemma $\text{xsimpl_r_hgc_or_htop} : \forall H \ Hra \ Hrg \ Hrt \ HL,$

$\text{Xsimpl } HL \ (Hra, (H \ * Hrg), Hrt) \rightarrow$

$\text{Xsimpl } HL \ (Hra, Hrg, (H \ * Hrt)).$

Proof using. $\text{xsimpl_r_start}' M$. Qed.

Lemma $\text{xsimpl_r_htop_replace_hgc} : \forall Hra \ Hrg \ Hrt \ HL,$

$\text{Xsimpl } HL \ (Hra, (\setminus Top \ * Hrg), Hrt) \rightarrow$

$\text{Xsimpl } HL (Hra, (\text{\textbackslash GC } \ast Hrg), (\text{\textbackslash Top } \ast Hrt)).$

Proof using. $\text{ximpl_r_start}' M.$ applics $\text{impl_hgc_r}.$ xaffine. Qed.

Lemma $\text{ximpl_r_hgc_drop} : \forall Hra Hrg Hrt HL,$

$\text{Xsimpl } HL (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL (Hra, Hrg, (\text{\textbackslash GC } \ast Hrt)).$

Proof using. $\text{ximpl_r_start}' M.$ applics $\text{impl_hgc_r}.$ xaffine. Qed.

Lemma $\text{ximpl_r_htop_drop} : \forall Hra Hrg Hrt HL,$

$\text{Xsimpl } HL (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } HL (Hra, Hrg, (\text{\textbackslash Top } \ast Hrt)).$

Proof using. $\text{ximpl_r_start}' M.$ applics $\text{impl_htop_r}.$ Qed.

Transition lemmas for LHS/RHS cancellation meant to be applied when Hlw and Hlt are empty

Ltac $\text{ximpl_lr_start } M :=$

$\text{introv } M; \text{ unfolds } \text{Xsimpl}; \text{ rew_heap in } *.$

Ltac $\text{ximpl_lr_start}' M :=$

$\text{ximpl_lr_start } M; \text{ hstars_simpl};$

$\text{try (applys } \text{impl_trans} (\text{rm } M); \text{ hstars_simpl}).$

Lemma $\text{ximpl_lr_cancel_same} : \forall H Hla Hlw Hlt Hra Hrg Hrt,$

$\text{Xsimpl } (Hla, Hlw, Hlt) (Hra, Hrg, Hrt) \rightarrow$

$\text{Xsimpl } ((H \ast Hla), Hlw, Hlt) (Hra, Hrg, (H \ast Hrt)).$

Proof using. $\text{ximpl_lr_start}' M.$ Qed.

Lemma $\text{ximpl_lr_cancel_htop} : \forall H Hla Hlw Hlt Hra Hrg Hrt,$

$\text{Xsimpl } (Hla, Hlw, Hlt) (Hra, (\text{\textbackslash Top } \ast Hrg), Hrt) \rightarrow$

$\text{Xsimpl } ((H \ast Hla), Hlw, Hlt) (Hra, (\text{\textbackslash Top } \ast Hrg), Hrt).$

Proof using.

$\text{ximpl_lr_start } M. \text{ rewrite (hstar_comm_assoc } Hra) \text{ in } *.$

$\text{rewrite } \leftarrow \text{hstar_htop_htop}. \text{ rew_heap. applics } \text{impl_frame_lr } M.$

$\text{applys } \text{impl_htop_r}.$

Qed.

Lemma $\text{ximpl_lr_cancel_hgc} : \forall Hla Hlw Hlt Hra Hrg Hrt,$

$\text{Xsimpl } (Hla, Hlw, Hlt) (Hra, (\text{\textbackslash GC } \ast Hrg), Hrt) \rightarrow$

$\text{Xsimpl } ((\text{\textbackslash GC } \ast Hla), Hlw, Hlt) (Hra, (\text{\textbackslash GC } \ast Hrg), Hrt).$

Proof using.

$\text{ximpl_lr_start } M. \text{ rewrite (hstar_comm_assoc } Hra).$

$\text{rewrite } \leftarrow \text{hstar_hgc_hgc at } 2. \text{ rew_heap.}$

$\text{applys } \text{impl_frame_lr}. \text{ applics } \text{impl_trans } M. \text{ hstars_simpl}.$

Qed.

Lemma $\text{ximpl_lr_cancel_eq} : \forall H1 H2 Hla Hlw Hlt Hra Hrg Hrt,$

$(H1 = H2) \rightarrow$

$\text{Xsimpl } (Hla, Hlw, Hlt) (Hra, Hrg, Hrt) \rightarrow$

$\text{Ximpl}((H1 \setminus * Hla), Hlw, Hlt) (Hra, Hrg, (H2 \setminus * Hrt)).$

Proof using. *introv* \rightarrow . *apply* \neg *ximpl_lr_cancel_same*. Qed.

Lemma *ximpl_lr_cancel_eq_repr* : $\forall A p (E1 E2:A \rightarrow hprop) Hla Hlw Hlt Hra Hrg Hrt, E1 = E2 \rightarrow$

$\text{Ximpl}(Hla, Hlw, Hlt) (Hra, Hrg, Hrt) \rightarrow$

$\text{Ximpl}((p \simgt E1) \setminus * Hla), Hlw, Hlt) (Hra, Hrg, ((p \simgt E2) \setminus * Hrt)).$

Proof using. *introv M*. *subst*. *apply* \neg *ximpl_lr_cancel_same*. Qed.

Lemma *ximpl_lr_hwand* : $\forall H1 H2 Hla,$

$\text{Ximpl}(\[], \[], (H1 \setminus * Hla)) (\[], \[], H2 \setminus * \[]) \rightarrow$

$\text{Ximpl}(Hla, \[], \[]) ((H1 \dashv * H2) \setminus * \[], \[], \[]).$

Proof using.

ximpl_lr_start' M. *rewrite hwand_equiv*.

applys himpl_trans (*rm M*). *hstars_simpl*.

Qed.

Lemma *ximpl_lr_hwand_hfalse* : $\forall Hla H1,$

$\text{Ximpl}(Hla, \[], \[]) ((\![\text{False}]\!] \dashv * H1) \setminus * \[], \[], \[]).$

Proof using.

intros. *generalize True*. *ximpl_lr_start M*. *rewrite hwand_equiv*.

applys himpl_hstar_hpure_l. *auto_false*.

Qed.

Lemma *ximpl_lr_qwand* : $\forall A (Q1 Q2:A \rightarrow hprop) Hla,$

$(\forall x, \text{Ximpl}(\[], \[], (Q1 x \setminus * Hla)) (\[], \[], Q2 x \setminus * \[])) \rightarrow$

$\text{Ximpl}(Hla, \[], \[]) ((Q1 \dashv * Q2) \setminus * \[], \[], \[]).$

Proof using.

ximpl_lr_start M. *rewrite qwand_equiv*. *intros x*.

specializes M x. *rew_heap* \neg *in M*.

Qed.

Lemma *ximpl_lr_qwand_unit* : $\forall (Q1 Q2:\text{unit} \rightarrow hprop) Hla,$

$\text{Ximpl}(\[], \[], (Q1 \text{tt} \setminus * Hla)) (\[], \[], (Q2 \text{tt} \setminus * \[])) \rightarrow$

$\text{Ximpl}(Hla, \[], \[]) ((Q1 \dashv * Q2) \setminus * \[], \[], \[]).$

Proof using. *introv M*. *applys ximpl_lr_qwand*. *intros []*. *applys M*. Qed.

Lemma *ximpl_lr_hforall* : $\forall A (J:A \rightarrow hprop) Hla,$

$(\forall x, \text{Ximpl}(\[], \[], Hla)) (\[], \[], J x \setminus * \[])) \rightarrow$

$\text{Ximpl}(Hla, \[], \[]) ((hforall J) \setminus * \[], \[], \[]).$

Proof using.

ximpl_lr_start M. *applys himpl_hforall_r*. *intros x*.

specializes M x. *rew_heap* \neg *in M*.

Qed.

Lemma *himpl_lr_refl* : $\forall Hla,$

$\text{Ximpl}(Hla, \[], \[]) (Hla, \[], \[]).$

Proof using. *intros*. *unfolds Ximpl*. *hstars_simpl*. Qed.

Lemma himpl_lr_qwand_unify : $\forall A (Q:A \rightarrow hprop) Hla,$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) ((Q \text{--*} (Q \text{*+} Hla)) \text{*} \text{[]}, \text{[]}, \text{[]}).$
Proof using. intros. unfolds Ximpl. hstars_simpl. rewrite \neg qwand_equiv. Qed.

Lemma himpl_lr_htop : $\forall Hla Hrg,$
 $\text{Ximpl}(\text{[]}, \text{[]}, \text{[]}) (\text{[]}, Hrg, \text{[]}) \rightarrow$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) (\text{[]}, (\text{\Top} \text{*} Hrg), \text{[]}).$

Proof using.

xsimpl_lr_start M. rewrite \leftarrow (hstar_hempty_l Hla).
applys himpl_hstar_trans_l M. hstars_simpl. apply himpl_htop_r.

Qed.

Lemma himpl_lr_hgc : $\forall Hla Hrg,$
 $haffine Hla \rightarrow$
 $\text{Ximpl}(\text{[]}, \text{[]}, \text{[]}) (\text{[]}, Hrg, \text{[]}) \rightarrow$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) (\text{[]}, (\text{\GC} \text{*} Hrg), \text{[]}).$

Proof using.

introv N. xsimpl_lr_start M. rewrite \leftarrow (hstar_hempty_l Hla).
applys himpl_hstar_trans_l M. hstars_simpl. apply \times himpl_hgc_r.

Qed.

Lemma xsimpl_lr_exit_nogc : $\forall Hla Hra,$
 $Hla ==> Hra \rightarrow$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) (Hra, \text{[]}, \text{[]}).$

Proof using. introv M. unfolds Ximpl. hstars_simpl. auto. Qed.

Lemma xsimpl_lr_exit : $\forall Hla Hra Hrg,$
 $Hla ==> Hra \text{*} Hrg \rightarrow$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) (Hra, Hrg, \text{[]}).$

Proof using. introv M. unfolds Ximpl. hstars_simpl. rewrite \neg hstar_comm. Qed.

Lemmas to flip accumulators back in place

Lemma xsimpl_flip_acc_l : $\forall Hla Hra Hla' Hrg,$
 $Hla = Hla' \rightarrow$
 $\text{Ximpl}(Hla', \text{[]}, \text{[]}) (Hra, Hrg, \text{[]}) \rightarrow$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) (Hra, Hrg, \text{[]}).$

Proof using. introv E1 M. subst \times . Qed.

Lemma xsimpl_flip_acc_r : $\forall Hla Hra Hra' Hrg,$
 $Hra = Hra' \rightarrow$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) (Hra', Hrg, \text{[]}) \rightarrow$
 $\text{Ximpl}(Hla, \text{[]}, \text{[]}) (Hra, Hrg, \text{[]}).$

Proof using. introv E1 M. subst \times . Qed.

Ltac xsimpl_flip_acc_l tt :=
 $\text{eapply xsimpl_flip_acc_l; [hstars_flip tt |].}$

Ltac xsimpl_flip_acc_r tt :=

```

eapply xsimpl_flip_acc_r; [ hstars_flip tt | ].

Ltac xsimpl_flip_acc_lr tt :=
  xsimpl_flip_acc_l tt; xsimpl_flip_acc_r tt.

```

Lemmas to pick the hypothesis to cancel

xsimpl_pick i applies to a goal of the form $\text{Xsimpl}((H_1 \setminus^* \dots \setminus^* H_i \setminus^* \dots \setminus^* H_n), H_{lw}, H_{lt}) HR$ and turns it into $\text{Xsimpl}((H_i \setminus^* H_1 \dots \setminus^* H_{i-1} \setminus^* H_{i+1} \setminus^* \dots \setminus^* H_n), H_{lw}, H_{lt}) HR$.

Lemma xsimpl_pick_lemma : $\forall H_{la1} H_{la2} H_{lw} H_{lt} HR,$

```

H_{la1} = H_{la2} →
Xsimpl (H_{la2}, H_{lw}, H_{lt}) HR →
Xsimpl (H_{la1}, H_{lw}, H_{lt}) HR.
```

Proof using. introv M. subst \neg . Qed.

```

Ltac xsimpl_pick i :=
let L := hstars_pick_lemma i in
eapply xsimpl_pick_lemma; [ apply L | ].
```

xsimpl_pick_st f applies to a goal of the form $\text{Xsimpl}((H_1 \setminus^* \dots \setminus^* H_i \setminus^* \dots \setminus^* H_n), H_{lw}, H_{lt}) HR$ and turns it into $\text{Xsimpl}((H_i \setminus^* H_1 \dots \setminus^* H_{i-1} \setminus^* H_{i+1} \setminus^* \dots \setminus^* H_n), H_{lw}, H_{lt}) HR$ for the first *i* such that *f Hi* returns **true**.

```

Ltac xsimpl_pick_st f :=
match goal with ⊢ Xsimpl (?H_{la}, ?H_{lw}, ?H_{lt}) ?HR ⇒
  hstars_search H_{la} ltac:(fun i H ⇒
    match f H with true ⇒ xsimpl_pick i end)
  end.
```

xsimpl_pick_syntactically H is a variant of the above that only checks for syntactic equality, not unifiability.

```

Ltac xsimpl_pick_syntactically H :=
xsimpl_pick_st ltac:(fun H' ⇒
  match H' with H ⇒ constr:(true) end).
```

xsimpl_pick_unifiable H applies to a goal of the form $\text{Xsimpl}(H_{la}, H_{lw}, H_{lt}) HR$, where *Hla* is of the form $H_1 \setminus^* \dots \setminus^* H_n \setminus^* []$. It searches for *H* among the *Hi*. If it finds it, it moves this *Hi* to the front, just before *H1*. Else, it fails.

```

Ltac xsimpl_pick_unifiable H :=
match goal with ⊢ Xsimpl (?H_{la}, ?H_{lw}, ?H_{lt}) ?HR ⇒
  hstars_search H_{la} ltac:(fun i H' ⇒
    unify H H'; xsimpl_pick i)
  end.
```

xsimpl_pick_same H is a choice for one of the above two, it is the default version used by *xsimpl*. Syntactic matching is faster but less expressive.

```
Ltac xsimpl_pick_same H :=
  xsimpl_pick_unifiable H.
```

xsimpl_pick_applied Q applies to a goal of the form $\text{Xsimpl}(Hla, Hlw, Hlt) HR$, where Hla is of the form $H1 \dots \text{Hn} \dots \text{Hl}$. It searches for $Q ?x$ among the Hi . If it finds it, it moves this Hi to the front, just before $H1$. Else, it fails.

```
Ltac xsimpl_pick_applied Q :=
  xsimpl_pick_st ltac:(fun H' =>
    match H' with Q _ => constr:(true) end).
```

repr_get_predicate H applies to a H of the form $p \sim R \dots$ and it returns R .

```
Ltac repr_get_predicate H :=
  match H with ?p \sim ?E => get_head E end.
```

xsimpl_pick_repr H applies to a goal of the form $\text{Xsimpl}(Hla, Hlw, Hlt) HR$, where Hla is of the form $H1 \dots \text{Hn} \dots \text{Hl}$, and where H is of the form $p \sim R$ (same as *repr p*). It searches for $p \sim R$ among the Hi . If it finds it, it moves this Hi to the front, just before $H1$. Else, it fails.

```
Ltac xsimpl_pick_repr H :=
  match H with ?p \sim ?E =>
    let R := get_head E in
    xsimpl_pick_st ltac:(fun H' =>
      match H' with (p \sim ?E') =>
        let R' := get_head E' in
        match R' with R => constr:(true) end end)
  end.
```

Tactic start and stop

Opaque Xsimpl.

```
Ltac xsimpl_handle_qimpl tt :=
  match goal with
  | ⊢ @qimpl _ _ ?Q2 ⇒ is_evar Q2; apply qimpl_refl
  | ⊢ @qimpl unit ?Q1 ?Q2 ⇒ let t := fresh "_tt" in intros t; destruct t
  | ⊢ @qimpl _ _ _ ⇒ let r := fresh "r" in intros r
  | ⊢ himpl _ ?H2 ⇒ is_evar H2; apply himpl_refl
  | ⊢ himpl _ _ ⇒ idtac
  | ⊢ @eq hprop _ _ ⇒ applys himpl_antisym
  | ⊢ @eq (_ → hprop) _ _ ⇒ applys fun_ext_1; applys himpl_antisym
  | _ ⇒ fail 1 "not a goal for xsimpl/xpull"
  end.
```

```
Ltac xsimpl_intro tt :=
  applys xsimpl_start.
```

```

Ltac xpull_start tt :=
  pose ltac_mark;
  intros;
  ximpl_handle_qimpl tt;
  applys xpull_protect;
  ximpl_intro tt.

Ltac ximpl_start tt :=
  pose ltac_mark;
  intros;
  ximpl_handle_qimpl tt;
  ximpl_intro tt.

Ltac ximpl_post_before_generalize tt :=
  idtac.

Ltac ximpl_post_after_generalize tt :=
  idtac.

Ltac himl_post_processing_for_hyp H :=
  idtac.

Ltac ximpl_handle_false_subgoals tt :=
  tryfalse.

Ltac ximpl_clean tt :=
  try remove_empty_heaps_right tt;
  try remove_empty_heaps_left tt;
  try ximpl_hint_remove tt.

Ltac gen_until_mark_with_processing_and_cleaning cont :=
  match goal with H: ?T  $\vdash \_ \Rightarrow$ 
  match T with
  | ltac_mark  $\Rightarrow$  clear H
  |  $\_ \Rightarrow$  cont H;
    let T := type of H in
    generalize H; clear H;

    try (match goal with  $\vdash \_ \rightarrow \_ \Rightarrow$ 
      match type of T with
      | Prop  $\Rightarrow$  idtac
      |  $\_ \Rightarrow$  intros  $\_$ 
      end end);
  gen_until_mark_with_processing cont
  end end.

Ltac ximpl_generalize tt :=
  ximpl_post_before_generalize tt;

```

```

 $ximpl\_handle\_false\_subgoals$  tt;
 $gen\_until\_mark\_with\_processing\_and\_cleaning$ 
   $\text{ltac}:(\text{impl\_post\_processing\_for\_hyp})$ ;
 $ximpl\_post\_after\_generalize$  tt.

Ltac  $ximpl\_post$  tt :=
   $ximpl\_clean$  tt;
   $ximpl\_generalize$  tt.

Ltac  $xpull\_post$  tt :=
   $ximpl\_clean$  tt;
   $\text{unfold protect}$ ;
   $ximpl\_generalize$  tt.

```

Auxiliary functions step

$ximpl_lr_cancel_eq_repr_post$ tt is meant to simplify goals of the form $E1 = E2$ that arises from cancelling $p \sim > E1$ with $p \sim > E2$ in the case where $E1$ and $E2$ share the same head symbol, that is, the same representation predicate R .

```

Ltac  $ximpl\_lr\_cancel\_eq\_repr\_post$  tt :=
   $\text{try fequal; try reflexivity.}$ 

```

$ximpl_r_hexists_apply$ tt is a tactic to apply $ximpl_r_hexists$ by exploiting a hint if one is available (see the hint section above) to specify the instantiation of the existential.

```

Ltac  $ximpl\_r\_hexists\_apply$  tt :=
   $\text{first [ }$ 
     $ximpl\_hint\_next \text{ ltac:(fun } x \Rightarrow$ 
       $\text{match } x \text{ with }$ 
         $| \_\_ \Rightarrow nrapply ximpl\_r\_hexists$ 
         $| \_ \Rightarrow \text{apply } (@ximpl\_r\_hexists } \_ x)$ 
         $\text{end)$ 
     $| nrapply ximpl\_r\_hexists ].$ 

```

$ximpl_hook$ H can be customize to handle cancellation of specific kind of heap predicates (e.g., hsingle).

```

Ltac  $ximpl\_hook$  H := fail.

```

Tactic step

```

Ltac  $ximpl\_hwand\_hstars\_l$  tt :=
   $\text{match goal with } \vdash \text{Ximpl } (?Hla, ((?H1s \sim > ?H2) \sim > ?Hlw), \_[] ) ?HR \Rightarrow$ 
     $\text{hstars\_search } H1s \text{ ltac:(fun } i H \Rightarrow$ 
       $\text{let } L := \text{hstars\_pick\_lemma } i \text{ in }$ 
       $\text{eapply ximpl\_l\_hwand\_reorder; }$ 
       $[ \text{apply } L$ 

```

```

| match H with
  | [] => apply xsimpl_l_cancel_hwand_hstar_hempty
  | _ => xsimpl_pick_same H; apply xsimpl_l_cancel_hwand_hstar
end
])

end.

Ltac xsimpl_step_l tt :=
match goal with ⊢ Xsimpl ?HL ?HR ⇒
match HL with
| (?Hla, ?Hlw, (?H ∗ ?Hlt)) =>
  match H with
    | [] => apply xsimpl_l_hempty
    | [?P] => apply xsimpl_l_hpure; intro
    | ?H1 ∗ ?H2 => rewrite (@hstar_assoc H1 H2)
    | hexists ?J => apply xsimpl_l_hexists; intro
    | ?H1 ∗?H2 => apply xsimpl_l_acc_wand
    | ?Q1 ∗?Q2 => apply xsimpl_l_acc_wand
    | _ => apply xsimpl_l_acc_other
  end
| (?Hla, ((?H1 ∗?H2) ∗?Hlw), []) =>
  match H1 with
    | [] => apply xsimpl_l_cancel_hwand_hempty
    | (_ ∗ _) => xsimpl_hwand_hstars_l tt
    | _ => first [ xsimpl_pick_same H1; apply xsimpl_l_cancel_hwand
                    | apply xsimpl_l_keep_wand ]
  end
| (?Hla, ((?Q1 ∗?Q2) ∗?Hlw), []) =>
  first [ xsimpl_pick_applied Q1; eapply xsimpl_l_cancel_qwand
          | apply xsimpl_l_keep_wand ]
end end.

Ltac xsimpl_hgc_or_htop_cancel cancel_item cancel_lemma :=
repeat (xsimpl_pick_same cancel_item; apply cancel_lemma).

Ltac xsimpl_hgc_or_htop_step tt :=
match goal with ⊢ Xsimpl (?Hla, [], []) (?Hra, ?Hrg, (?H ∗ ?Hrt)) =>
match constr:(Hrg, H) with
| ([], \GC) => applys xsimpl_r_hgc_or_htop;
                 xsimpl_hgc_or_htop_cancel (\GC) xsimpl_lr_cancel_hgc
| ([], \Top) => applys xsimpl_r_hgc_or_htop;
                 xsimpl_hgc_or_htop_cancel (\Top) xsimpl_lr_cancel_htop;
                 xsimpl_hgc_or_htop_cancel (\Top) xsimpl_lr_cancel_htop
| (\GC ∗ [], \Top) => applys xsimpl_r_htop_replace_hgc;

```

```

          xsimpl_hgc_or_htop_cancel (\Top) xsimpl_lr_cancel_htop
| (\GC \* [] , \GC) ⇒ applys xsimpl_r_hgc_drop
| (\Top \* [] , \GC) ⇒ applys xsimpl_r_hgc_drop
| (\Top \* [] , \Top) ⇒ applys xsimpl_r_htop_drop
end end.

Ltac xsimpl_cancel_same H :=
  xsimpl_pick_same H; apply xsimpl_lr_cancel_same.

Ltac xsimpl_step_r tt :=
  match goal with ⊢ Ximpl (?Hla, [] , []) (?Hra, ?Hrg, (?H \* ?Hrt)) ⇒
    match H with
    | ?H' ⇒ xsimpl_hook H
    | [] ⇒ apply xsimpl_r_hempty
    | [?P] ⇒ apply xsimpl_r_hpure
    | ?H1 \* ?H2 ⇒ rewrite (@hstar_assoc H1 H2)
    | ?H \-* ?H'eqH ⇒
        match H with
        | [?P] ⇒ fail 1
        | _ ⇒
            match H'eqH with
            | H ⇒ apply xsimpl_r_hwand_same
              ...
            end
        end
    end
  | hexists ?J ⇒ xsimpl_r_hexists_apply tt
  | \GC ⇒ xsimpl_hgc_or_htop_step tt
  | \Top ⇒ xsimpl_hgc_or_htop_step tt
  | protect ?H' ⇒ apply xsimpl_r_keep
  | protect ?Q' _ ⇒ apply xsimpl_r_keep
  | ?H' ⇒ is_not_evar H; xsimpl_cancel_same H
  | ?p ~> _ ⇒ xsimpl_pick_repr H; apply xsimpl_lr_cancel_eq_repr;
    [ xsimpl_lr_cancel_eq_repr_post tt | ]
  | ?x ~> Id ?X ⇒ has_no_evar x; apply xsimpl_r_id
  | ?x ~> ?T_evar ?X_evar ⇒ has_no_evar x; is_evar T_evar; is_evar X_evar;
    apply xsimpl_r_id_unify
  | _ ⇒ apply xsimpl_r_keep
end end.

Ltac xsimpl_step_lr tt :=
  match goal with ⊢ Ximpl (?Hla, [] , []) (?Hra, ?Hrg, []) ⇒
    match Hrg with
    | [] ⇒
        match Hra with
        | ?H1 \* [] ⇒

```

```

match H1 with
| ?Hra_evar => is_evar Hra_evar; rew_heap; apply himpl_lr_refl
| ?Q1 \-* ?Q2 => is_evar Q2; eapply himpl_lr_qwand_unify
| \[False] \-* ?H2 => apply xsimpl_lr_hwand_hfalse
| ?H1 \-* ?H2 => xsimpl_flip_acc_l tt; apply xsimpl_lr_hwand
| ?Q1 \-* ?Q2 =>
  xsimpl_flip_acc_l tt;
  match H1 with
  | @qwand unit ?Q1' ?Q2' => apply xsimpl_lr_qwand_unit
  | _ => apply xsimpl_lr_qwand; intro
  end
| hforall _ => xsimpl_flip_acc_l tt; apply xsimpl_lr_hforall; intro
  end
| [] => apply himpl_lr_refl
| _ => xsimpl_flip_acc_lr tt; apply xsimpl_lr_exit_nogc
end
| (\Top \* _) => apply himpl_lr_htop
| (\GC \* _) => apply himpl_lr_hgc;
  [ try remove_empty_heaps_haffine tt; xaffine | ]
| ?Hrg' => xsimpl_flip_acc_lr tt; apply xsimpl_lr_exit
end end.

```

```

Ltac xsimpl_step tt :=
  first [ xsimpl_step_l tt
    | xsimpl_step_r tt
    | xsimpl_step_lr tt ].

```

Tactic Notation

```

Ltac xpull_core tt :=
  xpull_start tt;
  repeat (xsimpl_step tt);
  xpull_post tt.

```

```

Tactic Notation "xpull" := xpull_core tt.
Tactic Notation "xpull" "~~" := xpull; auto_tilde.
Tactic Notation "xpull" "*" := xpull; auto_star.

```

```

Ltac xsimpl_core tt :=
  xsimpl_start tt;
  repeat (xsimpl_step tt);
  xsimpl_post tt.

```

```

Tactic Notation "xsimpl" := xsimpl_core tt.
Tactic Notation "xsimpl" "~~" := xsimpl; auto_tilde.

```

```

Tactic Notation "xsimpl" "*" := xsimpl; auto_star.

Tactic Notation "xsimpl" constr(L) :=
  match type of L with
  | list Boxer => xsimpl_hint_put L
  | _ => xsimpl_hint_put (boxer L :: nil)
  end; xsimpl.

Tactic Notation "xsimpl" constr(X1) constr(X2) :=
  xsimpl (» X1 X2).

Tactic Notation "xsimpl" constr(X1) constr(X2) constr(X3) :=
  xsimpl (» X1 X2 X3).

Tactic Notation "xsimpl" "~" constr(L) :=
  xsimpl L; auto_tilde.

Tactic Notation "xsimpl" "~" constr(X1) constr(X2) :=
  xsimpl X1 X2; auto_tilde.

Tactic Notation "xsimpl" "~" constr(X1) constr(X2) constr(X3) :=
  xsimpl X1 X2 X3; auto_tilde.

Tactic Notation "xsimpl" "*" constr(L) :=
  xsimpl L; auto_star.

Tactic Notation "xsimpl" "*" constr(X1) constr(X2) :=
  xsimpl X1 X2; auto_star.

Tactic Notation "xsimpl" "*" constr(X1) constr(X2) constr(X3) :=
  xsimpl X1 X2 X3; auto_star.

```

Tactic *xchange*

xchange performs rewriting on the LHS of an entailment. Essentially, it applies to a goal of the form $H1 \setminus^* H2 ==> H3$, and exploits an entailment $H1 ==> H1'$ to replace the goal with $H1' \setminus^* H2 ==> H3$.

The tactic is actually a bit more flexible in two respects:

- it does not force the LHS to be exactly of the form $H1 \setminus^* H2$
- it takes as argument any lemma, whose instantiation result in a heap entailment of the form $H1 ==> H1'$.

Concretely, the tactic is just a wrapper around an application of the lemma called `xchange_lemma`, which appears below.

xchanges combines a call to *xchange* with calls to *xsimpl* on the subgoals.

```

Lemma xchange_lemma : ∀ H1 H2 H3 H4,
  H1 ==> H2 →
  H3 ==> H1 \setminus^* (H2 \setminus^* protect H4) →
  H3 ==> H4.

```

Proof using.

```
introv M1 M2. applys himpl_trans (rm M2).
applys himpl_hstar_trans_I (rm M1). applys hwand_cancel.
```

Qed.

```
Ltac xchange_apply L :=
  eapply xchange_lemma; [ eapply L | ].
```

```
Ltac xchange_build_entailment modifier K :=
  match modifier with
  | -->
    match type of K with
    | _ = _ => constr:(@impl_of_eq _ _ K)
    | _ => constr:(K)
    end
  | _ => constr:(@modifier _ _ K)
  end.
```

```
Ltac xchange_perform L modifier cont :=
  forwards_nounfold_then L ltac:(fun K =>
  let X := fresh "TEMP" in
  set (X := K);
  let M := xchange_build_entailment modifier K in
  clear X;
  xchange_apply M;
  cont tt).
```

```
Ltac xchange_core L modifier cont :=
  pose ltac_mark;
  intros;
  match goal with
  | ⊢ _ ==> _ => idtac
  | ⊢ _ ==> _ => let x := fresh "r" in intros x
  end;
  xchange_perform L modifier cont;
  gen_until_mark.
```

Error reporting support for *xchange* (not for *xchanges*)

Definition xchange_hidden (*P*:Type) (*e*:*P*) := *e*.

Notation "'__XCHANGE FAILED TO MATCH PRECONDITION__'" :=
 (@xchange_hidden _ _).

```
Ltac xchange_report_error tt :=
  match goal with ⊢ context [?H1 \-* protect ?H2] =>
  change (H1 \-* protect H2) with (@xchange_hidden _ (H1 \-* protect H2)) end.
```

```
Ltac xchange_xpull_cont tt :=
```

```

 $xsimpl$ ; first
[  $xchange\_report\_error tt$ 
|  $\text{unfold protect; try solve [ apply himpl\_refl ]}.$ 

Ltac  $xchange\_xpull\_cont\_basic tt :=$ 
 $xsimpl$ ;  $\text{unfold protect; try solve [ apply himpl\_refl ]}.$ 

Ltac  $xchange\_xsimpl\_cont tt :=$ 
 $\text{unfold protect; } xsimpl; \text{try solve [ apply himpl\_refl ].}$ 

Ltac  $xchange\_nosimpl\_base E modifier :=$ 
 $xchange\_core E modifier \text{ltac:(idcont)}.$ 

Tactic Notation "xchange_nosimpl" constr( $E$ ) :=
 $xchange\_nosimpl\_base E \_\_.$ 

Tactic Notation "xchange_nosimpl" " $\rightarrow$ " constr( $E$ ) :=
 $xchange\_nosimpl\_base E himpl\_of\_eq.$ 

Tactic Notation "xchange_nosimpl" " $<-$ " constr( $E$ ) :=
 $xchange\_nosimpl\_base himpl\_of\_eq\_sym.$ 

Ltac  $xchange\_base E modif :=$ 
 $xchange\_core E modif \text{ltac:(xchange\_xpull\_cont)}.$ 

Tactic Notation "xchange" constr( $E$ ) :=
 $xchange\_base E \_\_.$ 

Tactic Notation "xchange" " $\sim$ " constr( $E$ ) :=
 $xchange E; auto\_tilde.$ 

Tactic Notation "xchange" "*" constr( $E$ ) :=
 $xchange E; auto\_star.$ 

Tactic Notation "xchange" " $\rightarrow$ " constr( $E$ ) :=
 $xchange\_base E himpl\_of\_eq.$ 

Tactic Notation "xchange" " $\sim$ " " $\rightarrow$ " constr( $E$ ) :=
 $xchange \rightarrow E; auto\_tilde.$ 

Tactic Notation "xchange" "*" " $\rightarrow$ " constr( $E$ ) :=
 $xchange \rightarrow E; auto\_star.$ 

Tactic Notation "xchange" " $<-$ " constr( $E$ ) :=
 $xchange\_base E himpl\_of\_eq\_sym.$ 

Tactic Notation "xchange" " $\sim$ " " $<-$ " constr( $E$ ) :=
 $xchange \leftarrow E; auto\_tilde.$ 

Tactic Notation "xchange" "*" " $<-$ " constr( $E$ ) :=
 $xchange \leftarrow E; auto\_star.$ 

Ltac  $xchanges\_base E modif :=$ 
 $xchange\_core E modif \text{ltac:(xchange\_xsimpl\_cont)}.$ 

Tactic Notation "xchanges" constr( $E$ ) :=
 $xchanges\_base E \_\_.$ 

Tactic Notation "xchanges" " $\sim$ " constr( $E$ ) :=
```

```

xchanges E; auto_tilde.
Tactic Notation "xchanges" "*" constr(E) :=
  xchanges E; auto_star.

Tactic Notation "xchanges" "->" constr(E) :=
  xchanges_base E himpl_of_eq.

Tactic Notation "xchanges" "~" "->" constr(E) :=
  xchanges → E; auto_tilde.

Tactic Notation "xchanges" "*" "->" constr(E) :=
  xchanges → E; auto_star.

Tactic Notation "xchanges" "<-" constr(E) :=
  xchanges_base E himpl_of_eq_sym.

Tactic Notation "xchanges" "~" "<-" constr(E) :=
  xchanges ← E; auto_tilde.

Tactic Notation "xchanges" "*" "<-" constr(E) :=
  xchanges ← E; auto_star.

Tactic Notation "xchange" constr(E1) "," constr(E2) :=
  xchange E1; try xchange E2.

Tactic Notation "xchange" constr(E1) "," constr(E2) "," constr(E3) :=
  xchange E1; try xchange E2; try xchange E3.

Tactic Notation "xchange" constr(E1) "," constr(E2) "," constr(E3) "," constr(E4) :=
  xchange E1; try xchange E2; try xchange E3; try xchange E4.

```

33.5 Demos

rew_heap demos

```

Lemma rew_heap_demo_with_evar : ∀ H1 H2 H3,
  (forall H, H1 \* (H \* H2) \* [] = H3 → True) → True.

```

Proof using.

```

  introv M. dup 3.
  { eapply M. rewrite hstar_assoc. rewrite hstar_assoc_demo. }
  { eapply M. rew_heap_assoc_demo. }
  { eapply M. rew_heap_demo. }
Abort.

```

hstars demos

```

Lemma hstars_flip_demo : ∀ H1 H2 H3 H4,
  (forall H, H1 \* H2 \* H3 \* H4 \* [] = H → H = H → True) → True.

```

Proof using.

```

intros M. eapply M. hstars_flip tt.
Abort.

Lemma hstars_flip_demo_0 :
  ( $\forall H, \text{[]} = H \rightarrow H = H \rightarrow \text{True}) \rightarrow \text{True}.$ 
Proof using.

```

```

  intros M. eapply M. hstars_flip tt.
Abort.

```

xsimpl_hint demos

```

Lemma xsimpl_demo_hints :  $\exists n, n = 3.$ 

```

```

Proof using.
  xsimpl_hint_put (» 3 true).
  xsimpl_hint_next ltac:(fun x =>  $\exists x$ ).
  xsimpl_hint_remove tt.

```

```

Abort.

```

hstars_pick demos

```

Lemma demo_hstars_pick_1 :  $\forall H1 H2 H3 H4 Hresult,$ 
   $(\forall H, H1 \text{ *} H2 \text{ *} H3 \text{ *} H4 = H \rightarrow H = Hresult \rightarrow \text{True}) \rightarrow \text{True}.$ 

```

```

Proof using.
  intros M. dup 2.
  { eapply M. let L := hstars_pick_lemma 3 in eapply L. demo. }
  { eapply M. let L := hstars_pick_lemma (hstars_last 4) in eapply L. demo. }
Qed.

```

hstars_simpl demos

```

Lemma demo_hstars_simpl_1 :  $\forall H1 H2 H3 H4 H5,$ 
   $H2 ==> H5 \rightarrow$ 
   $H1 \text{ *} H2 \text{ *} H3 \text{ *} H4 ==> H4 \text{ *} H5 \text{ *} H3 \text{ *} H1.$ 

```

```

Proof using.
  intros. dup.
  { hstars_simpl_start tt.
    hstars_simpl_step tt.
    hstars_simpl_step tt.
    hstars_simpl_step tt.
    hstars_simpl_step tt.
    hstars_simpl_post tt. auto. }
  { hstars_simpl. auto. }

```

```

Qed.

```

```
Lemma demo_hstars_simpl_2 : ∀ H1 H2 H3 H4 H5,
  (∀ H, H ∗ H2 ∗ H3 ∗ H4 ==> H4 ∗ H5 ∗ H3 ∗ H1 → True) → True.
```

Proof using.

introv M. eapply M. hstars_simpl.

Abort.

xsimpl_pick demos

```
Lemma xsimpl_pick_demo : ∀ (Q:bool→hprop) (P:Prop) H1 H2 H3 Hlw Hlt Hra Hrg Hrt,
  (∀ HX HY,
    Xsimpl ((H1 ∗ H2 ∗ H3 ∗ Q true ∗ ([P] ∗ HX) ∗ HY ∗ []), Hlw, Hlt)
    (Hra, Hrg, Hrt)
  → True) → True.
```

Proof using.

introv M. applys (rm M).

let L := hstars_pick_lemma 2%nat in set (X:=L).

eapply xsimpl_pick_lemma. apply X.

xsimpl_pick 2%nat.

xsimpl_pick_same H3.

xsimpl_pick_applied Q.

xsimpl_pick_same H2.

xsimpl_pick_unifiable H3.

xsimpl_pick_unifiable \[True].

xsimpl_pick_unifiable (\[P] ∗ H1).

Abort.

xpull and *xsimpl* demos

Tactic Notation "xpull0" := xpull_start tt.

Tactic Notation "xsimpl0" := xsimpl_start tt.

Tactic Notation "xsimpl1" := xsimpl_step tt.

Tactic Notation "xsimpl2" := xsimpl_post tt.

Tactic Notation "xsimpll" := xsimpl_step_l tt.

Tactic Notation "xsimplr" := xsimpl_step_r tt.

Tactic Notation "xsimpllr" := xsimpl_step_lr tt.

Declare Scope xsimpl_scope.

```
Notation "'HSIMPL' Hla Hlw Hlt =====> Hra Hrg Hrt" := (Xsimpl (Hla, Hlw, Hlt)
  (Hra, Hrg, Hrt))
```

(at level 69, Hla, Hlw, Hlt, Hra, Hrg, Hrt at level 0,

format "[v' 'HSIMPL' '/ Hla '/ Hlw '/ Hlt '/ =====> '/ Hra '/ Hrg '/ Hrt]"]"

: xsimpl_scope.

Local Open Scope xsimpl_scope.

Lemma `xpull_demo` : $\forall H1 H2 H3 H,$
 $(H1 \ast \square \ast (H2 \ast \exists (y:\text{int}) z (n:\text{nat}), [y = y + z + n]) \ast H3) ==> H.$

Proof using.

dup.

```
{ intros. xpull0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl2. demo. }
{ xpull. intros. demo. }
```

Abort.

Lemma `xsimpl_demo_stars` : $\forall H1 H2 H3 H4 H5,$
 $H1 \ast H2 \ast H3 \ast H4 ==> H4 \ast H3 \ast H5 \ast H2.$

Proof using.

dup 3.

```
{ xpull. demo. }
{ intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. demo. }
{ intros. xsimpl. demo. }
```

Abort.

Lemma `xsimpl_demo_keep_order` : $\forall H1 H2 H3 H4 H5 H6 H7,$
 $H1 \ast H2 \ast H3 \ast H4 ==> H5 \ast H3 \ast H6 \ast H7.$

Proof using. `intros. xsimpl. demo. Abort.`

Lemma `xsimpl_demo_stars_top` : $\forall H1 H2 H3 H4 H5,$
 $H1 \ast H2 \ast H3 \ast H4 \ast H5 ==> H3 \ast H1 \ast H2 \ast \text{Top}.$

Proof using.

dup.

```
{ intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
{ intros. xsimpl. }
```

Abort.

Lemma `xsimpl_demo_hint` : $\forall H1 (Q:\text{int} \rightarrow hprop),$
 $Q 4 ==> Q 3 \rightarrow$
 $H1 \ast Q 4 ==> \exists x, Q x \ast H1.$

Proof using.

introv W. dup.

```
{ intros. xsimpl_hint_put (» 3).
    xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl2. auto. }
{ xsimpl 3. auto. }
```

Qed.

Lemma `xsimpl_demo_stars_gc` : $\forall H1 H2,$
 $\text{haffine } H2 \rightarrow$

$H1 \ast H2 ==> H1 \ast \text{GC}$.

Proof using.

dup.

```
{ intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. }
{ intros. xsimpl~. }
```

Abort.

Lemma xsimpl_demo_evar_1 : $\forall H1 H2,$

$(\forall H, H1 \ast H2 ==> H \rightarrow \text{True}) \rightarrow \text{True}$.

Proof using. intros. eapply H. xsimpl. Abort.

Lemma xsimpl_demo_evar_2 : $\forall H1,$

$(\forall H, H1 ==> H1 \ast H \rightarrow \text{True}) \rightarrow \text{True}$.

Proof using.

introv M. dup.

```
{ eapply M. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    { eapply M. xsimpl~. }
```

Abort.

Lemma xsimpl_demo_htop_both_sides : $\forall H1 H2,$

$H1 \ast H2 \ast \text{Top} ==> H1 \ast \text{Top}$.

Proof using.

dup.

```
{ intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
{ intros. xsimpl~. }
```

Abort.

Lemma xsimpl_demo_htop_multiple : $\forall H1 H2,$

$H1 \ast H2 \ast \text{Top} ==> H1 \ast \text{Top} \ast \text{Top}$.

Proof using. intros. xsimpl~. Abort.

Lemma xsimpl_demo_hgc_multiple : $\forall H1 H2,$

haffine H2 \rightarrow

$H1 \ast H2 \ast \text{GC} ==> H1 \ast \text{GC} \ast \text{GC}$.

Proof using. intros. xsimpl~. Qed.

Lemma xsimpl_demo_hwand : $\forall H1 H2 H3 H4,$

$(H1 \dashv\ast (H2 \dashv\ast H3)) \ast H1 \ast H4 ==> (H2 \dashv\ast (H3 \ast H4))$.

Proof using.

dup.

```
{ intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
{ intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
```

```

{ intros. xsimpl ⊥. }
Qed.

Lemma xsimpl_demo_qwand : ∀ A (x:A) (Q1 Q2:A → hprop) H1,
  H1 \* (H1 \-* (Q1 \-* Q2)) \* (Q1 x) ==> (Q2 x).
Proof using. intros. xsimpl ⊥. Qed.

Lemma xsimpl_demo_hwand_r : ∀ H1 H2 H3,
  H1 \* H2 ==> H1 \* (H3 \-* (H2 \* H3)).
Proof using. intros. xsimpl ⊥. Qed.

Lemma xsimpl_demo_qwand_r : ∀ A (x:A) (Q1 Q2:A → hprop) H1 H2,
  H1 \* H2 ==> H1 \* (Q1 \-* (Q1 \** H2)).
Proof using. intros. xsimpl. Qed.

Lemma xsimpl_demo_hwand_multiple_1 : ∀ H1 H2 H3 H4 H5,
  H1 \-* H4 ==> H5 →
  (H2 \* ((H1 \* H2 \* H3) \-* H4)) \* H3 ==> H5.
Proof using. introv M. xsimpl. auto. Qed.

Lemma xsimpl_demo_hwand_multiple_2 : ∀ H1 H2 H3 H4 H5,
  (H1 \* H2 \* ((H1 \* H3) \-* (H4 \-* H5))) \* (H2 \-* H3) \* H4 ==> H5.
Proof using. intros. xsimpl. Qed.

Lemma xsimpl_demo_hwand_hempty : ∀ H1 H2 H3,
  ([] \-* H1) \* H2 ==> H3.
Proof using. intros. xsimpl. Abort.

Lemma xsimpl_demo_hwand_hstar_hempty : ∀ H0 H1 H2 H3,
  ((H0 \* []) \-* [] \-* H1) \* H2 ==> H3.
Proof using. intros. xsimpl. rew_heap. Abort.

Lemma xsimpl_demo_hwand_iter : ∀ H1 H2 H3 H4 H5,
  H1 \* H2 \* ((H1 \* H3) \-* (H4 \-* H5)) \* H4 ==> ((H2 \-* H3) \-* H5).
Proof using.
  intros. dup.
  { xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
  { xsimpl. }
Qed.

Lemma xsimpl_demo_repr_1 : ∀ p q (R:int → int → hprop),
  p ~> R 3 \* q ~> R 4 ==> ∃ n m, p ~> R n \* q ~> R m.
Proof using.
  intros. dup.
  { xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1.

```

xsiml1. *xsiml1.* *xsiml1.* *xsiml1.* *xsiml1.* }
 { *xsiml¬.* }

Qed.

Lemma xsimpl_demo_repr_2 : $\forall p (R R':\text{int} \rightarrow \text{int} \rightarrow hprop),$
 $R = R' \rightarrow$

$p \sim R' 3 \implies \exists n, p \sim R n.$
 Proof using. introv E. xsimpl. subst R'. xsimpl. Qed.

```

Lemma xsimpl_demo_repr_3 : ∀ p (R:int→int→hprop),
  let R' := R in
  p ~> R' 3 ==> \exists n n ~> R n

```

Proof using

```
intros. dup.  
{ xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }  
{ xsimpl1. }
```

Qed

```
Lemma xsimpl_demo_repr_4 : ∀ p n m (R:int→int→hprop),
  n = m + 0 →
  n ~> R n ==> n ~> R m
```

Proof using intros *xsimp! math* Qed.

Lemma xsimpl_demo_0 : $\forall H1 H2$

*H1 --> H2 * \CC * \CC*

Proof using intros *xsimp!* Abort

floor using: inches. *simp.* *Ascrip.*

*H1 --> H2 * \CC * \Top * \Tau*

sof using

Plot using:
intmax(d)

```
{ xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
  xsimpl1. xsimpl1. xsimpl1. xsimpl2. demo. }
{ xsimpl¬. demo. }
```

Abort.

Lemma xsimpl_demo_gc_2 : $\forall H1 H2 H3,$

$H1 \ast H2 \ast \text{Top} \ast \text{GC} \ast \text{Top} ==> H3 \ast \text{GC} \ast \text{GC}$.

Proof using. intros. xsimpl. Abort.

Lemma xsimpl_demo_gc_3 : $\forall H1 H2,$

$H1 \setminus* H2 \setminus* \text{GC} \setminus* \text{GC} ==> H2 \setminus* \text{GC} \setminus* \text{GC} \setminus* \text{GC}$.

Proof using. intros. xsimpl. xaffine. Abort.

Lemma xsimpl_demo_gc_4 : $\forall H1 H2,$

$H1 \ * H2 \ * \text{\GC} ==> H2 \ * \text{\GC} \ * \text{\Top} \ * \text{\Top} \ * \text{\GC}$.

Proof using. intros. xsimpl. Abort.

xchange demos

Lemma *xchange_demo_1* : $\forall H1 H2 H3 H4 H5 H6,$
 $H1 ==> H2 \ast H3 \rightarrow$
 $H1 \ast H4 ==> (H5 \ast H6).$

Proof using.

introv M. dup 3.
{ *xchange_nosimpl M. xsimpl. demo.* }
{ *xchange M. xsimpl. demo.* }
{ *xchanges M. demo.* }

Qed.

Lemma *xchange_demo_2* : $\forall A (Q:A \rightarrow hprop) H1 H2 H3,$
 $H1 ==> \exists x, Q x \ast (H2 \ast H3) \rightarrow$
 $H1 \ast H2 ==> \exists x, Q x \ast H3.$

Proof using.

introv M. dup 3.
{ *xchange_nosimpl M. xsimpl. unfold protect. xsimpl.* }
{ *xchange M. xsimpl.* }
{ *xchanges M.* }

Qed.

Lemma *xchange_demo_4* : $\forall A (Q1 Q2:A \rightarrow hprop) H,$
 $Q1 ==> Q2 \rightarrow$
 $Q1 \ast H ==> Q2 \ast H.$

Proof using. *introv M. xchanges M.* Qed.

Lemma *xsimpl_demo_hfalse* : $\forall H1 H2,$
 $H1 ==> \text{[False]} \rightarrow$
 $H1 \ast H2 ==> \text{[False]}.$

Proof using.

introv M. dup.
{ *xchange_nosimpl M. xsimpl0. xsimpl1. xsimpl1. xsimpl1.*
xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
{ *xchange M.* }

Qed.

Lemma *xchange_demo_hforall_l* :
 $\forall (hforall_specialize : \forall A (x:A) (J:A \rightarrow hprop), (hforall J) ==> (J x)),$
 $\forall (Q:\text{int} \rightarrow hprop) H1,$
 $(\forall x, Q x) \ast H1 ==> Q 2 \ast \text{Top}.$

Proof using.

intros. *xchange (» hforall_specialize 2). xsimpl.*

Qed.

End XSIMPLSETUP.

Chapter 34

Library **SLF.LibSepMinimal**

34.1 LibSepMinimal: Appendix - Minimalistic Soundness Proof

This file contains a stand-alone, minimalistic formalization of the soundness of Separation Logic reasoning rules with respect to the big-step semantics of an imperative lambda-calculus. This formalization accompanies the ICFP'20 paper *Separation Logic for Sequential Programs*: http://www.chargueraud.org/research/2020/seq_seplogic/seq_seplogic.pdf.

34.2 Source Language

34.2.1 Syntax

Set Implicit Arguments.

From *SLF* Require Export LibString LibCore.

From *SLF* Require Export LibSepTLCbuffer LibSepFmap.

Module FMAP := LIBSEPFMAP.

Variables are defined as strings, `var_eq` denotes boolean comparison.

Definition `var` : Type := **string**.

Definition `var_eq` := **String.string_dec**.

Locations are defined as natural numbers.

Definition `loc` : Type := **nat**.

Primitive operations include memory operations, as well as addition and division to illustrate a total and a partial arithmetic operations.

Inductive `prim` : Type :=

| `val_ref` : **prim**
| `val_get` : **prim**

```

| val_set : prim
| val_free : prim
| val_add : prim
| val_div : prim.

```

The grammar of closed values (assumed to contain no free variables) includes values of ground types, primitive operations, and closures.

Inductive val : Type :=

```

| val_unit : val
| val_bool : bool → val
| val_int : int → val
| val_loc : loc → val
| val_prim : prim → val
| val_fix : var → var → trm → val

```

The grammar of terms includes closed values, variables, functions, applications, let-binding and conditional.

with trm : Type :=

```

| trm_val : val → trm
| trm_var : var → trm
| trm_fix : var → var → trm → trm
| trm_app : trm → trm → trm
| trm_let : var → trm → trm → trm
| trm_if : trm → trm → trm → trm.

```

Coercions are used to improve conciseness in the statement of evaluation rules.

```

Coercion val_bool : bool >-> val.
Coercion val_int : Z >-> val.
Coercion val_loc : loc >-> val.
Coercion val_prim : prim >-> val.
Coercion trm_val : val >-> trm.
Coercion trm_var : var >-> trm.
Coercion trm_app : trm >-> Funclass.

```

The type of values is inhabited (useful for finite map operations).

```

Global Instance Inhab_val : Inhab val.
Proof. apply (Inhab_of_val val_unit). Qed.

```

A heap, a.k.a. state, consists of a finite map from location to values. Finite maps are formalized in the file *LibSepFmap.v*. (We purposely do not use TLC's finite map library to avoid complications with typeclasses.)

Definition heap : Type := fmap loc val.

Implicit types associate meta-variable with types.

Implicit Types f : var.

```

Implicit Types  $b$  : bool.
Implicit Types  $p$  : loc.
Implicit Types  $n$  : int.
Implicit Types  $v w r$  : val.
Implicit Types  $t$  : trm.
Implicit Types  $s h$  : heap.

```

34.2.2 Semantics

The standard capture-avoiding substitution is written `subst y w t`.

```

Fixpoint subst ( $y:\text{var}$ ) ( $w:\text{val}$ ) ( $t:\text{trm}$ ) : trm :=
  let aux  $t := \text{subst } y w t$  in
  let if_y_eq  $x t1 t2 := \text{if var\_eq } x y \text{ then } t1 \text{ else } t2$  in
  match  $t$  with
  | trm_val  $v \Rightarrow \text{trm\_val } v$ 
  | trm_var  $x \Rightarrow \text{if\_y\_eq } x (\text{trm\_val } w) t$ 
  | trm_fix  $f x t1 \Rightarrow \text{trm\_fix } f x (\text{if\_y\_eq } f t1 (\text{if\_y\_eq } x t1 (\text{aux } t1)))$ 
  | trm_app  $t1 t2 \Rightarrow \text{trm\_app } (\text{aux } t1) (\text{aux } t2)$ 
  | trm_let  $x t1 t2 \Rightarrow \text{trm\_let } x (\text{aux } t1) (\text{if\_y\_eq } x t2 (\text{aux } t2))$ 
  | trm_if  $t0 t1 t2 \Rightarrow \text{trm\_if } (\text{aux } t0) (\text{aux } t1) (\text{aux } t2)$ 
  end.

```

The big-step evaluation judgement, written `eval s t s' v`, asserts that the evaluation of term t in state s terminates on a value v in a state s' .

```

Inductive eval : heap → trm → heap → val → Prop :=
| eval_val : ∀  $s v$ ,
  eval  $s (\text{trm\_val } v) s v$ 
| eval_fix : ∀  $s f x t1$ ,
  eval  $s (\text{trm\_fix } f x t1) s (\text{val\_fix } f x t1)$ 
| eval_app : ∀  $s1 s2 v1 v2 f x t1 v$ ,
   $v1 = \text{val\_fix } f x t1 \rightarrow$ 
  eval  $s1 (\text{subst } x v2 (\text{subst } f v1 t1)) s2 v \rightarrow$ 
  eval  $s1 (\text{trm\_app } v1 v2) s2 v$ 
| eval_let : ∀  $s1 s2 s3 x t1 t2 v1 r$ ,
  eval  $s1 t1 s2 v1 \rightarrow$ 
  eval  $s2 (\text{subst } x v1 t2) s3 r \rightarrow$ 
  eval  $s1 (\text{trm\_let } x t1 t2) s3 r$ 
| eval_if : ∀  $s1 s2 b v t1 t2$ ,
  eval  $s1 (\text{if } b \text{ then } t1 \text{ else } t2) s2 v \rightarrow$ 
  eval  $s1 (\text{trm\_if } (\text{val\_bool } b) t1 t2) s2 v$ 
| eval_add : ∀  $m n1 n2$ ,
  eval  $m (\text{val\_add } (\text{val\_int } n1) (\text{val\_int } n2)) m (\text{val\_int } (n1 + n2))$ 
| eval_div : ∀  $m n1 n2$ ,

```

```

 $n2 \neq 0 \rightarrow$ 
eval m (val_div (val_int n1) (val_int n2)) m (val_int (Z.quot n1 n2))
| eval_ref :  $\forall s v p,$ 
   $\neg \text{Fmap.indom } s p \rightarrow$ 
    eval s (val_ref v) (Fmap.update s p v) (val_loc p)
| eval_get :  $\forall s p,$ 
  Fmap.indom s p  $\rightarrow$ 
    eval s (val_get (val_loc p)) s (Fmap.read s p)
| eval_set :  $\forall s p v,$ 
  Fmap.indom s p  $\rightarrow$ 
    eval s (val_set (val_loc p) v) (Fmap.update s p v) val_unit
| eval_free :  $\forall s p,$ 
  Fmap.indom s p  $\rightarrow$ 
    eval s (val_free (val_loc p)) (Fmap.remove s p) val_unit.

```

34.2.3 Automation for Heap Equality and Heap Disjointness

For goals asserting equalities between heaps, i.e., of the form $h1 = h2$, we set up automation so that it performs some tidying: substitution, removal of empty heaps, normalization with respect to associativity.

```

Hint Rewrite union_assoc union_empty_l union_empty_r : fmap.
Hint Extern 1 (_ = _ :> heap)  $\Rightarrow$  subst; autorewrite with fmap.

```

For goals asserting disjointness between heaps, i.e., of the form $Fmap.disjoint h1 h2$, we set up automation to perform simplifications: substitution, exploit distributivity of the disjointness predicate over unions of heaps, and exploit disjointness with empty heaps. The tactic *jauto_set* used here comes from the TLC library; essentially, it destructs conjunctions and existentials.

```

Hint Resolve Fmap.disjoint_empty_l Fmap.disjoint_empty_r.
Hint Rewrite disjoint_union_eq_l disjoint_union_eq_r : disjoint.
Hint Extern 1 (Fmap.disjoint _ _)  $\Rightarrow$ 
  subst; autorewrite with rew_disjoint in *; jauto_set.

```

34.3 Heap Predicates and Entailment

34.3.1 Extensionality Axioms

Extensionality axioms are essential to assert equalities between heap predicates of type `hprop`, and between postconditions, of type `val → hprop`.

```

Axiom functional_extensionality :  $\forall A B (f g:A \rightarrow B),$ 
   $(\forall x, f x = g x) \rightarrow$ 
   $f = g.$ 

```

Axiom *propositional_extensionality* : $\forall (P \ Q:\text{Prop}),$
 $(P \leftrightarrow Q) \rightarrow$
 $P = Q.$

34.3.2 Core Heap Predicates

The type of heap predicates is named `hprop`.

Definition `hprop := heap → Prop.`

We bind a few more meta-variables.

Implicit Types $P : \text{Prop}.$

Implicit Types $H : \text{hprop}.$

Implicit Types $Q : \text{val} \rightarrow \text{hprop}.$

Core heap predicates, and their associated notations:

- \emptyset denotes the empty heap predicate
- $\{P\}$ denotes a pure fact
- $\{p \sim v\}$ denotes a singleton heap
- $H_1 \setminus^* H_2$ denotes the separating conjunction
- $Q_1 \setminus^*+ H_2$ denotes the separating conjunction extending a postcondition
- $\exists x, H$ denotes an existential
- $\forall x, H$ denotes a universal.

Definition `hempty : hprop :=`
`fun h ⇒ (h = Fmap.empty).`

Definition `hsingle (p:loc) (v:val) : hprop :=`
`fun h ⇒ (h = Fmap.single p v).`

Definition `hstar (H1 H2 : hprop) : hprop :=`
`fun h ⇒ ∃ h1 h2, H1 h1`
 $\wedge H2 h2$
 $\wedge \text{Fmap.disjoint } h1 h2$
 $\wedge h = \text{Fmap.union } h1 h2.$

Definition `hexists A (J:A→hprop) : hprop :=`
`fun h ⇒ ∃ x, J x h.`

Definition `hforall (A : Type) (J : A → hprop) : hprop :=`
`fun h ⇒ ∀ x, J x h.`

Definition `hpure (P:Prop) : hprop :=`

```

hexists (fun (p:P) => hempty).

Declare Scope hprop_scope.

Notation "\[] := (hempty)
(at level 0) : hprop_scope.

Notation "\[ P ] := (hpure P)
(at level 0, format "\[ P ]") : hprop_scope.

Notation "p '~~> v" := (hsingle p v) (at level 32) : hprop_scope.

Notation "H1 '\*' H2" := (hstar H1 H2)
(at level 41, right associativity) : hprop_scope.

Notation "Q \*+ H" := (fun x => hstar (Q x) H)
(at level 40) : hprop_scope.

Notation "'\exists' x1 .. xn , H := 
(hexists (fun x1 => .. (hexists (fun xn => H)) ..))
(at level 39, x1 binder, H at level 50, right associativity,
format "'[ '\exists' '/ ' x1 .. xn , '/ ' H ']'"') : hprop_scope.

Notation "'\forall' x1 .. xn , H := 
(hforall (fun x1 => .. (hforall (fun xn => H)) ..))
(at level 39, x1 binder, H at level 50, right associativity,
format "'[ '\forall' '/ ' x1 .. xn , '/ ' H ']'"') : hprop_scope.

```

34.3.3 Entailment

```

Declare Scope hprop_scope.

Open Scope hprop_scope.

Entailment for heap predicates, written  $H1 ==> H2$ .

Definition himpl (H1 H2:hprop) : Prop :=
 $\forall h, H1 h \rightarrow H2 h$ .

Notation "H1 ==> H2" := (himpl H1 H2) (at level 55) : hprop_scope.

Entailment between postconditions, written  $Q1 ===> Q2$ 

Definition qimpl A (Q1 Q2:A→hprop) : Prop :=
 $\forall (v:A), Q1 v ==> Q2 v$ .

Notation "Q1 ===> Q2" := (qimpl Q1 Q2) (at level 55) : hprop_scope.

Entailment defines an order on heap predicates

Lemma himpl_refl :  $\forall H, H ==> H$ .
Proof. introv M. auto. Qed.

Lemma himpl_trans :  $\forall H2 H1 H3, H1 ==> H2 \wedge H2 ==> H3 \Rightarrow H1 ==> H3$ .

```

```
(H1 ==> H2) →
(H2 ==> H3) →
(H1 ==> H3).
```

Proof. *intros* M1 M2. *unfold*s × *himpl*. *Qed*.

Lemma himpl_antisym : $\forall H1 H2,$

```
(H1 ==> H2) →
(H2 ==> H1) →
(H1 = H2).
```

Proof. *intros* M1 M2. *apply*s *pred_ext_1*. *intros* h. *iff* ×. *Qed*.

Lemma qimpl_refl : $\forall Q,$

```
Q ==> Q.
```

Proof. *intros* Q v. *apply*s *himpl_refl*. *Qed*.

Hint *Resolve* *himpl_refl* *qimpl_refl*.

34.3.4 Properties of **hstar**

Lemma hstar_intro : $\forall H1 H2 h1 h2,$

```
H1 h1 →
H2 h2 →
Fmap.disjoint h1 h2 →
(H1 \* H2) (Fmap.union h1 h2).
```

Proof. *intros*. $\exists \times h1 h2$. *Qed*.

Lemma hstar_comm : $\forall H1 H2,$

```
H1 \* H2 = H2 \* H1.
```

Proof.

```
unfold hprop, hstar. intros H1 H2. applys himpl_antisym.
{ intros h (h1&h2&M1&M2&D&U).
  rewrite × Fmap.union_comm_of_disjoint in U.  $\exists \times h2 h1.$  }
{ intros h (h1&h2&M1&M2&D&U).
  rewrite × Fmap.union_comm_of_disjoint in U.  $\exists \times h2 h1.$  }
```

Qed.

Lemma hstar_assoc : $\forall H1 H2 H3,$

```
(H1 \* H2) \* H3 = H1 \* (H2 \* H3).
```

Proof.

```
intros H1 H2 H3. applys himpl_antisym; intros h.
{ intros (h'&h3&(h1&h2&M3&M4&D'&U')&M2&D&U). subst h'.
   $\exists h1 (h2 \oplus h3).$  splits ×. { applys × hstar_intro. } }
{ intros (h1&h'&M1&(h2&h3&M3&M4&D'&U')&D&U). subst h'.
   $\exists (h1 \oplus h2) h3.$  splits ×. { applys × hstar_intro. } }
```

Qed.

Lemma hstar_hempty_l : $\forall H,$

```
\[] \* H = H.
```

Proof.

```
intros. applys himpl_antisym; intros h.
{ intros (h1&h2&M1&M2&D&U). hnf in M1. subst. rewrite× Fmap.union_empty_l.
}
{ intros M. ∃ (@Fmap.empty loc val) h. splits×. { hnfs×. } }
Qed.
```

Lemma hstar_hexists : ∀ A (J:A→hprop) H,
(hexists J) * H = hexists (fun x ⇒ (J x) * H).

Proof.

```
intros. applys himpl_antisym; intros h.
{ intros (h1&h2&(x&M1)&M2&D&U). ∃ x h1 h2. }
{ intros (x&(h1&h2&M1&M2&D&U)). ∃ h1 h2. splits×. { ∃ x. } }
Qed.
```

Lemma hstar_hforall : ∀ H A (J:A→hprop),
(hforall J) * H ==> hforall (J *+ H).

Proof.

```
intros. intros h M. destruct M as (h1&h2&M1&M2&D&U). intros x. ∃ x h1 h2.
Qed.
```

Lemma himpl_frame_l : ∀ H2 H1 H1',
H1 ==> H1' →
(H1 * H2) ==> (H1' * H2).

Proof. introv W (h1&h2&?). ∃ x h1 h2. Qed.

Additional, symmetric results, useful for tactics

Lemma hstar_hempty_r : ∀ H,
H * [] = H.

Proof.

```
applys neutral_r_of_comm_neutral_l. applys× hstar_comm. applys× hstar_hempty_l.
Qed.
```

Lemma himpl_frame_r : ∀ H1 H2 H2',
H2 ==> H2' →
(H1 * H2) ==> (H1 * H2').

Proof.

```
introv M. do 2 rewrite (@hstar_comm H1). applys× himpl_frame_l.
Qed.
```

34.3.5 Properties of **hpure**

Lemma hstar_hpure_l : ∀ P H h,
(\[P] * H) h = (P ∧ H h).

Proof.

```

intros. apply prop_ext. unfold hpure.
rewrite hstar_hexists. rewrite× hstar_hempty_l.
iff (p&M) (p&M). { split×. } { ∃× p. }
Qed.

```

```

Lemma himpl_hstar_hpure_r : ∀ P H H',
P →
(H ==> H') →
H ==> (\[P] \* H').

```

Proof. *introv HP W. intros h K. rewrite× hstar_hpure_l.* Qed.

```

Lemma himpl_hstar_hpure_l : ∀ P H H',
(P → H ==> H') →
(\[P] \* H) ==> H'.

```

Proof. *introv W Hh. rewrite hstar_hpure_l in Hh. applys× W.* Qed.

34.3.6 Properties of hexists

```

Lemma himpl_hexists_l : ∀ A H (J:A→hprop),
(∀ x, J x ==> H) →
(hexists J) ==> H.

```

Proof. *introv W. intros h (x&Hh). applys× W.* Qed.

```

Lemma himpl_hexists_r : ∀ A (x:A) H J,
(H ==> J x) →
H ==> (hexists J).

```

Proof. *introv W. intros h. ∃ x. apply× W.* Qed.

```

Lemma himpl_hexists : ∀ A (J1 J2:A→hprop),
J1 ==> J2 →
hexists J1 ==> hexists J2.

```

Proof.

introv W. applys himpl_hexists_l. intros x. applys himpl_hexists_r W.

Qed.

34.3.7 Properties of hforall

```

Lemma himpl_hforall_r : ∀ A (J:A→hprop) H,
(∀ x, H ==> J x) →
H ==> (hforall J).

```

Proof. *introv M. intros h Hh x. apply× M.* Qed.

```

Lemma himpl_hforall_l : ∀ A x (J:A→hprop) H,
(J x ==> H) →
(hforall J) ==> H.

```

Proof. *introv M. intros h Hh. apply× M.* Qed.

```

Lemma himpl_hforall : ∀ A (J1 J2:A→hprop),
  J1 ==> J2 →
  hforall J1 ==> hforall J2.

```

Proof.

```
  introv W. applys himpl_hforall_r. intros x. applys himpl_hforall_l W.
```

Qed.

34.3.8 Properties of hsingle

```

Lemma hstar_hsingl_same_loc : ∀ p v1 v2,
  (p ~~> v1) ∗ (p ~~> v2) ==> \[False].

```

Proof.

```
  intros. unfold hsingl. intros h (h1&h2&E1&E2&D&E). false.
  subst. applys× Fmap.disjoint_single_single_same_inv.
```

Qed.

34.3.9 Basic Tactics for Simplifying Entailments

xsiml performs immediate simplifications on entailment relations.

```
Hint Rewrite hstar_assoc hstar_hempty_l hstar_hempty_r : hstar.
```

Tactic Notation "xsiml" :=

```

  try solve [ apply qimpl_refl ];
  try match goal with ⊢ _ ==> _ ⇒ intros ? end;
  autorewrite with hstar; repeat match goal with
  | ⊢ ?H ∗ _ ==> ?H ∗ _ ⇒ apply himpl_frame_r
  | ⊢ _ ∗ ?H ==> _ ∗ ?H ⇒ apply himpl_frame_l
  | ⊢ _ ∗ ?H ==> ?H ∗ _ ⇒ rewrite hstar_comm; apply himpl_frame_r
  | ⊢ ?H ∗ _ ==> _ ∗ ?H ⇒ rewrite hstar_comm; apply himpl_frame_l
  | ⊢ ?H ==> ?H ⇒ apply himpl_refl
  | ⊢ ?H ==> ?H' ⇒ is_evar H'; apply himpl_refl end.

```

Tactic Notation "xsiml" "*" := xsiml; auto_star.

xchange helps rewriting in entailments.

```

Lemma xchange_lemma : ∀ H1 H1',
  H1 ==> H1' → ∀ H H' H2,
  H ==> H1 ∗ H2 →
  H1' ∗ H2 ==> H' →
  H ==> H'.

```

Proof.

```
  introv M1 M2 M3. applys himpl_trans M2. applys himpl_trans M3.
  applys himpl_frame_l. applys M1.
```

Qed.

```
Tactic Notation "xchange" constr(M) :=
  forwards_nounfold_then M ltac:(fun K =>
    eapply xchange_lemma; [ eapply K | xsimpl | ]).
```

34.3.10 Reformulation of the Evaluation Rules for Primitive Operations.

```
Lemma eval_ref_sep : ∀ s1 s2 v p,
  s2 = Fmap.single p v →
  Fmap.disjoint s2 s1 →
  eval s1 (val_ref v) (Fmap.union s2 s1) (val_loc p).
```

Proof.

```
introv → D. forwards Dv: Fmap.indom_single p v.
rewrite ← Fmap.update_eq_union_single. applys× eval_ref.
{ intros N. applys× Fmap.disjoint_inv_not_indom_both D N. }
```

Qed.

```
Lemma eval_get_sep : ∀ s s2 p v,
  s = Fmap.union (Fmap.single p v) s2 →
  eval s (val_get (val_loc p)) s v.
```

Proof.

```
introv →. forwards Dv: Fmap.indom_single p v. applys_eq eval_get.
{ rewrite× Fmap.read_union_l. rewrite× Fmap.read_single. }
{ applys× Fmap.indom_union_l. }
```

Qed.

```
Lemma eval_set_sep : ∀ s1 s2 h2 p v1 v2,
  s1 = Fmap.union (Fmap.single p v1) h2 →
  s2 = Fmap.union (Fmap.single p v2) h2 →
  Fmap.disjoint (Fmap.single p v1) h2 →
  eval s1 (val_set (val_loc p) v2) s2 val_unit.
```

Proof.

```
introv → → D. forwards Dv: Fmap.indom_single p v1. applys_eq eval_set.
{ rewrite× Fmap.update_union_l. fequals. rewrite× Fmap.update_single. }
{ applys× Fmap.indom_union_l. }
```

Qed.

```
Lemma eval_free_sep : ∀ s1 s2 v p,
  s1 = Fmap.union (Fmap.single p v) s2 →
  Fmap.disjoint (Fmap.single p v) s2 →
  eval s1 (val_free p) s2 val_unit.
```

Proof.

```
introv → D. forwards Dv: Fmap.indom_single p v. applys_eq eval_free.
{ rewrite× Fmap.remove_union_single_l. intros Dl.
```

```

    applys Fmap.disjoint_inv_not_indom_both D Dl. applys Fmap.indom_single. }
{ applys× Fmap.indom_union_l. }

Qed.

```

34.4 Hoare Logic

34.4.1 Definition of Total Correctness Hoare Triples

Definition hoare (*t:trm*) (*H:hprop*) (*Q:val* \rightarrow *hprop*) :=
 $\forall h, H \ h \rightarrow \exists h' v, \text{eval } h \ t \ h' \ v \wedge Q \ v \ h'.$

34.4.2 Structural Rules for Hoare Triples

Lemma hoare_conseq : $\forall t H' Q' H Q,$
 $\text{hoare } t H' Q' \rightarrow$
 $H ==> H' \rightarrow$
 $Q' ==> Q \rightarrow$
 $\text{hoare } t H Q.$

Proof.

introv M MH MQ HF. forwards (h'&v&R&K): M h. { applys× MH. }
 $\exists h' v. \text{splits} \times. \{ \text{applys} \times MQ. \}$

Qed.

Lemma hoare_hexists : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\forall x, \text{hoare } t (J x) Q) \rightarrow$
 $\text{hoare } t (\text{hexists } J) Q.$

Proof. introv M. intros h (x&Hh). applys M Hh. Qed.

Lemma hoare_hpure : $\forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{hoare } t H Q) \rightarrow$
 $\text{hoare } t (\setminus [P] \setminus * H) Q.$

Proof.

introv M. intros h (h1&h2&(M1&HP)&M2&D&U). hnf in HP. subst.
 rewrite Fmap.union_empty_l. applys× M.

Qed.

34.4.3 Reasoning Rules for Terms, for Hoare Triples

Lemma hoare_val : $\forall v H Q,$
 $H ==> Q \ v \rightarrow$
 $\text{hoare } (\text{trm_val } v) H Q.$

Proof.

introv M. intros h K. $\exists h v. \text{splits}.$

$\{ \text{applys eval_val.} \}$
 $\{ \text{applys } M. \}$

Qed.

Lemma hoare_fix : $\forall f x t1 H Q,$
 $H ==> Q (\text{val_fix } f x t1) \rightarrow$
 $\text{hoare} (\text{trm_fix } f x t1) H Q.$

Proof.

$\text{intros } M. \text{ intros } h K. \exists h (\text{val_fix } f x t1). \text{ splits.}$
 $\{ \text{applys } \times \text{ eval_fix.} \}$
 $\{ \text{applys } \times M. \}$

Qed.

Lemma hoare_app : $\forall v1 v2 f x t1 H Q,$
 $v1 = \text{val_fix } f x t1 \rightarrow$
 $\text{hoare} (\text{subst } x v2 (\text{subst } f v1 t1)) H Q \rightarrow$
 $\text{hoare} (\text{trm_app } v1 v2) H Q.$

Proof.

$\text{intros } E M. \text{ intros } s K0. \text{ forwards } (s' \& v \& R1 \& K1): (\text{rm } M) K0.$
 $\exists s' v. \text{ splits. } \{ \text{applys eval_app } E R1. \} \{ \text{applys } K1. \}$

Qed.

Lemma hoare_let : $\forall x t1 t2 H Q Q1,$
 $\text{hoare} t1 H Q1 \rightarrow$
 $(\forall v1, \text{hoare} (\text{subst } x v1 t2) (Q1 v1) Q) \rightarrow$
 $\text{hoare} (\text{trm_let } x t1 t2) H Q.$

Proof.

$\text{intros } M1 M2 Hh. \text{ forwards } \times (h1' \& v1 \& R1 \& K1): (\text{rm } M1).$
 $\text{forwards } \times (h2' \& v2 \& R2 \& K2): (\text{rm } M2).$
 $\exists h2' v2. \text{ splits } \times. \{ \text{applys } \times \text{ eval_let.} \}$

Qed.

Lemma hoare_if : $\forall (b:\text{bool}) t1 t2 H Q,$
 $\text{hoare} (\text{if } b \text{ then } t1 \text{ else } t2) H Q \rightarrow$
 $\text{hoare} (\text{trm_if } b t1 t2) H Q.$

Proof.

$\text{intros } M1. \text{ intros } h Hh. \text{ forwards } \times (h1' \& v1 \& R1 \& K1): (\text{rm } M1).$
 $\exists h1' v1. \text{ splits } \times. \{ \text{applys } \times \text{ eval_if.} \}$

Qed.

34.4.4 Specification of Primitive Operations, for Hoare Triples.

Lemma hoare_add : $\forall H n1 n2,$
 $\text{hoare} (\text{val_add } n1 n2)$
 H

```
(fun r => \[r = val_int (n1 + n2)] \* H).
```

Proof.

```
intros. intros s K0. \exists s (val_int (n1 + n2)). split.  
{ applys eval_add. }  
{ rewrite hstar_hpure_l. }
```

Qed.

Lemma hoare_div : \forall H n1 n2,

```
n2 \neq 0 →  
hoare (val_div n1 n2)  
H  
(fun r => \[r = val_int (Z.quot n1 n2)] \* H).
```

Proof.

```
introv N. intros s K0. \exists s (val_int (Z.quot n1 n2)). split.  
{ applys eval_div N. }  
{ rewrite hstar_hpure_l. }
```

Qed.

Lemma hoare_ref : \forall H v,

```
hoare (val_ref v)  
H  
(fun r => (\exists p, \[r = val_loc p] \* p \simv v) \* H).
```

Proof.

```
intros. intros s1 K0. forwards (p & D & N): (Fmap.single_fresh 0%nat s1 v).  
\exists (Fmap.union (Fmap.single p v) s1) (val_loc p). split.  
{ applys eval_ref_sep D. }  
{ applys hstar_intro.  
{ \exists p. rewrite hstar_hpure_l. split. { hnfs. } } }
```

Qed.

Lemma hoare_get : \forall H v p,

```
hoare (val_get p)  
((p \simv v) \* H)  
(fun r => \[r = v] \* (p \simv v) \* H).
```

Proof.

```
intros. intros s K0. \exists s v. split.  
{ destruct K0 as (s1 & s2 & P2 & D & U). applys eval_get_sep U. }  
{ rewrite hstar_hpure_l. }
```

Qed.

Lemma hoare_set : \forall H w p v,

```
hoare (val_set (val_loc p) w)  
((p \simv v) \* H)  
(fun r => (p \simv w) \* H).
```

Proof.

```

intros. intros s1 (h1&h2&->&P2&D&U).
 $\exists$  (Fmap.union (Fmap.single p w) h2) val_unit. split.
{ applys eval_sep U D. auto. }
{ applys  $\times$  hstar_intro.
{ hnfs  $\times$ . }
{ applys Fmap.disjoint_single_set D. } }

```

Qed.

```

Lemma hoare_free :  $\forall H p v,$ 
  hoare (val_free (val_loc p))
    ((p  $\sim\sim>$  v)  $\backslash*$  H)
    (fun r  $\Rightarrow$  H).

```

Proof.

```

intros. intros s1 (h1&h2&->&P2&D&U).  $\exists$  h2 val_unit. split.
{ applys eval_free_sep U D. }
{ auto. }

```

Qed.

From now on, all operators have opaque definitions

Global Opaque hempty hpure hstar hsingle hexists hforall.

34.5 Separation Logic

34.5.1 Definition of Separation Logic triples

```

Definition triple (t:trm) (H:hprop) (Q:val  $\rightarrow$  hprop) : Prop :=
   $\forall (H':hprop)$ , hoare t (H  $\backslash*$  H') (Q  $\backslash*+$  H').

```

34.5.2 Structural Rules

```

Lemma triple_conseq :  $\forall t H' Q' H Q,$ 
  triple t H' Q'  $\rightarrow$ 
  H ==> H'  $\rightarrow$ 
  Q' ==> Q  $\rightarrow$ 
  triple t H Q.

```

Proof.

```

introv M MH MQ. intros HF. applys hoare_conseq M.
{ xchange MH. xsimpl. }
{ intros x. xchange (MQ x). xsimpl. }

```

Qed.

```

Lemma triple_frame :  $\forall t H Q H',$ 
  triple t H Q  $\rightarrow$ 

```

$\text{triple } t (H \setminus * H') (Q \setminus *+ H').$

Proof.

*introv M. intros HF. applys hoare_conseq (M (HF \setminus * H')); xsimpl.*
Qed.

Lemma triple_hexists : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\forall x, \text{triple } t (J x) Q) \rightarrow$
 $\text{triple } t (\text{hexists } J) Q.$

Proof.

introv M. intros HF. rewrite hstar_hexists.
applys hoare_hexists. intros. applys \times M.

Qed.

Lemma triple_hpure : $\forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{triple } t H Q) \rightarrow$
 $\text{triple } t (\setminus [P] \setminus * H) Q.$

Proof.

introv M. intros HF. rewrite hstar_assoc.
applys hoare_hpure. intros. applys \times M.

Qed.

34.5.3 Reasoning Rules for Terms

Lemma triple_val : $\forall v H Q,$
 $H ==> Q v \rightarrow$
 $\text{triple } (\text{trm_val } v) H Q.$

Proof.

introv M. intros HF. applys hoare_val. { xchange M. xsimpl. }

Qed.

Lemma triple_fix : $\forall f x t1 H Q,$
 $H ==> Q (\text{val_fix } f x t1) \rightarrow$
 $\text{triple } (\text{trm_fix } f x t1) H Q.$

Proof.

introv M. intros HF. applys hoare_fix. { xchange M. xsimpl. }

Qed.

Lemma triple_let : $\forall x t1 t2 Q1 H Q,$
 $\text{triple } t1 H Q1 \rightarrow$
 $(\forall v1, \text{triple } (\text{subst } x v1 t2) (Q1 v1) Q) \rightarrow$
 $\text{triple } (\text{trm_let } x t1 t2) H Q.$

Proof.

introv M1 M2. intros HF. applys hoare_let.
{ applys M1. }
{ intros v. applys hoare_conseq M2; xsimpl. }

Qed.

```
Lemma triple_if : ∀ (b:bool) t1 t2 H Q,  
  triple (if b then t1 else t2) H Q →  
  triple (trm_if b t1 t2) H Q.
```

Proof.

```
  introv M1. intros HF. applys hoare_if. applys M1.
```

Qed.

```
Lemma triple_app : ∀ v1 v2 f x t1 H Q,  
  v1 = val_fix f x t1 →  
  triple (subst x v2 (subst f v1 t1)) H Q →  
  triple (trm_app v1 v2) H Q.
```

Proof.

```
  introv E M1. intros H'. applys hoare_app E. applys M1.
```

Qed.

34.5.4 Specification of Primitive Operations

```
Lemma triple_add : ∀ n1 n2,  
  triple (val_add n1 n2)  
  \[]  
  (fun r ⇒ \[r = val_int (n1 + n2)]).
```

Proof. intros. intros H'. applys hoare_conseq hoare_add; xsimpl×. Qed.

```
Lemma triple_div : ∀ n1 n2,  
  n2 ≠ 0 →  
  triple (val_div n1 n2)  
  \[]  
  (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Proof. intros. intros H'. applys× hoare_conseq hoare_div; xsimpl×. Qed.

```
Lemma triple_ref : ∀ v,  
  triple (val_ref v)  
  \[]  
  (fun r ⇒ \exists p, \[r = val_loc p] \* p ~~> v).
```

Proof. intros. intros HF. applys hoare_conseq hoare_ref; xsimpl×. Qed.

```
Lemma triple_get : ∀ v p,  
  triple (val_get (val_loc p))  
  (p ~~> v)  
  (fun x ⇒ \[x = v] \* (p ~~> v)).
```

Proof. intros. intros HF. applys hoare_conseq hoare_get. xsimpl×. xsimpl×. Qed.

```
Lemma triple_set : ∀ w p v,  
  triple (val_set (val_loc p) w)  
  (p ~~> v)
```

```
(fun _ => p ~~> w).
```

Proof. intros. intros *HF*. applys hoare_conseq hoare_set; xsimpl. Qed.

```
Lemma triple_free : ∀ p v,  
  triple (val_free (val_loc p))  
  (p ~~> v)  
  (fun _ => [] ).
```

Proof. intros. intros *HF*. applys hoare_conseq hoare_free; xsimpl. Qed.

34.6 Bonus: Example Proof

See chapter Rules for comments on this proof.

OCaml:

```
let incr p = let n = !p in let m = n+1 in p := m
```

Definition of the function *incr*, using low-level syntax.

Open Scope string_scope.

```
Definition incr : val :=  
  val_fix "f" "p" (  
    trm_let "n" (trm_app val_get (trm_var "p")) (  
      trm_let "m" (trm_app (trm_app val_add  
        (trm_var "n")) (val_int 1)) (  
          trm_app (trm_app val_set (trm_var "p")) (trm_var "m")))).
```

Specification of the function *incr*.

```
Lemma triple_incr : ∀ (p:loc) (n:int),  
  triple (trm_app incr p)  
  (p ~~> n)  
  (fun _ => p ~~> (n+1)).
```

Verification of *incr*, applying the reasoning rules by hand.

Proof using.

```
intros. applys triple_app. { reflexivity. } simpl.  
applys triple_let. { apply triple_get. }  
intros n'. simpl. apply triple_hpure. intros →.  
applys triple_let. { applys triple_conseq.  
  { applys triple_frame. applys triple_add. }  
  { xsimpl. }  
  { xsimpl. } }  
intros m'. simpl. apply triple_hpure. intros →.  
{ applys triple_set. }
```

Qed.

Chapter 35

Library `SLF.LibSepReference`

35.1 LibSepReference: Appendix - The Full Construction

This file provides a pretty-much end-to-end construction of a weakest-precondition style characteristic formula generator (the function named `wpgen` in `WPgen`), for a core programming language with programs assumed to be in A-normal form.

This file is included by the chapters from the course.

Set Implicit Arguments.

From *SLF* Require Export `LibCore`.

From *SLF* Require Export `LibSepTLCbuffer` `LibSepVar` `LibSepFmap`.

From *SLF* Require `LibSepSimpl`.

Module `FMAP` := `LIBSEPFMAP`.

35.2 Imports

35.2.1 Extensionality Axioms

These standard extensionality axioms may also be found in the `LibAxiom.v` file associated with the TLC library.

Axiom *functional_extensionality* : $\forall A B (f g:A \rightarrow B)$,
 $(\forall x, f x = g x) \rightarrow$
 $f = g$.

Axiom *propositional_extensionality* : $\forall (P Q:\text{Prop})$,
 $(P \leftrightarrow Q) \rightarrow$
 $P = Q$.

35.2.2 Variables

The file *LibSepVar.v*, whose definitions are imported in the header to the present file, defines the type `var` as an alias for `string`. It also provides the boolean function `var_eq x y` to compare two variables, and the tactic `case_var` to perform case analysis on expressions of the form `if var_eq x y then .. else ..` that appear in the goal.

35.2.3 Finite Maps

The file *LibSepFmap.v*, which is bound by the short name `FMAP` in the header, provides a formalization of finite maps. These maps are used to represent heaps in the semantics. The library provides a tactic called `fmap_disjoint` to automate disjointness proofs, and a tactic called `fmap_eq` for proving equalities between heaps modulo associativity and commutativity. Without these two tactics, proofs involving finite maps would be much more tedious and fragile.

35.3 Source Language

35.3.1 Syntax

The grammar of primitive operations includes a number of operations.

Inductive `prim` : Type :=

```
| val_ref : prim
| val_get : prim
| val_set : prim
| val_free : prim
| val_rand : prim
| val_neg : prim
| val_opp : prim
| val_eq : prim
| val_add : prim
| val_neq : prim
| val_sub : prim
| val_mul : prim
| val_div : prim
| val_mod : prim
| val_le : prim
| val_lt : prim
| val_ge : prim
| val_gt : prim
| val_ptr_add : prim.
```

Locations are defined as natural numbers.

Definition `loc` : Type := `nat`.

The null location corresponds to address zero.

Definition `null` : loc := 0%nat.

The grammar of closed values includes includes basic values such as `int` and `bool`, but also locations, closures. It also includes two special values, `val_uninit` used in the formalization of arrays, and `val_error` used for stating semantics featuring error-propagation.

Inductive `val` : Type :=

- | `val_unit` : `val`
- | `val_bool` : `bool` → `val`
- | `val_int` : `int` → `val`
- | `val_loc` : `loc` → `val`
- | `val_prim` : `prim` → `val`
- | `val_fun` : `var` → `trm` → `val`
- | `val_fix` : `var` → `var` → `trm` → `val`
- | `val_uninit` : `val`
- | `val_error` : `val`

The grammar of terms includes values, variables, functions, applications, sequence, let-bindings, and conditions. Sequences are redundant with let-bindings, but are useful in practice to avoid binding dummy names, and serve on numerous occasion as a warm-up before proving results on let-bindings.

with `trm` : Type :=

- | `trm_val` : `val` → `trm`
- | `trm_var` : `var` → `trm`
- | `trm_fun` : `var` → `trm` → `trm`
- | `trm_fix` : `var` → `var` → `trm` → `trm`
- | `trm_app` : `trm` → `trm` → `trm`
- | `trm_seq` : `trm` → `trm` → `trm`
- | `trm_let` : `var` → `trm` → `trm` → `trm`
- | `trm_if` : `trm` → `trm` → `trm` → `trm`.

A state consists of a finite map from location to values. Records and arrays are represented as sets of consecutive cells, preceeded by a header field describing the length of the block.

Definition `state` : Type := `fmap` loc `val`.

The type `heap`, a.k.a. `state`. By convention, the “state” refers to the full memory state when describing the semantics, while the “heap” potentially refers to only a fraction of the memory state, when defining Separation Logic predicates.

Definition `heap` : Type := `state`.

35.3.2 Coq Tweaks

$h1 \setminus\cup h2$ is a notation for union of two heaps.

Notation "h1 $\setminus\cup$ h2" := (Fmap.union h1 h2)
(at level 37, right associativity).

Implicit types associated with meta-variables.

```
Implicit Types f : var.  
Implicit Types b : bool.  
Implicit Types p : loc.  
Implicit Types n : int.  
Implicit Types v w r vf vx : val.  
Implicit Types t : trm.  
Implicit Types h : heap.  
Implicit Types s : state.
```

The types of values and heap values are inhabited.

Global Instance Inhab_val : Inhab val.
Proof using. apply (Inhab_of_val val_unit). Qed.

Coercions to improve conciseness in the statement of evaluation rules.

```
Coercion val_bool : bool >-> val.  
Coercion val_int : Z >-> val.  
Coercion val_loc : loc >-> val.  
Coercion val_prim : prim >-> val.  
  
Coercion trm_val : val >-> trm.  
Coercion trm_var : var >-> trm.  
Coercion trm_app : trm >-> Funclass.
```

35.3.3 Substitution

The standard capture-avoiding substitution, written subst x v t.

```
Fixpoint subst (y:var) (v':val) (t:trm) : trm :=  
  let aux t := subst y v' t in  
  let if_y_eq x t1 t2 := if var_eq x y then t1 else t2 in  
  match t with  
  | trm_val v => trm_val v  
  | trm_var x => if_y_eq x (trm_val v') t  
  | trm_fun x t1 => trm_fun x (if_y_eq x t1 (aux t1))  
  | trm_fix f x t1 => trm_fix f x (if_y_eq f t1 (if_y_eq x t1 (aux t1)))  
  | trm_app t1 t2 => trm_app (aux t1) (aux t2)  
  | trm_seq t1 t2 => trm_seq (aux t1) (aux t2)  
  | trm_let x t1 t2 => trm_let x (aux t1) (if_y_eq x t2 (aux t2))
```

```
| trm_if t0 t1 t2 ⇒ trm_if (aux t0) (aux t1) (aux t2)
end.
```

35.3.4 Semantics

Evaluation rules for unary operations are captured by the predicate *redupop op v1 v2*, which asserts that *op v1* evaluates to *v2*.

```
Inductive evalunop : prim → val → val → Prop :=
| evalunop_neg : ∀ b1,
  evalunop val_neg (val_bool b1) (val_bool (neg b1))
| evalunop_opp : ∀ n1,
  evalunop val_opp (val_int n1) (val_int (- n1)).
```

Evaluation rules for binary operations are captured by the predicate *redupop op v1 v2 v3*, which asserts that *op v1 v2* evaluates to *v3*.

```
Inductive evalbinop : val → val → val → val → Prop :=
| evalbinop_eq : ∀ v1 v2,
  evalbinop val_eq v1 v2 (val_bool (isTrue (v1 = v2)))
| evalbinop_neq : ∀ v1 v2,
  evalbinop val_neq v1 v2 (val_bool (isTrue (v1 ≠ v2)))
| evalbinop_add : ∀ n1 n2,
  evalbinop val_add (val_int n1) (val_int n2) (val_int (n1 + n2))
| evalbinop_sub : ∀ n1 n2,
  evalbinop val_sub (val_int n1) (val_int n2) (val_int (n1 - n2))
| evalbinop_mul : ∀ n1 n2,
  evalbinop val_mul (val_int n1) (val_int n2) (val_int (n1 × n2))
| evalbinop_div : ∀ n1 n2,
  n2 ≠ 0 →
  evalbinop val_div (val_int n1) (val_int n2) (val_int (Z.quot n1 n2))
| evalbinop_mod : ∀ n1 n2,
  n2 ≠ 0 →
  evalbinop val_mod (val_int n1) (val_int n2) (val_int (Z.rem n1 n2))
| evalbinop_le : ∀ n1 n2,
  evalbinop val_le (val_int n1) (val_int n2) (val_bool (isTrue (n1 ≤ n2)))
| evalbinop_lt : ∀ n1 n2,
  evalbinop val_lt (val_int n1) (val_int n2) (val_bool (isTrue (n1 < n2)))
| evalbinop_ge : ∀ n1 n2,
  evalbinop val_ge (val_int n1) (val_int n2) (val_bool (isTrue (n1 ≥ n2)))
| evalbinop_gt : ∀ n1 n2,
  evalbinop val_gt (val_int n1) (val_int n2) (val_bool (isTrue (n1 > n2)))
| evalbinop_ptr_add : ∀ p1 p2 n,
  (p2:int) = p1 + n →
  evalbinop val_ptr_add (val_loc p1) (val_int n) (val_loc p2).
```

The predicate `trm_is_val t` asserts that `t` is a value.

```
Definition trm_is_val (t:trm) : Prop :=
  match t with trm_val v => True | _ => False end.
```

Big-step evaluation judgement, written `eval s t s' v`.

```
Inductive eval : heap → trm → heap → val → Prop :=
| eval_val : ∀ s v,
  eval s (trm_val v) s v
| eval_fun : ∀ s x t1,
  eval s (trm_fun x t1) s (val_fun x t1)
| eval_fix : ∀ s f x t1,
  eval s (trm_fix f x t1) s (val_fix f x t1)
| eval_app_args : ∀ s1 s2 s3 s4 t1 t2 v1 v2 r,
  (¬ trm_is_val t1 ∨ ¬ trm_is_val t2) →
  eval s1 t1 s2 v1 →
  eval s2 t2 s3 v2 →
  eval s3 (trm_app v1 v2) s4 r →
  eval s1 (trm_app t1 t2) s4 r
| eval_app_fun : ∀ s1 s2 v1 v2 x t1 v,
  v1 = val_fun x t1 →
  eval s1 (subst x v2 t1) s2 v →
  eval s1 (trm_app v1 v2) s2 v
| eval_app_fix : ∀ s1 s2 v1 v2 f x t1 v,
  v1 = val_fix f x t1 →
  eval s1 (subst x v2 (subst f v1 t1)) s2 v →
  eval s1 (trm_app v1 v2) s2 v
| eval_seq : ∀ s1 s2 s3 t1 t2 v1 v,
  eval s1 t1 s2 v1 →
  eval s2 t2 s3 v →
  eval s1 (trm_seq t1 t2) s3 v
| eval_let : ∀ s1 s2 s3 x t1 t2 v1 r,
  eval s1 t1 s2 v1 →
  eval s2 (subst x v1 t2) s3 r →
  eval s1 (trm_let x t1 t2) s3 r
| eval_if : ∀ s1 s2 b v t1 t2,
  eval s1 (if b then t1 else t2) s2 v →
  eval s1 (trm_if (val_bool b) t1 t2) s2 v
| eval_unop : ∀ op m v1 v,
  evalunop op v1 v →
  eval m (op v1) m v
| eval_binop : ∀ op m v1 v2 v,
  evalbinop op v1 v2 v →
  eval m (op v1 v2) m v
```

```

| eval_ref : ∀ s v p,
  ↗ Fmap.indom s p →
  eval s (val_ref v) (Fmap.update s p v) (val_loc p)
| eval_get : ∀ s p v,
  Fmap.indom s p →
  v = Fmap.read s p →
  eval s (val_get (val_loc p)) s v
| eval_set : ∀ s p v,
  Fmap.indom s p →
  eval s (val_set (val_loc p) v) (Fmap.update s p v) val_unit
| eval_free : ∀ s p,
  Fmap.indom s p →
  eval s (val_free (val_loc p)) (Fmap.remove s p) val_unit.

```

Specialized evaluation rules for addition and division, for avoiding the indirection via eval_binop in the course.

Lemma eval_add : $\forall s n1 n2,$
 $\quad \text{eval } s (\text{val_add} (\text{val_int } n1) (\text{val_int } n2)) s (\text{val_int } (n1 + n2)).$

Proof using. intros. applys eval_binop. applys evalbinop_add. Qed.

Lemma eval_div : $\forall s n1 n2,$
 $n2 \neq 0 \rightarrow$
 $\quad \text{eval } s (\text{val_div} (\text{val_int } n1) (\text{val_int } n2)) s (\text{val_int } (\text{Z.quot } n1 n2)).$

Proof using. intros. applys eval_binop. applys evalbinop_div. Qed.

eval_like $t1 t2$ asserts that $t2$ evaluates like $t1$. In particular, this relation hold whenever $t2$ reduces in small-step to $t1$.

Definition eval_like ($t1 t2:\text{trm}$) : Prop :=
 $\quad \forall s s' v, \text{eval } s t1 s' v \rightarrow \text{eval } s t2 s' v.$

35.4 Heap Predicates

We next define heap predicates and establish their properties. All this material is wrapped in a module, allowing us to instantiate the functor from `LibSepSimpl` that defines the tactic `xsimpl`. Module `SEPSIMPLARGS`.

35.4.1 Hprop and Entailment

*Declare Scope hprop_scope.
Open Scope hprop_scope.*

The type of heap predicates is named `hprop`.

Definition `hprop` := `heap → Prop`.

Implicit types for meta-variables.

Implicit Types $P : \text{Prop}$.

Implicit Types $H : \text{hprop}$.

Implicit Types $Q : \text{val} \rightarrow \text{hprop}$.

Entailment for heap predicates, written $H1 ==> H2$. This entailment is linear.

Definition $\text{himpl } (H1\ H2:\text{hprop}) : \text{Prop} :=$

$\forall h, H1\ h \rightarrow H2\ h$.

Notation " $H1 ==> H2$ " := ($\text{himpl } H1\ H2$) (at level 55) : hprop_scope .

Entailment between postconditions, written $Q1 ==> Q2$.

Definition $\text{qimpl } A\ (Q1\ Q2:A \rightarrow \text{hprop}) : \text{Prop} :=$

$\forall (v:A), Q1\ v ==> Q2\ v$.

Notation " $Q1 ==> Q2$ " := ($\text{qimpl } Q1\ Q2$) (at level 55) : hprop_scope .

35.4.2 Definition of Heap Predicates

The core heap predicates are defined next, together with the associated notation:

- \emptyset denotes the empty heap predicate
- $\lceil P \rceil$ denotes a pure fact
- $\lceil Top \rceil$ denotes the true heap predicate (affine)
- $p \sim \sim > v$ denotes a singleton heap
- $H1 \setminus^* H2$ denotes the separating conjunction
- $Q1 \setminus^*+ H2$ denotes the separating conjunction extending a postcondition
- $\exists x, H$ denotes an existential quantifier
- $\forall x, H$ denotes a universal quantifier
- $H1 \setminus^* H2$ denotes a magic wand between heap predicates
- $Q1 \setminus^* Q2$ denotes a magic wand between postconditions.

Definition $\text{hempty} : \text{hprop} :=$

$\text{fun } h \Rightarrow (h = \text{Fmap.empty})$.

Definition $\text{hsingle } (p:\text{loc})\ (v:\text{val}) : \text{hprop} :=$

$\text{fun } h \Rightarrow (h = \text{Fmap.single } p\ v)$.

Definition $\text{hstar } (H1\ H2 : \text{hprop}) : \text{hprop} :=$

$\text{fun } h \Rightarrow \exists h1\ h2, H1\ h1$

$\wedge H2 h2$
 $\wedge \text{Fmap.disjoint } h1 h2$
 $\wedge h = \text{Fmap.union } h1 h2.$

Definition `hexists A (J:A→hprop) : hprop :=`
`fun h ⇒ ∃ x, J x h.`

Definition `hforall (A : Type) (J : A → hprop) : hprop :=`
`fun h ⇒ ∀ x, J x h.`

Notation "`\[]`" := (`hempty`)
(at level 0) : `hprop_scope`.

Notation "`p '~~> v`" := (`hsingle p v`) (at level 32) : `hprop_scope`.

Notation "`H1 '*' H2`" := (`hstar H1 H2`)
(at level 41, right associativity) : `hprop_scope`.

Notation "`'\exists' x1 .. xn , H`" :=
(`hexists (fun x1 ⇒ .. (hexists (fun xn ⇒ H)) ..)`)
(at level 39, `x1 binder, H at level 50, right associativity,`
format "`'['\exists' / ' x1 .. xn , / ' H]'`") : `hprop_scope`.

Notation "`'\forall' x1 .. xn , H`" :=
(`hforall (fun x1 ⇒ .. (hforall (fun xn ⇒ H)) ..)`)
(at level 39, `x1 binder, H at level 50, right associativity,`
format "`'['\forall' / ' x1 .. xn , / ' H]'`") : `hprop_scope`.

The remaining operators are defined in terms of the previous above, rather than directly as functions over heaps. Doing so reduces the amount of proofs, by allowing to better leverage the tactic `xsimpl`.

Definition `hpure (P:Prop) : hprop :=`
`\exists (p:P), \[].`

Definition `htop : hprop :=`
`\exists (H:hprop), H.`

Definition `hwand (H1 H2 : hprop) : hprop :=`
`\exists H0, H0 * hpure ((H1 * H0) ==> H2).`

Definition `qwand A (Q1 Q2:A→hprop) : hprop :=`
`\forall x, hwand (Q1 x) (Q2 x).`

Notation "`\[P]`" := (`hpure P`)
(at level 0, format "`\[P]`") : `hprop_scope`.

Notation "`\Top`" := (`htop`) : `hprop_scope`.

Notation "`Q *+ H`" := (`fun x ⇒ hstar (Q x) H`)
(at level 40) : `hprop_scope`.

Notation "`H1 \-* H2`" := (`hwand H1 H2`)
(at level 43, right associativity) : `hprop_scope`.

```
Notation "Q1 \-* Q2" := (qwand Q1 Q2)
  (at level 43) : hprop_scope.
```

35.4.3 Basic Properties of Separation Logic Operators

Tactic for Automation

We set up `auto` to process goals of the form $\mathsf{h1} = \mathsf{h2}$ by calling `fmap_eq`, which proves equality modulo associativity and commutativity.

```
Hint Extern 1 (_ = _ :> heap) ⇒ fmap_eq.
```

We also set up `auto` to process goals of the form `Fmap.disjoint h1 h2` by calling the tactic `fmap_disjoint`, which essentially normalizes all disjointness goals and hypotheses, split all conjunctions, and invokes proof search with a base of hints specified in `LibSepFmap.v`.

```
Import Fmap.DisjointHints.
```

```
Tactic Notation "fmap_disjoint_pre" :=
  subst; rew_disjoint; jauto_set.
```

```
Hint Extern 1 (Fmap.disjoint _ _) ⇒ fmap_disjoint_pre.
```

Properties of `himpl` and `qimpl`

```
Lemma himpl_refl : ∀ H,
  H ==> H.
```

```
Proof using. introv M. auto. Qed.
```

```
Hint Resolve himpl_refl.
```

```
Lemma himpl_trans : ∀ H2 H1 H3,
  (H1 ==> H2) →
  (H2 ==> H3) →
  (H1 ==> H3).
```

```
Proof using. introv M1 M2. unfolds× himpl. Qed.
```

```
Lemma himpl_trans_r : ∀ H2 H1 H3,
  H2 ==> H3 →
  H1 ==> H2 →
  H1 ==> H3.
```

```
Proof using. introv M1 M2. applys× himpl_trans M2 M1. Qed.
```

```
Lemma himpl_antisym : ∀ H1 H2,
  (H1 ==> H2) →
  (H2 ==> H1) →
  (H1 = H2).
```

```
Proof using. introv M1 M2. applys pred_ext_1. intros h. iff×. Qed.
```

```
Lemma hprop_op_comm : ∀ (op:hprop→hprop→hprop),
```

$(\forall H1 H2, op H1 H2 ==> op H2 H1) \rightarrow$
 $(\forall H1 H2, op H1 H2 = op H2 H1).$

Proof using. introv M. intros. applys himpl_antisym; applys M. Qed.

Lemma qimpl_refl : $\forall A (Q:A \rightarrow \text{hprop}),$
 $Q ==> Q.$

Proof using. intros. unfolds \times . Qed.

Hint Resolve qimpl_refl.

Properties of hempty

Lemma hempty_intro :

$\lambda [] \text{Fmap.empty}.$

Proof using. unfolds \times . Qed.

Lemma hempty_inv : $\forall h,$

$\lambda [] h \rightarrow$

$h = \text{Fmap.empty}.$

Proof using. auto. Qed.

Properties of hstar

Lemma hstar_intro : $\forall H1 H2 h1 h2,$

$H1 h1 \rightarrow$

$H2 h2 \rightarrow$

$\text{Fmap.disjoint } h1 h2 \rightarrow$

$(H1 \setminus * H2) (\text{Fmap.union } h1 h2).$

Proof using. intros. $\exists \neg h1 h2.$ Qed.

Lemma hstar_inv : $\forall H1 H2 h,$

$(H1 \setminus * H2) h \rightarrow$

$\exists h1 h2, H1 h1 \wedge H2 h2 \wedge \text{Fmap.disjoint } h1 h2 \wedge h = \text{Fmap.union } h1 h2.$

Proof using. introv M. applys M. Qed.

Lemma hstar_comm : $\forall H1 H2,$

$H1 \setminus * H2 = H2 \setminus * H1.$

Proof using.

applys hprop_op_comm. unfold hprop, hstar. intros H1 H2.

intros h (h1&h2&M1&M2&D&U). rewrite $\neg \text{Fmap.union_comm_of_disjoint}$ in U.

$\exists \times h2 h1.$

Qed.

Lemma hstar_assoc : $\forall H1 H2 H3,$

$(H1 \setminus * H2) \setminus * H3 = H1 \setminus * (H2 \setminus * H3).$

Proof using.

intros H1 H2 H3. applys himpl_antisym; intros h.

```

{ intros (h'&h3&(h1&h2&M3&M4&D'&U')&M2&D&U). subst h'.
  ∃ h1 (h2 \+ h3). splits¬. { applys× hstar_intro. } }
{ intros (h1&h'&M1&(h2&h3&M3&M4&D'&U')&D&U). subst h'.
  ∃ (h1 \+ h2) h3. splits¬. { applys× hstar_intro. } }

```

Qed.

Lemma hstar_hempty_l : $\forall H, H \setminus\ast \emptyset = H$.

Proof using.

```

intros. applys himpl_antisym; intros h.
{ intros (h1&h2&M1&M2&D&U). forwards E: hempty_inv M1. subst.
  rewrite¬ Fmap.union_empty_l. }
{ intros M. ∃ (Fmap.empty:heap) h. splits¬. { applys hempty_intro. } }

```

Qed.

Lemma hstar_hempty_r : $\forall H, H \setminus\ast \emptyset = H$.

Proof using.

applys neutral_r_of_comm_neutral_l. applys¬ hstar_comm. applys¬ hstar_hempty_l.

Qed.

Lemma hstar_hexists : $\forall A (J:A \rightarrow \text{hprop}) H, (\text{hexists } J) \setminus\ast H = \text{hexists } (\text{fun } x \Rightarrow (J x) \setminus\ast H)$.

Proof using.

```

intros. applys himpl_antisym; intros h.
{ intros (h1&h2&(x&M1)&M2&D&U). ∃¬ x h1 h2. }
{ intros (x&(h1&h2&M1&M2&D&U)). ∃ h1 h2. splits¬. { ∃¬ x. } }

```

Qed.

Lemma hstar_hforall : $\forall H A (J:A \rightarrow \text{hprop}), (\text{hforall } J) \setminus\ast H ==> \text{hforall } (J \setminus\ast H)$.

Proof using.

```

intros. intros h M. destruct M as (h1&h2&M1&M2&D&U). intros x. ∃¬ h1 h2.
Qed.

```

Lemma himpl_frame_l : $\forall H2 H1 H1', H1 ==> H1' \rightarrow (H1 \setminus\ast H2) ==> (H1' \setminus\ast H2)$.

Proof using. introv W (h1&h2&?). ∃× h1 h2. Qed.

Lemma himpl_frame_r : $\forall H1 H2 H2', H2 ==> H2' \rightarrow (H1 \setminus\ast H2) ==> (H1 \setminus\ast H2')$.

Proof using.

introv M. do 2 rewrite (@hstar_comm H1). applys¬ himpl_frame_l.

Qed.

Lemma himpl_frame_lr : $\forall H1 H1' H2 H2', H1 ==> H1' \rightarrow (H1 \setminus\ast H2) ==> (H1' \setminus\ast H2)$.

$H1 \implies H1' \rightarrow$
 $H2 \implies H2' \rightarrow$
 $(H1 \star H2) \implies (H1' \star H2').$

Proof using.

introv $M1 M2$. applys himpl_trans. applys¬ himpl_frame_l $M1$. applys¬ himpl_frame_r.

Qed.

Lemma himpl_hstar_trans_l : $\forall H1 H2 H3 H4$,

$H1 \implies H2 \rightarrow$
 $H2 \star H3 \implies H4 \rightarrow$
 $H1 \star H3 \implies H4.$

Proof using.

introv $M1 M2$. applys himpl_trans $M2$. applys himpl_frame_l $M1$.

Qed.

Lemma himpl_hstar_trans_r : $\forall H1 H2 H3 H4$,

$H1 \implies H2 \rightarrow$
 $H3 \star H2 \implies H4 \rightarrow$
 $H3 \star H1 \implies H4.$

Proof using.

introv $M1 M2$. applys himpl_trans $M2$. applys himpl_frame_r $M1$.

Qed.

Properties of hpure

Lemma hpure_intro : $\forall P$,
 $P \rightarrow$

$\set{P} \text{Fmap.empty}.$

Proof using. introv M . $\exists M$. unfolds \times . Qed.

Lemma hpure_inv : $\forall P h$,

$\set{P} h \rightarrow$
 $P \wedge h = \text{Fmap.empty}.$

Proof using. introv $(p \& M)$. split¬. Qed.

Lemma hstar_hpure_l : $\forall P H h$,

$(\set{P} \star H) h = (P \wedge H) h.$

Proof using.

```

intros. apply prop_ext. unfold hpure.
rewrite hstar_hexists. rewrite $\times$  hstar_hempty_l.
iff  $(p \& M) (p \& M)$ . { split¬. } {  $\exists p$ . }

```

Qed.

Lemma hstar_hpure_r : $\forall P H h$,
 $(H \star \set{P}) h = (H h \wedge P).$

Proof using.

```

intros. rewrite hstar_comm. rewrite hstar_hpure_l. apply× prop_ext.
Qed.
```

```

Lemma himpl_hstar_hpure_r : ∀ P H H',
  P →
  (H ==> H') →
  H ==> (\\[P] \* H').
```

Proof using. introv HP W. intros h K. rewrite× hstar_hpure_l. Qed.

```

Lemma hpure_inv_hempty : ∀ P h,
  \\[P] h →
  P ∧ \\[] h.
```

Proof using.

```
introv M. rewrite ← hstar_hpure_l. rewrite¬ hstar_hempty_r.
```

Qed.

```

Lemma hpure_intro_hempty : ∀ P h,
  \\[] h →
  P →
  \\[P] h.
```

Proof using.

```
introv M N. rewrite ← (hstar_hempty_l \\[P]). rewrite¬ hstar_hpure_r.
```

Qed.

```

Lemma himpl_hempty_hpure : ∀ P,
  P →
  \\[] ==> \\[P].
```

Proof using. introv HP. intros h Hh. applys× hpure_intro_hempty. Qed.

```

Lemma himpl_hstar_hpure_l : ∀ P H H',
  (P → H ==> H') →
  (\\[P] \* H) ==> H'.
```

Proof using.

```
introv W Hh. rewrite hstar_hpure_l in Hh. applys× W.
```

Qed.

```

Lemma hempty_eq_hpure_true :
  \\[] = \\[True].
```

Proof using.

```
applys himpl_antisym; intros h M.
{ applys× hpure_intro_hempty. }
{ forwards*: hpure_inv_hempty M. }
```

Qed.

```

Lemma hfalse_hstar_any : ∀ H,
  \\[False] \* H = \\[False].
```

Proof using.

```
intros. applys himpl_antisym; intros h; rewrite hstar_hpure_l; intros M.
```

$\{ \text{false} \times . \} \{ \text{lets: hpure_inv_hempty } M. \text{false} \times . \}$
 Qed.

Properties of hexists

Lemma hexists_intro : $\forall A (x:A) (J:A \rightarrow \text{hprop}) h,$
 $J x h \rightarrow$
 $(\text{hexists } J) h.$

Proof using. intros. $\exists \neg x.$ Qed.

Lemma hexists_inv : $\forall A (J:A \rightarrow \text{hprop}) h,$
 $(\text{hexists } J) h \rightarrow$
 $\exists x, J x h.$

Proof using. intros. destruct H as [x H]. $\exists \neg x.$ Qed.

Lemma himpl_hexists_l : $\forall A H (J:A \rightarrow \text{hprop}),$
 $(\forall x, J x ==> H) \rightarrow$
 $(\text{hexists } J) ==> H.$

Proof using. introv W. intros h (x&Hh). applys \times W. Qed.

Lemma himpl_hexists_r : $\forall A (x:A) H J,$
 $(H ==> J x) \rightarrow$
 $H ==> (\text{hexists } J).$

Proof using. introv W. intros h. $\exists x.$ apply \neg W. Qed.

Lemma himpl_hexists : $\forall A (J1 J2:A \rightarrow \text{hprop}),$
 $J1 ==> J2 \rightarrow$
 $\text{hexists } J1 ==> \text{hexists } J2.$

Proof using.

introv W. applys himpl_hexists_l. intros x. applys himpl_hexists_r W.
Qed.

Properties of hforall

Lemma hforall_intro : $\forall A (J:A \rightarrow \text{hprop}) h,$
 $(\forall x, J x h) \rightarrow$
 $(\text{hforall } J) h.$

Proof using. introv M. applys \times M. Qed.

Lemma hforall_inv : $\forall A (J:A \rightarrow \text{hprop}) h,$
 $(\text{hforall } J) h \rightarrow$
 $\forall x, J x h.$

Proof using. introv M. applys \times M. Qed.

Lemma himpl_hforall_r : $\forall A (J:A \rightarrow \text{hprop}) H,$
 $(\forall x, H ==> J x) \rightarrow$
 $H ==> (\text{hforall } J).$

Proof using. *intros* M . *intros* h Hh x . *apply* \neg M . Qed.

Lemma $\text{himpl_hforall_l} : \forall A x (J:A \rightarrow \text{hprop}) H,$

$$(J x ==> H) \rightarrow$$

$$(\text{hforall } J) ==> H.$$

Proof using. *intros* M . *intros* h Hh . *apply* \neg M . Qed.

Lemma $\text{hforall_specialize} : \forall A (x:A) (J:A \rightarrow \text{hprop}),$

$$(\text{hforall } J) ==> (J x).$$

Proof using. *intros*. *applys* \times himpl_hforall_l x . Qed.

Lemma $\text{himpl_hforall} : \forall A (J1 J2:A \rightarrow \text{hprop}),$

$$J1 ==> J2 \rightarrow$$

$$\text{hforall } J1 ==> \text{hforall } J2.$$

Proof using.

intros W . *applys* himpl_hforall_r . *intros* x . *applys* himpl_hforall_l W .

Qed.

Properties of **hwand**

Lemma $\text{hwand_equiv} : \forall H0 H1 H2,$

$$(H0 ==> H1 \dashv\ast H2) \leftrightarrow (H1 \dashv\ast H0 ==> H2).$$

Proof using.

unfold hwand. iff M.

{ *rewrite hstar_comm. applys himpl_hstar_trans_l (rm M).*

rewrite hstar_heexists. applys himpl_heexists_l. intros H.

rewrite (hstar_comm H). rewrite hstar_assoc.

rewrite (hstar_comm H H1). applys \neg himpl_hstar_hpure_l. }

{ *applys himpl_heexists_r H0.*

rewrite \leftarrow (hstar_hempty_r H0) at 1.

applys himpl_frame_r. applys himpl_hempty_hpure M. }

Qed.

Lemma $\text{himpl_hwand_r} : \forall H1 H2 H3,$

$$H2 \dashv\ast H1 ==> H3 \rightarrow$$

$$H1 ==> (H2 \dashv\ast H3).$$

Proof using. *intros* M . *rewrite* \neg hwand_equiv . Qed.

Lemma $\text{himpl_hwand_r_inv} : \forall H1 H2 H3,$

$$H1 ==> (H2 \dashv\ast H3) \rightarrow$$

$$H2 \dashv\ast H1 ==> H3.$$

Proof using. *intros* M . *rewrite* \neg \leftarrow hwand_equiv . Qed.

Lemma $\text{hwand_cancel} : \forall H1 H2,$

$$H1 \dashv\ast (H1 \dashv\ast H2) ==> H2.$$

Proof using. *intros*. *applys* himpl_hwand_r_inv . *applys* himpl_refl . Qed.

Arguments hwand_cancel : clear implicits.

```

Lemma himpl_hempty_hwand_same : ∀ H,
  \[] ==> (H \-* H).
Proof using. intros. apply himpl_hwand_r. rewrite¬ hstar_hempty_r. Qed.

Lemma hwand_hempty_l : ∀ H,
  (\[] \-* H) = H.
Proof using.
  intros. applys himpl_antisym.
  { rewrite ← hstar_hempty_l at 1. applys hwand_cancel. }
  { rewrite hwand_equiv. rewrite¬ hstar_hempty_l. }
Qed.

Lemma hwand_hpure_l : ∀ P H,
  P →
  (\[P] \-* H) = H.
Proof using.
  introv HP. applys himpl_antisym.
  { lets K: hwand_cancel \[P] H. applys himpl_trans K.
    applys× himpl_hstar_hpure_r. }
  { rewrite hwand_equiv. applys× himpl_hstar_hpure_l. }
Qed.

Lemma hwand_curry : ∀ H1 H2 H3,
  (H1 \* H2) \-* H3 ==> H1 \-* (H2 \-* H3).
Proof using.
  intros. apply himpl_hwand_r. apply himpl_hwand_r.
  rewrite ← hstar_assoc. rewrite (hstar_comm H1 H2).
  applys hwand_cancel.
Qed.

Lemma hwand_uncurry : ∀ H1 H2 H3,
  H1 \-* (H2 \-* H3) ==> (H1 \* H2) \-* H3.
Proof using.
  intros. rewrite hwand_equiv. rewrite (hstar_comm H1 H2).
  rewrite hstar_assoc. applys himpl_hstar_trans_r.
  { applys hwand_cancel. } { applys hwand_cancel. }
Qed.

Lemma hwand_curry_eq : ∀ H1 H2 H3,
  (H1 \* H2) \-* H3 = H1 \-* (H2 \-* H3).
Proof using.
  intros. applys himpl_antisym.
  { applys hwand_curry. }
  { applys hwand_uncurry. }
Qed.

Lemma hwand_inv : ∀ h1 h2 H1 H2,

```

```
(H1 \-* H2) h2 →
H1 h1 →
Fmap.disjoint h1 h2 →
H2 (h1 \u h2).
```

Proof using.

```
intros M2 M1 D. unfolds hwand. lets (H0&M3): hexists_inv M2.
lets (h0&h3&P1&P3&D'&U): hstar_inv M3. lets (P4&E3): hpure_inv P3.
subst h2 h3. rewrite union_empty_r in *. applys P4. applys× hstar_intro.
```

Qed.

Properties of qwand

```
Lemma qwand_equiv : ∀ H A (Q1 Q2:A→hprop),
H ==> (Q1 \-* Q2) ↔ (Q1 \*+ H) ==> Q2.
```

Proof using.

```
unfold qwand. iff M.
{ intros x. rewrite hstar_comm. applys himpl_hstar_trans_l (rm M).
  applys himpl_trans. applys hstar_hforall. simpl.
  applys himpl_hforall_l x. rewrite hstar_comm. applys hwand_cancel. }
{ applys himpl_hforall_r. intros x. rewrite× hwand_equiv. }
```

Qed.

```
Lemma qwand_cancel : ∀ A (Q1 Q2:A→hprop),
Q1 \*+ (Q1 \-* Q2) ==> Q2.
```

Proof using. intros. rewrite ← qwand_equiv. applys qimpl_refl. Qed.

```
Lemma himpl_qwand_r : ∀ A (Q1 Q2:A→hprop) H,
Q1 \*+ H ==> Q2 →
H ==> (Q1 \-* Q2).
```

Proof using. introv M. rewrite¬ qwand_equiv. Qed.

Arguments himpl_qwand_r [A].

```
Lemma qwand_specialize : ∀ A (x:A) (Q1 Q2:A→hprop),
(Q1 \-* Q2) ==> (Q1 x \-* Q2 x).
```

Proof using. intros. applys× himpl_hforall_l x. Qed.

Arguments qwand_specialize [A].

Properties of htop

```
Lemma htop_intro : ∀ h,
\Top h.
```

Proof using. intros. ∃¬ (=h). Qed.

```
Lemma himpl_htop_r : ∀ H,
H ==> \Top.
```

Proof using. intros. intros h Hh . *applys* \times htop_intro. Qed.

Lemma htop_eq :

$$\text{\textbackslash Top} = (\exists H, H).$$

Proof using. auto. Qed.

Lemma hstar_htop_htop :

$$\text{\textbackslash Top} \text{\textbackslash *} \text{\textbackslash Top} = \text{\textbackslash Top}.$$

Proof using.

$$\text{applys himpl_antisym}.$$

$$\{ \text{applys himpl_htop_r. } \}$$

$$\{ \text{rewrite } \leftarrow \text{hstar_hempty_r at 1. applys himpl_frame_r. applys himpl_htop_r. } \}$$

Qed.

Properties of hsingle

Lemma hsingle_intro : $\forall p v,$

$$(p \sim\sim v) (\text{Fmap.single } p v).$$

Proof using. intros. hnfs \times . Qed.

Lemma hsingle_inv: $\forall p v h,$

$$(p \sim\sim v) h \rightarrow$$

$$h = \text{Fmap.single } p v.$$

Proof using. auto. Qed.

Lemma hstar_hsingle_same_loc : $\forall p w1 w2,$

$$(p \sim\sim w1) \text{\textbackslash *} (p \sim\sim w2) ==> \text{\textbackslash [False]}.$$

Proof using.

intros. unfold hsingle. intros h ($h1 \& h2 \& E1 \& E2 \& D \& E$). false.

subst. *applys* \times Fmap.disjoint_single_single_same_inv.

Qed.

Arguments hstar_hsingle_same_loc : clear implicits.

Definition and Properties of haffine and hgc

Definition haffine ($H:\text{hprop}$) :=

True.

Lemma haffine_hany : $\forall (H:\text{hprop}),$

haffine H .

Proof using. unfold haffine. auto. Qed.

Lemma haffine_hempty :

haffine $\text{\textbackslash []}.$

Proof using. *applys* haffine_hany. Qed.

Definition hgc :=

htop.

Notation "\GC" := (hgc) : hprop_scope.

Lemma haffine_hgc :

 haffine \GC.

Proof using. applys haffine_hany. Qed.

Lemma himpl_hgc_r : $\forall H,$

 haffine $H \rightarrow$

$H ==> \GC.$

Proof using. introv M. applys himpl_htop_r. Qed.

Lemma hstar_hgc_hgc :

 \GC * \GC = \GC.

Proof using. applys hstar_htop_htop. Qed.

Functor Instantiation to Obtain *xsiml*

End SEPSIMPLARGS.

We are now ready to instantiate the functor that defines *xsiml*. Demos of *xsiml* are presented in chapter *Himpl.v*.

Module Export HS := LIBSEPSIMPL.XSIMPLSETUP(SEPSIMPLARGS).

Export SepSimplArgs.

From now on, all operators have opaque definitions.

Global Opaque hempty hpure hstar hsingle hexists hforall
hwand qwand htop hgc haffine.

At this point, the tactic *xsiml* is defined. There remains to customize the tactic so that it recognizes the predicate $p \sim> v$ in a special way when performing simplifications.

The tactic *xsiml_hook_hsingl* $p v$ operates as part of *xsiml*. The specification that follows makes sense only in the context of the presentation of the invariants of *xsiml* described in *LibSepSimpl.v*. This tactic is invoked on goals of the form $\text{Xsiml}(Hla, Hlw, Hlt) HR$, where Hla is of the form $H1 * .. * Hn * []$. The tactic *xsiml_hook_hsingl* $p v$ searches among the Hi for a heap predicate of the form $p \sim> v'$. If it finds one, it moves this Hi to the front, just before $H1$. Then, it cancels it out with the $p \sim> v$ that occurs in HR . Otherwise, the tactic fails.

```
Ltac xsiml_hook_hsingl p :=
  xsiml_pick_st ltac:(fun H' =>
    match H' with (hsingle p ?v') =>
      constr:(true) end);
  apply xsiml_lr_cancel_eq;
  [ xsiml_lr_cancel_eq_repr_post tt | ].
```

The tactic `xsimpl_hook` handles cancellation of heap predicates of the form $p \sim\!> v$. It forces their cancellation against heap predicates of the form $p \sim\!> w$, by asserting the equality $v = w$. Note: this tactic is later refined to also handle records.

```
Ltac xsimpl_hook H ::=  
  match H with  
  | hsingle ?p ?v => xsimpl_hook_hsingle p  
  end.
```

One last hack is to improve the `math` tactic so that it is able to handle the `val_int` coercion in goals and hypotheses of the form `val_int ?n = val_int ?m`, and so that it is able to process the well-founded relations `downto` and `upto` for induction on integers.

```
Ltac math_0 ::=  
  unfolds downto, upto;  
  repeat match goal with  
  | ⊢ val_int _ = val_int _ => fequal  
  | H: val_int _ = val_int _ ⊢ _ => inverts H  
  end.
```

35.4.4 Properties of `haffine`

In this file, we set up an affine logic. The lemma `haffine_any` asserts that `haffine H` holds for any `H`. Nevertheless, let us state corollaries of `haffine_any` for specific heap predicates, to illustrate how the `xaffine` tactic works in chapter `Affine`.

`Lemma haffine_hempty :`

`haffine [].`

`Proof using. intros. applys haffine_hany. Qed.`

`Lemma haffine_hpure : ∀ P,`

`haffine [P].`

`Proof using. intros. applys haffine_hany. Qed.`

`Lemma haffine_hstar : ∀ H1 H2,`

`haffine H1 →`

`haffine H2 →`

`haffine (H1 ∗ H2).`

`Proof using. intros. applys haffine_hany. Qed.`

`Lemma haffine_hexists : ∀ A (J:A→hprop),`

`(∀ x, haffine (J x)) →`

`haffine (∃ x, (J x)).`

`Proof using. intros. applys haffine_hany. Qed.`

`Lemma haffine_hforall : ∀ A ‘{Inhab A} (J:A→hprop),`

`(∀ x, haffine (J x)) →`

`haffine (∀ x, (J x)).`

```
Proof using. intros. applys haffine_hany. Qed.
```

```
Lemma haffine_hstar_hpure :  $\forall (P:\text{Prop}) H,$   
   $(P \rightarrow \text{haffine } H) \rightarrow$   
   $\text{haffine } (\lambda [P] \ \backslash * \ H).$ 
```

```
Proof using. intros. applys haffine_hany. Qed.
```

```
Lemma haffine_hgc :
```

```
   $\text{haffine } \backslash \text{GC}.$ 
```

```
Proof using. intros. applys haffine_hany. Qed.
```

Using these lemmas, we are able to configure the *xaffine* tactic.

```
Ltac xaffine_core tt ::=  
  repeat match goal with  $\vdash \text{haffine } ?H \Rightarrow$   
    match  $H$  with  
    | ( $\text{hempty}$ )  $\Rightarrow$  apply haffine_hempty  
    | ( $\text{hpure } _$ )  $\Rightarrow$  apply haffine_hpure  
    | ( $\text{hstar } _ \ _$ )  $\Rightarrow$  apply haffine_hstar  
    | ( $\text{hexists } _$ )  $\Rightarrow$  apply haffine_hexists  
    | ( $\text{hforall } _$ )  $\Rightarrow$  apply haffine_hforall  
    | ( $\text{hgc}$ )  $\Rightarrow$  apply haffine_hgc  
    |  $_ \Rightarrow$  eauto with haffine  
    end  
  end.
```

35.5 Reasoning Rules

Implicit Types $P : \text{Prop}.$

Implicit Types $H : \text{hprop}.$

Implicit Types $Q : \text{val} \rightarrow \text{hprop}.$

35.5.1 Evaluation Rules for Primitives in Separation Style

These lemmas reformulated the big-step evaluation rule in a Separation-Logic friendly presentation, that is, by using disjoint unions as much as possible.

```
Lemma eval_ref_sep :  $\forall s1 \ s2 \ v \ p,$   
   $s2 = \text{Fmap.single } p \ v \rightarrow$   
   $\text{Fmap.disjoint } s2 \ s1 \rightarrow$   
   $\text{eval } s1 \ (\text{val\_ref } v) \ (\text{Fmap.union } s2 \ s1) \ (\text{val\_loc } p).$ 
```

Proof using.

```
  introv  $\rightarrow D.$  forwards  $Dv : \text{Fmap.indom\_single } p \ v.$   
  rewrite  $\leftarrow \text{Fmap.update\_eq\_union\_single}.$  applys  $\neg \text{eval\_ref}.$   
  { intros N. applys  $\neg \text{Fmap.disjoint\_inv\_not\_indom\_both } D \ N.$  }
```

Qed.

Lemma eval_get_sep : $\forall s s2 p v,$
 $s = \text{Fmap.union}(\text{Fmap.single } p v) s2 \rightarrow$
 $\text{eval } s (\text{val_get } (\text{val_loc } p)) s v.$

Proof using.

intros \rightarrow . *forwards* $Dv : \text{Fmap.indom_single } p v.$
applys eval_get.
 $\{ \text{applys} \neg \text{Fmap.indom_union_l.} \}$
 $\{ \text{rewrite} \neg \text{Fmap.read_union_l. rewrite} \neg \text{Fmap.read_single.} \}$

Qed.

Lemma eval_set_sep : $\forall s1 s2 h2 p w v,$
 $s1 = \text{Fmap.union}(\text{Fmap.single } p w) h2 \rightarrow$
 $s2 = \text{Fmap.union}(\text{Fmap.single } p v) h2 \rightarrow$
 $\text{Fmap.disjoint}(\text{Fmap.single } p w) h2 \rightarrow$
 $\text{eval } s1 (\text{val_set } (\text{val_loc } p) v) s2 \text{ val_unit.}$

Proof using.

intros $\rightarrow \rightarrow D.$ *forwards* $Dv : \text{Fmap.indom_single } p w.$
applys_eq eval_set.
 $\{ \text{rewrite} \neg \text{Fmap.update_union_l. fequals.}$
 $\quad \text{rewrite} \neg \text{Fmap.update_single.} \}$
 $\{ \text{applys} \neg \text{Fmap.indom_union_l.} \}$

Qed.

Lemma eval_free_sep : $\forall s1 s2 v p,$
 $s1 = \text{Fmap.union}(\text{Fmap.single } p v) s2 \rightarrow$
 $\text{Fmap.disjoint}(\text{Fmap.single } p v) s2 \rightarrow$
 $\text{eval } s1 (\text{val_free } p) s2 \text{ val_unit.}$

Proof using.

intros $\rightarrow D.$ *forwards* $Dv : \text{Fmap.indom_single } p v.$
applys_eq eval_free.
 $\{ \text{rewrite} \neg \text{Fmap.remove_union_single_l.}$
 $\quad \text{intros } Dl. \text{ applys Fmap.disjoint_inv_not_indom_both } D Dl.$
 $\quad \text{applys Fmap.indom_single.} \}$
 $\{ \text{applys} \neg \text{Fmap.indom_union_l.} \}$

Qed.

35.5.2 Hoare Reasoning Rules

35.6 Definition of total correctness Hoare Triples.

Definition hoare ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) :=
 $\forall h, H h \rightarrow \exists h' v, \text{eval } h t h' v \wedge Q v h'.$

Structural rules for hoare triples.

Lemma `hoare_conseq` : $\forall t H' Q' H Q,$
 $\text{hoare } t H' Q' \rightarrow$
 $H ==> H' \rightarrow$
 $Q' ===> Q \rightarrow$
 $\text{hoare } t H Q.$

Proof using.

introv M MH MQ HF. forwards (h'&v&R&K): M h. { applys × MH. }
 $\exists h' v. \text{splits} \neg. \{ \text{applys} \times MQ. \}$

Qed.

Lemma `hoare_hexists` : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\forall x, \text{hoare } t (J x) Q) \rightarrow$
 $\text{hoare } t (\text{hexists } J) Q.$

Proof using. *introv M. intros h (x&Hh). applys M Hh.* Qed.

Lemma `hoare_hpure` : $\forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{hoare } t H Q) \rightarrow$
 $\text{hoare } t (\setminus [P] \setminus * H) Q.$

Proof using.

introv M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).
 $\text{lets } E: \text{hempty_inv } HP. \text{subst. rewrite Fmap.union_empty_l. applys} \neg M.$

Qed.

Other structural rules, not required for setting up wpgen.

Lemma `hoare_hforall` : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\exists x, \text{hoare } t (J x) Q) \rightarrow$
 $\text{hoare } t (\text{hforall } J) Q.$

Proof using.

introv (x&M) Hh. applys × hoare_conseq (hforall J) Q M.
 $\text{applys} \times \text{himpl_hforall_l}.$

Qed.

Lemma `hoare_hwand_hpure_l` : $\forall t (P:\text{Prop}) H Q,$
 $P \rightarrow$
 $\text{hoare } t H Q \rightarrow$
 $\text{hoare } t (\setminus [P] \setminus * H) Q.$

Proof using. *introv HP M. rewrite × hwand_hpure_l.* Qed.

Reasoning rules for hoare triples. These rules follow directly from the big-step evaluation rules.

Lemma `hoare_eval_like` : $\forall t1 t2 H Q,$
 $\text{eval_like } t1 t2 \rightarrow$
 $\text{hoare } t1 H Q \rightarrow$
 $\text{hoare } t2 H Q.$

Proof using.

introv E M1 K0. forwards ($s' \& v \& R1 \& K1$): M1 K0.

$\exists s' v. \text{split. } \{ \text{applys } E R1. \} \{ \text{applys } K1. \}$

Qed.

Lemma hoare_val : $\forall v H Q,$

$H ==> Q v \rightarrow$

hoare (trm_val v) H Q.

Proof using.

introv M. intros h K. $\exists h v. \text{splits.}$

$\{ \text{applys eval_val.} \}$

$\{ \text{applys } M. \}$

Qed.

Lemma hoare_fun : $\forall x t1 H Q,$

$H ==> Q (\text{val_fun } x t1) \rightarrow$

hoare (trm_fun x t1) H Q.

Proof using.

introv M. intros h K. $\exists h _. \text{splits.}$

$\{ \text{applys eval_fun.} \}$

$\{ \text{applys } M. \}$

Qed.

Lemma hoare_fix : $\forall f x t1 H Q,$

$H ==> Q (\text{val_fix } f x t1) \rightarrow$

hoare (trm_fix f x t1) H Q.

Proof using.

introv M. intros h K. $\exists h _. \text{splits.}$

$\{ \text{applys eval_fix.} \}$

$\{ \text{applys } M. \}$

Qed.

Lemma hoare_app_fun : $\forall v1 v2 x t1 H Q,$

$v1 = \text{val_fun } x t1 \rightarrow$

hoare (subst x v2 t1) H Q \rightarrow

hoare (trm_app v1 v2) H Q.

Proof using.

introv E M. intros s K0. forwards ($s' \& v \& R1 \& K1$): (rm M) K0.

$\exists s' v. \text{splits. } \{ \text{applys eval_app_fun } E R1. \} \{ \text{applys } K1. \}$

Qed.

Lemma hoare_app_fix : $\forall v1 v2 f x t1 H Q,$

$v1 = \text{val_fix } f x t1 \rightarrow$

hoare (subst x v2 (subst f v1 t1)) H Q \rightarrow

hoare (trm_app v1 v2) H Q.

Proof using.

*introv E M. intros s K0. forwards (s'&v&R1&K1): (rm M) K0.
 $\exists s' v.$ splits. { applys eval_app_fix E R1. } { applys K1. }*

Qed.

Lemma hoare_seq : $\forall t1 t2 H Q H1,$
*hoare t1 H (fun r \Rightarrow H1) \rightarrow
 hoare t2 H1 Q \rightarrow
 hoare (trm_seq t1 t2) H Q.*

Proof using.

*introv M1 M2 Hh.
 forwards \times (h1'&v1&R1&K1): (rm M1).
 forwards \times (h2'&v2&R2&K2): (rm M2).
 $\exists h2' v2.$ splits \neg . { applys \neg eval_seq R1 R2. }*

Qed.

Lemma hoare_let : $\forall x t1 t2 H Q Q1,$
*hoare t1 H Q1 \rightarrow
 $(\forall v1,$ hoare (subst x v1 t2) (Q1 v1) Q \rightarrow
 hoare (trm_let x t1 t2) H Q.*

Proof using.

*introv M1 M2 Hh.
 forwards \times (h1'&v1&R1&K1): (rm M1).
 forwards \times (h2'&v2&R2&K2): (rm M2).
 $\exists h2' v2.$ splits \neg . { applys \neg eval_let R2. }*

Qed.

Lemma hoare_if : $\forall (b:\text{bool}) t1 t2 H Q,$
*hoare (if b then t1 else t2) H Q \rightarrow
 hoare (trm_if b t1 t2) H Q.*

Proof using.

*introv M1. intros h Hh. forwards \times (h1'&v1&R1&K1): (rm M1).
 $\exists h1' v1.$ splits \neg . { applys \times eval_if. }*

Qed.

Operations on the state.

Lemma hoare_ref : $\forall H v,$
hoare (val_ref v)
 H
(fun r \Rightarrow ($\exists p,$ $\exists [r = \text{val_loc } p] \ \forall p \sim\sim> v$) $\forall H$).

Proof using.

*intros. intros s1 K0.
 forwards \neg (p&D&N): (Fmap.single_fresh 0%nat s1 v).
 $\exists (\text{Fmap.union} (\text{Fmap.single } p \ v) \ s1) (\text{val_loc } p).$ split.
 { applys \neg eval_ref_sep D. }
 { applys \neg hstar_intro.*

$\{ \exists p. \text{rewrite} \sqsubset \text{hstar_hpure_l}. \text{split} \sqsubset. \{ \text{applys} \sqsubset \text{hsingle_intro}. \} \} \}$
 Qed.

Lemma hoare_get : $\forall H v p,$
 hoare (val_get p)
 $((p \rightsquigarrow v) \ast H)$
 $(\text{fun } r \Rightarrow \lambda[r = v] \ast (p \rightsquigarrow v) \ast H).$

Proof using.

intros. intros s K0. $\exists s v. \text{split}.$
 $\{ \text{destruct } K0 \text{ as } (s1 \& s2 \& P1 \& P2 \& D \& U).$
 $\quad \text{lets } E1: \text{hsingle_inv } P1. \text{subst } s1. \text{applys eval_get_sep } U. \}$
 $\{ \text{rewrite} \sqsubset \text{hstar_hpure_l}. \}$

Qed.

Lemma hoare_set : $\forall H w p v,$
 hoare (val_set (val_loc p) v)
 $((p \rightsquigarrow w) \ast H)$
 $(\text{fun } r \Rightarrow \lambda[r = \text{val_unit}] \ast (p \rightsquigarrow v) \ast H).$

Proof using.

intros. intros s1 K0.
 $\text{destruct } K0 \text{ as } (h1 \& h2 \& P1 \& P2 \& D \& U).$
 $\text{lets } E1: \text{hsingle_inv } P1.$
 $\exists (\text{Fmap.union } (\text{Fmap.single } p v) h2) \text{ val_unit. split}.$
 $\{ \text{subst } h1. \text{applys eval_set_sep } U D. \text{ auto}. \}$
 $\{ \text{rewrite hstar_hpure_l. split} \sqsubset.$
 $\quad \{ \text{applys} \sqsubset \text{hstar_intro}.$
 $\quad \{ \text{applys} \sqsubset \text{hsingle_intro}. \}$
 $\quad \{ \text{subst } h1. \text{applys Fmap.disjoint_single_set } D. \} \} \}$

Qed.

Lemma hoare_free : $\forall H p v,$
 hoare (val_free (val_loc p))
 $((p \rightsquigarrow v) \ast H)$
 $(\text{fun } r \Rightarrow \lambda[r = \text{val_unit}] \ast H).$

Proof using.

intros. intros s1 K0.
 $\text{destruct } K0 \text{ as } (h1 \& h2 \& P1 \& P2 \& D \& U).$
 $\text{lets } E1: \text{hsingle_inv } P1.$
 $\exists h2 \text{ val_unit. split}.$
 $\{ \text{subst } h1. \text{applys eval_free_sep } U D. \}$
 $\{ \text{rewrite hstar_hpure_l. split} \sqsubset. \}$

Qed.

Other operations.

Lemma hoare_unop : $\forall v H op v1,$

```

evalunop op v1 v →
hoare (op v1)
H
(fun r ⇒ \[r = v] \* H).

```

Proof using.

```

intros R. intros h Hh. ∃ h v. splits.
{ applys eval_unop. }
{ rewrite hstar_hpure_l. }

```

Qed.

Lemma hoare_binop : ∀ v H op v1 v2,
evalbinop op v1 v2 v →
hoare (op v1 v2)
H
(fun r ⇒ \[r = v] * H).

Proof using.

```

intros R. intros h Hh. ∃ h v. splits.
{ applys eval_binop. }
{ rewrite hstar_hpure_l. }

```

Qed.

Bonus: example of proofs for a specific primitive operation.

Lemma hoare_add : ∀ H n1 n2,
hoare (val_add n1 n2)
H
(fun r ⇒ \[r = val_int (n1 + n2)] * H).

Proof.

```

dup.
{ intros. applys hoare_binop. applys evalbinop_add. }
{ intros. intros s K0. ∃ s (val_int (n1 + n2)). split.
  { applys eval_binop. applys evalbinop_add. }
  { rewrite hstar_hpure_l. } }

```

Qed.

35.6.1 Definition of Separation Logic Triples.

A Separation Logic triple $t \; H \; Q$ may be defined either in terms of Hoare triples, or in terms of wp , as $H ==> \text{wp} \; t \; Q$. We prefer the former route, which we find more elementary.

Definition triple ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : Prop :=
 $\forall (H':\text{hprop}), \text{hoare } t (H \; \text{*\;} H') (Q \; \text{*\;} H')$.

We introduce a handy notation for postconditions of functions that return a pointer:
 $\text{funloc } p \Rightarrow H$ is short for $\text{fun } (r:\text{val}) \Rightarrow \exists (p:\text{loc}), \exists [r = \text{val_loc } p] \; \text{*\;} H$.

Notation "'funloc' p '=>' H" :=

```
(fun r => \exists p, \[r = val_loc p] \* H)
(at level 200, p ident, format "'funloc' p '=>' H") : hprop_scope.
```

35.6.2 Structural Rules

Consequence and frame rule.

```
Lemma triple_conseq : \forall t H' Q' H Q,
  triple t H' Q' \rightarrow
  H ==> H' \rightarrow
  Q' ===> Q \rightarrow
  triple t H Q.
```

Proof using.

```
intros M MH MQ. intros HF. applys hoare_conseq M.
{ xchanges MH. }
{ intros x. xchanges (MQ x). }
```

Qed.

```
Lemma triple_frame : \forall t H Q H',
  triple t H Q \rightarrow
  triple t (H \* H') (Q \**+ H').
```

Proof using.

```
intros M. intros HF. applys hoare_conseq (M (HF \* H')); xsimpl.
```

Qed.

Details for the proof of the frame rule.

```
Lemma triple_frame' : \forall t H Q H',
  triple t H Q \rightarrow
  triple t (H \* H') (Q \**+ H').
```

Proof using.

```
intros M. unfold triple in *. rename H' into H1. intros H2.
applys hoare_conseq. applys M (H1 \* H2).
{ rewrite hstar_assoc. auto. }
{ intros v. rewrite hstar_assoc. auto. }
```

Qed.

Extraction rules.

```
Lemma triple_hexists : \forall t (A:Type) (J:A \rightarrow hprop) Q,
  (\forall x, triple t (J x) Q) \rightarrow
  triple t (hexists J) Q.
```

Proof using.

```
intros M. intros HF. rewrite hstar_hexists.
applys hoare_hexists. intros. applys \times M.
```

Qed.

```
Lemma triple_hpure : \forall t (P:Prop) H Q,
```

$(P \rightarrow \text{triple } t H Q) \rightarrow$
 $\text{triple } t (\setminus [P] \setminus * H) Q.$

Proof using.

*introv M. intros HF. rewrite hstar_assoc.
applys hoare_hpure. intros. applys× M.*

Qed.

Lemma triple_hforall : $\forall A (x:A) t (J:A \rightarrow \text{hprop}) Q,$
 $\text{triple } t (J x) Q \rightarrow$
 $\text{triple } t (\text{hforall } J) Q.$

Proof using.

introv M. applys× triple_conseq M. applys hforall_specialize.

Qed.

Lemma triple_hwand_hpure_l : $\forall t (P:\text{Prop}) H Q,$
 $P \rightarrow$
 $\text{triple } t H Q \rightarrow$
 $\text{triple } t (\setminus [P] \setminus -* H) Q.$

Proof using.

introv HP M. applys× triple_conseq M. rewrite× hwand_hpure_l.

Qed.

Combined and ramified rules.

Lemma triple_conseq_frame : $\forall H2 H1 Q1 t H Q,$
 $\text{triple } t H1 Q1 \rightarrow$
 $H ==> H1 \setminus * H2 \rightarrow$
 $Q1 \setminus *+ H2 ==> Q \rightarrow$
 $\text{triple } t H Q.$

Proof using.

introv M WH WQ. applys triple_conseq WH WQ. applys triple_frame M.

Qed.

Lemma triple_ramified_frame : $\forall H1 Q1 t H Q,$
 $\text{triple } t H1 Q1 \rightarrow$
 $H ==> H1 \setminus * (Q1 \setminus -* Q) \rightarrow$
 $\text{triple } t H Q.$

Proof using.

introv M W. applys triple_conseq_frame (Q1 \setminus - Q) M W.
{ rewrite← qwand_equiv. }*

Qed.

Named heaps.

Lemma hexists_named_eq : $\forall H,$
 $H = (\exists h, \setminus [H h] \setminus * (= h)).$

Proof using.

intros. apply himpl_antisym.

```

{ intros h K. applys hexists_intro h.
  rewrite× hstar_hpure_l. }
{ xpull. intros h K. intros ? →. auto. }
Qed.

```

Lemma triple_named_heap : $\forall t H Q,$
 $(\forall h, H h \rightarrow \text{triple } t (= h) Q) \rightarrow$
 $\text{triple } t H Q.$

Proof using.

```

intros M. rewrite (hexists_named_eq H).
applys triple_hexists. intros h.
applys× triple_hpure.

```

Qed.

35.6.3 Rules for Terms

Lemma triple_eval_like : $\forall t1 t2 H Q,$
 $\text{eval_like } t1 t2 \rightarrow$
 $\text{triple } t1 H Q \rightarrow$
 $\text{triple } t2 H Q.$

Proof using.

```

intros E M1. intros H. applys hoare_eval_like E. applys M1.

```

Qed.

Lemma triple_val : $\forall v H Q,$
 $H ==> Q v \rightarrow$
 $\text{triple } (\text{trm_val } v) H Q.$

Proof using.

```

intros M. intros HF. applys hoare_val. { xchanges M. }

```

Qed.

Lemma triple_fun : $\forall x t1 H Q,$
 $H ==> Q (\text{val_fun } x t1) \rightarrow$
 $\text{triple } (\text{trm_fun } x t1) H Q.$

Proof using.

```

intros M. intros HF. applys¬ hoare_fun. { xchanges M. }

```

Qed.

Lemma triple_fix : $\forall f x t1 H Q,$
 $H ==> Q (\text{val_fix } f x t1) \rightarrow$
 $\text{triple } (\text{trm_fix } f x t1) H Q.$

Proof using.

```

intros M. intros HF. applys¬ hoare_fix. { xchanges M. }

```

Qed.

Lemma triple_seq : $\forall t1 t2 H Q H1,$

```

triple t1 H (fun v => H1) →
triple t2 H1 Q →
triple (trm_seq t1 t2) H Q.

```

Proof using.

```

intros M1 M2. intros HF. applys hoare_seq.
{ applys M1. }
{ applys hoare_conseq M2; xsimpl. }

```

Qed.

```

Lemma triple_let : ∀ x t1 t2 Q1 H Q,
triple t1 H Q1 →
(∀ v1, triple (subst x v1 t2) (Q1 v1) Q) →
triple (trm_let x t1 t2) H Q.

```

Proof using.

```

intros M1 M2. intros HF. applys hoare_let.
{ applys M1. }
{ intros v. applys hoare_conseq M2; xsimpl. }

```

Qed.

```

Lemma triple_let_val : ∀ x v1 t2 H Q,
triple (subst x v1 t2) H Q →
triple (trm_let x v1 t2) H Q.

```

Proof using.

```

intros M. applys triple_let (fun v => \[v = v1] \* H).
{ applys triple_val. xsimpl. }
{ intros v. applys triple_hpure. intros →. applys M. }

```

Qed.

```

Lemma triple_if : ∀ (b:bool) t1 t2 H Q,
triple (if b then t1 else t2) H Q →
triple (trm_if b t1 t2) H Q.

```

Proof using.

```

intros M1. intros HF. applys hoare_if. applys M1.

```

Qed.

```

Lemma triple_app_fun : ∀ x v1 v2 t1 H Q,
v1 = val_fun x t1 →
triple (subst x v2 t1) H Q →
triple (trm_app v1 v2) H Q.

```

Proof using.

```

unfold triple. intros E M1. intros H'.
applys hoare_app_fun E. applys M1.

```

Qed.

```

Lemma triple_app_fun_direct : ∀ x v2 t1 H Q,
triple (subst x v2 t1) H Q →

```

`triple (trm_app (val_fun x t1) v2) H Q.`
 Proof using. *introv M. applys* `triple_app_fun`. Qed.

`Lemma triple_app_fix : ∀ v1 v2 f x t1 H Q,`
`v1 = val_fix f x t1 →`
`triple (subst x v2 (subst f v1 t1)) H Q →`
`triple (trm_app v1 v2) H Q.`

Proof using.
`unfold triple. introv E M1. intros H'.`
`applys hoare_app_fix E. applys M1.`

Qed.

`Lemma triple_app_fix_direct : ∀ v2 f x t1 H Q,`
`triple (subst x v2 (subst f (val_fix f x t1) t1)) H Q →`
`triple (trm_app (val_fix f x t1) v2) H Q.`

Proof using. *introv M. applys* `triple_app_fix`. Qed.

35.6.4 Triple-Style Specification for Primitive Functions

Operations on the state.

`Lemma triple_ref : ∀ v,`
`triple (val_ref v)`
`\[]`
`(funloc p ⇒ p ~~> v).`

Proof using.
`intros. unfold triple. intros H'. applys hoare_conseq hoare_ref; xsimpl¬.`
 Qed.

`Lemma triple_get : ∀ v p,`
`triple (val_get p)`
`(p ~~> v)`
`(fun r ⇒ \[r = v] * (p ~~> v)).`

Proof using.
`intros. unfold triple. intros H'. applys hoare_conseq hoare_get; xsimpl¬.`
 Qed.

`Lemma triple_set : ∀ w p v,`
`triple (val_set (val_loc p) v)`
`(p ~~> w)`
`(fun _ ⇒ p ~~> v).`

Proof using.
`intros. unfold triple. intros H'. applys hoare_conseq hoare_set; xsimpl¬.`
 Qed.

`Lemma triple_free : ∀ p v,`
`triple (val_free (val_loc p))`

```
(p ~~> v)
  (fun _ => []).
```

Proof using.

```
intros. unfold triple. intros H'. applys hoare_conseq hoare_free; xsimpl¬.
Qed.
```

Other operations.

```
Lemma triple_unop : ∀ v op v1,
  evalunop op v1 v →
  triple (op v1) [] (fun r => \[r = v]).
```

Proof using.

```
introv R. unfold triple. intros H'.
applys× hoare_conseq hoare_unop. xsimpl×.
```

Qed.

```
Lemma triple_binop : ∀ v op v1 v2,
  evalbinop op v1 v2 v →
  triple (op v1 v2) [] (fun r => \[r = v]).
```

Proof using.

```
introv R. unfold triple. intros H'.
applys× hoare_conseq hoare_binop. xsimpl×.
```

Qed.

```
Lemma triple_add : ∀ n1 n2,
  triple (val_add n1 n2)
  []
  (fun r => \[r = val_int (n1 + n2)]).
```

Proof using. intros. applys× triple_binop. applys× evalbinop_add. Qed.

```
Lemma triple_div : ∀ n1 n2,
  n2 ≠ 0 →
  triple (val_div n1 n2)
  []
  (fun r => \[r = val_int (Z.quot n1 n2)]).
```

Proof using. intros. applys× triple_binop. applys× evalbinop_div. Qed.

```
Lemma triple_neg : ∀ (b1:bool),
  triple (val_neg b1)
  []
  (fun r => \[r = val_bool (neg b1)]).
```

Proof using. intros. applys× triple_unop. applys× evalunop_neg. Qed.

```
Lemma triple_opp : ∀ n1,
  triple (val_opp n1)
  []
  (fun r => \[r = val_int (- n1)]).
```

Proof using. intros. applys× triple_unop. applys× evalunop_opp. Qed.

```

Lemma triple_eq : ∀ v1 v2,
  triple (val_eq v1 v2)
  []
  (fun r ⇒ \[r = isTrue (v1 = v2)]).

Proof using. intros. applys× triple_binop. applys evalbinop_eq. Qed.

Lemma triple_neq : ∀ v1 v2,
  triple (val_neq v1 v2)
  []
  (fun r ⇒ \[r = isTrue (v1 ≠ v2)]).

Proof using. intros. applys× triple_binop. applys evalbinop_neq. Qed.

Lemma triple_sub : ∀ n1 n2,
  triple (val_sub n1 n2)
  []
  (fun r ⇒ \[r = val_int (n1 - n2)]).

Proof using. intros. applys× triple_binop. applys× evalbinop_sub. Qed.

Lemma triple_mul : ∀ n1 n2,
  triple (val_mul n1 n2)
  []
  (fun r ⇒ \[r = val_int (n1 × n2)]).

Proof using. intros. applys× triple_binop. applys× evalbinop_mul. Qed.

Lemma triple_mod : ∀ n1 n2,
  n2 ≠ 0 →
  triple (val_mod n1 n2)
  []
  (fun r ⇒ \[r = val_int (Z.rem n1 n2)]).

Proof using. intros. applys× triple_binop. applys× evalbinop_mod. Qed.

Lemma triple_le : ∀ n1 n2,
  triple (val_le n1 n2)
  []
  (fun r ⇒ \[r = isTrue (n1 ≤ n2)]).

Proof using. intros. applys× triple_binop. applys× evalbinop_le. Qed.

Lemma triple_lt : ∀ n1 n2,
  triple (val_lt n1 n2)
  []
  (fun r ⇒ \[r = isTrue (n1 < n2)]).

Proof using. intros. applys× triple_binop. applys× evalbinop_lt. Qed.

Lemma triple_ge : ∀ n1 n2,
  triple (val_ge n1 n2)
  []
  (fun r ⇒ \[r = isTrue (n1 ≥ n2)]).

Proof using. intros. applys× triple_binop. applys× evalbinop_ge. Qed.

```

```

Lemma triple_gt : ∀ n1 n2,
  triple (val_gt n1 n2)
  ([] []
    (fun r ⇒ \[r = isTrue (n1 > n2)]).

```

Proof using. intros. applys× triple_binop. applys× evalbinop_gt. Qed.

```

Lemma triple_ptr_add : ∀ p n,
  p + n ≥ 0 →
  triple (val_ptr_add p n)
  ([] []
    (fun r ⇒ \[r = val_loc (abs (p + n))]).

```

Proof using.

```

  intros. applys× triple_binop. applys× evalbinop_ptr_add.
  { rewrite¬ abs_nonneg. }

```

Qed.

```

Lemma triple_ptr_add_nat : ∀ p (f:nat),
  triple (val_ptr_add p f)
  ([] []
    (fun r ⇒ \[r = val_loc (p+f)%nat]).
```

Proof using.

```

  intros. applys triple_conseq triple_ptr_add. { math. } { xsimpl. }
  { xsimpl. intros. subst. fequals.
    applys eq_nat_of_eq_int. rewrite abs_nonneg; math. }

```

Qed.

35.6.5 Alternative Definition of wp

Definition of the Weakest-Precondition Judgment.

wp is defined on top of hoare triples. More precisely $\text{wp } t \ Q$ is a heap predicate such that $H ==> \text{wp } t \ Q$ if and only if $\text{triple } t \ H \ Q$, where $\text{triple } t \ H \ Q$ is defined as $\forall H', \text{hoare } t (H \setminus^* H') (Q \setminus^{*+} H')$.

```

Definition wp (t:trm) := fun (Q:val→hprop) ⇒
  ∃ H, H \setminus^* \[∀ H', \text{hoare } t (H \setminus^* H') (Q \setminus^{*+} H')].

```

Equivalence with triples.

```

Lemma wp_equiv : ∀ t H Q,
  (H ==> wp t Q) ↔ (triple t H Q).

```

Proof using.

```

  unfold wp, triple. iff M.
  { intros H'. applys hoare_conseq. 2:{ applys himpl_frame_l M. }
    { clear M. rewrite hstar_hexists. applys hoare_hexists. intros H''.
      rewrite (hstar_comm H'). rewrite hstar_assoc.

```

```

    applys hoare_hpure. intros N. applys N. }
    { auto. } }
{ xsimpl H. apply M. }
Qed.

```

Structural Rule for wp

The ramified frame rule.

```
Lemma wp_ramified : ∀ Q1 Q2 t,
  (wp t Q1) \* (Q1 \-* Q2) ==> (wp t Q2).
```

Proof using.

```

  intros. unfold wp. xpull. intros H M.
  xsimpl (H \* (Q1 \-* Q2)). intros H'.
  applys hoare_conseq M; xsimpl.

```

Qed.

Arguments wp_ramified : clear implicits.

Corollaries.

```
Lemma wp_conseq : ∀ t Q1 Q2,
  Q1 ==> Q2 →
  wp t Q1 ==> wp t Q2.
```

Proof using.

```
  introv M. applys himpl_trans_r (wp_ramified Q1 Q2). xsimpl. xchanges M.
```

Qed.

```
Lemma wp_frame : ∀ t H Q,
  (wp t Q) \* H ==> wp t (Q \*+ H).
```

Proof using. intros. applys himpl_trans_r wp_ramified. xsimpl. Qed.

```
Lemma wp_ramified_frame : ∀ t Q1 Q2,
  (wp t Q1) \* (Q1 \-* Q2) ==> (wp t Q2).
```

Proof using. intros. applys himpl_trans_r wp_ramified. xsimpl. Qed.

```
Lemma wp_ramified_trans : ∀ t H Q1 Q2,
  H ==> (wp t Q1) \* (Q1 \-* Q2) →
  H ==> (wp t Q2).
```

Proof using. introv M. xchange M. applys wp_ramified. Qed.

Weakest-Precondition Style Reasoning Rules for Terms.

```
Lemma wp_eval_like : ∀ t1 t2 Q,
  eval_like t1 t2 →
  wp t1 Q ==> wp t2 Q.
```

Proof using.

```
  introv E. unfold wp. xpull. intros H M. xsimpl H.
```

intros H' . applys hoare_eval_like $E M$.

Qed.

Lemma wp_val : $\forall v Q$,

$Q v \Rightarrow wp(\text{trm_val } v) Q$.

Proof using. intros. unfold wp. xsimpl; intros H' . applys hoare_val. xsimpl. Qed.

Lemma wp_fun : $\forall x t Q$,

$Q (\text{val_fun } x t) \Rightarrow wp(\text{trm_fun } x t) Q$.

Proof using. intros. unfold wp. xsimpl; intros H' . applys hoare_fun. xsimpl. Qed.

Lemma wp_fix : $\forall f x t Q$,

$Q (\text{val_fix } f x t) \Rightarrow wp(\text{trm_fix } f x t) Q$.

Proof using. intros. unfold wp. xsimpl; intros H' . applys hoare_fix. xsimpl. Qed.

Lemma wp_app_fun : $\forall x v1 v2 t1 Q$,

$v1 = \text{val_fun } x t1 \rightarrow$

$wp(\text{subst } x v2 t1) Q \Rightarrow wp(\text{trm_app } v1 v2) Q$.

Proof using. introv EQ1. unfold wp. xsimpl; intros. applys \times hoare_app_fun. Qed.

Lemma wp_app_fix : $\forall f x v1 v2 t1 Q$,

$v1 = \text{val_fix } f x t1 \rightarrow$

$wp(\text{subst } x v2 (\text{subst } f v1 t1)) Q \Rightarrow wp(\text{trm_app } v1 v2) Q$.

Proof using. introv EQ1. unfold wp. xsimpl; intros. applys \times hoare_app_fix. Qed.

Lemma wp_seq : $\forall t1 t2 Q$,

$wp t1 (\text{fun } r \Rightarrow wp t2 Q) \Rightarrow wp(\text{trm_seq } t1 t2) Q$.

Proof using.

intros. unfold wp at 1. xsimpl. intros $H' M1$.

unfold wp at 1. xsimpl. intros H'' .

applys hoare_seq. applys (rm $M1$). unfold wp.

repeat rewrite hstar_hexists. applys hoare_hexists; intros H''' .

rewrite (hstar_comm H'''); repeat rewrite hstar_assoc.

applys hoare_hpure; intros $M2$. applys hoare_conseq $M2$; xsimpl.

Qed.

Lemma wp_let : $\forall x t1 t2 Q$,

$wp t1 (\text{fun } v \Rightarrow wp(\text{subst } x v t2) Q) \Rightarrow wp(\text{trm_let } x t1 t2) Q$.

Proof using.

intros. unfold wp at 1. xsimpl. intros $H' M1$.

unfold wp at 1. xsimpl. intros H'' .

applys hoare_let. applys (rm $M1$). intros v . simpl. unfold wp.

repeat rewrite hstar_hexists. applys hoare_hexists; intros H''' .

rewrite (hstar_comm H'''). rewrite hstar_assoc.

applys hoare_hpure; intros $M2$. applys hoare_conseq $M2$; xsimpl.

Qed.

Lemma wp_if : $\forall b t1 t2 Q$,

```
wp (if b then t1 else t2) Q ==> wp (trm_if b t1 t2) Q.
```

Proof using.

```
intros. repeat unfold wp. xsimpl; intros H M H'.
```

```
applys hoare_if. applys M.
```

Qed.

Lemma wp_if_case : $\forall b t1 t2 Q$,

```
(if b then wp t1 Q else wp t2 Q) ==> wp (trm_if b t1 t2) Q.
```

Proof using. intros. applys himpl_trans wp_if. case_if \neg . Qed.

35.7 WP Generator

This section defines a “weakest-precondition style characteristic formula generator”. This technology adapts the technique of “characteristic formulae” (originally developed in CFML 1.0) to produce weakest preconditions. (The formulae, their manipulation, and their correctness proofs are simpler in wp-style.)

The goal of the section is to define a function `wpgen t`, recursively over the structure of `t`, such that `wpgen t Q` entails `wp t Q`. Unlike `wp t Q`, which is defined semantically, `wpgen t Q` is defined following the syntax of `t`.

Technically, we define `wpgen E t`, where `E` is a list of bindings, to compute a formula that entails `wp (isubst E t)`, where `isubst E t` denotes the iterated substitution of bindings from `E` inside `t`.

35.7.1 Definition of Context as List of Bindings

In order to define a structurally recursive and relatively efficient characteristic formula generator, we need to introduce contexts, that essentially serve to apply substitutions lazily.

Open Scope `liblist_scope`.

A context is an association list from variables to values.

Definition `ctx : Type := list (var \times val)`.

`lookup x E` returns `Some v` if `x` is bound to a value `v`, and `None` otherwise.

Fixpoint `lookup (x:var) (E:ctx) : option val :=`

```
match E with
```

```
| nil  $\Rightarrow$  None
```

```
| (y, v)::E1  $\Rightarrow$  if var_eq x y  
then Some v  
else lookup x E1
```

```
end.
```

`rem x E` denotes the removal of bindings on `x` from `E`.

Fixpoint `rem (x:var) (E:ctx) : ctx :=`

```
match E with
```

```

| nil ⇒ nil
| (y, v)::E1 ⇒
  let E1' := rem x E1 in
    if var_eq x y then E1' else (y, v)::E1'
end.

```

ctx_disjoint $E_1 \ E_2$ asserts that the two contexts have disjoint domains.

Definition ctx_disjoint ($E_1 \ E_2:\text{ctx}$) : Prop :=
 $\forall x \ v_1 \ v_2, \text{lookup } x \ E_1 = \text{Some } v_1 \rightarrow \text{lookup } x \ E_2 = \text{Some } v_2 \rightarrow \text{False}.$

ctx_equiv $E_1 \ E_2$ asserts that the two contexts bind same keys to same values.

Definition ctx_equiv ($E_1 \ E_2:\text{ctx}$) : Prop :=
 $\forall x, \text{lookup } x \ E_1 = \text{lookup } x \ E_2.$

Basic properties of context operations follow.

Section CtxOps.

Lemma lookup_app : $\forall E_1 \ E_2 \ x,$
 $\text{lookup } x (E_1 ++ E_2) = \text{match } \text{lookup } x \ E_1 \text{ with}$

- | None ⇒ $\text{lookup } x \ E_2$
- | Some $v \Rightarrow \text{Some } v$

end.

Proof using.

```

intros. induction E1 as [(y,w) E1']; rew_list; simpl; intros.
{ auto. } { case_var¬. }

```

Qed.

Lemma lookup_rem : $\forall x \ y \ E,$
 $\text{lookup } x (\text{rem } y \ E) = \text{If } x = y \text{ then None else } \text{lookup } x \ E.$

Proof using.

```

intros. induction E as [(z,v) E].
{ simpl. case_var¬. }
{ simpl. case_var~; simpl; case_var¬. }

```

Qed.

Lemma rem_app : $\forall x \ E_1 \ E_2,$
 $\text{rem } x (E_1 ++ E_2) = \text{rem } x \ E_1 ++ \text{rem } x \ E_2.$

Proof using.

```

intros. induction E1 as [(y,w) E1']; rew_list; simpl. { auto. }
{ case_var¬. { rew_list. fequals. } }

```

Qed.

Lemma ctx_equiv_rem : $\forall x \ E_1 \ E_2,$
 $\text{ctx_equiv } E_1 \ E_2 \rightarrow$
 $\text{ctx_equiv } (\text{rem } x \ E_1) (\text{rem } x \ E_2).$

Proof using.

```

intros M. unfolds ctx_equiv. intros y.

```

```
do 2 rewrite lookup_rem. case_var¬.
Qed.
```

```
Lemma ctx_disjoint_rem : ∀ x E1 E2,
  ctx_disjoint E1 E2 →
  ctx_disjoint (rem x E1) (rem x E2).
```

Proof using.

```
intros D. intros y v1 v2 K1 K2. rewrite lookup_rem in *.
case_var¬. applys× D K1 K2.
```

Qed.

```
Lemma ctx_disjoint_equiv_app : ∀ E1 E2,
  ctx_disjoint E1 E2 →
  ctx_equiv (E1 ++ E2) (E2 ++ E1).
```

Proof using.

```
intros D. intros x. do 2 rewrite¬ lookup_app.
case_eq (lookup x E1); case_eq (lookup x E2); auto.
{ intros v2 K2 v1 K1. false× D. }
```

Qed.

End CtxOps.

35.7.2 Multi-Substitution

Definition of Multi-Substitution

The specification of the characteristic formula generator is expressed using the multi-substitution function, which substitutes a list of bindings inside a term.

```
Fixpoint isubst (E:ctx) (t:trm) : trm :=
  match t with
  | trm_val v ⇒
    v
  | trm_var x ⇒
    match lookup x E with
    | None ⇒ t
    | Some v ⇒ v
    end
  | trm_fun x t1 ⇒
    trm_fun x (isubst (rem x E) t1)
  | trm_fix f x t1 ⇒
    trm_fix f x (isubst (rem x (rem f E)) t1)
  | trm_if t0 t1 t2 ⇒
    trm_if (isubst E t0) (isubst E t1) (isubst E t2)
  | trm_seq t1 t2 ⇒
    trm_seq (isubst E t1) (isubst E t2)
```

```

| trm_let x t1 t2 =>
  trm_let x (isubst E t1) (isubst (rem x E) t2)
| trm_app t1 t2 =>
  trm_app (isubst E t1) (isubst E t2)
end.

```

Properties of Multi-Substitution

The goal of this entire section is only to establish `isubst_nil` and `isubst_rem`, which assert:

$$\text{isubst nil } t = t \text{ and } \text{isubst } ((x,v)::E) t = \text{subst } x v (\text{isubst } (\text{rem } x E) t)$$

The first targeted lemma.

Lemma `isubst_nil` : $\forall t,$
 $\text{isubst nil } t = t.$

Proof using. `intros t. induction t; simpl; f_equal.` Qed.

The next lemma relates `subst` and `isubst`.

Lemma `subst_eq_isubst_one` : $\forall x v t,$
 $\text{subst } x v t = \text{isubst } ((x,v)::\text{nil}) t.$

Proof using.

```

intros. induction t; simpl.
{ f_equal. }
{ case_var. }
{ f_equal. case_var. { rewrite_invert isubst_nil. } }
{ f_equal. case_var; try case_var; simpl; try case_var; try rewrite isubst_nil; auto. }
{ f_equal. }
{ f_equal. }
{ f_equal. case_var. { rewrite_invert isubst_nil. } }
{ f_equal. }
```

Qed.

The next lemma shows that equivalent contexts produce equal results for `isubst`.

Lemma `isubst_ctx_equiv` : $\forall t E1 E2,$
 $\text{ctx_equiv } E1 E2 \rightarrow$
 $\text{isubst } E1 t = \text{isubst } E2 t.$

Proof using.

```

hint ctx_equiv_rem.
intros t. induction t; introv EQ; simpl; f_equal_invert.
{ rewrite_invert EQ. }
```

Qed.

The next lemma asserts that `isubst` distribute over concatenation.

Lemma `isubst_app` : $\forall t E1 E2,$
 $\text{isubst } (E1 ++ E2) t = \text{isubst } E2 (\text{isubst } E1 t).$

Proof using.

```

hint ctx_disjoint_rem.

intros t. induction t; simpl; intros.
{ fequals. }
{ rename v into x. rewrite¬ lookup_app.
  case_eq (lookup x E1); introv K1; case_eq (lookup x E2); introv K2; auto.
  { simpl. rewrite¬ K2. }
  { simpl. rewrite¬ K2. } }
{ fequals. rewrite× rem_app. }
{ fequals. do 2 rewrite× rem_app. }
{ fequals×. }
{ fequals×. }
{ fequals×. { rewrite× rem_app. } }
{ fequals×. }
Qed.

```

The next lemma asserts that the concatenation order is irrelevant in a substitution if the contexts have disjoint domains.

```

Lemma isubst_app_swap : ∀ t E1 E2,
  ctx_disjoint E1 E2 →
  isubst (E1 ++ E2) t = isubst (E2 ++ E1) t.

```

Proof using.

```
  introv D. applys isubst_ctx_equiv. applys¬ ctx_disjoint_equiv_app.
```

Qed.

We are ready to derive the second targeted property of isubst.

```

Lemma isubst_rem : ∀ x v E t,
  isubst ((x, v)::E) t = subst x v (isubst (rem x E) t).

```

Proof using.

```
  intros. rewrite subst_eq_isubst_one. rewrite ← isubst_app.
  rewrite isubst_app_swap.
  { applys isubst_ctx_equiv. intros y. rew_list. simpl. rewrite lookup_rem. case_var¬. }
  { intros y v1 v2 K1 K2. simpls. rewrite lookup_rem in K1. case_var. }
```

Qed.

A variant useful for trm_fix is proved next.

```

Lemma isubst_rem_2 : ∀ f x vf vx E t,
  isubst ((f, vf)::(x, vx)::E) t = subst x vx (subst f vf (isubst (rem x (rem f E)) t)).

```

Proof using.

```
  intros. do 2 rewrite subst_eq_isubst_one. do 2 rewrite ← isubst_app.
  rewrite isubst_app_swap.
  { applys isubst_ctx_equiv. intros y. rew_list. simpl. do 2 rewrite lookup_rem. case_var¬. }
  { intros y v1 v2 K1 K2. rew_listx in *. simpls. do 2 rewrite lookup_rem in K1. case_var. }
```

Qed.

35.7.3 Definition of `wpgen`

The definition of `wpgen E t` comes next. It depends on a predicate called `mkstruct` for handling structural rules, and on one auxiliary definition for each term rule.

Definition of `mkstruct`

Let `formula` denote the type of `wp t` and `wpgen t`.

Definition `formula := (val → hprop) → hprop.`

Implicit Type `F : formula.`

`mkstruct F` transforms a formula `F` into one that satisfies structural rules of Separation Logic. This predicate transformer enables integrating support for the frame rule (and other structural rules), in characteristic formulae.

Definition `mkstruct (F:formula) : formula :=`
`fun Q ⇒ ∃ Q', F Q' ∗ (Q' ∗ Q).`

Lemma `mkstruct_ramified : ∀ Q1 Q2 F,`
`(mkstruct F Q1) ∗ (Q1 ∗ Q2) ==> (mkstruct F Q2).`

Proof using. `unfold mkstruct. xsimpl.` Qed.

Arguments `mkstruct_ramified : clear implicits.`

Lemma `mkstruct_erase : ∀ Q F,`
`F Q ==> mkstruct F Q.`

Proof using. `unfolds mkstruct. xsimpl.` Qed.

Arguments `mkstruct_erase : clear implicits.`

Lemma `mkstruct_conseq : ∀ F Q1 Q2,`
`Q1 ==> Q2 →`
`mkstruct F Q1 ==> mkstruct F Q2.`

Proof using.

`intros WQ. unfolds mkstruct. xpull. intros Q. xsimpl Q. xchanges WQ.`

Qed.

Lemma `mkstruct_frame : ∀ F H Q,`
`(mkstruct F Q) ∗ H ==> mkstruct F (Q ∗+ H).`

Proof using.

`intros. unfold mkstruct. xpull. intros Q'. xsimpl Q'.`

Qed.

Lemma `mkstruct_monotone : ∀ F1 F2 Q,`
`(∀ Q, F1 Q ==> F2 Q) →`
`mkstruct F1 Q ==> mkstruct F2 Q.`

Proof using.

intros WF . *unfolds* `mkstruct`. *xpull*. *intros* Q' . *xchange* WF . *xsimpl* Q' .
Qed.

Definition of Auxiliary Definition for `wpgen`

we state auxiliary definitions for `wpgen`, one per term construct. For simplicity, we here assume the term t to be in A-normal form. If it is not, the formula generated will be incomplete, that is, useless to prove triples about the term t . Note that the actual generator in CFML2 does support terms that are not in A-normal form.

Definition `wpgen_fail` : formula := fun $Q \Rightarrow \text{False}$.

Definition `wpgen_val` ($v:\text{val}$) : formula := fun $Q \Rightarrow Q v$.

Definition `wpgen_fun` ($Fof:\text{val} \rightarrow \text{formula}$) : formula := fun $Q \Rightarrow \forall vf, \forall vx Q' \Rightarrow wp(\text{trm_app } vf vx) Q' \dashv^\ast Q vf$.

Definition `wpgen_fix` ($Fof:\text{val} \rightarrow \text{val} \rightarrow \text{formula}$) : formula := fun $Q \Rightarrow \forall vf, \forall vx Q' \Rightarrow wp(\text{trm_app } vf vx) Q' \dashv^\ast Q vf$.

Definition `wpgen_var` ($E:\text{ctx}$) ($x:\text{var}$) : formula :=
 match lookup $x E$ with
 | `None` \Rightarrow `wpgen_fail`
 | `Some` $v \Rightarrow$ `wpgen_val` v
 end.

Definition `wpgen_seq` ($F1 F2:\text{formula}$) : formula := fun $Q \Rightarrow F1 (\text{fun } v \Rightarrow F2 Q)$.

Definition `wpgen_let` ($F1:\text{formula}$) ($F2of:\text{val} \rightarrow \text{formula}$) : formula := fun $Q \Rightarrow F1 (\text{fun } v \Rightarrow F2of v Q)$.

Definition `wpgen_if` ($t:\text{trm}$) ($F1 F2:\text{formula}$) : formula := fun $Q \Rightarrow \exists (b:\text{bool}), \exists (t = \text{trm_val } (\text{val_bool } b)) \dashv^\ast (\text{if } b \text{ then } F1 Q \text{ else } F2 Q)$.

Definition `wpgen_if_trm` ($F0 F1 F2:\text{formula}$) : formula :=
`wpgen_let` $F0 (\text{fun } v \Rightarrow \text{mkstruct}(\text{wpgen_if } v F1 F2))$.

Recursive Definition of `wpgen`

`wpgen` $E t$ is structurally recursive on t . Note that this function does not recurse through values. Note also that the context E gets extended when traversing bindings, in the let-binding and the function cases.

Fixpoint `wpgen` ($E:\text{ctx}$) ($t:\text{trm}$) : formula :=
`mkstruct` **match** t **with**
 | `trm_val` $v \Rightarrow$ `wpgen_val` v
 | `trm_var` $x \Rightarrow$ `wpgen_var` $E x$

```

| trm_fun x t1 => wp gen_fun (fun v => wp gen ((x,v)::E) t1)
| trm_fix f x t1 => wp gen_fix (fun vf v => wp gen ((f,vf)::(x,v)::E) t1)
| trm_if t0 t1 t2 => wp gen_if (isubst E t0) (wp gen E t1) (wp gen E t2)
| trm_seq t1 t2 => wp gen_seq (wp gen E t1) (wp gen E t2)
| trm_let x t1 t2 => wp gen_let (wp gen E t1) (fun v => wp gen ((x,v)::E) t2)
| trm_app t1 t2 => wp (isubst E t)
end.

```

Soundness of `wp gen`

`formula_sound t F` asserts that, for any Q , the Separation Logic judgment triple $(F\ Q)\ t\ Q$ is valid. In other words, it states that F is a stronger formula than `wp t`.

The soundness theorem that we are ultimately interested in asserts that `formula_sound (isubst E t) (wp gen E t)` holds for any E and t .

Definition `formula_sound (t:trm) (F:formula) : Prop :=`

```
  ∀ Q, F Q ==> wp t Q.
```

Lemma `wp_sound : ∀ t,`

```
  formula_sound t (wp t).
```

Proof using. `intros. intros Q. applys himpl_refl. Qed.`

One soundness lemma for `mkstruct`.

Lemma `mkstruct_wp : ∀ t,`

```
  mkstruct (wp t) = (wp t).
```

Proof using.

```
  intros. applys fun_ext_1. intros Q. applys himpl_antisym.
  { unfold mkstruct. xpull. intros Q'. applys wp_ramified. }
  { applys mkstruct_erase. }
```

Qed.

Lemma `mkstruct_sound : ∀ t F,`

```
  formula_sound t F →
```

```
  formula_sound t (mkstruct F).
```

Proof using.

```
  introv M. unfolds formula_sound. intros Q'.
  rewrite ← mkstruct_wp. applys× mkstruct_monotone M.
```

Qed.

One soundness lemma for each term construct.

Lemma `wp gen_fail_sound : ∀ t,`

```
  formula_sound t wp gen_fail.
```

Proof using. `intros. intros Q. unfold wp gen_fail. xpull. Qed.`

Lemma `wp gen_val_sound : ∀ v,`

```
  formula_sound (trm_val v) (wp gen_val v).
```

Proof using. intros. intros Q . unfolds wp_gen_val. applys wp_val. Qed.

Lemma wp_gen_fun_sound : $\forall x t1 Fof$,

$$(\forall vx, formula_sound (subst x vx t1) (Fof vx)) \rightarrow formula_sound (trm_fun x t1) (wp_gen_fun Fof).$$

Proof using.

introv M . intros Q . unfolds wp_gen_fun. applys himpl_hforall_l (val_fun x t1).

xchange hwand_hpure_l.

{ intros. applys himpl_trans_r. { applys× wp_app_fun. } { applys× M. } }
{ applys wp_fun. }

Qed.

Lemma wp_gen_fix_sound : $\forall f x t1 Fof$,

$$(\forall vf vx, formula_sound (subst x vx (subst f vf t1)) (Fof vf vx)) \rightarrow formula_sound (trm_fix f x t1) (wp_gen_fix Fof).$$

Proof using.

introv M . intros Q . unfolds wp_gen_fix. applys himpl_hforall_l (val_fix f x t1).

xchange hwand_hpure_l.

{ intros. applys himpl_trans_r. { applys× wp_app_fix. } { applys× M. } }
{ applys wp_fix. }

Qed.

Lemma wp_gen_seq_sound : $\forall F1 F2 t1 t2$,

formula_sound $t1 F1 \rightarrow$

formula_sound $t2 F2 \rightarrow$

formula_sound (trm_seq $t1 t2$) (wp_gen_seq $F1 F2$).

Proof using.

introv $S1 S2$. intros Q . unfolds wp_gen_seq. applys himpl_trans wp_seq.

applys himpl_trans $S1$. applys wp_conseq. intros v . applys $S2$.

Qed.

Lemma wp_gen_let_sound : $\forall F1 F2of x t1 t2$,

formula_sound $t1 F1 \rightarrow$

($\forall v, formula_sound (subst x v t2) (F2of v)$) \rightarrow

formula_sound (trm_let $x t1 t2$) (wp_gen_let $F1 F2of$).

Proof using.

introv $S1 S2$. intros Q . unfolds wp_gen_let. applys himpl_trans wp_let.

applys himpl_trans $S1$. applys wp_conseq. intros v . applys $S2$.

Qed.

Lemma wp_gen_if_sound : $\forall F1 F2 t0 t1 t2$,

formula_sound $t1 F1 \rightarrow$

formula_sound $t2 F2 \rightarrow$

formula_sound (trm_if $t0 t1 t2$) (wp_gen_if $t0 F1 F2$).

Proof using.

introv $S1 S2$. intros Q . unfold wp_gen_if. xpull. intros $b \rightarrow$.

applys himpl_trans wp_if. case_if. { applys S1. } { applys S2. }
 Qed.

The main inductive proof for the soundness theorem.

Lemma `wpgen_sound : ∀ E t,`
`formula_sound (isubst E t) (wpgen E t).`

Proof using.

```
intros. gen E. induction t; intros; simpl;
  applys mkstruct_sound.
  { applys wpgen_val_sound. }
  { rename v into x. unfold wpgen_var. case_eq (lookup x E).
    { intros v EQ. applys wpgen_val_sound. }
    { intros N. applys wpgen_fail_sound. } }
  { rename v into x. applys wpgen_fun_sound.
    { intros vx. rewrite× ← isubst_rem. } }
  { rename v into f, v0 into x. applys wpgen_fix_sound.
    { intros vf vx. rewrite× ← isubst_rem_2. } }
  { applys wp_sound. }
  { applys× wpgen_seq_sound. }
  { rename v into x. applys× wpgen_let_sound.
    { intros v. rewrite× ← isubst_rem. } }
  { applys× wpgen_if_sound. }
```

Qed.

Lemma `himpl_wpgen_wp : ∀ t Q,`
`wpgen nil t Q ==> wp t Q.`

Proof using.

```
introv M. lets N: (wpgen_sound nil t). rewrite isubst_nil in N. applys× N.
```

Qed.

The final theorem for closed terms.

Lemma `triple_of_wpgen : ∀ t H Q,`
`H ==> wpgen nil t Q →`
`triple t H Q.`

Proof using.

```
introv M. rewrite ← wp_equiv. xchange M. applys himpl_wpgen_wp.
```

Qed.

35.8 Practical Proofs

This last section shows the techniques involved in setting up the lemmas and tactics required to carry out verification in practice, through concise proof scripts.

35.8.1 Lemmas for Tactics to Manipulate `wpgen` Formulae

Lemma `xstruct_lemma` : $\forall F H Q,$

$H ==> F Q \rightarrow$

$H ==> \text{mkstruct } F Q.$

Proof using. `introv M. xchange M. applys mkstruct_erase.` Qed.

Lemma `xval_lemma` : $\forall v H Q,$

$H ==> Q v \rightarrow$

$H ==> \text{wpgen_val } v Q.$

Proof using. `introv M. applys M.` Qed.

Lemma `xlet_lemma` : $\forall H F1 F2of Q,$

$H ==> F1 (\text{fun } v \Rightarrow F2of v Q) \rightarrow$

$H ==> \text{wpgen_let } F1 F2of Q.$

Proof using. `introv M. xchange M.` Qed.

Lemma `xseq_lemma` : $\forall H F1 F2 Q,$

$H ==> F1 (\text{fun } v \Rightarrow F2 Q) \rightarrow$

$H ==> \text{wpgen_seq } F1 F2 Q.$

Proof using. `introv M. xchange M.` Qed.

Lemma `xif_lemma` : $\forall b H F1 F2 Q,$

$(b = \text{true} \rightarrow H ==> F1 Q) \rightarrow$

$(b = \text{false} \rightarrow H ==> F2 Q) \rightarrow$

$H ==> (\text{wpgen_if } b F1 F2) Q.$

Proof using. `introv M1 M2. unfold wpgen_if. xsimpl b. case_if.` Qed.

Lemma `xapp_lemma` : $\forall t Q1 H1 H Q,$

$\text{triple } t H1 Q1 \rightarrow$

$H ==> H1 \text{ \textit{*}} (Q1 \text{ \textit{-*}} \text{protect } Q) \rightarrow$

$H ==> \text{wp } t Q.$

Proof using.

`introv M W. rewrite ← wp_equiv in M. xchange W. xchange M.`

`applys wp_ramified_frame.`

Qed.

Lemma `xfun_spec_lemma` : $\forall (S:\text{val} \rightarrow \text{Prop}) H Q Fof,$

$(\forall vf,$

$(\forall vx H' Q', (H' ==> Fof vx Q') \rightarrow \text{triple } (\text{trm_app } vf vx) H' Q') \rightarrow$

$S vf) \rightarrow$

$(\forall vf, S vf \rightarrow (H ==> Q vf)) \rightarrow$

$H ==> \text{wpgen_fun } Fof Q.$

Proof using.

`introv M1 M2. unfold wpgen_fun. xsimpl. intros vf N.`

`applys M2. applys M1. introv K. rewrite ← wp_equiv. xchange K. applys N.`

Qed.

```

Lemma xfun_nospec_lemma : ∀ H Q Fof,
  (∀ vf,
    (∀ vx H' Q', (H' ==> Fof vx Q') → triple (trm_app vf vx) H' Q') →
    (H ==> Q vf)) →
  H ==> wp_gen_fun Fof Q.

```

Proof using.

```

  introv M. unfold wp_gen_fun. xsimpl. intros vf N. applys M.
  introv K. rewrite ← wp_equiv. xchange K. applys N.

```

Qed.

```

Lemma xwp_lemma_fun : ∀ v1 v2 x t H Q,
  v1 = val_fun x t →
  H ==> wp_gen ((x, v2)::nil) t Q →
  triple (trm_app v1 v2) H Q.

```

Proof using.

```

  introv M1 M2. rewrite ← wp_equiv. xchange M2.
  xchange (» wp_gen_sound ((x, v2)::nil) t Q).
  rewrite ← subst_eq_isubst_one. applys× wp_app_fun.

```

Qed.

```

Lemma xwp_lemma_fix : ∀ v1 v2 f x t H Q,
  v1 = val_fix f x t →
  H ==> wp_gen (((f, v1)::(x, v2)::nil) t Q) →
  triple (trm_app v1 v2) H Q.

```

Proof using.

```

  introv M1 M2. rewrite ← wp_equiv. xchange M2.
  xchange (» wp_gen_sound (((f, v1)::nil) ++ (x, v2)::nil) t Q).
  rewrite isubst_app. do 2 rewrite ← subst_eq_isubst_one.
  applys× wp_app_fix.

```

Qed.

```

Lemma xtriple_lemma : ∀ t H (Q:val→hprop),
  H ==> mkstruct (wp t) Q →
  triple t H Q.

```

Proof using.

```

  introv M. rewrite ← wp_equiv. xchange M. unfold mkstruct.
  xpull. intros Q'. applys wp_ramified_frame.

```

Qed.

35.8.2 Tactics to Manipulate **wp_gen** Formulae

The tactic are presented in WPgen.

Database of hints for *xapp*.

```
Hint Resolve triple_get triple_set triple_ref triple_free : triple.
```

```
Hint Resolve triple_add triple_div triple_neg triple_opp triple_eq
  triple_neq triple_sub triple_mul triple_mod triple_le triple_lt
  triple_ge triple_gt triple_ptr_add triple_ptr_add_nat : triple.
```

xstruct removes the leading `mkstruct`.

```
Tactic Notation "xstruct" :=
  applics xstruct_lemma.
```

xstruct_if_needed removes the leading `mkstruct` if there is one.

```
Tactic Notation "xstruct_if_needed" :=
  try match goal with ⊢ ?H ==> mkstruct ?F ?Q ⇒ xstruct end.
```

```
Tactic Notation "xval" :=
  xstruct_if_needed; applics xval_lemma.
```

```
Tactic Notation "xlet" :=
  xstruct_if_needed; applics xlet_lemma.
```

```
Tactic Notation "xseq" :=
  xstruct_if_needed; applics xseq_lemma.
```

```
Tactic Notation "xseq_xlet_if_needed" :=
  try match goal with ⊢ ?H ==> mkstruct ?F ?Q ⇒
    match F with
    | wpigen_seq ?F1 ?F2 ⇒ xseq
    | wpigen_let ?F1 ?F2of ⇒ xlet
    end end.
```

```
Tactic Notation "xif" :=
  xseq_xlet_if_needed; xstruct_if_needed;
  applics xif_lemma; rew_bool_eq.
```

xapp_try_clear_unit_result implements some post-processing for cleaning up unused variables.

```
Tactic Notation "xapp_try_clear_unit_result" :=
  try match goal with ⊢ val ↗ _ ⇒ intros _ end.
```

```
Tactic Notation "xtriple" :=
  intros; applics xtriple_lemma.
```

```
Tactic Notation "xtriple_if_needed" :=
  try match goal with ⊢ triple ?t ?H ?Q ⇒ applics xtriple_lemma end.
```

xapp_simpl performs the final step of the tactic *xapp*.

```
Lemma xapp_simpl_lemma : ∀ F H Q,
  H ==> F Q →
  H ==> F Q \* (Q \-* protect Q).
```

Proof using. introv M. xchange M. unfold protect. xsimpl. Qed.

```
Tactic Notation "xapp_simpl" :=
```

```
first [ applys xapp_simpl_lemma
       | xsimpl; unfold protect; xapp_try_clear_unit_result ].
```

Tactic Notation "xapp_pre" :=
xtriple_if_needed; xseq_xlet_if_needed; xstruct_if_needed.

xapp_nosubst E implements the heart of *xapp E*. If the argument *E* was always a triple, it would suffice to run *applys xapp_lemma E; xapp_simpl*. Yet, *E* might be a specification involving quantifiers. These quantifiers need to be first instantiated. This instantiation is achieved by means of the tactic *forwards_nounfold_then* offered by the TLC library.

Tactic Notation "xapp_nosubst" constr(*E*) :=
xapp_pre;
forwards_nounfold_then E ltac:(fun K => applys xapp_lemma K; xapp_simpl).

xapp_apply_spec implements the heart of *xapp*, when called without argument. It finds out the specification triple, either in the hint data base named *triple*, or in the context by looking for an induction hypothesis. Disclaimer: as explained in *WPgen*, the simple implementation of *xapp_apply_spec* which we use here does not apply when the specification includes premises that cannot be solved by *eauto*; in such cases, the tactic *xapp E* must be called, providing the specification *E* explicitly. This limitation is overcome using more involved Hint Extern tricks in CFML 2.0.

Tactic Notation "xapp_apply_spec" :=
first [solve [eauto with triple]
| match goal with H: _ ⊢ _ => eapply H end].

xapp_nosubst_for_records is place holder for implementing *xapp* on records. It is implemented further on.

Ltac xapp_nosubst_for_records tt :=
fail.

xapp first calls *xtriple* if the goal is *triple t H Q* instead of *H ==> wp t Q*.

Tactic Notation "xapp_nosubst" :=
xapp_pre;
first [applys xapp_lemma; [xapp_apply_spec | xapp_simpl]
| xapp_nosubst_for_records tt].

xapp_try_subst checks if the goal is of the form:

- either $\forall (r:\text{val}), (r = \dots) \rightarrow \dots$
- or $\forall (r:\text{val}), \forall x, (r = \dots) \rightarrow \dots$

in which case it substitutes *r* away.

Tactic Notation "xapp_try_subst" :=
try match goal with
| ⊢ \forall (r:\text{val}), (r = _) \rightarrow _ => intros ? →

```

| ⊢ ∀ (r:val), ∀ x, (r = _) → _ ⇒
  let y := fresh x in intros ? y ->; revert y
end.

Tactic Notation "xapp" constr(E) :=
  xapp_nosubst E; xapp_try_subst.

Tactic Notation "xapp" :=
  xapp_nosubst; xapp_try_subst.

Tactic Notation "xapp_debug" :=
  xseq_xlet_if_needed; xstruct_if_needed; applys xapp_lemma.
  xapp is essentially equivalent to xapp_debug; [ xapp_apply_spec | xapp_simpl ] .

Tactic Notation "xfun" constr(S) :=
  xseq_xlet_if_needed; xstruct_if_needed; applys xfun_spec_lemma S.

Tactic Notation "xfun" :=
  xseq_xlet_if_needed; xstruct_if_needed; applys xfun_nospec_lemma.

xvars may be called for unfolding “program variables as definitions”, which take the form Vars.x, and revealing the underlying string.

Tactic Notation "xvars" :=
  DefinitionsForVariables.libsepvar_unfold.

xwp_simpl is a specialized version of simpl to be used for getting the function wp to compute properly.

Ltac xwp_simpl :=
  xvars;
  cbn beta delta [
    wpgen wp_gen_var isubst lookup var_eq
    string_dec string_rec string_rect
    sumbool_rec sumbool_rect
    Ascii.ascii_dec Ascii.ascii_rec Ascii.ascii_rect
    Bool.bool_dec bool_rec bool_rect ] iota zeta;
  simpl.

```

Tactic Notation "xwp" :=
 intros;
 first [applys xwp_lemma_fun; [reflexivity |]
 | applys xwp_lemma_fix; [reflexivity |]];
 xwp_simpl.

35.8.3 Notations for Triples and **wpgen**

Declare Scope wp_scope.

Notation for printing proof obligations arising from **wpgen**.

```

Notation "'PRE' H 'CODE' F 'POST' Q :=
  (H ==> (mkstruct F) Q)
  (at level 8, H at level 0, F, Q at level 0,
   format "[v' 'PRE' H '/' 'CODE' F '/' 'POST' Q ]") : wp_scope.

```

```

Notation "' F" :=
  (mkstruct F)
  (at level 10,
   format "' F") : wp_scope.

```

Custom grammar for the display of characteristic formulae.

Declare Custom Entry wp.

```

Notation "<[ e ]>" :=
  e
  (at level 0, e custom wp at level 99) : wp_scope.

```

```

Notation "' F" :=
  (mkstruct F)
  (in custom wp at level 10,
   format "' F") : wp_scope.

```

```

Notation "( x )" :=
  x
  (in custom wp,
   x at level 99) : wp_scope.

```

```

Notation "{ x }" :=
  x
  (in custom wp at level 0,
   x constr,
   only parsing) : wp_scope.

```

```

Notation "x" :=
  x
  (in custom wp at level 0,
   x constr at level 0) : wp_scope.

```

```

Notation "'Fail'" :=
  ((wpgen_fail))
  (in custom wp at level 69) : wp_scope.

```

```

Notation "'Val' v" :=
  ((wpgen_val v))
  (in custom wp at level 69) : wp_scope.

```

```

Notation "'Let' x ':=' F1 'in' F2" :=
  ((wpgen_let F1 (fun x => F2)))
  (in custom wp at level 69,

```

x name,
 $F1$ custom wp at level 99,
 $F2$ custom wp at level 99,
right associativity,
format "[v '[' Let' x ':=' F1 'in' ']' '/ '[' F2 ']']'" : wp_scope.

Notation "'Seq' F1 ; F2" :=
((wp_gen_seq F1 F2))
(in custom wp at level 68,
 $F1$ custom wp at level 99,
 $F2$ custom wp at level 99,
right associativity,
format "'[v 'Seq' '[' F1 '] ; '/ '[' F2 ']']'" : wp_scope.

Notation "'App' f v1" :=
((wp (trm_app f v1)))
(in custom wp at level 68, f , $v1$ at level 0) : wp_scope.

Notation "'App' f v1 v2" :=
((wp (trm_app (trm_app f v1) v2)))
(in custom wp at level 68, f , $v1$, $v2$ at level 0) : wp_scope.

Notation "'If_-' v 'Then' F1 'Else' F2" :=
((wp_gen_if v F1 F2))
(in custom wp at level 69,
 $F1$ custom wp at level 99,
 $F2$ custom wp at level 99,
left associativity,
format "[v '[' If_-' v 'Then' ']' '/ '[' F1 ']' '/ 'Else' '/ '[' F2 ']']'" : wp_scope.

Notation "'Fun' x '=>' F1" :=
((wp_gen_fun (fun x => F1)))
(in custom wp at level 69,
 x name,
 $F1$ custom wp at level 99,
right associativity,
format "[v '[' Fun' x '=>' F1 ']']'" : wp_scope.

Notation "'Fix' f x '=>' F1" :=
((wp_gen_fix (fun f x => F1)))
(in custom wp at level 69,
 f name, x name,
 $F1$ custom wp at level 99,
right associativity,
format "[v '[' Fix' f x '=>' F1 ']']'" : wp_scope.

35.8.4 Notation for Concrete Terms

The custom grammar for **trm** is defined in **LibSepVar**.

Declare Scope val_scope.

Terms

Notation "<{ e }>" :=

e
(at level 0, *e* custom **trm** at level 99) : **trm_scope**.

Notation "(x)" :=

x
(in custom **trm**, *x* at level 99) : **trm_scope**.

Notation "'begin' e 'end'" :=

e
(in custom **trm**, *e* custom **trm** at level 99, only parsing) : **trm_scope**.

Notation "{ x }" :=

x
(in custom **trm**, *x* constr) : **trm_scope**.

Notation "x" := *x*

(in custom **trm** at level 0,
x constr at level 0) : **trm_scope**.

Notation "t1 t2" := (**trm_app** *t1* *t2*)

(in custom **trm** at level 30,
left associativity,
only parsing) : **trm_scope**.

Notation "'if' t0 'then' t1 'else' t2" :=

(**trm_if** *t0* *t1* *t2*)
(in custom **trm** at level 69,
t0 custom **trm** at level 99,
t1 custom **trm** at level 99,
t2 custom **trm** at level 99,
left associativity,
format "[v '[' 'if' *t0* 'then' ']' '/ '[' *t1* ']' '/ 'else' ']' '[' *t2* ']' ']'") : **trm_scope**.

Notation "'if' t0 'then' t1 'end'" :=

(**trm_if** *t0* *t1* (**trm_val** **val_unit**))
(in custom **trm** at level 69,
t0 custom **trm** at level 99,
t1 custom **trm** at level 99,
left associativity,
format "[v '[' 'if' *t0* 'then' ']' '/ '[' *t1* ']' '/ 'end' ']'") : **trm_scope**.

Notation "t1 ';' t2" :=

```

(trm_seq t1 t2)
  (in custom trm at level 68,
   t2 custom trm at level 99,
   right associativity,
   format "[v' [ t1 '] ';' '/ [ t2 '] ']") : trm_scope.

Notation "'let' x '=' t1 'in' t2" :=
  (trm_let x t1 t2)
  (in custom trm at level 69,
   x at level 0,
   t1 custom trm at level 99,
   t2 custom trm at level 99,
   right associativity,
   format "[v' [ 'let' x '=' t1 'in' ']' '/ [ t2 '] ']") : trm_scope.

Notation "'fix' f x1 '=>' t" :=
  (val_fix f x1 t)
  (in custom trm at level 69,
   f, x1 at level 0,
   t custom trm at level 99,
   format "'fix' f x1 '=>' t") : val_scope.

Notation "'fix_-' f x1 '=>' t" :=
  (trm_fix f x1 t)
  (in custom trm at level 69,
   f, x1 at level 0,
   t custom trm at level 99,
   format "'fix_-' f x1 '=>' t") : trm_scope.

Notation "'fun' x1 '=>' t" :=
  (val_fun x1 t)
  (in custom trm at level 69,
   x1 at level 0,
   t custom trm at level 99,
   format "'fun' x1 '=>' t") : val_scope.

Notation "'fun_-' x1 '=>' t" :=
  (trm_fun x1 t)
  (in custom trm at level 69,
   x1 at level 0,
   t custom trm at level 99,
   format "'fun_-' x1 '=>' t") : trm_scope.

Notation "()" :=
  (trm_val val_unit)
  (in custom trm at level 0) : trm_scope.

Notation "()" :=

```

(val_unit)
(at level 0) : val_scope.

Notation for Primitive Operations.

Notation "'ref'" :=
(trm_val (val_prim val_ref))
(in custom trm at level 0) : trm_scope.

Notation "'free'" :=
(trm_val (val_prim val_free))
(in custom trm at level 0) : trm_scope.

Notation "'not'" :=
(trm_val (val_prim val_neg))
(in custom trm at level 0) : trm_scope.

Notation "!" t" :=
(val_get t)
(in custom trm at level 67,
t custom trm at level 99) : trm_scope.

Notation "t1 := t2" :=
(val_set t1 t2)
(in custom trm at level 67) : trm_scope.

Notation "t1 + t2" :=
(val_add t1 t2)
(in custom trm at level 58) : trm_scope.

Notation "'-' t" :=
(val_opp t)
(in custom trm at level 57,
t custom trm at level 99) : trm_scope.

Notation "t1 - t2" :=
(val_sub t1 t2)
(in custom trm at level 58) : trm_scope.

Notation "t1 * t2" :=
(val_mul t1 t2)
(in custom trm at level 57) : trm_scope.

Notation "t1 / t2" :=
(val_div t1 t2)
(in custom trm at level 57) : trm_scope.

Notation "t1 'mod' t2" :=
(val_div t1 t2)
(in custom trm at level 57) : trm_scope.

Notation "t1 = t2" :=

```

(val_eq t1 t2)
  (in custom trm at level 58) : trm_scope.

Notation "t1 <> t2" :=
  (val_neq t1 t2)
  (in custom trm at level 58) : trm_scope.

Notation "t1 <= t2" :=
  (val_le t1 t2)
  (in custom trm at level 60) : trm_scope.

Notation "t1 < t2" :=
  (val_lt t1 t2)
  (in custom trm at level 60) : trm_scope.

Notation "t1 >= t2" :=
  (val_ge t1 t2)
  (in custom trm at level 60) : trm_scope.

Notation "t1 > t2" :=
  (val_gt t1 t2)
  (in custom trm at level 60) : trm_scope.

```

35.8.5 Scopes, Coercions and Notations for Concrete Programs

```

Module PROGRAMSYNTAX.

Export NotationForVariables.

Module VARS := DEFINITIONSFORVARIABLES.

Close Scope fmap_scope.
Open Scope string_scope.
Open Scope val_scope.
Open Scope trm_scope.
Open Scope wp_scope.
Coercion string_to_var (x:string) : var := x.

End PROGRAMSYNTAX.

```

35.9 Bonus

35.9.1 Disjunction: Definition and Properties of hor

```

Definition hor (H1 H2 : hprop) : hprop :=
  \exists (b:bool), if b then H1 else H2.

Lemma hor_sym : \forall H1 H2,

```

```
hor H1 H2 = hor H2 H1.
```

Proof using.

```
intros. unfold hor. applys himpl_antisym.  
{ applys himpl_hexists_l. intros b.  
  applys himpl_hexists_r (neg b). destruct $\times$  b. }  
{ applys himpl_hexists_l. intros b.  
  applys himpl_hexists_r (neg b). destruct $\times$  b. }
```

Qed.

```
Lemma himpl_hor_r_r :  $\forall H1 H2,$   
   $H1 \Rightarrow\!\!> \text{hor } H1 H2.$ 
```

Proof using. intros. unfolds hor. $\exists \times \text{true}$. Qed.

```
Lemma himpl_hor_r_l :  $\forall H1 H2,$   
   $H2 \Rightarrow\!\!> \text{hor } H1 H2.$ 
```

Proof using. intros. unfolds hor. $\exists \times \text{false}$. Qed.

```
Lemma himpl_hor_l :  $\forall H1 H2 H3,$   
   $H1 \Rightarrow\!\!> H3 \rightarrow$   
   $H2 \Rightarrow\!\!> H3 \rightarrow$   
   $\text{hor } H1 H2 \Rightarrow\!\!> H3.$ 
```

Proof using.

```
introv M1 M2. unfolds hor. applys himpl_hexists_l. intros b. case_if $\times$ .
```

Qed.

```
Lemma triple_hor :  $\forall t H1 H2 Q,$   
  triple t H1 Q  $\rightarrow$   
  triple t H2 Q  $\rightarrow$   
  triple t ( $\text{hor } H1 H2$ ) Q.
```

Proof using.

```
introv M1 M2. unfold hor. applys triple_hexists.  
  intros b. destruct $\times$  b.
```

Qed.

35.9.2 Conjunction: Definition and Properties of hand

```
Definition hand (H1 H2 : hprop) : hprop :=  
   $\lambda b:\text{bool}, \text{if } b \text{ then } H1 \text{ else } H2.$ 
```

```
Lemma hand_sym :  $\forall H1 H2,$   
  hand H1 H2 = hand H2 H1.
```

Proof using.

```
intros. unfold hand. applys himpl_antisym.  
{ applys himpl_hforall_r. intros b.  
  applys himpl_hforall_l (neg b). destruct $\times$  b. }  
{ applys himpl_hforall_r. intros b.
```

```

    applys himpl_hforall_l (neg b). destruct× b. }

Qed.

Lemma himpl_hand_l_r : ∀ H1 H2,
  hand H1 H2 ==> H1.
Proof using. intros. unfolds hand. applys× himpl_hforall_l true. Qed.

Lemma himpl_hand_l_l : ∀ H1 H2,
  hand H1 H2 ==> H2.
Proof using. intros. unfolds hand. applys× himpl_hforall_l false. Qed.

Lemma himpl_hand_r : ∀ H1 H2 H3,
  H1 ==> H2 →
  H1 ==> H3 →
  H1 ==> hand H2 H3.

Proof using.
  introv M1 M2. unfold hand. applys himpl_hforall_r. intros b. case_if×.

Qed.

Lemma triple_hand_l : ∀ t H1 H2 Q,
  triple t H1 Q →
  triple t (hand H1 H2) Q.

Proof using. introv M1. unfold hand. applys¬ triple_hforall true. Qed.

Lemma triple_hand_r : ∀ t H1 H2 Q,
  triple t H2 Q →
  triple t (hand H1 H2) Q.

Proof using. introv M1. unfold hand. applys¬ triple_hforall false. Qed.

```

35.9.3 Treatment of Functions of 2 and 3 Arguments

As explained in chapter **Struct**, there are different ways to support functions of several arguments: curried functions, n-ary functions, or functions expecting a tuple as argument.

For simplicity, we here follow the approach based on curried function, specialized for arity 2 and 3. It is possible to state arity-generic definitions and lemmas, but the definitions become much more technical.

From an engineering point of view, it is easier and more efficient to follow the approach using n-ary functions, as the CFML tool does.

Syntax for Functions of 2 or 3 Arguments.

```

Notation "'fun' x1 x2 '=>' t :=
  (val_fun x1 (trm_fun x2 t))
  (in custom trm at level 69,
  x1, x2 at level 0,
  format "'fun' x1 x2 '=>' t") : val_scope.

```

```

Notation "'fix' f x1 x2 '=>' t :=
  (val_fix f x1 (trm_fun x2 t))
  (in custom trm at level 69,
   f, x1, x2 at level 0,
   format "'fix' f x1 x2 '=>' t") : val_scope.

Notation "'fun_-' x1 x2 '=>' t :=
  (trm_fun x1 (trm_fun x2 t))
  (in custom trm at level 69,
   x1, x2 at level 0,
   format "'fun_-' x1 x2 '=>' t") : trm_scope.

Notation "'fix_-' f x1 x2 '=>' t :=
  (trm_fix f x1 (trm_fun x2 t))
  (in custom trm at level 69,
   f, x1, x2 at level 0,
   format "'fix_-' f x1 x2 '=>' t") : trm_scope.

Notation "'fun' x1 x2 x3 '=>' t :=
  (val_fun x1 (trm_fun x2 (trm_fun x3 t)))
  (in custom trm at level 69,
   x1, x2, x3 at level 0,
   format "'fun' x1 x2 x3 '=>' t") : val_scope.

Notation "'fix' f x1 x2 x3 '=>' t :=
  (val_fix f x1 (trm_fun x2 (trm_fun x3 t)))
  (in custom trm at level 69,
   f, x1, x2, x3 at level 0,
   format "'fix' f x1 x2 x3 '=>' t") : val_scope.

Notation "'fun_-' x1 x2 x3 '=>' t :=
  (trm_fun x1 (trm_fun x2 (trm_fun x3 t)))
  (in custom trm at level 69,
   x1, x2, x3 at level 0, format "'fun_-' x1 x2 x3 '=>' t") : trm_scope.

Notation "'fix_-' f x1 x2 x3 '=>' t :=
  (trm_fix f x1 (trm_fun x2 (trm_fun x3 t)))
  (in custom trm at level 69,
   f, x1, x2, x3 at level 0, format "'fix_-' f x1 x2 x3 '=>' t") : trm_scope.

```

Evaluation Rules for Applications to 2 or 3 Arguments.

eval_like judgment for applications to several arguments.

```

Lemma eval_like_app_fun2 : ∀ v0 v1 v2 x1 x2 t1,
  v0 = val_fun x1 (trm_fun x2 t1) →
  x1 ≠ x2 →
  eval_like (subst x2 v2 (subst x1 v1 t1)) (v0 v1 v2).

```

Proof using.

```
intros E N. intros R. applys× eval_app_args.  
{ applys eval_app_fun E. simpl. rewrite var_eq_spec. case_if. applys eval_fun. }  
{ applys× eval_val. }  
{ applys× eval_app_fun. }
```

Qed.

Lemma eval_like_app_fix2 : $\forall v_0 v_1 v_2 f x_1 x_2 t_1,$
 $v_0 = \text{val_fix } f x_1 (\text{trm_fun } x_2 t_1) \rightarrow$
 $(x_1 \neq x_2 \wedge f \neq x_2) \rightarrow$
 $\text{eval_like} (\text{subst } x_2 v_2 (\text{subst } x_1 v_1 (\text{subst } f v_0 t_1))) (v_0 v_1 v_2).$

Proof using.

```
intros E (N1&N2). intros R. applys× eval_app_args.  
{ applys eval_app_fix E. simpl. do 2 (rewrite var_eq_spec; case_if). applys eval_fun. }  
{ applys× eval_val. }  
{ applys× eval_app_fun. }
```

Qed.

Lemma eval_like_app_fun3 : $\forall v_0 v_1 v_2 v_3 x_1 x_2 x_3 t_1,$
 $v_0 = \text{val_fun } x_1 (\text{trm_fun } x_2 (\text{trm_fun } x_3 t_1)) \rightarrow$
 $(x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3) \rightarrow$
 $\text{eval_like} (\text{subst } x_3 v_3 (\text{subst } x_2 v_2 (\text{subst } x_1 v_1 t_1))) (v_0 v_1 v_2 v_3).$

Proof using.

```
intros E (N1&N2&N3). intros R. applys× eval_app_args.  
{ applys× eval_like_app_fun2 E. simpl. do 2 (rewrite var_eq_spec; case_if). applys eval_fun. }  
{ applys eval_val. }  
{ applys× eval_app_fun. }
```

Qed.

Lemma eval_like_app_fix3 : $\forall v_0 v_1 v_2 v_3 f x_1 x_2 x_3 t_1,$
 $v_0 = \text{val_fix } f x_1 (\text{trm_fun } x_2 (\text{trm_fun } x_3 t_1)) \rightarrow$
 $(x_1 \neq x_2 \wedge f \neq x_2 \wedge f \neq x_3 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3) \rightarrow$
 $\text{eval_like} (\text{subst } x_3 v_3 (\text{subst } x_2 v_2 (\text{subst } x_1 v_1 (\text{subst } f v_0 t_1)))) (v_0 v_1 v_2 v_3).$

Proof using.

```
intros E (N1&N2&N3&N4&N5). intros R. applys× eval_app_args.  
{ applys× eval_like_app_fix2 E. simpl. do 3 (rewrite var_eq_spec; case_if). applys eval_fun. }  
{ applys eval_val. }  
{ applys× eval_app_fun. }
```

Qed.

Reasoning Rules for Applications to 2 or 3 Arguments

Weakest preconditions for applications to several arguments.

Lemma wp_app_fun2 : $\forall x_1 x_2 v_0 v_1 v_2 t_1 Q,$
 $v_0 = \text{val_fun } x_1 (\text{trm_fun } x_2 t_1) \rightarrow$
 $x_1 \neq x_2 \rightarrow$
 $\wp(\text{subst } x_2 v_2 (\text{subst } x_1 v_1 t_1)) Q ==> \wp(\text{trm_app } v_0 v_1 v_2) Q.$

Proof using. introv EQ1 N. applys wp_eval_like. applys× eval_like_app_fun2. Qed.

Lemma wp_app_fix2 : $\forall f x_1 x_2 v_0 v_1 v_2 t_1 Q,$
 $v_0 = \text{val_fix } f x_1 (\text{trm_fun } x_2 t_1) \rightarrow$
 $(x_1 \neq x_2 \wedge f \neq x_2) \rightarrow$
 $\wp(\text{subst } x_2 v_2 (\text{subst } x_1 v_1 (\text{subst } f v_0 t_1))) Q ==> \wp(\text{trm_app } v_0 v_1 v_2) Q.$

Proof using. introv EQ1 N. applys wp_eval_like. applys× eval_like_app_fix2. Qed.

Lemma wp_app_fun3 : $\forall x_1 x_2 x_3 v_0 v_1 v_2 v_3 t_1 Q,$
 $v_0 = \text{val_fun } x_1 (\text{trm_fun } x_2 (\text{trm_fun } x_3 t_1)) \rightarrow$
 $(x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3) \rightarrow$
 $\wp(\text{subst } x_3 v_3 (\text{subst } x_2 v_2 (\text{subst } x_1 v_1 t_1))) Q ==> \wp(\text{trm_app } v_0 v_1 v_2 v_3) Q.$

Proof using. introv EQ1 N. applys wp_eval_like. applys× eval_like_app_fun3. Qed.

Lemma wp_app_fix3 : $\forall f x_1 x_2 x_3 v_0 v_1 v_2 v_3 t_1 Q,$
 $v_0 = \text{val_fix } f x_1 (\text{trm_fun } x_2 (\text{trm_fun } x_3 t_1)) \rightarrow$
 $(x_1 \neq x_2 \wedge f \neq x_2 \wedge f \neq x_3 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3) \rightarrow$
 $\wp(\text{subst } x_3 v_3 (\text{subst } x_2 v_2 (\text{subst } x_1 v_1 (\text{subst } f v_0 t_1)))) Q ==> \wp(\text{trm_app } v_0 v_1 v_2 v_3) Q.$

Proof using. introv EQ1 N. applys wp_eval_like. applys× eval_like_app_fix3. Qed.

Generalization of *xwp* to Handle Functions of Two Arguments

Generalization of *xwp* to handle functions of arity 2 and 3, and to handle weakening of an existing specification.

Lemma xwp_lemma_fun2 : $\forall v_0 v_1 v_2 x_1 x_2 t H Q,$
 $v_0 = \text{val_fun } x_1 (\text{trm_fun } x_2 t) \rightarrow$
 $\text{var_eq } x_1 x_2 = \text{false} \rightarrow$
 $H ==> \text{wpgen } ((x_1, v_1) :: (x_2, v_2) :: \text{nil}) t Q \rightarrow$
 $\text{triple } (v_0 v_1 v_2) H Q.$

Proof using.

```
introv M1 N M2. rewrite var_eq_spec in N. rew_bool_eq in *.
rewrite ← wp_equiv. xchange M2.
xchange (» wp_gen_sound (((x1, v1) :: nil) ++ ((x2, v2) :: nil)) t Q).
rewrite isubst_app. do 2 rewrite ← subst_eq_isubst_one.
applys× wp_app_fun2.
```

Qed.

Lemma xwp_lemma_fix2 : $\forall f v_0 v_1 v_2 x_1 x_2 t H Q,$
 $v_0 = \text{val_fix } f x_1 (\text{trm_fun } x_2 t) \rightarrow$
 $(\text{var_eq } x_1 x_2 = \text{false} \wedge \text{var_eq } f x_2 = \text{false}) \rightarrow$

$H \implies \text{wp_gen}((f, v_0) :: (x_1, v_1) :: (x_2, v_2) :: \text{nil}) t Q \rightarrow$
 $\text{triple}(v_0 v_1 v_2) H Q.$

Proof using.

```
intros M1 N M2. repeat rewrite var_eq_spec in N. rew_bool_eq in *.
rewrite ← wp_equiv. xchange M2.
xchange (» wp_gen_sound(((f, v0) :: nil) ++ ((x1, v1) :: nil) ++ ((x2, v2) :: nil)) t Q).
do 2 rewrite isubst_app. do 3 rewrite ← subst_eq_isubst_one.
applys× wp_app_fix2.
```

Qed.

Lemma xwp_lemma_fun3 : $\forall v_0 v_1 v_2 v_3 x_1 x_2 x_3 t H Q,$
 $v_0 = \text{val_fun } x_1 (\text{trm_fun } x_2 (\text{trm_fun } x_3 t)) \rightarrow$
 $(\text{var_eq } x_1 x_2 = \text{false} \wedge \text{var_eq } x_1 x_3 = \text{false} \wedge \text{var_eq } x_2 x_3 = \text{false}) \rightarrow$
 $H \implies \text{wp_gen}((x_1, v_1) :: (x_2, v_2) :: (x_3, v_3) :: \text{nil}) t Q \rightarrow$
 $\text{triple}(v_0 v_1 v_2 v_3) H Q.$

Proof using.

```
intros M1 N M2. repeat rewrite var_eq_spec in N. rew_bool_eq in *.
rewrite ← wp_equiv. xchange M2.
xchange (» wp_gen_sound(((x1, v1) :: nil) ++ ((x2, v2) :: nil) ++ ((x3, v3) :: nil)) t Q).
do 2 rewrite isubst_app. do 3 rewrite ← subst_eq_isubst_one.
applys× wp_app_fun3.
```

Qed.

Lemma xwp_lemma_fix3 : $\forall f v_0 v_1 v_2 v_3 x_1 x_2 x_3 t H Q,$
 $v_0 = \text{val_fix } f x_1 (\text{trm_fun } x_2 (\text{trm_fun } x_3 t)) \rightarrow$
 $(\text{var_eq } x_1 x_2 = \text{false} \wedge \text{var_eq } f x_2 = \text{false} \wedge \text{var_eq } f x_3 = \text{false}$
 $\wedge \text{var_eq } x_1 x_3 = \text{false} \wedge \text{var_eq } x_2 x_3 = \text{false}) \rightarrow$
 $H \implies \text{wp_gen}((f, v_0) :: (x_1, v_1) :: (x_2, v_2) :: (x_3, v_3) :: \text{nil}) t Q \rightarrow$
 $\text{triple}(v_0 v_1 v_2 v_3) H Q.$

Proof using.

```
intros M1 N M2. repeat rewrite var_eq_spec in N. rew_bool_eq in *.
rewrite ← wp_equiv. xchange M2.
xchange (» wp_gen_sound(((f, v0) :: nil) ++ ((x1, v1) :: nil) ++ ((x2, v2) :: nil) ++ ((x3, v3) :: nil)) t Q).
do 3 rewrite isubst_app. do 4 rewrite ← subst_eq_isubst_one.
applys× wp_app_fix3.
```

Qed.

Tactic Notation "xwp" :=

```
intros;
first [ applys xwp_lemma_fun; [ reflexivity | ]
| applys xwp_lemma_fix; [ reflexivity | ]
| applys xwp_lemma_fun2; [ reflexivity | reflexivity | ]
| applys xwp_lemma_fix2; [ reflexivity | splits; reflexivity | ]
| applys xwp_lemma_fun3; [ reflexivity | splits; reflexivity | ]
```

```
| applys xwp_lemma_fix3; [ reflexivity | splits; reflexivity | ] ];
xwp_simpl.
```

35.9.4 Bonus: Triples for Applications to Several Arguments

Triples for applications to 2 arguments. Similar rules can be stated and proved for 3 arguments. These rules are not needed for the verification framework.

```
Lemma triple_app_fun2 : ∀ v0 v1 v2 x1 x2 t1 H Q,
  v0 = val_fun x1 (trm_fun x2 t1) →
  x1 ≠ x2 →
  triple (subst x2 v2 (subst x1 v1 t1)) H Q →
  triple (v0 v1 v2) H Q.
```

Proof using.

```
introv E N M1. applys triple_eval_like M1. applys× eval_like_app_fun2.
```

Qed.

```
Lemma triple_app_fix2 : ∀ f x1 x2 v0 v1 v2 t1 H Q,
  v0 = val_fix f x1 (trm_fun x2 t1) →
  (x1 ≠ x2 ∧ f ≠ x2) →
  triple (subst x2 v2 (subst x1 v1 (subst f v0 t1))) H Q →
  triple (v0 v1 v2) H Q.
```

Proof using.

```
introv E N M1. applys triple_eval_like M1. applys× eval_like_app_fix2.
```

Qed.

35.9.5 Specification of Record Operations

The chapter Struct shows how to these specifications may be realized.

Implicit Types $k : \text{nat}$.

Representation of Records

A field name is implemented as a natural number

Definition field : Type := **nat**.

A record field is described as the pair of a field and a value stored in that field.

Definition hrecord_field : Type := (field × **val**).

A record consists of a list of fields.

Definition hrecord_fields : Type := **list** hrecord_field.

Implicit Types $kvs : \text{hrecord_fields}$.

Record fields syntax, e.g., ‘{ head := x; tail := q }’.

Notation “{ k1 := v1 }” :=

```

((k1 , (v1:val))::nil)
(at level 0, k1 at level 0, only parsing)
: val_scope.

Notation "{ k1 := v1 ; k2 := v2 }" :=
((k1 , (v1:val))::(k2 , (v2:val))::nil)
(at level 0, k1, k2 at level 0, only parsing)
: val_scope.

Notation "{ k1 := v1 ; k2 := v2 ; k3 := v3 }" :=
((k1 , (v1:val))::(k2 , (v2:val))::(k3 , (v3:val))::nil)
(at level 0, k1, k2, k3 at level 0, only parsing)
: val_scope.

Notation "{ k1 := v1 }" :=
((k1 , v1)::nil)
(at level 0, k1 at level 0, only printing)
: val_scope.

Notation "{ k1 := v1 ; k2 := v2 }" :=
((k1 , v1)::(k2 , v2)::nil)
(at level 0, k1, k2 at level 0, only printing)
: val_scope.

Notation "{ k1 := v1 ; k2 := v2 ; k3 := v3 }" :=
((k1 , v1)::(k2 , v2)::(k3 , v3)::nil)
(at level 0, k1, k2, k3 at level 0, only printing)
: val_scope.

```

`Open Scope val_scope.`

`hrecord kvs p`, written $p \sim\sim> kvs$, describes a record at location p , with fields described by the list kvs .

`Parameter hrecord : ∀ (kvs:hrecord_fields) (p:loc), hprop.`

`Notation "p '~~~>' kvs" := (hrecord kvs p)`
`(at level 32) : hprop_scope.`

The heap predicate `hrecord kvs p` captures in particular the invariant that the location p is not null.

`Parameter hrecord_not_null : ∀ p kvs,`
`hrecord kvs p ==> hrecord kvs p * \[p ≠ null].`

Read Operation on Record Fields

`val_get_field k p`, written $p'.k$, reads field k from the record at location p .

`Parameter val_get_field : field → val.`

`Notation "t1 '.' k" :=`

```
(val_get_field k t1)
  (in custom trm at level 56, k at level 0, format "t1 `.` k" ) : trm_scope.
```

Generator of specifications of val_get_field.

```
Fixpoint hfields_lookup (k:field) (kvs:hrecord_fields) : option val :=
  match kvs with
  | nil => None
  | (ki, vi)::kvs' => if Nat.eq_dec k ki
    then Some vi
    else hfields_lookup k kvs'
  end.
```

Specification of val_get_field in terms of hrecord.

```
Parameter triple_get_field_hrecord : ∀ kvs p k v,
  hfields_lookup k kvs = Some v →
  triple (val_get_field k p)
  (hrecord kvs p)
  (fun r => \[r = v] \* hrecord kvs p).
```

Write Operation on Record Fields

val_set_field k p v, written Set p'.k ':= v, update the contents of the field k from the record at location p, with the value v.

Parameter val_set_field : field → val.

```
Notation "t1 `.` k ':= t2" :=
  (val_set_field k t1 t2)
  (in custom trm at level 56,
  k at level 0, format "t1 `.` k ':= t2")
  : trm_scope.
```

Generator of specifications for val_set_field.

```
Fixpoint hfields_update (k:field) (v:val) (kvs:hrecord_fields)
  : option hrecord_fields :=
  match kvs with
  | nil => None
  | (ki, vi)::kvs' => if Nat.eq_dec k ki
    then Some ((k, v)::kvs')
    else match hfields_update k v kvs' with
      | None => None
      | Some LR => Some ((ki, vi)::LR)
    end
  end.
```

Specification of val_set_field in terms of hrecord.

```

Parameter triple_set_field_hrecord : ∀ kvs kvs' k p v,
  hfields_update k v kvs = Some kvs' →
  triple (val_set_field k p v)
    (hrecord kvs p)
  (fun _ ⇒ hrecord kvs' p).

```

Allocation of Records

`val_alloc_hrecord ks` allocates a record with fields `ks`, storing uninitialized contents.

Parameter `val_alloc_hrecord` : ∀ (ks:list field), **trm**.

```

Parameter triple_alloc_hrecord : ∀ ks,
  ks = nat_seq 0 (LibListExec.length ks) →
  triple (val_alloc_hrecord ks)
    []
  (funloc p ⇒ hrecord (LibListExec.map (fun k ⇒ (k, val_uninit)) ks) p).

```

An arity-generic version of the definition of allocation combined with initialization is beyond the scope of this course. We only include constructors for arity 2 and 3.

`val_new_record_2 k1 k2 v1 v2`, written ‘{ k1 := v1 ; k2 := v2 }’, allocates a record with two fields and initialize the fields.

Parameter `val_new_hrecord_2` : ∀ (k1:field) (k2:field), **val**.

```

Notation "{'{ k1 := v1 ; k2 := v2 }'" :=
  (val_new_hrecord_2 k1 k2 v1 v2)
  (in custom trm at level 65,
  k1, k2 at level 0,
  v1, v2 at level 65) : trm_scope.

```

Open Scope `trm_scope`.

```

Parameter triple_new_hrecord_2 : ∀ k1 k2 v1 v2,
  k1 = 0%nat →
  k2 = 1%nat →
  triple <{'{ k1 := v1 ; k2 := v2 }'}>
    []
  (funloc p ⇒ p ~~~> {'{ k1 := v1 ; k2 := v2 }'}).

```

Hint Resolve `val_new_hrecord_2` : `triple`.

`val_new_record_3 k1 k2 k3 v1 v2 v3`, written ‘{ k1 := v1 ; k2 := v2 ; k3 := v3 }’, allocates a record with three fields and initialize the fields.

Parameter `val_new_hrecord_3` : ∀ (k1:field) (k2:field) (k3:field), **val**.

```

Notation "{'{ k1 := v1 ; k2 := v2 ; k3 := v3 }'" :=
  (val_new_hrecord_3 k1 k2 k3 v1 v2 v3)
  (in custom trm at level 65,
  k1, k2, k3 at level 0,
  v1, v2, v3 at level 65) : trm_scope.

```

```

 $v1, v2, v3$  at level 65) : trm_scope.
Parameter triple_new_hrecord_3 :  $\forall k1 k2 k3 v1 v2 v3,$ 
   $k1 = 0\%nat \rightarrow$ 
   $k2 = 1\%nat \rightarrow$ 
   $k3 = 2\%nat \rightarrow$ 
  triple <{ '{  $k1 := v1 ; k2 := v2 ; k3 := v3$  } }>
     $\lambda []$ 
  (funloc p  $\Rightarrow p$   $\sim\sim\sim$  '{  $k1 := v1 ; k2 := v2 ; k3 := v3$  }).
Hint Resolve val_new_hrecord_3 : triple.

```

Deallocation of Records

val_dealloc_hrecord p, written *delete p*, deallocates the record at location *p*.

```
Parameter val_dealloc_hrecord : val.
```

```
Notation "'delete'" :=
  (trm_val val_dealloc_hrecord)
  (in custom trm at level 0) : trm_scope.
```

```
Parameter triple_dealloc_hrecord :  $\forall kvs p,$ 
  triple (val_dealloc_hrecord p)
    (hrecord kvs p)
    (fun _  $\Rightarrow \lambda []$ ).
```

```
Hint Resolve triple_dealloc_hrecord : triple.
```

Extending *xapp* to Support Record Access Operations

The tactic *xapp* is refined to automatically invoke the lemmas *triple_get_field_hrecord* and *triple_set_field_hrecord*.

```
Parameter xapp_get_field_lemma :  $\forall H k p Q,$ 
   $H ==> \exists kvs, (\text{hrecord } kvs p) \ast$ 
    match hfields_lookup k kvs with
    | None  $\Rightarrow \lambda [\text{False}]$ 
    | Some v  $\Rightarrow ((\text{fun } r \Rightarrow \lambda [r = v] \ast \text{hrecord } kvs p) \text{ -* protect } Q)$  end  $\rightarrow$ 
   $H ==> \text{wp } (\text{val_get_field } k p) Q.$ 
```

```
Parameter xapp_set_field_lemma :  $\forall H k p v Q,$ 
   $H ==> \exists kvs, (\text{hrecord } kvs p) \ast$ 
    match hfields_update k v kvs with
    | None  $\Rightarrow \lambda [\text{False}]$ 
    | Some kvs'  $\Rightarrow ((\text{fun } _ \Rightarrow \text{hrecord } kvs' p) \text{ -* protect } Q)$  end  $\rightarrow$ 
   $H ==> \text{wp } (\text{val_set_field } k p v) Q.$ 
```

```
Ltac xapp_nosubst_for_records tt ::=
```

```

first [ applys xapp_set_field_lemma; ximpl; simpl; xapp_simpl
      | applys xapp_get_field_lemma; ximpl; simpl; xapp_simpl ].
```

Extending *ximpl* to Simplify Record Equalities

fequals-fields simplifies equalities between values of types *hrecord_fields*, i.e. list of pairs of field names and values.

```

Ltac f>equals-fields :=
  match goal with
  |  $\vdash \text{nil} = \text{nil}$   $\Rightarrow$  reflexivity
  |  $\vdash \text{cons } \dots = \text{cons } \dots \Rightarrow$  applys args_eq_2; [f>equals | f>equals-fields] 
  end.
```

At this point, the tactic *ximpl* is refined to take into account simplifications of predicates of the form $p \sim\sim> kvs$. The idea is to find a matching predicate of the form $p \sim\sim> kvs'$ on the right-hand side of the entailment, then to simplify the list equality $kvs = kvs'$.

```

Ltac ximpl_hook_hrecord p :=
  ximpl_pick_st ltac:(fun H' =>
    match H' with hrecord ?kvs' p  $\Rightarrow$ 
      constr:(true) end;
    apply ximpl_lr_cancel_eq;
    [f_equal; first [ reflexivity | try f>equals-fields ] | ].
```

```

Ltac ximpl_hook H ::=
  match H with
  | hsingle ?p ?v  $\Rightarrow$  ximpl_hook_hssingle p
  | hrecord ?kvs ?p  $\Rightarrow$  ximpl_hook_hrecord p
  end.
```

35.10 Demo Programs

Module DEMOPROGRAMS.

Export *ProgramSyntax*.

Definition and Verification of *incr*.

Here is an implementation of the increment function, written in A-normal form.

OCaml:

```
let incr p = let n = !p in let m = n + 1 in p := m
```

The notation '*p*' stands for ("x":var). It is defined in *LibSepVar.v*.

```

Definition incr : val :=
<{ fun 'p =>
  let 'n = ! 'p in
```

```
let 'm = 'n + 1 in
'p := 'm }>.
```

Here is the Separation Logic triple specifying increment. And the proof follows. Note that the script contains explicit references to the specification lemmas of the functions being called (e.g. `triple_get` for the `get` operation). The actual CFML2 setup is able to automatically infer those names.

```
Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (incr p)
  (p ~~> n)
  (fun _ ⇒ (p ~~> (n+1))).
```

Proof using.

```
xwp. xapp. xapp. xapp. xsimpl ×.
```

Qed.

We register this specification so that it may be automatically invoked in further examples.

Hint Resolve `triple_incr` : `triple`.

Alternative definition using variable names without quotes, obtained by importing the module `Vars` from `LibSepVar.v`.

Module Export DEF_INCR. **Import** Vars.

```
Definition incr' : val :=
<{ fun p ⇒
  let n = ! p in
  let m = n + 1 in
  p := m }>.
```

End DEF_INCR.

```
Lemma triple_incr' : ∀ (p:loc) (n:int),
  triple (incr' p)
  (p ~~> n)
  (fun _ ⇒ (p ~~> (n+1))).
```

Proof using.

```
xwp. xapp. xapp. xapp. xsimpl ×.
```

Qed.

35.10.1 The Decrement Function

```
Definition decr : val :=
<{ fun 'p ⇒
  let 'n = ! 'p in
  let 'm = 'n - 1 in
  'p := 'm }>.
```

```
Module Export DEF_DECL. Import Vars.
```

```
Definition decr : val :=  
<{ fun p =>  
    let n = !p in  
    let m = n - 1 in  
    p := m }>.
```

```
End DEF_DECL.
```

```
Lemma triple_decr : ∀ (p:loc) (n:int),  
  triple (trm_app decr p)  
  (p ~~> n)  
  (fun _ => p ~~> (n-1)).
```

Proof using.

```
xwp. xapp. xapp. xapp. xsimpl ×.
```

Qed.

```
Hint Resolve triple_decr : triple.
```

Definition and Verification of mysucc.

Another example, involving a call to *incr*.

```
Module Export DEF_MYSUCC. Import Vars.
```

```
Definition mysucc : val :=  
<{ fun n =>  
    let r = ref n in  
    incr r;  
    let x = !r in  
    free r;  
    x }>.
```

```
End DEF_MYSUCC.
```

```
Lemma triple_mysucc : ∀ n,  
  triple (trm_app mysucc n)  
  []  
  (fun v => \[v = n+1]).
```

Proof using.

```
xwp. xapp. intros r. xapp. xapp. xapp. xval. xsimpl. auto.
```

Qed.

Definition and Verification of myrec.

Another example, involving a function involving 3 arguments, as well as record operations.

OCaml:

```
let myrec r n1 n2 = r.myfield := r.myfield + n1 + n2
```

```

Definition myfield : field := 0%nat.

Module Export DEF_MYREC. Import Vars.

Definition myrec : val :=
<{ fun p n1 n2 =>
  let n = (p'.myfield) in
  let m1 = n + n1 in
  let m2 = m1 + n2 in
  p'.myfield := m2 }>.

Lemma triple_myrec : ∀ (p:loc) (n n1 n2:int),
  triple (myrec p n1 n2)
  (p ~~> '{ myfield := n})
  (fun _ => p ~~> '{ myfield := (n+n1+n2) }).

```

Proof using.

xwp. xapp. xapp. xapp. xapp. xsimpl.

Qed.

End DEF_MYREC.

Definition and Verification of myfun.

Another example involving a local function definition.

Module Export DEF_MYFUN. Import Vars.

```

Definition myfun : val :=
<{ fun p =>
  let f = (fun_ u => incr p) in
  f();
  f() }>.

```

End DEF_MYFUN.

```

Lemma triple_myfun : ∀ (p:loc) (n:int),
  triple (myfun p)
  (p ~~> n)
  (fun _ => p ~~> (n+2)).

```

Proof using.

```

xwp.
xfun (fun (f:val) => ∀ (m:int),
  triple (f())
  (p ~~> m)
  (fun _ => p ~~> (m+1))); intros f Hf.
{ intros. applys Hf. clear Hf. xapp. xsimpl. }
xapp.
xapp.
replace (n+1+1) with (n+2); [|math]. xsimpl.

```

Qed.

End DEMOPROGRAMS.

Chapter 36

Library `SLF.Preface`

36.1 Preface: Introduction to the Course

36.2 Welcome

This electronic book is Volume 6 of the *Software Foundations* series, which presents the mathematical underpinnings of reliable software.

In this volume, you will learn about the foundations of Separation Logic, a practical approach to the modular verification of imperative programs. In particular, this volume presents the building blocks for constructing a program verification tool. It does not, however, focus on reasoning about data structures and algorithms using Separation Logic. This aspect is covered to some extent by Volume 5 of *Software Foundations*, which presents Verifiable C, a program logic and proof system for C. For OCaml programs, this aspect will be covered in a yet-to-be-written volume presenting CFML, a tool that builds upon all the techniques presented in this volume.

You are only assumed to understand the material in *Software Foundations* Volume 1 (*Logical Foundations*), and the two chapters on Hoare Logic (Hoare and Hoare2) from Software Foundations Volume 2 (*PL Foundations*). The reading of Volume 5 is not a prerequisite. The exposition here is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers.

A large fraction of the contents of this course is also written up in traditional LaTeX-style presentation, in the ICFP’20 article: *Separation Logic for Sequential Programs*, by Arthur Charguéraud. The long version of this paper is available at this link: <http://www.chargueraud.org/research/>

This paper includes, in particular, a 5-page historical survey of contributions to mechanized presentations of Separation Logic, featuring 100+ citations. For a broader survey of Separation Logic, we recommend Peter O’Hearn’s 2019 survey, which is available from: <https://dl.acm.org/doi/10.1145/3211968> – including the *supplemental material* linked near the bottom of that page.

36.3 Separation Logic

Separation Logic is a *program logic*: it enables one to establish that a program satisfies its specification. Specifications are expressed using triples of the form $\{H\} t \{Q\}$. Whereas in Hoare logic the precondition H and the postcondition Q describe the whole memory state, in Separation Logic, H and Q describe only a fragment of the memory state. This fragment includes the resources necessary to the execution of t .

A key ingredient of Separation Logic is the frame rule, which enables modular proofs. It is stated as follows.

$$\{H\} t \{Q\}$$

$$\{H \setminus^* H'\} t \{Q \setminus^* H'\}$$

The above rule asserts that if a term t executes correctly with the resources H and produces Q , then the term t admits the same behavior in a larger memory state, described by the union of H with a disjoint component H' , producing the postcondition Q extended with that same resource H' unmodified. The star symbol \setminus^* denotes the *separating conjunction* operator of Separation Logic.

Separation Logic can be exploited in three kind of tools.

- Automated proofs: the user provides only the code, and the tool locates sources of potential bugs. A good automated tool provides feedback that, most of time, is relevant.
- Semi-automated proofs: the user provides not just the code, but also specifications and invariants. The tool then leverages automated solvers (e.g., SMT solvers) to discharge proof obligations.
- Interactive proofs: the user provides not just the code and its specifications, but also a detailed proof script justifying the correctness of the code. These proofs may be worked on interactively using a proof assistant such as Coq.

The present course focuses on the third approach, that is, the integration of Separation Logic in an interactive proof assistant. This approach has been successfully put to practice throughout the world, using various proof assistants (Coq, Isabelle/HOL, HOL), targeting different languages (Assembly, C, SML, OCaml, Rust...), and for verifying various kind of programs, ranging from low-level operating system kernels to high-level data structures and algorithms.

36.4 Separation Logic in a proof assistant

The benefits of exploiting Separation Logic in a proof assistant include at least four major points:

- higher-order logic provides virtually unlimited expressiveness that enables formulating arbitrarily complex specifications and invariants;

- a proof assistant provides a unified framework to prove both the implementation details of the code and the underlying mathematical results form, e.g., results from theory or graph theory;
- proof scripts may be easily maintained to reflect on a change to the source code;
- the fact that Separation Logic is formalized in the proof assistant provides high confidence in the correctness of the tool.

Pretty much all the tools that leverage Separation Logic in a proof assistant are constructed following the same schema:

- A formalization of the syntax and semantics of the source language. This is called a *deep embedding* of the programming language.
- A definition of Separation Logic predicates as predicates from higher-order logic. This is called a *shallow embedding* of the program logic.
- A definition of Separation Logic triples as a predicate, the statements of the reasoning rules as lemmas, and the proof of these reasoning rules with respect to the semantics.
- An infrastructure that consists of lemmas, tactics and notation, allowing for the verification of concrete programs to be carried out through relatively concise proof scripts.
- Applications of this infrastructure to the verification of concrete programs.

The purpose of this course is to explain how to set up such a construction. To that end, we consider in this course the simplest possible variant of Separation Logic, and apply it to a minimalistic imperative programming language. The language essentially consists of a lambda-calculus with references. This language admits a simple semantics and avoids in particular the need to distinguish between stack variables and heap- allocated variables. Advanced chapters from the course explains how to add support for loops, records, arrays, and n-ary functions.

36.5 Several Possible Depths of Reading

The material is organized in such a way that it can be easily adapted to the amount of time that the reader is ready to invest in the course.

The course contains 13 chapters, not counting the *Preface*, *Postscript*, and *Bib* chapters. The course is organized in 3 major parts, as pictured in the roadmap diagram.

- The short curriculum includes only the 5 first chapters (ranging from chapter **Basic** to chapter **Rules**).
- The medium curriculum includes 3 additional chapters (ranging from chapter **WPsem** to chapter **Wand**).

- The full curriculum includes 5 more chapters (ranging from chapter `Partial` to chapter `Rich`).

In addition, each chapter except `Basic` is decomposed in three parts.

- The *First Pass* section presents the most important ideas only. The course is designed in such a way that it is possible to read only the *First Pass* section of every chapter. The reader may be interested in going through all these sections to get the big picture, before revisiting each chapter in more details.
- The *More Details* section presents additional material explaining in more depth the meaning and the consequences of the key results. This section also contains descriptions of the most important proofs. By default, readers would eventually read all this material.
- The *Optional Material* section typically contains the remaining proofs, as well as discussions of alternative definitions. The *Optional Material* sections are all independent from each other. They would typically be of interest to readers who want to understand every detail, readers who are seeking for a deep understanding of a particular notion, and readers who are looking for answers to specific questions.

36.6 List of Chapters

The first two chapters, namely chapters `Basic` and `Repr` give a primer on how to prove imperative programs in Separation Logic, thus focusing on the end user's perspective. All the other chapters focus on the implementor's perspective, explaining how Separation Logic is defined and how a practical verification tool can be constructed.

The list of chapters appears below. The numbering corresponds to *teaching units*: if the chapters were taught as part of a University course, one could presumably aim to cover one teaching unit per week.

- (1) `Basic`: introduction to the core features of Separation Logic, illustrated using short programs manipulating references.
- (1) `Repr`: introduction to representation predicates in Separation Logic, in particular for describing mutable lists and trees.
- (2) `Hprop`: definition of the core operators of Separation Logic, of Hoare triples, and of Separation Logic triples.
- (2) `Himpl`: definition of the entailment relation, statement and proofs of its fundamental properties, and description of the simplification tactic for entailment.

- (3) **Rules**: statement and proofs of the reasoning rules of Separation Logic, and example proofs of programs using these rules.
- (4) **WPsem**: definition of the semantic notion of weakest precondition, and statement of rules in weakest-precondition style.
- (4) **WPgen**: presentation of a function that effectively computes the weakest precondition of a term, independently of its specification.
- (5) **Wand**: introduction of the magic wand operator and of the ramified frame rule, and extension of the weakest-precondition generator for handling local function definitions.
- (5) **Affine**: description of a generalization of Separation Logic with affine heap predicates, which are useful, in particular, for handling garbage-collected programming languages.
- (6) **Struct**: specification of array and record operations, and encoding of these operations using pointer arithmetic.
- (6) **Rich**: description of the treatment of additional language constructs, including loops, assertions, and n-ary functions.
- (7) **Nondet**: definition of triples for non-deterministic programming languages.
- (7) **Partial**: definition of triples for partial correctness only, i.e., for not requiring termination proofs.

36.7 Other Distributed Files

The chapters listed above depend on a number of auxiliary files, which the reader does not need to go through, but might be interested in looking at, either by curiosity, or for checking out a specific implementation detail.

- **LibSepReference**: a long file that defines the program verification tool that is used in the first two chapters, and whose implementation is discussed throughout the other chapters. Each chapter from the course imports this module, as opposed to importing the chapters that precedes it.
- **LibSepVar**: a formalization of program variables, together with a bunch of notations for parsing variables.
- **LibSepFmap**: a formalization of finite maps, which are used for representing the memory state.

- `LibSepSimpl`: a functor that implements a powerful tactic for automatically simplifying entailments in Separation Logic.
- `LibSepMinimal`: a minimalistic formalization of a soundness proof for Separation Logic, corresponding to the definitions and proofs presented in the ICFP’20 paper mentioned earlier.
- All other `Lib` files are imports from the TLC library, which is described next.

The TLC library is a collection of general purpose theory and tactics developed over the years by Arthur Charguéraud. The TLC library is exploited in this course to streamline the presentation. TLC provides, in particular, extensions for classical logic and tactics that are particularly well-suited for meta-theory. Prior knowledge of TLC is not required, and all exercises can be completed without using TLC tactics.

The classical logic aspects of TLC are presented at the moment they appear in the course. The TLC tactics are also briefly described upon their first occurrence. Moreover, most of these tactics are presented in the chapter *UseTactics* of Software Foundations Volume 2 (*Programming Language Foundations*).

36.8 Practicalities

36.8.1 System Requirements

The *Preface* of Software Foundations Volume 1 (*Logical Foundations*) describes how to install Coq. The files you are reading have been tested with Coq version 8.13.

If you use Coq version 8.12, you will first need to patch the sources by executing the following command:

```
sed 's/name,/ident,/g' -i LibSepReference.v
```

36.8.2 Note for CoqIDE Users

CoqIDE typically works better with its *asynchronous* proof mode disabled. To load all the course files in CoqIDE, use the following command line.

```
coqide -async-proofs off -async-proofs-command-error-resilience off Basic.v &
```

36.8.3 Feedback Welcome

If you intend to use this course either in class or for self-study, the author, Arthur Charguéraud, would love to hear from you. Just knowing in which context the course has been used and how much of the course students were able to cover is very valuable information.

You can send feedback at: slf –at– chargueraud.org.

If you plan on providing any non-small amount of feedback, do not hesitate to ask the author to be added as contributor to the github repository. In particular, please do not

hesitate to improve the formulation of the English sentences throughout this volume, as the author is not a native speaker.

36.8.4 Exercises

Each chapter includes numerous exercises. The star rating scheme is described in the *Preface* of Software Foundations Volume 1 (*Logical Foundations*).

Disclaimer: the difficulty ratings currently in place are highly speculative! Your feedback is very much welcome.

Disclaimer: the auto-grading system has not been tested for this volume. If you are interested in using auto-grading for this volume, please contact the author.

36.8.5 Recommended Citation Format

If you want to refer to this volume in your own writing, please do so as follows:

```
@book {Chargueraud:SF6, author = {Arthur Charguéraud}, editor = {Benjamin C. Pierce}, title = "Separation Logic Foundations", series = "Software Foundations", volume = "6", year = "2021", publisher = "Electronic textbook", note = {Version 1.1, \URL{http://softwarefoundations}, }}
```

36.9 Thanks

The development of the technical infrastructure for the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep Specification*.

Chapter 37

Library `SLF.Basic`

37.1 Basic: Basic Proofs in Separation Logic

```
Set Implicit Arguments.
```

```
From SLF Require Import LibSepReference.
```

```
Import ProgramSyntax DemoPrograms.
```

```
Implicit Types n m : int.
```

```
Implicit Types p q : loc.
```

37.2 A First Taste

This chapter gives an overview of the basic features of Separation Logic, by means of examples. The examples are specified and verified using a Separation Logic framework whose construction is explained throughout the course.

37.2.1 Parsing of Programs

The source code of the programs considered are written within Coq, using a “custom grammar” that allows writing code that reads almost like OCaml code. For example, consider the function *incr*, which increments the contents of a mutable cell that stores an integer. In OCaml syntax, this function could be defined as:

OCaml:

```
let incr = fun p -> let n = !p in let m = n + 1 in p := m
```

In Coq, we describe the corresponding program as shown below. Observe that all variable names are prefixed with a quote symbol. This presentation avoids conflict between program variables and Coq constants. The function defined, named *incr*, admits the type **val**. This type is defined by the framework.

```
Definition incr : val :=
```

```
<{ fun 'p =>
```

```

let 'n = ! 'p in
let 'm = 'n + 1 in
'p := 'm }>.

```

There is no need to learn how to write programs in this custom syntax: source code is provided for all the programs involved in this course.

To simplify the implementation of the framework and the reasoning about programs, we make throughout the course the simplifying assumption that programs are written in “A-normal form”: all intermediate expressions must be named using a let-binding.

37.2.2 Specification of the Increment Function

The specification of *incr p* is expressed using a “Separation Logic triple”, that is, a predicate of the form *triple t H Q*. The term *t* here corresponds to the application of the function *incr* to the argument *p*. We could write this application in the form $\langle\{ \text{incr } p \} \rangle$, using the custom syntax for parsing programs. That said, we can also write the same term in the form (*incr p*), leveraging Coq’s coercion facility to reuse Coq’s syntax for application. In other words, the specification of *incr* can be written in the form *triple (incr p) H Q*.

The components *H* and *Q* correspond to the precondition and to the postcondition, which are explained next. To improve readability, we follow the convention of writing both the precondition and the postcondition on separate lines.

```

Lemma triple_incr : ∀ (p:loc) (n:int),
  triple <{ incr p }>
    (p ~~> n)
    (fun _ ⇒ (p ~~> (n+1))).

```

In the specification above, *p* denotes the “location”, that is, the address in memory, of the reference cell provided as argument to the increment function. Locations have type *loc* in the framework.

The precondition is written $p ~~> n$. This Separation Logic predicate describes a memory state in which the contents of the location *p* is the value *n*. In the present example, *n* stands for an integer value.

The behavior of the operation *incr p* consists of updating the memory state by incrementing the contents of the cell at location *p*, updating its contents to *n+1*. Thus, the memory state posterior to the increment operation is described by the predicate $p ~~> (n+1)$.

The result value returned by *incr p* is the unit value, which does not carry any useful information. In the specification of *incr*, the postcondition is of the form *fun _ ⇒ ...* to indicate that there is no need to bind a name for the result value.

The general pattern of a specification thus includes:

- Quantification of the arguments of the functions—here, the variable *p*.
- Quantification of the “ghost variables” used to describe the input state—here, the variable *n*.

- The application of the predicate `triple` to the function application `incr p`—here, the term being specified by the triple.
- The precondition describing the input state—here, the predicate $p \sim\sim > n$.
- The postcondition describing both the output value and the output state. The general pattern is `fun r => H'`, where `r` names the result and `H'` describes the final state. Here, `r` is just an underscore symbol, and the final state is described by $p \sim\sim > (n+1)$.

Note that we have to write $p \sim\sim > (n+1)$ using parentheses around $n+1$, because $p \sim\sim > n+1$ would get parsed as $(p \sim\sim > n) + 1$.

37.2.3 Verification of the Increment Function

Our next step is to prove the specification lemma `triple_incr` which specifies the behavior of the function `incr`. We conduct the proof using tactics provided by the frameworks, called the “x-tactics” because their name start with the letter “x”. These tactics include `xwp` for starting a proof, `xapp` for reasoning about a function call, and `xsimpl` for proving that a description of a state entails another one.

Proof.

`xwp` begins the verification proof. `xwp`.

The proof obligation is displayed using a custom notation of the form `PRE H CODE F POST Q`. In the `CODE` section, one should be able to somewhat recognize the body of `incr`. Indeed, if we ignore the back-ticks and perform the alpha-renaming from `v` to `n` and `v0` to `m`, the `CODE` section reads like:

`< Let n := App val_get p in Let m := App val_add n 1 in App val_set p) >`

which is somewhat similar to the original source code, but displayed using a special syntax whose meaning will be explained later on, in chapter WPgen.

The remainder of the proof performs essentially a symbolic execution of the code. At each step, one should not attempt to read the full proof obligation, but instead only look at the current state, described by the `PRE` part (here, $p \sim\sim > n$), and at the first line only of the `CODE` part, which corresponds to the next operation to reason about. Each of the operations involved here is handled using the tactic `xapp`.

First, we reason about the operation `!p` that reads into `p`; this read operation returns the value `n`. `xapp`.

Second, we reason about the addition operation `n+1`. `xapp`.

Third, we reason about the update operation `p := n+1`, thereby updating the state to $p \sim\sim > (n+1)$. `xapp`.

At this stage, the proof obligation takes the form `H1 ==> H2`. It requires us to check that the final state matches what is claimed in the postcondition. We discharge it using the tactic `xsimpl`. `xsimpl`.

Qed.

This completes the verification of the lemma *triple_incr*, which establishes a formal specification for the increment function. Before moving on to another function, we associate using the command shown below the lemma *triple_incr* with the function *incr* in a hint database called *triple*. Doing so, when we verify a function that features a call to *incr*, the *xapp* tactic will be able to automatically invoke the lemma *triple_incr*.

```
Hint Resolve triple_incr : triple.
```

The reader may be curious to know what the notation *PRE H CODE F POST Q* stands for, and what the x-tactics are doing. Everything will be explained throughout the course. This chapter and the next one focus presenting the features of Separation Logic, and on showing how x-tactics can be used to verify programs in Separation Logic.

37.2.4 A Function with a Return Value

As a second example, let us specify a function that performs simple arithmetic computations. The function, whose code appears below, expects an integer argument *n* (in \mathbb{Z}). It evaluates *a* as $n+1$, then evaluates *b* as $n-1$, and finally returns the sum *a+b*. The function thus always returns $2*n$.

```
Definition example_let : val :=  
<{ fun 'n =>  
  let 'a = 'n + 1 in  
  let 'b = 'n - 1 in  
  'a + 'b }>.
```

The specification takes the form *triple (example_let n) H (fun r => H')*, where *r*, of type **val**, denotes the output value.

The precondition *H* describes what we need to assume about the input state. For this function, we need not assume anything, hence we write \emptyset to denote the empty precondition. The program might have allocated data prior to the call to the function *example_let*, however this function will not interfere in any way with this previously-allocated data.

The postcondition describes what the function produces. More precisely, it describes in general both the output that the function returns, and the data from memory that the function has allocated, accessed, or updated. The function *example_let* does not interact with the state, thus the postcondition could be described using the empty predicate \emptyset .

Yet, if we write just *fun (r:val) => \emptyset* as postcondition, we would have said nothing about the output value *r* produced by a call *example_let*. Instead, we would like to specify that the result *r* is equal to $2*n$. To that end, we write the postcondition *fun r => $\set{r = 2*n}$* . Here, we use the predicate \set{P} , which allows to embed “pure facts”, of type **Prop** in preconditions and postconditions.

The equality *r = 2*n* actually resolves to *r = val_int (2*n)*, where *val_int* is a coercion that translates the integer value $2*n$ into the corresponding integer value, of type **val**, from the programming language. If you do not know what a coercion is, just ignore the previous sentence, and wait until chapter *Rules* to learn about coercions.

```

Lemma triple_example_let : ∀ (n:int),
  triple <{ example_let n }>
  ([] []
  (fun r ⇒ \[r = 2×n]).)

```

The proof script is quite similar to the previous one: *xwp* begins the proof, *xapp* performs symbolic execution, and *xsimpl* simplifies the entailment. Ultimately, we need to check that the expression computed, $(n + 1) + (n - 1)$, is equal to the specified result, that is, $2 \times n$. To prove this equality, we invoke the tactic *math*, which is a variant of the tactic *lia* provided by the TLC library. Recall from the preface that this course leverages TLC for enhanced definitions and tactics.

Proof.

xwp. xapp. xapp. xapp. xsimpl. math.

Qed.

From here on, we use the command **Proof using** for introducing a proof instead of writing just **Proof**. Doing so enables asynchronous proof checking, a feature that may enable faster processing of scripts. Moreover, to minimize the amount of syntactic noise in specifications, we leverage the coercion mechanism to allow writing the specified term, here `example_let n` simply surrounded with parentheses, as opposed to the heavier form `<{ example_let n }>`. (See **Rules** for details). For example, we would format the previous proof in the following form.

```

Lemma triple_example_let' : ∀ (n:int),
  triple (example_let n)
  ([] []
  (fun r ⇒ \[r = 2×n]).)

```

Proof using.

xwp. xapp. xapp. xapp. xsimpl. math.

Qed.

Exercise: 1 star, standard, especially useful (triple_quadruple) Consider the function `quadruple`, which expects an integer `n` and returns its quadruple, that is, the value $4 \times n$.

Definition `quadruple` : **val** :=

```

<{ fun 'n ⇒
  let 'm = 'n + 'n in
  'm + 'm }>.

```

Specify and verify the function `quadruple` to express that it returns $4 \times n$. Hint: follow the pattern of the previous proof.

□

Exercise: 2 stars, standard, especially useful (triple_inplace_double) Consider the function `inplace_double`, which expects a reference on an integer, reads its contents, then

updates the contents with the double of the original value.

```
Definition inplace_double : val :=
<{ fun 'p =>
  let 'n = !'p in
  let 'm = 'n + 'n in
  'p := 'm }>.
```

Specify and verify the function `inplace_double`. Hint: follow the pattern of the first example, namely `triple_incr`.

□

37.3 Separation Logic Operators

37.3.1 Increment of Two References

Consider the following function, which expects the addresses of two reference cells, and increments both of them.

```
Definition incr_two : val :=
<{ fun 'p 'q =>
  incr 'p;
  incr 'q }>.
```

The specification of this function takes the form `triple (incr_two p q) H (fun _ => H')`, where the underscore symbol denotes the result value. We do not bother binding a name for that result value because it always consists of the unit value.

The precondition describes two references cells: $p \sim\!> n$ and $q \sim\!> m$. To assert that the two cells are distinct from each other, we separate their description with the operator $\setminus*$. Thus, the precondition is $(p \sim\!> n) \setminus* (q \sim\!> m)$, or simply $p \sim\!> n \setminus* q \sim\!> m$. The operator $\setminus*$ is called the “separating conjunction” of Separation Logic. It is also known as the “star” operator.

The postcondition describes the final state in a similar way, as $p \sim\!> (n+1) \setminus* q \sim\!> (m+1)$. This predicate reflects the fact that the contents of both references gets increased by one unit.

The specification triple for `incr_two` is thus as follows. The proof follows the same pattern as in the previous examples.

```
Lemma triple_incr_two : ∀ (p q:loc) (n m:int),
  triple (incr_two p q)
  (p \sim\!> n \setminus* q \sim\!> m)
  (fun _ => p \sim\!> (n+1) \setminus* q \sim\!> (m+1)).
```

Proof using.

`xwp. xapp. xapp. xsimpl.`

Qed.

Because we will make use of the function `incr_two` later in this chapter, we register the specification `triple_incr_two` in the `triple` database.

Hint Resolve `triple_incr_two : triple`.

A quick point of vocabulary before moving on: Separation Logic expressions such as $p \sim\!> n$ or \emptyset or $H1 \ast H2$ are called “heap predicates”, because they correspond to predicates over “heaps”, i.e., predicates over memory states.

37.3.2 Aliased Arguments

The specification `triple_incr_two` stated above describes the behavior of calls to the function `incr_two` only for cases where the two arguments provided correspond to distinct reference cells. It says nothing, however, about a call of the form `incr_two p p`. Indeed, in Separation Logic, a state described by $p \sim\!> n$ cannot be matched against a state described by $p \sim\!> n \ast p \sim\!> n$, because the star operator requires its operand to correspond to disjoint pieces of state.

What happens if we nevertheless try to exploit `triple_incr_two` to reason about a call of the form `incr_two p p`, that is, with aliased arguments? Let’s find out, by considering the operation `aliased_call p`, which does execute such a call.

```
Definition aliased_call : val :=
<{ fun 'p =>
  incr_two 'p 'p }>.
```

A call to `aliased_call p` increases the contents of `p` by 2. This property can be specified as follows.

```
Lemma triple_aliased_call : ∀ (p:loc) (n:int),
  triple (aliased_call p)
  (p \sim\!> n)
  (fun _ => p \sim\!> (n+2)).
```

If we attempt the proof, we get stuck. The tactic `xapp` reports its failure by issuing a proof obligation of the form $\emptyset ==> (p \sim\!> ?m) \ast _$. This proof obligation requires us to show that, from the empty heap predicate state, one can extract a heap predicate $p \sim\!> ?m$ describing a reference at location `p` with some integer contents `?m`.

Proof using.

`xwp. xapp.`

Abort.

On the one hand, the precondition of the specification `triple_incr_two`, with $q = p$, requires providing $p \sim\!> ?n \ast p \sim\!> ?m$. On the other hand, the current state is described as $p \sim\!> n$. When trying to match the two, the internal simplification tactic `xsimpl` is able to cancel out one occurrence of $p \sim\!> n$ from both expressions, but then there remains to match the empty heap predicate \emptyset against $(p \sim\!> ?m)$. The issue here is that the specification `triple_incr_two` is specialized for the case of “non-aliased” references.

One thing we can do is to state and prove an alternative specification for the function `incr_two`, to cover the case of aliased arguments. The precondition of this alternative specification mentions a single reference, $p \sim\!> n$, and the postcondition asserts that the contents of that reference gets increased by two units. This alternative specification is stated and proved as follows.

```
Lemma triple_incr_two_aliased : ∀ (p:loc) (n:int),
  triple (incr_two p p)
  (p ∼\!> n)
  (fun _ ⇒ p ∼\!> (n+2)).
```

Proof using.

xwp. xapp. xapp. xsimpl. math.

Qed.

By exploiting the alternative specification for `incr_two`, we are able to prove the specification of the function `aliased_call`. In order to indicate to the tactic `xapp` that it should not invoke the lemma `triple_incr_two` registered for `incr_two`, but instead invoke the lemma `triple_incr_two_aliased`, we provide that lemma as argument to `xapp`. Concretely, we write `xapp triple_incr_two_aliased`.

```
Lemma triple_aliased_call : ∀ (p:loc) (n:int),
  triple (aliased_call p)
  (p ∼\!> n)
  (fun _ ⇒ p ∼\!> (n+2)).
```

Proof using.

xwp. xapp triple_incr_two_aliased. xsimpl.

Qed.

Taking a step back, it may appear somewhat disappointing that we need two different specifications for a same function, depending on whether its arguments are aliased or not. There exists advanced features of Separation Logic that allow handling the two cases through a single specification. However, for such a simple function it is easiest to just state and prove the two specifications separately.

37.3.3 A Function that Takes Two References and Increments One

Consider the following function, which expects the addresses of two reference cells and increments only the first one. What is interesting about this function is precisely the fact that it does nothing with its second argument.

```
Definition incr_first : val :=
<{ fun 'p 'q ⇒
  incr 'p }>.
```

We can specify this function by describing its input state as $p \sim\!> n \setminus^* q \sim\!> m$, and describing its output state as $p \sim\!> (n+1) \setminus^* q \sim\!> m$. Formally:

```
Lemma triple_incr_first : ∀ (p q:loc) (n m:int),
```

```

triple (incr_first p q)
  (p ~~> n \* q ~~> m)
  (fun _ => p ~~> (n+1) \* q ~~> m).

```

Proof using.

xwp. xapp. xsimpl.

Qed.

The second reference plays absolutely no role in the execution of the function. Thus, we could equally well consider a specification that mentions only the existence of the first reference.

```

Lemma triple_incr_first' : ∀ (p q:loc) (n:int),
  triple (incr_first p q)
    (p ~~> n)
    (fun _ => p ~~> (n+1)).

```

Proof using.

xwp. xapp. xsimpl.

Qed.

Interestingly, the specification `triple_incr_first`, which mentions the two references, is derivable from the specification `triple_incr_first'`, which mentions only the first reference. To prove the implication, it suffices to invoke the tactic `xapp` with argument `triple_incr_first'`.

```

Lemma triple_incr_first_derived : ∀ (p q:loc) (n m:int),
  triple (incr_first p q)
    (p ~~> n \* q ~~> m)
    (fun _ => p ~~> (n+1) \* q ~~> m).

```

Proof using.

intros. xapp triple_incr_first'. xsimpl.

Qed.

More generally, in Separation Logic, if a specification triple holds, then this triple remains valid when we add the same heap predicate to both the precondition and the postcondition. This is the “frame” principle, a key modularity feature that we’ll come back to later on in the course.

37.3.4 Transfer from one Reference to Another

Consider the `transfer` function, whose code appears below. Recall that, to simplify the implementation of the framework used in the course, we need to assign a name to every intermediate result.

```

Definition transfer : val :=
<{ fun 'p 'q =>
  let 'n = !'p in
  let 'm = !'q in
  let 's = 'n + 'm in

```

```
'p := 's;
'q := 0 }>.
```

Exercise: 1 star, standard, especially useful (triple_transfer) State and prove a lemma called *triple_transfer*, to specify the behavior of transfer $p \rightarrow q$ in the case where p and q denote two distinct references.

□

Exercise: 1 star, standard, especially useful (triple_transfer_aliased) State and prove a lemma called *triple_transfer_aliased* specifying the behavior of transfer when it is applied twice to the same argument. It should take the form triple (transfer p p) _ _.

□

37.3.5 Specification of Allocation

Consider the operation *ref v*, which allocates a memory cell with contents v . How can we specify this operation using a triple? The precondition of this triple should be the empty heap predicate, written \emptyset , because the allocation can execute in an empty state. The postcondition should assert that the output value is a pointer p , such that the final state is described by $p \rightsquigarrow v$.

It would be tempting to write the postcondition $\text{fun } p \Rightarrow p \rightsquigarrow v$. Yet, the triple would be ill-typed, because the postcondition of a triple must be a predicate over values, of type **val** in the framework, whereas here p is an address, of type **loc**. We thus need to write the postcondition in the form $\text{fun } (r:\text{val}) \Rightarrow H'$, where r denotes the result value, and somehow assert that r is the value that corresponds to the location p . This value is written $\text{val_loc } p$, where val_loc denotes the constructor that injects locations into the grammar of program values.

To formally quantify the variable p , we use the existential quantifier for heap predicates, written \exists . The correct postcondition for *ref v* is thus $\text{fun } r \Rightarrow \exists (p:\text{loc}), \exists [r = \text{val_loc } p] \forall p \rightsquigarrow v$. The complete statement of the specification appears below. It appears as a **Parameter** instead of a **Lemma**, because the semantics of the primitive operation *ref* is established in another file, directly with respect to the semantics—in **Rules**.

```
Parameter triple_ref : ∀ (v:val),
  triple <{ ref v }>
  [] []
  (fun r ⇒ ∃ p, ∃ [r = val_loc p] ∀ p �rightsquigarrow v).
```

The pattern $\text{fun } r \Rightarrow \exists p, \exists [r = \text{val_loc } p] \forall H$ occurs whenever a function returns a pointer. To improve concision for this frequent pattern, we introduce a specific notation, of the form *funloc p ⇒ H*.

Notation "'funloc' p '=>' H" :=

```
(fun r => \exists p, \[r = val_loc p] \* H)
(at level 200, p ident, format "'funloc' p => H").
```

Using this notation, the specification *triple_ref* can be reformulated more concisely, as follows.

```
Parameter triple_ref' : \forall (v:val),
  triple <\{ ref v \}>
  [] []
  (funloc p => p ~~> v).
```

The tool CFML, which leverages similar techniques as described in this course, leverages type-classes to generalize the notation *funloc* to all return types. Yet, in order to avoid technical difficulties associated with type-classes, we will not go for the general presentation, but instead exploit the *funloc* notation, specific to the case where the return type is a location. For other types, we can quantify over the result value explicitly.

37.3.6 Allocation of a Reference with Greater Contents

Consider the following function, *ref_greater*, which takes as argument the address *p* of a memory cell with contents *n*, allocates a fresh memory cell with contents *n+1*, then returns the address of that fresh cell.

```
Definition ref_greater : val :=
<\{ fun 'p =>
  let 'n = !'p in
  let 'm = 'n + 1 in
  ref 'm \}.
```

The precondition of *ref_greater* needs to assert the existence of a cell $p ~~> n$. The postcondition of *ref_greater* should assert the existence of two cells, $p ~~> n$ and $q ~~> (n+1)$, where *q* denotes the location returned by the function. The postcondition is thus written *funloc q => p ~~> n * q ~~> (n+1)*, which is a shorthand for *fun (r:val) => \exists q, \[r = val_loc q] * p ~~> n * q ~~> (n+1)*.

```
Lemma triple_ref_greater : \forall (p:loc) (n:int),
  triple (ref_greater p)
  (p ~~> n)
  (funloc q => p ~~> n \* q ~~> (n+1)).
```

Proof using.

```
xwp. xapp. xapp. xapp. intros q. xsimpl. auto.
```

Qed.

□

Exercise: 2 stars, standard, especially useful (*triple_ref_greater_abstract*) State another specification for the function *ref_greater* with a postcondition that does not reveal the

contents of the fresh reference q , but instead only asserts that it is greater than the contents of p . To that end, introduce in the postcondition an existentially quantified variable called m , with $m > n$. This new specification, to be called *triple_ref_greater_abstract*, should be derived from *triple_ref_greater*, following the proof pattern employed in *triple_incr_first_derived*.

□

37.3.7 Deallocation in Separation Logic

Separation Logic, in its simplest form, enforces that every piece of allocated data is eventually deallocated. Yet, OCaml is a programming language equipped with a garbage collector: programs do not contain explicit deallocation operations. Thus, concretely, if we consider an OCaml program that allocates a reference and that this reference is not described in the postcondition, we get stuck in the proof. In what follows, we will first describe how the proof gets stuck, then we will see how the problem goes away by adding an explicit deallocation operation, and we'll point at a later chapter (*Affine*) for the presentation of a generic solution to handling implicit deallocation.

To begin with, consider the following function, which makes local use of a reference, without exposing that reference in the postcondition. This function computes the successor of a integer n . It first stores n into a reference cell, then it increments that reference, and finally it returns the new contents of the reference.

```
Definition succ_using_incr_attempt :=
<{ fun 'n =>
  let 'p = ref 'n in
  incr 'p;
  ! 'p }>.
```

A call to that function can be specified using an empty precondition and a postcondition asserting that the final result is equal to $n+1$. Let us investigate how we get stuck on the last step when trying to prove that specification.

```
Lemma triple_succ_using_incr_attempt : ∀ (n:int),
  triple (succ_using_incr_attempt n)
  \[]
  (fun r ⇒ \[r = n+1]).
```

Proof using.

```
xwp. xapp. intros p. xapp. xapp. xsimpl. { auto. }
```

Abort.

We get stuck with the unprovable entailment $p \sim\sim > (n+1) ==> \[],$ where the left-hand side describes a state with one reference, whereas the right-hand side describes an empty state. There are three possibilities to work around the issue.

The first solution consists of extending the postcondition to account for the existence of the reference p . This yields a provable specification.

```
Lemma triple_succ_using_incr_attempt' : ∀ (n:int),
```

```

triple (succ_using_incr_attempt n)
  []
  (fun r => \[r = n+1] \* \exists p, (p ~~> (n+1))).

```

Proof using.

```
xwp. xapp. intros p. xapp. xapp. xsimpl. { auto. }
```

Qed.

However, while the specification above is provable, it is totally unsatisfying. Indeed, the piece of postcondition $\exists p, p ~~> (n+1)$ is of absolutely no use to the caller of the function. Worse, the caller will get its own heap predicate polluted with $\exists p, p ~~> (n+1)$, with no way of throwing away that predicate.

A second solution is to alter the code of the program to include an explicit free operation, written *free p*, for deallocating the reference. This operation does not exist in OCaml, but let us nevertheless assume it to be able to demonstrate how Separation Logic supports reasoning about explicit deallocation.

Definition `succ_using_incr` :=

```
<{ fun 'n =>
  let 'p = ref 'n in
  incr 'p;
  let 'x = ! 'p in
  free 'p;
  'x }>.
```

This program may be proved correct with respect to the intended postcondition `fun r => \[r = n+1]`, without the need to mention *p*. In the proof, shown below, the key step is the last call to *xapp*. This call is for reasoning about the operation *free p*, which consumes (i.e., removes) from the current state the heap predicate of the form $p ~~> ..$. At the last proof step, we invoke the tactic *xval* for reasoning about the return value.

```

Lemma triple_succ_using_incr : \forall n,
  triple (succ_using_incr n)
  []
  (fun r => \[r = n+1]).
```

Proof using.

```
xwp. xapp. intros p. xapp. xapp.
  xapp. xval. xsimpl. auto.
```

Qed.

The third solution consists of considering a generalized version of Separation Logic in which specific classes of heap predicates may be freely discarded from the current state, at any point during the proofs. This variant is described in the chapter *Affine*.

37.3.8 Combined Reading and Freeing of a Reference

The function `get_and_free` takes as argument the address `p` of a reference cell. It reads the contents of that cell, frees the cell, and returns its contents.

```
Definition get_and_free : val :=  
<{ fun 'p =>  
  let 'v = ! 'p in  
  free 'p;  
  'v }>.
```

Exercise: 2 stars, standard, especially useful (`triple_get_and_free`) Prove the correctness of the function `get_and_free`.

```
Lemma triple_get_and_free : ∀ p v,  
  triple (get_and_free p)  
  (p ~~> v)  
  (fun r => \[r = v]).
```

Proof using. Admitted.

□

Hint Resolve `triple_get_and_free` : `triple`.

37.4 Recursive Functions

37.4.1 Axiomatization of the Mathematical Factorial Function

Our next example consists of a program that evaluates the factorial function. To specify this function, we consider a Coq axiomatization of the mathematical factorial function, named `facto`. Here again, we use the `Parameter` keyword because we are not interested in the details of the implementation of this function. We wrap the axiomatization inside a module, so that we can later refer to it from other files.

Module Import FACTO.

Parameter `facto` : int → int.

The factorial of 0 and 1 is equal to 1, and the factorial of n for $n > 1$ is equal to $n \times \text{facto}(n-1)$. Note that we purposely leave unspecified the value of `facto` on negative arguments.

Parameter `facto_init` : ∀ n,

```
0 ≤ n ≤ 1 →  
facto n = 1.
```

Parameter `facto_step` : ∀ n,

```
n > 1 →  
facto n = n × (facto (n-1)).
```

End FACTO.

37.4.2 A Partial Recursive Function, Without State

As a warm-up, we first consider consider a recursive function that does not involve any mutable state. The program function `factorec` computes the factorial of its argument: it implements the logical function *facto*.

OCaml:

```
let rec factorec n = if n <= 1 then 1 else n * factorec (n-1)
```

The corresponding code in A-normal form is slightly more verbose.

Definition `factorec : val :=`

```
<{ fix 'f 'n =>
    let 'b = 'n ≤ 1 in
    if 'b
        then 1
    else let 'x = 'n - 1 in
        let 'y = 'f 'x in
        'n × 'y }>.
```

A call to `factorec n` can be specified as follows:

- the initial state is empty,
- the final state is empty,
- the result value r is such that $r = \text{facto } n$, when $n \geq 0$.

In case the argument is negative (i.e., $n < 0$), we have two choices:

- either we explicitly specify that the result is 1 in this case,
- or we rule out this possibility by requiring $n \geq 0$.

Let us follow the second approach, in order to illustrate the specification of partial functions.

There are two possibilities for expressing the constraint $n \geq 0$:

- either we use as precondition $\setminus[n \geq 0]$,
- or we we use the empty precondition, that is, $\setminus[]$, and we place an assumption $(n \geq 0) \rightarrow _$ to the front of the triple.

The two presentations are totally equivalent. We prefer the second presentation, which tends to improve both the readability of specifications and the conciseness of proof scripts. In that style, the specification of `factorec` is stated as follows.

Lemma `triple_factorec : ∀ n,`

$n \geq 0 \rightarrow$

`triple (factorec n)`

```
\[]  
(fun r => \[r = facto n]).
```

In general, we prove specifications for recursive functions by exploiting a strong induction principle statement (“well-founded induction”) that allows us to assume, while we try to prove the specification, that the specification already holds for any “smaller input”. The (well-founded) order relation that defines whether an input is smaller than another one is specified by the user.

In the specific case of the function `factorec`, the input is a nonnegative integer `n`, so we can assume, by induction hypothesis, that the specification already holds for any nonnegative integer smaller than `n`. Let’s walk through the proof script in detail, to see in particular how to set up the induction, how we exploit it for reasoning about the recursive call, and how we justify that the recursive call is made on a smaller input. **Proof using**. `unfold factorec`.

We set up a proof by induction on `n` to obtain an induction hypothesis for the recursive calls. The well-founded relation `downto 0` captures the order on arguments: `downto 0 i j` asserts that $0 \leq i < j$ holds. The tactic `induction_wf`, provided by the TLC library, helps setting up well-founded inductions. It is exploited as follows. `intros n. induction_wf IH: (downto 0) n.`

Observe the induction hypothesis `IH`. By unfolding `downto` as done in the next step, we can see that this hypothesis asserts that the specification that we are trying to prove already holds for arguments that are smaller than the current argument `n`, and that are greater than or equal to 0. `unfold downto in IH. intros Hn. xwp.`

We reason about the evaluation of the boolean condition $n \leq 1$. `xapp.`

The result of the evaluation of $n \leq 1$ in the source program is described by the boolean value `isTrue (n ≤ 1)`, which appears in the *CODE* section after *Ifval*. The operation `isTrue` is provided by the TLC library as a conversion function from `Prop` to `bool`. The use of such a conversion function (which leverages classical logic) greatly simplifies the process of automatically performing substitutions after calls to `xapp`.

We next perform the case analysis on the test $n \leq 1$. `xif.`

Doing so gives two cases.

In the “then” branch, we can assume $n \leq 1$. `{ intros C.`

Here, the return value is 1. `xval. xsimpl.`

We check that $1 = \text{facto } n$ when $n \leq 1$. `rewrite facto_init. math. math. }`

In the “else” branch, we can assume $n > 1$. `{ intros C.`

We reason about the evaluation of $n-1$ `xapp.`

We reason about the recursive call, implicitly exploiting the induction hypothesis `IH` with $n-1$. `xapp.`

We justify that the recursive call is indeed made on a smaller argument than the current one, that is, a nonnegative integer smaller than `n`. `{ math. }`

We justify that the recursive call is made to a nonnegative argument, as required by the specification. `{ math. }`

We reason about the multiplication $n \times \text{facto}(n-1)$. `xapp.`

We check that $n \times \text{facto } (n-1)$ matches `facto n`. `xsimpl. rewrite (@facto_step n).`

math. math. }

Qed.

37.4.3 A Recursive Function with State

Let's now tackle a recursive function involving some mutable state. The function `repeat_incr p m` makes m times a call to `incr p`. Here, m is assumed to be a nonnegative value.

OCaml:

```
let rec repeat_incr p m = if m > 0 then ( incr p; repeat_incr p (m - 1) )
```

In the concrete syntax for programs, conditionals without an 'else' branch are written `if t1 then t2 end`. The keyword `end` avoids ambiguities in cases where this construct is followed by a semi-column.

```
Definition repeat_incr : val :=
<{ fix 'f 'p 'm =>
  let 'b = 'm > 0 in
  if 'b then
    incr 'p;
    let 'x = 'm - 1 in
    'f 'p 'x
  end }>.
```

The specification for `repeat_incr p` requires that the initial state contains a reference `p` with some integer contents n , that is, $p \sim\!> n$. Its postcondition asserts that the resulting state is $p \sim\!> (n+m)$, which is the result after incrementing m times the reference `p`. Observe that this postcondition is only valid under the assumption that $m \geq 0$.

Lemma triple_repeat_incr : $\forall (m\ n:\text{int})\ (p:\text{loc}),$

$$\begin{aligned} m \geq 0 \rightarrow \\ \text{triple}(\text{repeat_incr } p\ m) \\ (p \sim\!> n) \\ (\text{fun } _ \Rightarrow p \sim\!> (n + m)). \end{aligned}$$

Exercise: 2 stars, standard, especially useful (`triple_repeat_incr`) Prove the specification of the function `repeat_incr`, by following the template of the proof of `triple_factorec`'. Hint: begin the proof with `intros m. induction_wf IH: ...`, but make sure to not leave `n` in the goal, otherwise the induction principle that you obtain is too weak.

Proof using. Admitted.

□

In the previous examples of recursive functions, the induction was always performed on the first argument quantified in the specification. When the decreasing argument is not the first one, additional manipulations are required for re-generalizing into the goal the variables that may change during the course of the induction. Here is an example illustrating how to deal with such a situation.

```

Lemma triple_repeat_incr' : ∀ (p:loc) (n m:int),
  m ≥ 0 →
  triple (repeat_incr p m)
    (p ~~> n)
    (fun _ => p ~~> (n + m)).

```

Proof using.

First, introduces all variables and hypotheses. `intros n m Hm.`

Next, generalize those that are not constant during the recursion. We use the TLC tactic `gen`, which is a shorthand for *generalized dependent*. `gen n Hm.`

Then, set up the induction. `induction_wf IH: (downto 0) m. unfold downto in IH.`

Finally, re-introduce the generalized hypotheses. `intros.`

The rest of the proof is exactly the same as before. `Abort.`

37.4.4 Trying to Prove Incorrect Specifications

We established for `repeat_incr p m` a specification featuring the hypothesis $m \geq 0$, but what if we did omit this hypothesis? At which step would we get stuck in the proof? What feedback would we get at that step?

Certainly, we expect the proof to get stuck because if $m < 0$. Indeed, in that case, the call to `repeat_incr p m` terminates immediately, thus the final state is $p ~~> n$, like the initial state, and this final state does not match the claimed postcondition $p ~~> (n + m)$. Let us investigate how the proof of lemma `triple_repeat_incr` breaks.

```

Lemma triple_repeat_incr_incorrect : ∀ (p:loc) (n m:int),
  triple (repeat_incr p m)
    (p ~~> n)
    (fun _ => p ~~> (n + m)).

```

Proof using.

`intros. revert n. induction_wf IH: (downto 0) m. unfold downto in IH.`

`intros. xwp. xapp. xif; intros C.`

{

`xapp. xapp. xapp. { math. } xsimpl. math. }`

{

`xval.`

At this point, we are requested to justify that the current state $p ~~> n$ matches the postcondition $p ~~> (n + m)$, which amounts to proving $n = n + m$. `xsimpl.`

`Abort.`

When the specification features the assumption $m \geq 0$, we can prove this equality because the fact that we are in the else branch means that $m \leq 0$, thus $m = 0$. However, without the assumption $m \geq 0$, the value of m could very well be negative. In that case, the equality $n = n + m$ is unprovable. As a user, the proof obligation $(m \leq 0) \rightarrow (n = n + m)$ gives us a very strong hint on the fact that either the code or the specification is not handling the case $m < 0$ properly. This concludes our example attempt at proving an incorrect specification.

In passing, we note that there exists a valid specification for `repeat_incr` that does not constrain `m` but instead specifies that, regardless of the value of `m`, the state evolves from $p \rightsquigarrow n$ to $p \rightsquigarrow (n + \max 0 m)$. The corresponding proof scripts exploits two characteristic properties of the function `max`.

```
Lemma max_l : ∀ n m,
  n ≥ m →
  max n m = n.
```

Proof using. `introv M. unfold max. case_if; math. Qed.`

```
Lemma max_r : ∀ n m,
  n ≤ m →
  max n m = m.
```

Proof using. `introv M. unfold max. case_if; math. Qed.`

Here is the most general specification for the function `repeat_incr`.

Exercise: 2 stars, standard, optional (`triple_repeat_incr'`) Prove the general specification for the function `repeat_incr`, covering also the case $m < 0$.

```
Lemma triple_repeat_incr' : ∀ (p:loc) (n m:int),
  triple (repeat_incr p m)
  (p ∽> n)
  (fun _ ⇒ p ∽> (n + max 0 m)).
```

Proof using. `Admitted.`

□

37.4.5 A Recursive Function Involving two References

Consider the function `step_transfer p q`, which repeatedly increments a reference `p` and decrements a reference `q`, as long as the contents of `q` is positive.

OCaml:

```
let rec step_transfer p q = if !q > 0 then ( incr p; decr q; step_transfer p q )
```

Definition `step_transfer` :=

```
<{ fix 'f 'p 'q ⇒
  let 'm = !'q in
  let 'b = 'm > 0 in
  if 'b then
    incr 'p;
    decr 'q;
    'f 'p 'q
  end }>.
```

The specification of `step_transfer` is essentially the same as that of the function `transfer` presented previously, the only difference being that we here assume the contents of `q` to be nonnegative.

```

Lemma triple_step_transfer : ∀ p q n m,
  m ≥ 0 →
  triple (step_transfer p q)
    (p ∼> n ∗ q ∼> m)
  (fun _ => p ∼> (n + m) ∗ q ∼> 0).

```

Exercise: 2 stars, standard, especially useful (triple_step_transfer) Verify the function `step_transfer`. Hint: to set up the induction, follow the pattern shown in the proof of `triple_repeat_incr'`.

Proof using. *Admitted.*

□

37.5 Summary

This chapter introduces the following notions:

- “Heap predicates”, which are used to describe memory states in Separation Logic.
- “Specification triples”, of the form `triple t H Q`, which relate a term `t`, a precondition `H`, and a postcondition `Q`.
- “Entailment proof obligations”, of the form `H ==> H'` or `Q ===> Q'`, which assert that a pre- or post-condition is weaker than another one.
- “Verification proof obligations”, of the form `PRE H CODE F POST Q`, which are produced by the framework, and capture triples by leveraging a notion of “weakest precondition”, presented further in the course.
- Custom proof tactics, called “x-tactics”, which are specialized tactics for carrying discharging these proof obligations.

Several “heap predicates”, used to describe memory states, were presented in this first chapter. They include:

- `p ∼> n`, which describes a memory cell at location `p` with contents `n`,
- `\[]`, which describes an empty state,
- `\[P]`, which also describes an empty state, and moreover asserts that the proposition `P` is true,
- `H1 ∗ H2`, which describes a state made of two disjoint parts, one satisfying `H1` and another satisfying `H2`,
- `\exists x, H`, which is used to quantify variables in postconditions.

All these heap predicates admit the type `hprop`, which describes predicates over memory states. Technically, `hprop` is defined as `state → Prop`.

The verification of practical programs is carried out using x-tactics, identified by the leading “x” letter in their name. These tactics include:

- `xwp` to begin a proof,
- `xapp` to reason about an application,
- `xval` to reason about a return value,
- `xif` to reason about a conditional,
- `xsimpl` to simplify or prove entailments ($H ==> H'$ and $Q ===> Q'$).

In addition to x-tactics, the proof scripts exploit standard Coq tactics, as well as tactics from the TLC library, which provides a bunch of useful, general purpose tactics. In this chapter, we used a few TLC tactics:

- `math`, which is a variant of `lia` for proving mathematical goals,
- `induction_wf`, which sets up proofs by well-founded induction,
- `gen`, which is a shorthand for `generalize dependent`, a tactic also useful to set up induction principles.

37.6 Historical Notes

The key ideas of Separation Logic were devised by John Reynolds, inspired in part by older work by *Burstall* 1972 (in Bib.v). Reynolds presented his ideas in lectures given in the fall of 1999. The proposed rules turned out to be unsound, but *Ishtiaq* and *O’Hearn* 2001 (in Bib.v) noticed a strong relationship with the logic of bunched implications by *O’Hearn* and *Pym* 1999 (in Bib.v), leading to ideas on how to set up a sound program logic. Soon afterwards, the seminal publications on Separation Logic appeared at the CSL workshop *O’Hearn, Reynolds, and Yang* 2001 (in Bib.v) and at the LICS conference *Reynolds* 2002 (in Bib.v).

The Separation Logic specifications and proof scripts using x-tactics presented in this file are directly adapted from the CFML tool (2010-2020), developed mainly by Arthur Charguéraud. The notations for Separation Logic predicates are directly inspired from those introduced in the Ynot project *Chlipala et al* 2009 (in Bib.v). See chapter *Bib* for references.

Chapter 38

Library `SLF.Repr`

38.1 Repr: Representation Predicates

Set Implicit Arguments.

From *SLF* Require Import LibSepReference.

Import ProgramSyntax DemoPrograms.

From *SLF* Require Import Basic.

Open Scope liblist_scope.

Implicit Types $n m : \text{int}$.

Implicit Types $p q s c : \text{loc}$.

Implicit Types $x : \text{val}$.

38.2 First Pass

A representation predicate is a heap predicate that describes a mutable data structure. For example, the heap predicate `MList L p` describes a mutable linked list whose head cell is at location `p`, and whose elements are described by the Coq list `L`. In this chapter, we'll see how to define `MList`, and to use this predicate for specifying and verifying functions that operate on mutable linked lists. We will also study representation predicates for mutable trees, as well as for counter functions, which feature an internal state.

As explained in the *Preface*, this chapter, like all the following ones, is decomposed in three parts:

- The *First Pass* section presents the most important ideas only.
- The *More Details* section presents additional material explaining in more depth the meaning and the consequences of the key results. By default, readers would eventually read all this material.
- The *Optional Material* section contains more advanced material, for readers who can afford to invest more time in the topic.

38.2.1 Formalization of the List Representation Predicate

The implementation of mutable lists and trees involves the use of records. For simplicity, field names are represented as natural numbers. For example, to represent a list cell with a head and a tail field, we define `head` as the constant zero, and `tail` as the constant one.

Definition `head : field := 0%nat.`

Definition `tail : field := 1%nat.`

The heap predicate $p \rightsquigarrow \{ \text{head} := x; \text{tail} := q \}$ describes a record allocated at location p , with a head field storing x and a tail field storing q . The arrow \rightsquigarrow is provided by the framework for describing records. The notation $\{ \dots \}$ is a handy notation for a list of pairs of field names and values of arbitrary types. (Further details on the formalization of records are presented in chapter `Struct`.)

A mutable list consists of a chain of cells. Each cell stores a tail pointer that gives the location of the next cell in the chain. The last cell stores the `null` value in its tail field.

The heap predicate `MList L p` describes a list whose head cell is at location p , and whose elements are described by the list L . This predicate is defined recursively on the structure of L :

- if L is empty, then p is the null pointer,
- if L is of the form $x::L'$, then p is not null, and the head field of p contains x , and the tail field of p contains a pointer q such that `MList L' q` describes the tail of the list.

This definition is formalized as follows.

```
Fixpoint MList (L:list val) (p:loc) : hprop :=
  match L with
  | nil ⇒ \[p = null]
  | x::L' ⇒ \exists q, (p \rightsquigarrow \{ head := x; tail := q \}) \* (MList L' q)
  end.
```

38.2.2 Alternative Characterizations of `MList`

Carrying out proofs directly with `MList` can be slightly cumbersome, mainly due to Coq's limited support for folding back definitions. We find it more practical to explicitly state equalities that paraphrase the definition of `MList`. There is one equality for the `nil` case, and one for the `cons` case.

```
Lemma MList_nil : ∀ p,
  (MList nil p) = \[p = null].
```

Proof using. auto. Qed.

```
Lemma MList_cons : ∀ p x L',
  MList (x::L') p =
  \exists q, (p \rightsquigarrow \{ head := x; tail := q \}) \* (MList L' q).
```

Proof using. auto. Qed.

In addition, it is also very useful in proofs to reformulate the definition of `MList L p` in the form of a case analysis on whether the pointer `p` is null or not. This corresponds to the programming pattern `if p == null then ... else`. This alternative characterization of `MList L p` asserts that:

- if `p` is null, then `L` is empty,
- otherwise, `L` decomposes as `x::L'`, the head field of `p` contains `x`, and the tail field of `p` contains a pointer `q` such that `MList L' q` describes the tail of the list.

The corresponding lemma, shown below, is stated using the `If P then X else Y` construction, which generalizes Coq's construction `if b then X else Y` to discriminate over a proposition `P` as opposed to a boolean value `b`. The `If` construct leverages (strong) classical logic. It is provided by the TLC library, just like the tactic `case_if` which is convenient for performing the case analysis on whether `P` is true or false.

Lemma `MList_if` : $\forall (p:\text{loc}) (L:\text{list val}),$

```
(MList L p)
==> (If p = null
      then \[L = nil]
      else \exists x q L', \[L = x::L']
           \* (p ~~~> '{ head := x; tail := q}) \* (MList L' q)).
```

The proof is a bit technical, it may be skipped for a first reading. Proof using.

Let's prove this result by case analysis on `L`. intros. destruct `L` as [|`x L'`].

Case `L = nil`. By definition of `MList`, we have `p = null`. { `xchange MList_nil`. intros `M`.

We have to justify `L = nil`, which is trivial. The TLC `case_if` tactic performs a case analysis on the argument of the first visible `if`. `case_if. xsimpl. auto. }`

Case `L = x::L'`.

One possibility is to perform a rewrite operation using `MList_cons` on its first occurrence. Then using CFML the tactic `xpull` to extract the existential quantifiers out from the precondition: `rewrite MList_cons. xpull. intros q`. A more efficient approach is to use the dedicated CFML tactic `xchange`, which is specialized for performing updates in the current state. { `xchange MList_cons`. intros `q`.

Because a record is allocated at location `p`, the pointer `p` cannot be null. The lemma `hrecord_not_null` allows us to exploit this property, extracting the hypothesis `p ≠ null`. We use again the tactic `case_if` to simplify the case analysis. `xchange hrecord_not_null. intros N. case_if.`

To conclude, it suffices to correctly instantiate the existential quantifiers. The tactic `xsimpl` is able to guess the appropriate instantiations. `xsimpl. auto. }`

Qed.

Note that the reciprocal entailment to the one stated in `MList_if` is also true, but we do not need it so we do not bother proving it here. In the rest of the course, we will never

unfold the definition `MList`, but only work using `MList_nil`, `MList_cons`, and `MList_if`. So, we can make `MList` opaque, thereby avoiding undesired simplifications.

`Global Opaque MList.`

38.2.3 In-place Concatenation of Two Mutable Lists

The function `append` expects two arguments: a pointer `p1` on a nonempty list, and a pointer `p2` on another list (possibly empty). The function updates the last cell from the first list in such a way that its tail points to the head cell of `p2`. After this operation, the pointer `p1` points to a list that corresponds to the concatenation of the two input lists.

OCaml:

```
let rec append p1 p2 = if p1.tail == null then p1.tail <- p2 else append p1.tail p2
```

```
Definition append : val :=
<{ fix 'f 'p1 'p2 =>
  let 'q1 = 'p1'.tail in
  let 'b = ('q1 = null) in
  if 'b
    then 'p1'.tail := 'p2
    else 'f 'q1 'p2 }>.
```

The `append` function is specified and verified as shown below. The proof pattern is representative of that of many list-manipulating functions, so it is essential that the reader follow through every step of this proof.

```
Lemma triple_append : ∀ (L1 L2:list val) (p1 p2:loc),
p1 ≠ null →
triple (append p1 p2)
(MList L1 p1 \* MList L2 p2)
(fun _ ⇒ MList (L1++L2) p1).
```

`Proof using.`

The induction principle provides an hypothesis for the tail of `L1`, i.e., for the list `L1'`, such that the relation `list_sub L1' L1` holds. *introv K. gen p1. induction_wf IH: list_sub L1. introv N. xwp.*

To begin the proof, we reveal the head cell of `p1`. We let `q1` denote the tail of `p1`, and `L1'` the tail of `L1`. *xchange (MList_if p1). case_if. xpull. intros x q1 L1' →.*

We then reason about the case analysis. *xapp. xapp. xif; intros Cq1.*

If `q1'` is null, then `L1'` is empty. *{ xchange (MList_if q1). case_if. xpull. intros →.*

In this case, we set the pointer, then we fold back the head cell. *xapp. xchange ← MList_cons. }*

Otherwise, if `q1'` is not null, we reason about the recursive call using the induction hypothesis, then we fold back the head cell. *{ xapp. xchange ← MList_cons. }*

`Qed.`

38.2.4 Smart Constructors for Linked Lists

We next introduce two smart constructors for linked lists, called `mnil` and `mcons`. The operation `mnil()` creates an empty list. Its implementation simply returns the value `null`. Its specification asserts that the return value is a pointer `p` such that `MList nil p` holds.

Definition `mnil : val :=`

```
<{ fun 'u =>
  null }>.
```

Lemma `triple_mnil :`

```
triple (mnil ())
  []
  (funloc p => MList nil p).
```

The proof uses the tactic `xchanges`, which is a shorthand for `xchange` followed with `xsimpl`. **Proof using.** `xwp. xval. xchanges ← (MList_nil null)`. **Qed.**

Hint Resolve `triple_mnil : triple`.

Observe that the specification `triple_mnil` does not mention the `null` pointer anywhere. This specification may thus be used to specify the behavior of operations on mutable lists without having to reveal low-level implementation details that involve the `null` pointer.

The operation `mcons x q` creates a fresh list cell, with `x` in the head field and `q` in the tail field. Its implementation allocates and initializes a fresh record made of two fields. The allocation operation leverages the allocation construct written `{ head := 'x; tail := 'q }` in the code. This construct is in fact a notation for an operation called `val_new_hrecord_2`, which we here view as a primitive operation.

Definition `mcons : val :=`

```
<{ fun 'x 'q =>
  '{ head := 'x; tail := 'q } }>.
```

The operation `mcons` admits two specifications. The first one describes only the production of a heap predicate for the freshly allocated record.

Lemma `triple_mcons : ∀ x q,`

```
triple (mcons x q)
  []
  (funloc p => p ~~~> '{ head := x ; tail := q }).
```

Proof using. `xwp. xapp triple_new_hrecord_2; auto. xsimpl`. **Qed.**

The second specification assumes that the argument `q` comes with a list representation of the form `Mlist q L`, and it specifies that the function `mcons` produces the representation predicate `Mlist p (x::L)`. This second specification is derivable from the first one, by folding the representation predicate `MList` using the tactic `xchange`.

Lemma `triple_mcons' : ∀ L x q,`

```
triple (mcons x q)
  (MList L q)
```

```
(funloc p ⇒ MList (x::L) p).
```

Proof using.

```
intros. xapp triple_mcons.  
intros p. xchange ← MList_cons. xsimpl ×.
```

Qed.

In practice, this second specification is more often useful than the first one, hence we register it in the database for *xapp*. It remains possible to invoke *xapp triple_mcons* for exploiting the first specification, where needed.

Hint Resolve *triple_mcons'* : *triple*.

38.2.5 Copy Function for Lists

The function **mcopy** takes a mutable linked list and builds an independent copy of it, with help of the functions **mnil** and **mcons**.

OCaml:

```
let rec mcopy p = if p == null then mnil () else mcons (p.head) (mcopy p.tail)
```

Definition **mcopy** : **val** :=

```
<{ fix 'f 'p ⇒  
    let 'b = ('p = null) in  
    if 'b  
    then mnil ()  
    else  
        let 'x = 'p'.head in  
        let 'q = 'p'.tail in  
        let 'q2 = ('f 'q) in  
        mcons 'x 'q2 }>.
```

The precondition of **mcopy** requires a linked list described as **MList L p**. The postcondition asserts that the function returns a pointer *p'* and a list described as **MList L p'**, in addition to the original list **MList L p**. The two lists are totally disjoint and independent, as captured by the separating conjunction symbol (the star).

Lemma **triple_mcopy** : $\forall L p,$

```
triple (mcopy p)  
(MList L p)  
(funloc p' ⇒ (MList L p) \* (MList L p')).
```

The proof is structure is like the previous ones. While playing the script, try to spot the places where:

- **mnil** produces an empty list of the form **MList nil p'**,
- the recursive call produces a list of the form **MList L' q'**,
- **mcons** produces a list of the form **MList (x::L') p'**.

Proof using.

```
intros. gen p. induction_wf IH: list_sub L.
xwp. xapp. xchange MList_if. xif; intros C; case_if; xpull.
{ intros →. xapp. xsimpl×. subst. xchange× ← MList_nil. }
{ intros x q L' →. xapp. xapp. xapp. intros q'.
  xapp. intros p'. xchange ← MList_cons. xsimpl×. }
```

Qed.

38.2.6 Length Function for Lists

The function `mlength` computes the length of a mutable linked list.

OCaml:

```
let rec mlength p = if p == null then 0 else 1 + mlength p.tail
```

Definition `mlength` : **val** :=

```
<{ fix 'f 'p =>
  let 'b = ('p = null) in
  if 'b
    then 0
    else (let 'q = 'p'.tail in
          let 'n = 'f 'q in
          'n + 1) }>.
```

Exercise: 3 stars, standard, especially useful (triple_mlength) Prove the correctness of the function `mlength`.

Lemma `triple_mlength` : $\forall L p,$

```
triple (mlength p)
(MList L p)
(fun r => \[r = val_int (length L)] \* (MList L p)).
```

Proof using. Admitted.

□

38.2.7 Alternative Length Function for Lists

In this section, we consider an alternative implementation of `mlength` that uses an auxiliary reference cell, called `c`, to keep track of the number of cells traversed so far. The list is traversed recursively, incrementing the contents of the reference once for every cell.

OCaml:

```
let rec listacc c p = if p == null then () else (incr c; listacc c p.tail)
let mlength' p = let c = ref 0 in listacc c p; !c
```

Definition `acclength` : **val** :=

```
<{ fix 'f 'c 'p =>
```

```

let 'b = ('p ≠ null) in
if 'b then
  incr 'c;
  let 'q = 'p'.tail in
    'f 'c 'q
end }>.

```

Definition `mlength'` : **val** :=
`<{ fun 'p =>`
 `let 'c = ref 0 in`
 `acclength 'c 'p;`
 `get_and_free 'c }>.`

Exercise: 3 stars, standard, especially useful (`triple_mlength'`) Prove the correctness of the function `mlength'`. Hint: start by stating a lemma `triple_acclength` expressing the specification of the recursive function `acclength`. Make sure to generalize the appropriate variables before applying the well-founded induction tactic. Then, complete the proof of the specification `triple_mlength'`, using `xapp triple_acclength` to reason about the call to the auxiliary function.

Lemma `triple_mlength'` : $\forall L p,$
`triple (mlength' p)`
`(MList L p)`
`(fun r => \[r = val_int (length L)] * (MList L p)).`

Proof using. Admitted.

□

38.2.8 In-Place Reversal Function for Lists

The function `mrev` takes as argument a pointer on a mutable list, and returns a pointer on the reverse list, that is, a list with elements in the reverse order. The cells from the input list are reused for constructing the output list. The operation is said to be “in place”.

OCaml:

```

let rec mrev_aux p1 p2 = if p2 == null then p1 else (let p3 = p2.tail in p2.tail <- p1;
mrev_aux p2 p3)
let mrev p = mrev_aux p null

```

Definition `mrev_aux` : **val** :=
`<{ fix 'f 'p1 'p2 =>`
 `let 'b = ('p2 = null) in`
 `if 'b`
 `then 'p1`
 `else (`
 `let 'p3 = 'p2'.tail in`

```
'p2'.tail := 'p1;
'f 'p2 'p3) }>.
```

```
Definition mrev : val :=
<{ fun 'p =>
  mrev_aux null 'p }>.
```

Exercise: 5 stars, standard, optional (triple_mrev) Prove the correctness of the functions `mrev_aux` and `mrev`. Hint: here again, start by stating a lemma `mtriple_rev_aux` expressing the specification of the recursive function `mrev_aux`. Make sure to generalize the appropriate variables before applying the well-founded induction tactic. Then, complete the proof of `triple_mrev`, using `xapp triple_mrev_aux`.

```
Lemma triple_mrev : ∀ L p,
  triple (mrev p)
  (MList L p)
  (funloc q ⇒ MList (rev L) q).
```

Proof using. Admitted.

□

38.3 More Details

38.3.1 Sized Stack

Module SIZEDSTACK.

In this section, we consider the implementation of a mutable stack featuring a constant-time access to the size of the stack. This stack structure consists of a 2-field record, storing a pointer on a mutable linked list, and an integer storing the length of that list. The implementation includes a function `create` to allocate an empty stack, a function `sizeof` for reading the size, and three functions `push`, `top` and `pop` for operating at the top of the stack.

OCaml:

```
type 'a stack = { data : 'a list; size : int }
let create () = { data = nil; size = 0 }
let sizeof s = s.size
let push p x = s.data <- mcons x s.data; s.size <- s.size + 1
let top s = let p = s.data in p.head
let pop s = let p = s.data in let x = p.head in let q = p.tail in delete p; s.data <- q in
  s.size <- s.size - 1; x
```

The representation predicate for the stack takes the form `Stack L s`, where `s` denotes the location of the record describing the stack, and where `L` denotes the list of items stored in the stack. The underlying mutable list is described as `MList L p`, where `p` is the location `p` stored in the first field of the record. The definition of `Stack` is as follows.

```

Definition data : field := 0%nat.
Definition size : field := 1%nat.

Definition Stack (L:list val) (s:loc) : hprop :=
  ∃ p, s ~~~>‘{ data := p; size := length L } ∗ (MList L p).

```

Observe that the predicate `Stack` does not expose the location of the mutable list; this location is existentially quantified in the definition. The predicate `Stack` also does not expose the size of the stack, as this value can be deduced by computing `length L`. Let's start with the specification and verification of `create` and `sizeof`.

```

Definition create : val :=
  <{ fun 'u ⇒
    ‘{ data := null; size := 0 } }>.

```

```

Lemma triple_create :
  triple (create ())
  []
  (funloc s ⇒ Stack nil s).

```

Proof using.

```

xwp. xapp triple_new_hrecord_2; auto. intros s.
unfold Stack. xsimpl×. xchange× ← (MList_nil null).

```

Qed.

The `sizeof` operation returns the contents of the `size` field of a stack.

```

Definition sizeof : val :=
  <{ fun 'p ⇒
    'p‘.size }>.

```

```

Lemma triple_sizeof : ∀ L s,
  triple (sizeof s)
  (Stack L s)
  (fun r ⇒ \[r = length L] ∗ Stack L s).

```

Proof using.

```

xwp. unfold Stack. xpull. intros p. xapp. xsimpl×.

```

Qed.

The `push` operation extends the head of the list, and increments the `size` field.

```

Definition push : val :=
  <{ fun 's 'x ⇒
    let 'p = 's‘.data in
    let 'p2 = mcons 'x 'p in
    's‘.data := 'p2;
    let 'n = 's‘.size in
    let 'n2 = 'n + 1 in
    's‘.size := 'n2 }>.

```

Exercise: 3 stars, standard, especially useful (triple_push) Prove the following specification for the push operation.

Lemma triple_push : $\forall L s x,$
triple (push $s x)$
(Stack $L s)$
(fun $u \Rightarrow \text{Stack } (x :: L) s).$

Proof using. *Admitted.*

□

The pop operation extracts the element at the head of the list, updates the data field to the tail of the list, and decrements the size field.

Definition pop : val :=
 $\langle\{ \text{fun } 's \Rightarrow$
 $\quad \text{let } 'p = 's'.\text{data} \text{ in}$
 $\quad \text{let } 'x = 'p'.\text{head} \text{ in}$
 $\quad \text{let } 'p2 = 'p'.\text{tail} \text{ in}$
 $\quad \text{delete } 'p;$
 $\quad 's'.\text{data} := 'p2;$
 $\quad \text{let } 'n = 's'.\text{size} \text{ in}$
 $\quad \text{let } 'n2 = 'n - 1 \text{ in}$
 $\quad 's'.\text{size} := 'n2;$
 $\quad 'x \} \rangle.$

Exercise: 4 stars, standard, especially useful (triple_pop) Prove the following specification for the pop operation.

Lemma triple_pop : $\forall L s,$
 $L \neq \text{nil} \rightarrow$
triple (pop $s)$
(Stack $L s)$
(fun $x \Rightarrow \exists L', \exists [L = x :: L'] \text{ Stack } L' s).$

Proof using. *Admitted.*

□

The top operation extracts the element at the head of the list.

Definition top : val :=
 $\langle\{ \text{fun } 's \Rightarrow$
 $\quad \text{let } 'p = 's'.\text{data} \text{ in}$
 $\quad 'p'.\text{head} \} \rangle.$

Exercise: 2 stars, standard, optional (triple_top) Prove the following specification for the top operation.

Lemma triple_top : $\forall L s,$

```

 $L \neq \text{nil} \rightarrow$ 
triple (top s)
  (Stack L s)
  (fun x => \exists L', [L = x :: L'] \* Stack L s).

```

Proof using. Admitted.

□

End SIZEDSTACK.

38.3.2 Formalization of the Tree Representation Predicate

In this section, we generalize the ideas presented for linked lists to binary trees. For simplicity, let us consider binary trees that store integer values in the nodes. Just like mutable lists are specified with respect to Coq’s purely functional lists, mutable binary trees are specified with respect to Coq trees. Consider the following inductive definition of the type **tree**. A leaf is represented by the constructor **Leaf**, and a node takes the form **Node n T1 T2**, where **n** is an integer and **T1** and **T2** denote the two subtrees.

```

Inductive tree : Type :=
| Leaf : tree
| Node : int → tree → tree → tree.

```

Implicit Types T : **tree**.

In a program manipulating a mutable tree, an empty tree is represented using the **null** pointer, and a node is represented in memory using a three-cell record. The first field, named “item”, stores an integer. The other two fields, named “left” and “right”, store pointers to the left and right subtrees, respectively.

Definition item : field := 0%nat.

Definition left : field := 1%nat.

Definition right : field := 2%nat.

The heap predicate $p \sim\sim>\{ \text{item} := n; \text{left} := p1; \text{right} := p2 \}$ describes a record allocated at location **p**, storing the integer **n** and the two pointers **p1** and **p2**.

The representation predicate **MTree T p**, of type **hprop**, asserts that the mutable tree structure with root at location **p** describes the logical tree **T**. The predicate is defined recursively on the structure of **T**:

- if **T** is a **Leaf**, then **p** is the null pointer,
- if **T** is a node **Node n T1 T2**, then **p** is not null, and at location **p** one finds a record with field contents **n**, **p1** and **p2**, with **MTree T1 p1** and **MTree T2 p2** describing the two subtrees.

```

Fixpoint MTree (T:tree) (p:loc) : hprop :=
  match T with

```

```

| Leaf  $\Rightarrow \set{p = \text{null}}$ 
| Node  $n\ T1\ T2 \Rightarrow \exists p1\ p2,$ 
   $(p \rightsquigarrow \{ \text{item} := n; \text{left} := p1; \text{right} := p2 \})$ 
   $\ast (\text{MTree } T1\ p1)$ 
   $\ast (\text{MTree } T2\ p2)$ 
end.

```

38.3.3 Alternative Characterization of MTree

Just like for MList, it is very useful for proofs to state three lemmas that paraphrase the definition of MTree. The first two lemmas correspond to the folding/unfolding rules for leaves and nodes.

Lemma MTree_Leaf : $\forall p,$
 $(\text{MTree Leaf } p) = \set{p = \text{null}}.$

Proof using. auto. Qed.

Lemma MTree_Node : $\forall p\ n\ T1\ T2,$
 $(\text{MTree (Node } n\ T1\ T2) p) =$
 $\exists p1\ p2,$
 $(p \rightsquigarrow \{ \text{item} := n; \text{left} := p1; \text{right} := p2 \})$
 $\ast (\text{MTree } T1\ p1) \ast (\text{MTree } T2\ p2).$

Proof using. auto. Qed.

The third lemma reformulates $\text{MTree } T\ p$ using a case analysis on whether p is the null pointer. This formulation matches the case analysis typically perform in the code of functions that operates on trees.

Lemma MTree_if : $\forall (p:\text{loc}) (T:\text{tree}),$
 $(\text{MTree } T\ p)$
 $\Rightarrow (\text{If } p = \text{null}$
 $\quad \text{then } \set{T = \text{Leaf}}$
 $\quad \text{else } \exists n\ p1\ p2\ T1\ T2, \set{T = \text{Node } n\ T1\ T2}$
 $\quad \ast (p \rightsquigarrow \{ \text{item} := n; \text{left} := p1; \text{right} := p2 \})$
 $\quad \ast (\text{MTree } T1\ p1) \ast (\text{MTree } T2\ p2)).$

Proof using.

```

intros. destruct T as [|n T1 T2].
{ xchange MTree_Leaf. intros M. case_if. xsimpl. }
{ xchange MTree_Node. intros p1 p2.
  xchange hrecord_not_null. intros N. case_if. xsimpl. }

```

Qed.

Opaque MTree.

38.3.4 Additional Tooling for MTree

For reasoning about recursive functions over trees, it is useful to exploit the well-founded order associated with “immediate subtrees”. Concretely, **tree_sub** $T_1 \ T$ asserts that the tree T_1 is either the left or the right subtree of the tree T . This order may be exploited for verifying recursive functions over trees using the tactic *induction_wf IH*: **tree_sub** T .

```
Inductive tree_sub : binary (tree) :=
| tree_sub_1 : ∀ n T1 T2,
  tree_sub T1 (Node n T1 T2)
| tree_sub_2 : ∀ n T1 T2,
  tree_sub T2 (Node n T1 T2).
```

Lemma tree_sub_wf : wf tree_sub.

Proof using.

```
intros T. induction T; constructor; intros t' H; inversions¬ H.
```

Qed.

Hint Resolve tree_sub_wf : wf.

For allocating fresh tree nodes as a 3-field record, we introduce the operation **mnode** n p_1 p_2 , defined and specified as follows.

```
Definition mnode : val :=
val_new_hrecord_3 item left right.
```

A first specification of **mnode** describes the allocation of a record.

```
Lemma triple_mnode : ∀ n p1 p2,
triple (mnode n p1 p2)
[][]
(funloc p ⇒ p ~~~> '{ item := n ; left := p1 ; right := p2 }).
```

Proof using. intros. *applys*× triple_new_hrecord_3. Qed.

A second specification, derived from the first, asserts that, if provided the description of two subtrees T_1 and T_2 at locations p_1 and p_2 , the operation **mnode** n p_1 p_2 builds, at a fresh location p , a tree described by *Mtree* [Node n T_1 T_2] p . Compared with the first specification, this second specification is said to perform “ownership transfer”.

Exercise: 2 stars, standard, optional (triple_mnode') Prove the specification **triple_mnode'** for node allocation.

```
Lemma triple_mnode' : ∀ T1 T2 n p1 p2,
triple (mnode n p1 p2)
(MTree T1 p1 * MTree T2 p2)
(funloc p ⇒ MTree (Node n T1 T2) p).
```

Proof using. Admitted.

Hint Resolve triple_mnode' : triple.

□

38.3.5 Tree Copy

The operation `tree_copy` takes as argument a pointer `p` on a mutable tree and returns a fresh copy of that tree. It is defined in a similar way to the function `mcopy` for lists.

OCaml:

```
let rec tree_copy p = if p = null then null else mnode p.item (tree_copy p.left) (tree_copy p.right)
```

Definition `tree_copy` :=

```
<{ fix 'f 'p =>
  let 'b = ('p = null) in
  if 'b then null else (
    let 'n = 'p'.item in
    let 'p1 = 'p'.left in
    let 'p2 = 'p'.right in
    let 'q1 = 'f 'p1 in
    let 'q2 = 'f 'p2 in
    mnode 'n 'q1 'q2
  ) }>.
```

Exercise: 3 stars, standard, optional (triple_tree_copy) Prove the specification of `tree_copy`.

Lemma `triple_tree_copy` : $\forall p \ T,$
 $\text{triple}(\text{tree_copy } p)$
 $(\text{MTree } T \ p)$
 $(\text{funloc } q \Rightarrow (\text{MTree } T \ p) \ \backslash * \ (\text{MTree } T \ q)).$

Proof using. *Admitted.*

□

38.3.6 Computing the Sum of the Items in a Tree

The operation `mtreesum` takes as argument the location `p` of a mutable tree, and it returns the sum of all the integers stored in the nodes of that tree. Consider the implementation that traverses the tree, using an auxiliary reference cell to maintain the sum of all the items visited so far.

OCaml:

```
let rec treeacc c p = if p <> null then (c := !c + p.item; treeacc c p.left; treeacc c p.right)
let mtreesum p = let c = ref 0 in treeacc c p; !c
```

Definition `treeacc` : **val** :=

```
<{ fix 'f 'c 'p =>
  let 'b = ('p ≠ null) in
  if 'b then
```

```

let 'm = ! 'c in
let 'x = 'p'.item in
let 'm2 = 'm + 'x in
  'c := 'm2;
let 'p1 = 'p'.left in
  'f 'c 'p1;
let 'p2 = 'p'.right in
  'f 'c 'p2
end }>.

```

```

Definition mtreesum : val :=
<{ fun 'p =>
  let 'c = ref 0 in
  treeacc 'c 'p;
  get_and_free 'c }>.

```

The specification of `mtreesum` is expressed in terms of the Coq function `treesum`, which computes the sum of the node items stored in a logical tree. This operation is defined by recursion over the tree.

```

Fixpoint treesum (T:tree) : int :=
match T with
| Leaf => 0
| Node n T1 T2 => n + treesum T1 + treesum T2
end.

```

Exercise: 4 stars, standard, optional (`triple_mtreesum`) Prove the correctness of the function `mlength'`. Hint: to begin with, state and prove the specification lemma `triple_treeacc`.

```

Lemma triple_mtreesum : ∀ T p,
  triple (mtreesum p)
  (MTree T p)
  (fun r => \[r = treesum T] \* (MTree T p)).

```

Proof using. Admitted.

□

38.3.7 Verification of a Counter Function with Local State

This section is concerned with the verification of counter functions, which feature internal, mutable state. A counter function `f` is a function that, each time it is called, returns the next integer. Concretely, the first call to `f()` returns 1, the second call to `f()` returns 2, the third call to `f()` returns 3, and so on.

A counter function can be implemented using a reference cell, named `p`, that stores the integer last returned by the counter. Initially, the contents of the reference `p` is zero. Each

time the counter function is called, the contents of p is increased by one unit, and the new value of the contents is returned to the caller.

The function `create_counter` produces a fresh counter function. Concretely, `create_counter()` returns a counter function f independent from any other previously existing counter function.

OCaml:

```
let create_counter () = let p = ref 0 in (fun () -> (incr p; !p))
```

Definition `create_counter : val :=`

```
<{ fun 'u =>
  let 'p = ref 0 in
  (fun_ 'u => (incr 'p; !'p)) }>.
```

In this section, we present two specifications for counter functions. The first specification is the most direct, however it exposes the existence of the reference cell, revealing implementation details about the counter function. The second specification is more abstract: it hides from the client the internal representation of the counter, by means of using an abstract representation predicate.

Let us begin with the simple, direct specification. The proposition `CounterSpec f p` asserts that f is a counter function f whose internal state is stored in a reference cell at location p . Thus, invoking f in a state $p \rightsquigarrow m$ updates the state to $p \rightsquigarrow (m+1)$, and produces the output value $m+1$.

Definition `CounterSpec (f:val) (p:loc) : Prop :=`

```
 $\forall m, \text{triple } (f ())$ 
 $(p \rightsquigarrow m)$ 
 $(\text{fun } r \Rightarrow \exists [r = m+1] \ \text{(* } p \rightsquigarrow (m+1)\text{)}).$ 
```

Implicit Type $f : \text{val}$.

The function `create_counter` creates a fresh counter. Its precondition is empty. Its post-condition asserts that the function f being returned satisfies `CounterSpec f p`, and the output state contains a cell $p \rightsquigarrow 0$ for some existentially quantified location p .

Lemma `triple_create_counter :`

```
triple (create_counter ())
\ []
(\fun f \Rightarrow \exists p, (p \rightsquigarrow 0) \text{(* } \text{[CounterSpec } f p]\text{)}).
```

Proof using.

```
xwp. xapp. intros p.
```

The proof involves the use of a new tactic, called *xfun*, for reasoning about local function definitions. Here, *xfun* gives us the hypothesis H_f that specifies the code of f *xfun. intros f Hf*.

```
xsimpl.
```

```
{ intros m.
```

To reason about the call to the function f , we can exploit H_f , either explicitly by calling *applys Hf*, or automatically by simply calling *xapp . xapp.*

```
xapp. xapp. xsimpl. auto. }
```

Qed.

Consider now a call to a counter function f , under the assumption $\text{CounterSpec } f \ p$. Assume the input state to be of the form $p \sim\!> n$ for some n . Then, the call produces a state $p \sim\!> (n+1)$, and returns the value $n+1$. The specification shown below captures this logic, for any f .

```
Lemma triple_apply_counter : ∀ f p n,
  CounterSpec f p →
  triple (f ())
  (p ∼> n)
  (fun r ⇒ \[r = n+1] \* (p ∼> (n+1))).
```

The specification above is in fact nothing but a reformulation of the definition of CounterSpec . Thus, its proof is straightforward. (The proof may actually be reduced to just `auto`, however in general one needs to use `xapp` for reasoning about an abstract function.) *Proof using.* `introv Hf. unfolds in Hf. xapp. xsimpl ×.` Qed.

38.3.8 Verification of a Counter Function with Local State, With Abstraction

Let us move on to the presentation of more abstract specifications. The goal is to hide from the client the existence of the reference cell used to represent the internal state of the counter functions. To that end, we introduce the heap predicate $\text{IsCounter } f \ n$, which relates a function f , its current value n , and the piece of memory state involved in the implementation of this function. This piece of memory is of the form $p \sim\!> n$, for some location p , such that $\text{CounterSpec } f \ p$ holds.

```
Definition IsCounter (f:val) (n:int) : hprop :=
  ∃ p, p ∼> n \* \[CounterSpec f p].
```

Using IsCounter , we can reformulate the specification `create_counter` with a postcondition asserting that the output function f is described by the heap predicate $\text{IsCounter } f \ 0$.

```
Lemma triple_create_counter_abstract :
  triple (create_counter ())
  []
  (fun f ⇒ IsCounter f 0).
```

This lemma is the same as the previous specification `triple_create_counter`, except that the reference cell p is no longer visible. *Proof using.* `unfold IsCounter. applies triple_create_counter.` Qed.

We can also reformulate the specification of a call to a counter function. A call to $f()$, in a state satisfying $\text{IsCounter } f \ n$, produces a state satisfying $\text{IsCounter } f \ (n+1)$, and returns $n+1$.

Exercise: 4 stars, standard, especially useful (triple_apply_counter_abstract)
 Prove the abstract specification for a counter function. You will need to begin the proof using the tactic *xtriple*, for turning goal into a form on which *xpull* can be invoked to extract facts from the precondition.

```
Lemma triple_apply_counter_abstract : ∀ f n,
  triple (f ())
  (IsCounter f n)
  (fun r ⇒ \[r = n+1] \* (IsCounter f (n+1))).
```

Proof using. Admitted.

□

Opaque IsCounter.

38.4 Optional Material

38.4.1 Specification of a Higher-Order Repeat Operator

Consider the higher-order iterator `repeat`: a call to `repeat f n` executes `n` times the call `f()`.

OCaml:

```
let rec repeat f n = if n > 0 then (f(); repeat f (n-1))
```

Definition `repeat` : **val** :=

```
<{ fix 'g 'f 'n ⇒
  let 'b = ('n > 0) in
  if 'b then
    'f ();
  let 'n2 = 'n - 1 in
    'g 'f 'n2
  end }>.
```

For simplicity, let us assume for now $n \geq 0$. The specification of `repeat n f` can be expressed in terms of an invariant, named I , describing the state in between every two calls to `f`. We assume that the initial state satisfies $I 0$. Moreover, we assume that, for every index i in the range from 0 (inclusive) to n (exclusive), a call `f()` can execute in a state that satisfies $I i$ and produce a state that satisfies $I (i+1)$. The specification below asserts that, under these two assumptions, after the n calls to `f()`, the final state satisfies $I n$. The specification takes the form:

$n \geq 0 \rightarrow \text{Hypothesis_on_f} \rightarrow \text{triple}(\text{repeat } f \ n) (I \ 0) (\text{fun } u \Rightarrow I \ n)$

where *Hypothesis_on_f* is a proposition that captures the following specification:

forall i, $0 \leq i < n \rightarrow \text{triple}(f()) (I \ i) (\text{fun } u \Rightarrow I \ (i+1))$

The complete specification of `repeat n f` is thus as shown below.

```
Lemma triple_repeat : ∀ (I:int→hprop) (f:val) (n:int),
  n ≥ 0 →
```

```

(∀ i, 0 ≤ i < n →
  triple (f ())
    (I i)
    (fun u ⇒ I (i+1))) →
  triple (repeat f n)
    (I 0)
    (fun u ⇒ I n).

```

Proof using.

intros $H_n H_f$.

To establish this specification, we carry out a proof by induction over a generalized specification, covering the case where there remains m iterations to perform, for any value of m between 0 and n inclusive.

forall m , $0 \leq m \leq n \rightarrow$ triple (repeat $f m$) ($I (n-m)$) ($\text{fun } u \Rightarrow I n$)

We use the TLC tactics *cuts*, a variant of *cut*, to state show that the generalized specification entails the statement of **triple_repeat**. *cuts G*: $(\forall m, 0 \leq m \leq n \rightarrow$

```

triple (repeat f m)
  (I (n-m))
  (fun u ⇒ I n)).

```

{ *applys_eq G*. { *fequals math.* } { *math.* } }

We then carry a proof by induction: during the execution, the value of m decreases step by step down to 0. *intros m. induction_wf IH*: (downto 0) m . *intros Hm.*

xwp. xapp. xif; intros C.

We reason about the call to f { *xapp. { math. } xapp.*

We next reason about the recursive call. *xapp. { math. } { math. }*

math_rewrite ((n - m) + 1 = n - (m - 1)). xsimpl. }

Finally, when m reaches zero, we check that we obtain $I n$. { *xval. math_rewrite (n - m = n). xsimpl.* }

Qed.

38.4.2 Specification of an Iterator on Mutable Lists

The operation **miter** takes as argument a function f and a pointer p on a mutable list, and invokes f on each of the items stored in that list.

OCaml:

```
let rec miter f p = if p <> null then (f p.head; miter f p.tail)
```

Definition **miter** : **val** :=

```

<{ fix 'g 'f 'p ⇒
  let 'b = ('p ≠ null) in
  if 'b then
    let 'x = 'p.head in
    'f 'x;
    let 'q = 'p.tail in

```

```
'g 'f 'q
end }>.
```

The specification of `miter` follows the same structure as that of the function `repeat` from the previous section, with two main differences. The first difference is that the invariant is expressed not in terms of an index `i` ranging from 0 to `n`, but in terms of a prefix of the list `L` being traversed. This prefix ranges from `nil` to the full list `L`. The second difference is that the operation `miter f p` requires in its precondition, in addition to `I nil`, the description of the mutable list `MList L p`. This predicate is returned in the postcondition, unchanged, reflecting the fact that the iteration process does not alter the contents of the list.

Exercise: 5 stars, standard, especially useful (`triple_miter`) Prove the correctness of `triple_miter`.

```
Lemma triple_miter : ∀ (I:list val → hprop) L (f:val) p,
  (∀ x L1 L2, L = L1++x::L2 →
    triple (f x)
    (I L1)
    (fun u ⇒ I (L1++(x::nil))) →
    triple (miter f p)
    (MList L p \* I nil)
    (fun u ⇒ MList L p \* I L)).
```

Proof using. Admitted.

□

For exploiting the specification `triple_miter` to reason about a call to `miter`, it is necessary to provide an invariant `I` of type `list val → hprop`, that is, of the form `fun (K:list val) ⇒ ...`. This invariant, which cannot be inferred automatically, should describe the state at the point where the iteration has traversed a prefix `K` of the list `L`. Concretely, for reasoning about a call to `miter`, one should exploit the tactic `xapp (triple_iter (fun K ⇒ ...))`. An example appears next.

38.4.3 Computing the Length of a Mutable List using an Iterator

The function `mlength_using_miter` computes the length of a mutable list by iterating over that list a function that increments a reference cell once for every item.

OCaml:

```
let mlength_using_miter p = let c = ref 0 in miter (fun x -> incr c) p; !c
```

Exercise: 4 stars, standard, especially useful (`triple_mlength_using_miter`) Prove the correctness of `mlength_using_iter`. Hint: as explained earlier, use `xfun`; `intros f Hf` for reasoning about the function definition, then use `xapp` for reasoning about a call to `f`.

```
Definition mlength_using_miter : val :=
<{ fun 'p ⇒
```

```

let 'c = ref 0 in
let 'f = (fun_ 'x => incr 'c) in
  miter 'f 'p;
  get_and_free 'c }>.

```

Lemma triple_mlength_using_miter : $\forall p L,$
 $\text{triple} (\text{mlength_using_miter } p)$
 $(\text{MList } L p)$
 $(\text{fun } r \Rightarrow \exists [r = \text{length } L] \ \text{MList } L p).$

Proof using. *Admitted.*

□

38.4.4 A Continuation-Passing-Style, In-Place Concatenation Function

This section presents an example verification of a function involving “continuations”. The function `cps_append` is similar to the function `append` presented previously: it also performs in-place concatenation of two lists. The main difference is that it is implemented using an auxiliary recursive function in “continuation-passing style” (CPS).

The presentation of `cps_append p1 p2` is also slightly different: this operation returns a pointer `p3` that describes the head of the result of the concatenation. In the general case, `p3` is equal to `p1`, but if `p1` is the null pointer, meaning that the first list is empty, then `p3` is equal to `p2`.

The code of `cps_append` involves the auxiliary function `cps_append_aux p1 p2 k`, which invokes the continuation function `k` on the result of concatenating the lists at locations `p1` and `p2`. Its code appears at first quite puzzling, because the recursive call is performed inside the continuation. It takes a good drawing and at least several minutes to figure out how the function works.

OCaml:

```

let rec cps_append_aux p1 p2 k = if p1 == null then k p2 else cps_append_aux p1.tail
p2 (fun r => (p1.tail <- r); k p1)
let cps_append p1 p2 = cps_append_aux p1 p2 (fun r => r)

```

Definition `cps_append_aux : val :=`
`<{ fix 'f 'p1 'p2 'k =>`
 `let 'b = ('p1 = null) in`
 `if 'b`
 `then 'k 'p2`
 `else`
 `let 'q1 = 'p1'.tail in`
 `let 'k2 = (fun_ 'r => ('p1'.tail := 'r; 'k 'p1)) in`
 `'f 'q1 'p2 'k2 }>.`

Definition `cps_append : val :=`

```
<{ fun 'p1 'p2 =>
  let 'f = (fun_ 'r => 'r) in
  cps_append_aux 'p1 'p2 'f }>.
```

Exercise: 5 stars, standard, optional (triple_cps_append) Specify and verify `cps_append_aux`, then verify `cps_append`.

```
Lemma triple_cps_append : ∀ (L1 L2:list val) (p1 p2:loc),
  triple (cps_append p1 p2)
  (MList L1 p1 \* MList L2 p2)
  (funloc p3 => MList (L1++L2) p3).
```

Proof using. Admitted.

□

38.4.5 Historical Notes

The representation predicate for lists appears in the seminal papers on Separation Logic: the notes by Reynolds from the summer 1999, updated the next summer *Reynolds* 2000 (in Bib.v), and the publication by O'Hearn, Reynolds, and Yang 2001 (in Bib.v). The function `cps_append` was proposed in Reynolds's article as an open challenge for verification.

Most presentations of Separation Logic target partial correctness, whereas this chapters targets total correctness. The specifications of recursive functions are established using the built-in induction mechanism offered by Coq.

The specification of higher-order iterators requires higher-order Separation Logic. Being embedded in the higher-order logic of Coq, the Separation Logic that we work with is inherently higher-order. Further information on the history of higher-order Separation Logic for higher-order programs may be found in section 10.2 of http://www.chargueraud.org/research/2020/seq_sep.html

Chapter 39

Library `SLF.Hprop`

39.1 Hprop: Heap Predicates

Set Implicit Arguments.

From *SLF* Require Export LibSepReference.

Import ProgramSyntax.

Tweak to simplify the use of definitions and lemmas from *LibSepFmap.v*. *Arguments* `Fmap.single {A} {B}`.

Arguments `Fmap.union {A} {B}`.

Arguments `Fmap.disjoint {A} {B}`.

Arguments `Fmap.union_comm_of_disjoint {A} {B}`.

Arguments `Fmap.union_empty_l {A} {B}`.

Arguments `Fmap.union_empty_r {A} {B}`.

Arguments `Fmap.union_assoc {A} {B}`.

Arguments `Fmap.disjoint_empty_l {A} {B}`.

Arguments `Fmap.disjoint_empty_r {A} {B}`.

Arguments `Fmap.disjoint_union_eq_l {A} {B}`.

Arguments `Fmap.disjoint_union_eq_r {A} {B}`.

39.2 First Pass

In the programming language that we consider, a concrete memory state is described as a finite map from locations to values.

- A location has type `loc`.
- A value has type `val`.
- A state has type `state`.

Details will be presented in the chapter Rules.

To help distinguish between full states and pieces of state, we let the type `heap` be a synonymous for `state` but with the intention of representing only a piece of state. Throughout the course, we write `s` for a full memory state (of type `state`), and we write `h` for a piece of memory state (of type `heap`).

In Separation Logic, a piece of state is described by a “heap predicate”, i.e., a predicate over heaps. A heap predicate has type `hprop`, defined as `heap → Prop`, which is equivalent to `state → Prop`.

By convention, throughout the course:

- H denotes a heap predicate of type `hprop`; it describes a piece of state,
- Q denotes a postcondition, of type `val → hprop`; it describes both a result value and a piece of state. Observe that `val → hprop` is equivalent to `val → state → Prop`.

This chapter presents the definition of the key heap predicate operators from Separation Logic:

- $\setminus[]$ denotes the empty heap predicate,
- $\setminus[P]$ denotes a pure fact,
- $p \sim\!> v$ denotes a singleton heap,
- $H1 \setminus^* H2$ denotes the separating conjunction,
- $Q1 \setminus^*+ H2$ denotes the separating conjunction, between a postcondition and a heap predicate,
- $\setminus\exists x, H$ denotes an existential.

This chapter also introduces the formal definition of triples:

- a Hoare triple, written `hoare t H Q`, features a precondition H and a postcondition Q that describe the whole memory state in which the execution of the term `t` takes place.
- a Separation Logic triple, written `triple t H Q`, features a pre- and a postcondition that describes only the piece of the memory state in which the execution of the term `t` takes place.

This chapter and the following ones exploit a few additional TLC tactics to enable concise proofs.

- `applys` is an enhanced version of `eapply`.
- `applys_eq` is a variant of `applys` that enables matching the arguments of the predicate that appears in the goal “up to equality” rather than “up to conversion”.

- *specializes* is an enhanced version of `specialize`.
- *lets* and *forwards* are forward-chaining tactics that enable instantiating a lemma.

What these tactics do should be fairly intuitive where they are used. Note that all exercises can be carried out without using TLC tactics. For details, the chapter *UseTactics.v* from the “Programming Language Foundations” volume explains the behavior of these tactics.

39.2.1 Syntax and Semantics

We assume an imperative programming language with a formal semantics. We do not need to know about the details of the language construct for now. All we need to know is that there exists:

- a type of terms, written `trm`,
- a type of values, written `val`,
- a type of states, written `state` (i.e., finite map from `loc` to `val`),
- a big-step evaluation judgment, written `eval h1 t h2 v`, asserting that, starting from state `s1`, the evaluation of the term `t` terminates in the state `s2` and produces the value `v`.

`Check eval : state -> trm -> state -> val -> Prop.`

The corresponding definitions are described in the chapter [Rules](#).

At this point, we don’t need to know the exact grammar of terms and values. Let’s just give one example to make things concrete. Consider the function: `fun x => if x then 0 else 1`.

In the language that we consider, it can be written in raw syntax as follows.

```
Definition example_val : val :=
  val_fun "x" (trm_if (trm_var "x")
    (trm_val (val_int 0))
    (trm_val (val_int 1))).
```

Thanks to a set of coercions and notation, this term can be written in a somewhat more readable way, as follows.

```
Definition example_val' : trm :=
  <{ fun "x" =>
    if "x" then 0 else 1 }>.
```

39.2.2 Description of the State

Locations, of type `loc`, denote the addresses of allocated objects. Locations are a particular kind of values.

A state is a finite map from locations to values.

The file `LibSepFmap.v` provides a self-contained formalization of finite maps, but we do need to know about the details.

Definition `state : Type := fmap loc val.`

By convention, we use the type `state` describes a full state of memory, and introduce the type `heap` to describe just a piece of state.

Definition `heap : Type := state.`

In particular, the library `LibSepFmap.v`, whose module name is abbreviated as `FMAP`, exports the following definitions.

- `Fmap.empty` denotes the empty state,
- `Fmap.single p v` denotes a singleton state, that is, a unique cell at location `p` with contents `v`,
- `Fmap.union h1 h2` denotes the union of the two states `h1` and `h2`.
- `Fmap.disjoint h1 h2` asserts that `h1` and `h2` have disjoint domains.

The types of these definitions are as follows.

`Check Fmap.empty : heap.` `Check Fmap.single : loc -> val -> heap.` `Check Fmap.union : heap -> heap -> heap.` `Check Fmap.disjoint : heap -> heap -> Prop.`

Note that the union operation is commutative only when its two arguments have disjoint domains. Throughout the Separation Logic course, we will only consider disjoint unions, for which commutativity holds.

39.2.3 Heap Predicates

In Separation Logic, the state is described using “heap predicates”. A heap predicate is a predicate over a piece of state. Let `hprop` denote the type of heap predicates.

Definition `hprop := heap → Prop.`

Thereafter, let `H` range over heap predicates.

Implicit Type `H : hprop.`

An essential aspect of Separation Logic is that all heap predicates defined and used in practice are built using a few basic combinators. In other words, except for the definition of the combinators themselves, we will never define a custom heap predicate directly as a function of the state.

We next describe the most important combinators of Separation Logic.

The `hempty` predicate, usually written `\[]`, characterizes an empty state.

Definition `hempty : hprop :=`
`fun (h:heap) => (h = Fmap.empty).`

Notation "`\[]`" := (`hempty`) (at level 0).

The pure fact predicate, written `\[P]`, characterizes an empty state and moreover asserts that the proposition P is true.

Definition `hpure (P:Prop) : hprop :=`
`fun (h:heap) => (h = Fmap.empty) \wedge P.`

Notation "`\[P]`" := (`hpure P`) (at level 0, format "`\[P]`").

The singleton heap predicate, written `p $\sim\sim>$ v`, characterizes a state with a single allocated cell, at location `p`, storing the value `v`.

Definition `hsingle (p:loc) (v:val) : hprop :=`
`fun (h:heap) => (h = Fmap.single p v).`

Notation "`p $\sim\sim>$ v`" := (`hsingle p v`) (at level 32).

The “separating conjunction”, written `H1 * H2`, characterizes a state that can be decomposed in two disjoint parts, one that satisfies $H1$, and another that satisfies $H2$. In the definition below, the two parts are named `h1` and `h2`.

Definition `hstar (H1 H2 : hprop) : hprop :=`
`fun (h:heap) => \exists h1 h2, H1 h1`
 `\wedge H2 h2`
 `\wedge Fmap.disjoint h1 h2`
 `\wedge h = Fmap.union h1 h2.`

Notation "`H1 * H2`" := (`hstar H1 H2`) (at level 41, right associativity).

The existential quantifier for heap predicates, written `\exists x, H` characterizes a heap that satisfies H for some x . The variable x has type A , for some arbitrary type A .

The notation `\exists x, H` stands for `hexists (fun x => H)`. The generalized notation `\exists x1 ... xn, H` is also available.

The definition of `hexists` is a bit technical. It is not essential to master it at this point. Additional explanations are provided near the end of this chapter.

Definition `hexists (A:Type) (J:A \rightarrow hprop) : hprop :=`
`fun (h:heap) => \exists x, J x h.`

Notation "`'\exists' x1 .. xn , H`" :=
`(hexists (fun x1 => .. (hexists (fun xn => H) ..)))`
`(at level 39, x1 binder, H at level 50, right associativity,`
`format "['\exists' x1 .. xn , H]").`

All the definitions above are eventually turned `Opaque`, after the appropriate introduction and elimination lemmas are established for them. Thus, at some point it is no longer possible to involve, say `unfold hstar`. Opacity ensures that all the proofs that are constructed do not depend on the details of how these definitions of heap predicates are set up.

39.2.4 Extensionality for Heap Predicates

To work in Separation Logic, it is extremely convenient to be able to state equalities between heap predicates. For example, in the next chapter, we will establish the associativity property for the star operator, that is:

```
Parameter hstar_assoc : ∀ H1 H2 H3,
  (H1 \* H2) \* H3 = H1 \* (H2 \* H3).
```

How can we prove a goal of the form $H1 = H2$ when $H1$ and $H2$ have type `hprop`, that is, `heap → Prop`?

Intuitively, H and H' are equal if and only if they characterize exactly the same set of heaps, that is, if $\forall (h:\text{heap}), H1\ h \leftrightarrow H2\ h$.

This reasoning principle, a specific form of extensionality property, is not available by default in Coq. But we can safely assume it if we extend the logic of Coq with a standard axiom called “predicate extensionality”.

```
Axiom predicate_extensionality : ∀ (A:Type) (P Q:A→Prop),
  (∀ x, P x ↔ Q x) →
  P = Q.
```

By specializing P and Q above to the type `hprop`, we obtain exactly the desired extensionality principle.

```
Lemma hprop_eq : ∀ (H1 H2:hprop),
  (∀ (h:heap), H1\ h ↔ H2\ h) →
  H1 = H2.
```

Proof using. *applys predicate_extensionality.* Qed.

39.2.5 Type and Syntax for Postconditions

A postcondition characterizes both an output value and an output state. In Separation Logic, a postcondition is thus a relation of type `val → state → Prop`, which is equivalent to `val → hprop`.

Thereafter, we let Q range over postconditions.

Implicit Type $Q : \text{val} \rightarrow \text{hprop}$.

One common operation is to augment a postcondition with a piece of state. This operation is described by the operator $Q \backslash^*+ H$, which is just a convenient notation for `fun x ⇒ (Q x * H)`.

Notation "Q *+ H" := (fun x ⇒ hstar (Q x) H) (at level 40).

Intuitively, in order to prove that two postconditions $Q1$ and $Q2$ are equal, it suffices to show that the heap predicates $Q1 \vee$ and $Q2 \vee$ (both of type `hprop`) are equal for any value v .

Again, the extensionality property that we need is not built-in to Coq. We need the axiom called “functional extensionality”, stated next.

```
Axiom functional_extensionality : ∀ A B (f g:A→B),
  (∀ x, f x = g x) →
  f = g.
```

The desired equality property for postconditions follows directly from that axiom.

```
Lemma qprop_eq : ∀ (Q1 Q2:val→hprop),
  (∀ (v:val), Q1 v = Q2 v) →
  Q1 = Q2.
```

Proof using. *applys functional_extensionality.* Qed.

39.2.6 Separation Logic Triples and the Frame Rule

A Separation Logic triple is a generalization of a Hoare triple that integrate built-in support for an essential rule called “the frame rule”. Before we give the definition of a Separation Logic triple, let us first give the definition of a Hoare triple and state the much-desired frame rule.

A (total correctness) Hoare triple, written $\{H\} t \{Q\}$ on paper, and here written `hoare t H Q`, asserts that starting from a state s satisfying the precondition H , the term t evaluates to a value v and to a state s' that, together, satisfy the postcondition Q . It is formalized in Coq as shown below.

```
Definition hoare (t:trm) (H:hprop) (Q:val→hprop) : Prop :=
  ∀ (s:state), H s →
  ∃ (s':state) (v:val), eval s t s' v ∧ Q v s'.
```

Note that Q has type `val→hprop`, thus $Q v$ has type `hprop`. Recall that `hprop = heap→Prop`. Thus $Q v s'$ has type `Prop`.

The frame rule asserts that if one can derive a specification of the form `triple H t Q` for a term t , then one should be able to automatically derive `triple (H ∗ H') t (Q ∗+ H')` for any H' .

Intuitively, if t executes safely in a heap H , it should behave similarly in any extension of H with a disjoint part H' . Moreover, its evaluation should leave this piece of state H' unmodified throughout the execution of t .

The following definition of a Separation Logic triple builds upon that of a Hoare triple by “baking in” the frame rule.

```
Definition triple (t:trm) (H:hprop) (Q:val→hprop) : Prop :=
  ∀ (H':hprop), hoare t (H ∗ H') (Q ∗+ H').
```

This definition inherently satisfies the frame rule, as we show below. The proof only exploits the associativity of the star operator.

```
Lemma triple_frame : ∀ t H Q H',
  triple t H Q →
  triple t (H ∗ H') (Q ∗+ H').
```

Proof using.

*introv M. unfold triple in *. rename H' into H1. intros H2.*

```

specializes M (H1 \* H2).
applys_eq M.
{ rewrite hstar_assoc. auto. }
{ applys functional_extensionality. intros v. rewrite hstar_assoc. auto. }
Qed.

```

39.2.7 Example of a Triple: the Increment Function

Recall the function *incr* introduced in the chapter Basic.

Parameter *incr* : val.

An application of this function, written *incr p*, is technically a term of the form `trm_app (trm_val incr) (trm_val (val_loc p))`, where `trm_val` injects values in the grammar of terms, and `val_loc` injects locations in the grammar of locations.

The abbreviation *incr p* parses correctly because `trm_app`, `trm_val`, and `val_loc` are registered as coercions. Let us check this claim with Coq.

```

Lemma incr_applied : ∀ (p:loc) (n:int),
  trm_app (trm_val incr) (trm_val (val_loc p))
  = incr p.

```

Proof using reflexivity. Qed.

The operation *incr p* is specified using a triple as shown below.

```

Parameter triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
  (p ~~> n)
  (fun r ⇒ p ~~> (n+1)).

```

39.3 More Details

39.3.1 Example Applications of the Frame Rule

The frame rule asserts that a triple remains true in any extended heap.

Calling *incr p* in a state where the memory consists of two memory cells, one at location *p* storing an integer *n* and one at location *q* storing an integer *m* has the effect of updating the contents of the cell *p* to *n+1*, while leaving the contents of *q* unmodified.

```

Lemma triple_incr_2 : ∀ (p q:loc) (n m:int),
  triple (incr p)
  ((p ~~> n) \* (q ~~> m))
  (fun _ ⇒ (p ~~> (n+1)) \* (q ~~> m)).

```

The above specification lemma is derivable from the specification lemma *triple_incr* by applying the frame rule to augment both the precondition and the postcondition with *q ~~> m*.

Proof using.

```
intros. let M: triple_incr p n.
let N: triple_frame (q ~~> m) M. apply N.
Qed.
```

Here, we have framed on $q ~~> m$, but we could similarly frame on any heap predicate H , as captured by the following specification lemma.

```
Parameter triple_incr_3 : ∀ (p:loc) (n:int) (H:hprop),
triple (incr p)
((p ~~> n) \* H)
(fun _ => (p ~~> (n+1)) \* H).
```

Remark: in practice, we always prefer writing “small-footprint specifications”, such as *triple_incr*, that describe the minimal piece of state necessary for the function to execute. Indeed, other specifications that describe a larger piece of state can be derived by application of the frame rule.

39.3.2 Power of the Frame Rule with Respect to Allocation

Consider the specification lemma for an allocation operation. This rule states that, starting from the empty heap, one obtains a single memory cell at some location p with contents v .

```
Parameter triple_ref : ∀ (v:val),
triple (val_ref v)
[] []
(funloc p => p ~~> v).
```

Applying the frame rule to the above specification, and to another memory cell, say $l' ~~> v'$, we obtain:

```
Parameter triple_ref_with_frame : ∀ (l':loc) (v':val) (v:val),
triple (val_ref v)
(l' ~~> v')
(funloc p => p ~~> v \* l' ~~> v').
```

This derived specification captures the fact that the newly allocated cell at address p is distinct from the previously allocated cell at address l' .

More generally, through the frame rule, we can derive that any piece of freshly allocated data is distinct from any piece of previously existing data.

This independence principle is extremely powerful. It is an inherent strength of Separation Logic.

39.3.3 Notation for Heap Union

Thereafter, to improve readability of statements in proofs, we introduce the following notation for heap union.

```
Notation "h1 \u h2" := (Fmap.union h1 h2) (at level 37, right associativity).
```

39.3.4 Introduction and Inversion Lemmas for Basic Operators

The following lemmas help getting a better understanding of the meaning of the Separation Logic combinators. For each operator, we present one introduction lemma and one inversion lemma.

`Implicit Types $P : \text{Prop}$.`

`Implicit Types $v : \text{val}$.`

The introduction lemmas show how to prove goals of the form $H \ h$, for various forms of the heap predicate H .

`Lemma hempty_intro :`

`\[] Fmap.empty.`

`Proof using. hnf. auto. Qed.`

`Lemma hpure_intro : $\forall P,$`

`$P \rightarrow$`

`\[P] Fmap.empty.`

`Proof using. introv M. hnf. auto. Qed.`

`Lemma hsingle_intro : $\forall p v,$`

`($p \sim\sim v$) (Fmap.single p v).`

`Proof using. intros. hnf. auto. Qed.`

`Lemma hstar_intro : $\forall H1 H2 h1 h2,$`

`$H1 h1 \rightarrow$`

`$H2 h2 \rightarrow$`

`Fmap.disjoint h1 h2 \rightarrow`

`($H1 \setminus\ast H2$) ($h1 \setminus\cup h2$).`

`Proof using. intros. $\exists \times h1 h2.$ Qed.`

`Lemma hexists_intro : $\forall A (x:A) (J:A \rightarrow \text{hprop}) h,$`

`$J x h \rightarrow$`

`($\forall \exists x, J x$) h.`

`Proof using. introv M. hnf. eauto. Qed.`

The inversion lemmas show how to extract information from hypotheses of the form $H \ h$, for various forms of the heap predicate H .

`Lemma hempty_inv : $\forall h,$`

`\[] h \rightarrow`

`h = Fmap.empty.`

`Proof using. introv M. hnf in M. auto. Qed.`

`Lemma hpure_inv : $\forall P h,$`

`\[P] h \rightarrow`

`P \wedge h = Fmap.empty.`

`Proof using. introv M. hnf in M. autos \times . Qed.`

`Lemma hsingle_inv: $\forall p v h,$`

```

 $(p \sim\sim v) h \rightarrow$ 
 $h = \text{Fmap.single } p \ v.$ 
Proof using. introv M. hnf in M. auto. Qed.

Lemma hstar_inv :  $\forall H1 H2 h,$ 
 $(H1 \setminus* H2) h \rightarrow$ 
 $\exists h1 h2, H1 h1 \wedge H2 h2 \wedge \text{Fmap.disjoint } h1 h2 \wedge h = h1 \cup h2.$ 
Proof using. introv M. hnf in M. eauto. Qed.

Lemma hexists_inv :  $\forall A (J:A \rightarrow \text{hprop}) h,$ 
 $(\exists x, J x) h \rightarrow$ 
 $\exists x, J x h.$ 
Proof using. introv M. hnf in M. eauto. Qed.

```

Exercise: 4 stars, standard, especially useful (hstar_hpure_l) Prove that a heap h satisfies $\setminus[P] \setminus^* H$ if and only if P is true and h it satisfies H . The proof requires two lemmas on finite maps from *LibSepFmap.v*:

Check $\text{Fmap.union_empty_l} : \text{forall } h, \text{Fmap.empty } \cup h = h$.
 Check $\text{Fmap.disjoint_empty_l} : \text{forall } h, \text{Fmap.disjoint } \text{Fmap.empty } h$.
 Hint: begin the proof by applying *propositional_extensionality*.

```

Lemma hstar_hpure_l :  $\forall P H h,$ 
 $(\setminus[P] \setminus^* H) h = (P \wedge H h).$ 

```

Proof using. *Admitted.*

□

39.4 Optional Material

39.4.1 Alternative, Equivalent Definitions for Separation Logic Triples

We have previously defined triple on top of hoare, with the help of the separating conjunction operator, as: $\forall (H':\text{hprop}), \text{hoare } (H \setminus^* H') t (Q \setminus^+ H')$. In what follows, we give an equivalent characterization, expressed directly in terms of heaps and heap unions.

The alternative definition of triple $t H Q$ asserts that if $h1$ satisfies the precondition H and $h2$ describes the rest of the state, then the evaluation of t produces a value v in a final state made that can be decomposed between a part $h1'$ and $h2$ unchanged, in such a way that v and $h1'$ together satisfy the postcondition Q . Formally:

```

Definition triple_lowlevel (t:trm) (H:hprop) (Q:val → hprop) : Prop :=
 $\forall h1 h2,$ 
 $\text{Fmap.disjoint } h1 h2 \rightarrow$ 
 $H h1 \rightarrow$ 
 $\exists h1' v,$ 
 $\text{Fmap.disjoint } h1' h2$ 

```

$\wedge \text{eval } (h1 \setminus u h2) t (h1' \setminus u h2) v$
 $\wedge Q v h1'.$

Let us establish the equivalence between this alternative definition of `triple` and the original one.

Exercise: 4 stars, standard, especially useful (`triple_iff_triple_lowlevel`) Prove the equivalence between `triple` and `triple_low_level`. Warning: this is probably a very challenging exercise.

Lemma `triple_iff_triple_lowlevel` : $\forall t H Q,$
 $\text{triple } t H Q \leftrightarrow \text{triple_lowlevel } t H Q.$

Proof using. *Admitted.*

□

39.4.2 Alternative Definitions for Heap Predicates

In what follows, we discuss alternative, equivalent definitions for the fundamental heap predicates. We write these equivalence using equalities of the form $H1 = H2$. Recall that lemma `hprop_eq` enables deriving such equalities by invoking predicate extensionality.

The empty heap predicate `\[]` is equivalent to the pure fact predicate `\[True]`.

Lemma `hempty_eq_hpure_true` :

`\[] = \[True].`

Proof using.

```
unfold hempty, hpure. apply hprop_eq. intros h. iff Hh.
{ auto. }
{ jauto. }
```

Qed.

Thus, `hempty` could be defined in terms of `hpure`, as `hpure True`, written `\[True]`.

Definition `hempty' : hprop :=`

`\[True].`

The pure fact predicate `\[P]` is equivalent to the existential quantification over a proof of P in the empty heap, that is, to the heap predicate $\exists (p:P), \[]$.

Lemma `hpure_eq_hexists_proof` : $\forall P,$

`\[P] = (\exists (p:P), \[]).`

Proof using.

```
unfold hempty, hpure, hexists. intros P.
apply hprop_eq. intros h. iff Hh.
{ destruct Hh as (E&p). ∃ p. auto. }
{ destruct Hh as (p&E). auto. }
```

Qed.

Thus, `hpure` could be defined in terms of `hexists` and `hempty`, as `hexists (fun (p:P) => hempty)`, also written $\exists (p:P), \text{[]}$.

Definition `hpure'` ($P:\text{Prop}$) : `hprop` :=
 $\exists (p:P), \text{[]}.$

It is useful to minimize the number of combinators, both for elegance and to reduce the proof effort.

Since we cannot do without `hexists`, we have a choice between considering either `hpure` or `hempty` as primitive, and the other one as derived. The predicate `hempty` is simpler and appears as more fundamental.

Hence, in the subsequent chapters (and in the CFML tool), we define `hpure` in terms of `hexists` and `hempty`, like in the definition of `hpure'` shown above. In other words, we assume the definition:

Definition `hpure` ($P:\text{Prop}$) : `hprop` := $\exists (p:P), \square$.

39.4.3 Additional Explanations for the Definition of \exists

The heap predicate $\exists (n:\text{int}), p \sim\sim > (\text{val_int } n)$ characterizes a state that contains a single memory cell, at address p , storing the integer value n , for “some” (unspecified) integer n .

Parameter ($p:\text{loc}$). Check $(\exists (n:\text{int}), p \sim\sim > (\text{val_int } n)) : \text{hprop}$.

The type of \exists , which operates on `hprop`, is very similar to that of \exists , which operates on `Prop`.

The notation $\exists x, P$ stands for `ex` ($\text{fun } x \Rightarrow P$), where `ex` has the following type:

Check `ex` : forall $A : \text{Type}$, $(A \rightarrow \text{Prop}) \rightarrow \text{Prop}$.

Likewise, $\exists x, H$ stands for `hexists` ($\text{fun } x \Rightarrow H$), where `hexists` has the following type:

Check `hexists` : forall $A : \text{Type}$, $(A \rightarrow \text{hprop}) \rightarrow \text{hprop}$.

Remark: the notation for \exists is directly adapted from that of \exists , which supports the quantification an arbitrary number of variables, and is defined in `Coq.Init.Logic` as follows.

Notation “exists’ $x .. y, p$ ” := $(\text{ex} (\text{fun } x \Rightarrow .. (\text{ex} (\text{fun } y \Rightarrow p) ..)))$ (at level 200, x binder, right associativity, format “‘exists’ ‘/’ $x .. y, /’ p$ ”).

39.4.4 Formulation of the Extensionality Axioms

Module EXTENSIONALITY.

To establish extensionality of entailment, we have used the predicate extensionality axiom. In fact, this axiom can be derived by combining the axiom of “functional extensionality” with another one called “propositional extensionality”.

The axiom of “propositional extensionality” asserts that two propositions that are logically equivalent (in the sense that they imply each other) can be considered equal.

Axiom `propositional_extensionality` : $\forall (P \ Q:\text{Prop}),$
 $(P \leftrightarrow Q) \rightarrow$
 $P = Q.$

The axiom of “functional extensionality” asserts that two functions are equal if they provide equal result for every argument.

Axiom functional_extensionality : $\forall A B (f g:A \rightarrow B),$
 $(\forall x, f x = g x) \rightarrow$
 $f = g.$

Exercise: 1 star, standard, especially useful (**predicate_extensionality_derived**)
Using the two axioms *propositional_extensionality* and *functional_extensionality*, show how to derive *predicate_extensionality*.

Lemma predicate_extensionality_derived : $\forall A (P Q:A \rightarrow \text{Prop}),$
 $(\forall x, P x \leftrightarrow Q x) \rightarrow$
 $P = Q.$

Proof using. *Admitted.*

□

End EXTENSIONALITY.

39.4.5 Historical Notes

In this chapter, we defined the predicate `triple t H Q` for Separation Logic triples on top of the predicate `hoare t H Q` for Hoare triples, by quantifying universally on a heap predicate `H'`, which describes the “rest of the word”. This technique, known as the “baked-in frame rule”, was introduced by *Birkedal, Torp-Smith and Yang 2006* (in Bib.v), who developed the first Separation Logic for a higher-order programming language. It was later employed successfully in numerous formalizations of Separation Logic.

Compared to the use of a “low-level” definition of Separation Logic triples such as the predicate `triple_lowlevel`, which quantifies over disjoint pieces of heaps, the “high-level” definition that bakes in the frame rule leads to more elegant, simpler proofs.

Chapter 40

Library `SLF.Himpl`

40.1 `Himpl`: Heap Entailment

Foundations of Separation Logic

Chapter: “`Himpl`”.

Author: Arthur Charguéraud. License: CC-by 4.0.

Set Implicit Arguments.

From *SLF* Require LibSepReference.

From *SLF* Require Export Hprop.

Implicit Types

Implicit Types $P : \text{Prop}$.

Implicit Types $H : \text{hprop}$.

Implicit Types $Q : \text{val} \rightarrow \text{hprop}$.

40.2 First Pass

In the previous chapter, we have introduced the key heap predicate operators, and we have defined the notion of Separation Logic triple.

Before we can state and prove reasoning rules for establishing triples, we need to introduce the “entailment relation”. This relation, written $H1 ==> H2$, asserts that any heap that satisfies $H1$ also satisfies $H2$.

We also need to extend the entailment relation to postconditions. We write $Q1 ===> Q2$ to asserts that, for any result value v , the entailment $Q1 v ==> Q2 v$ holds.

The two entailment relations appear in the statement of the rule of consequence, which admits the same statement in Separation Logic as in Hoare logic. It asserts that precondition can be strengthened and postcondition can be weakened in a specification triple.

Lemma triple_conseq : forall t H Q H' Q', triple t H' Q' -> H ==> H' -> Q' ===> Q -> triple t H Q.

This chapter presents:

- the formal definition of the entailment relations,
- the fundamental properties of the Separation Logic operators: these properties are expressed either as entailments, or as equalities, which denote symmetric entailments,
- the 4 structural rules of Separation Logic: the rule of consequence, the frame rule (which can be combined with the rule of consequence), and the extractions rules for pure facts and for quantifiers,
- the tactics *ximpl* and *xchange* that are critically useful for manipulating entailments in practice,
- (optional) details on how to prove the fundamental properties and the structural rules.

40.2.1 Definition of Entailment

The “entailment relationship” $H1 ==> H2$ asserts that any heap h that satisfies the heap predicate $H1$ also satisfies the heap predicate $H2$.

```
Definition himpl (H1 H2:hprop) : Prop :=
  ∀ (h:heap), H1 h → H2 h.
```

Notation "H1 ==> H2" := (himpl H1 H2) (at level 55).

$H1 ==> H2$ captures the fact that $H1$ is a stronger precondition than $H2$, in the sense that it is more restrictive.

As we show next, the entailment relation is reflexive, transitive, and antisymmetric. It thus forms an order relation on heap predicates.

```
Lemma himpl_refl : ∀ H,
  H ==> H.
Proof using. intros h. hnf. auto. Qed.
```

```
Lemma himpl_trans : ∀ H2 H1 H3,
  (H1 ==> H2) →
  (H2 ==> H3) →
  (H1 ==> H3).
```

Proof using. introv M1 M2. intros h H1h. eauto. Qed.

Exercise: 1 star, standard, especially useful (himpl_antisym) Prove the antisymmetry of entailment result shown below using extensionality for heap predicates, as captured by lemma *predicate_extensionality* (or lemma *hprop_eq*) introduced in the previous chapter (*Hprop*).

```
Lemma himpl_antisym : ∀ H1 H2,
  (H1 ==> H2) →
  (H2 ==> H1) →
```

$H1 = H2$.

Proof using. Admitted.

□

Remark: as the proof scripts show, the fact that entailment on `hprop` constitutes an order relation is a direct consequence of the fact that implication on `Prop`, that is, \rightarrow , is an order relation on `Prop` (when assuming the propositional extensionality axiom).

The lemma `himpl_antisym` may, for example, be used to establish commutativity of separating conjunction: $(H1 \setminus * H2) = (H2 \setminus * H1)$ by proving that each side entails the other side, that is, by proving $(H1 \setminus * H2) ==> (H2 \setminus * H1)$ and $(H2 \setminus * H1) ==> (H1 \setminus * H2)$.

40.2.2 Entailment for Postconditions

The entailment $==>$ relates heap predicates. It is used to capture that a precondition “entails” another one. We need a similar judgment to assert that a postcondition “entails” another one.

For that purpose, we introduce $Q1 ==> Q2$, which asserts that for any value v , the heap predicate $Q1 v$ entails $Q2 v$.

Definition `qimpl` ($Q1\ Q2:\text{val} \rightarrow \text{hprop}$) : `Prop` :=
 $\forall (v:\text{val}), Q1\ v ==> Q2\ v$.

Notation "Q1 ==> Q2" := (`qimpl` $Q1\ Q2$) (at level 55).

Remark: equivalently, $Q1 ==> Q2$ holds if for any value v and any heap h , the proposition $Q1 v h$ implies $Q2 v h$.

Entailment on postconditions also forms an order relation: it is reflexive, transitive, and antisymmetric.

Lemma `qimpl_refl` : $\forall Q,$

$Q ==> Q$.

Proof using. intros $Q\ v$. applys `himpl_refl`. Qed.

Lemma `qimpl_trans` : $\forall Q2\ Q1\ Q3,$

$(Q1 ==> Q2) \rightarrow$

$(Q2 ==> Q3) \rightarrow$

$(Q1 ==> Q3)$.

Proof using. introv $M1\ M2$. intros v . applys `himpl_trans`; eauto. Qed.

Lemma `qimpl_antisym` : $\forall Q1\ Q2,$

$(Q1 ==> Q2) \rightarrow$

$(Q2 ==> Q1) \rightarrow$

$(Q1 = Q2)$.

Proof using.

introv $M1\ M2$. apply *functional_extensionality*.

intros v . applys `himpl_antisym`; eauto.

Qed.

40.2.3 Fundamental Properties of Separation Logic Operators

The 5 fundamental properties of Separation Logic operators are described next. Many other properties are derivable from those.

(1) The star operator is associative.

`Parameter hstar_assoc : ∀ H1 H2 H3,`

$$(H1 \ast H2) \ast H3 = H1 \ast (H2 \ast H3).$$

(2) The star operator is commutative.

`Parameter hstar_comm : ∀ H1 H2,`

$$H1 \ast H2 = H2 \ast H1.$$

(3) The empty heap predicate is a neutral for the star. Because star is commutative, it is equivalent to state that `hempty` is a left or a right neutral for `hstar`. We chose, arbitrarily, to state the left-neutral property.

`Parameter hstar_hempty_l : ∀ H,`

$$\emptyset \ast H = H.$$

(4) Existentials can be “extruded” out of stars, that is: $(\exists x, J x) \ast H$ is equivalent to $\exists x, (J x \ast H)$.

`Parameter hstar_hexists : ∀ A (J:A → hprop) H,`

$$(\exists x, J x) \ast H = \exists x, (J x \ast H).$$

(5) The star operator is “monotone” with respect to entailment, meaning that if $H1 ==> H1'$ then $(H1 \ast H2) ==> (H1' \ast H2)$.

Viewed the other way around, if we have to prove the entailment relation $(H1 \ast H2) ==> (H1' \ast H2)$, we can “cancel out” $H2$ on both sides. In this view, the monotonicity property is a sort of “frame rule for the entailment relation”.

Here again, due to commutativity of star, it only suffices to state the left version of the monotonicity property.

`Parameter himpl_frame_l : ∀ H2 H1 H1',`

$$H1 ==> H1' \rightarrow$$

$$(H1 \ast H2) ==> (H1' \ast H2).$$

Exercise: 1 star, standard, especially useful (`hstar_comm_assoc`) The commutativity and associativity results combine into one result that is sometimes convenient to exploit in proofs.

`Lemma hstar_comm_assoc : ∀ H1 H2 H3,`

$$H1 \ast H2 \ast H3 = H2 \ast H1 \ast H3.$$

`Proof using. Admitted.`

□

Exercise: 1 star, standard, especially useful (`himpl_frame_r`) The monotonicity property of the star operator w.r.t. entailment can also be stated in a symmetric fashion, as shown next. Prove this result. Hint: exploit the transitivity of entailment (`himpl_trans`) and the asymmetric monotonicity result (`himpl_frame_l`).

`Lemma himpl_frame_r : ∀ H1 H2 H2',
 $H2 ==> H2' \rightarrow$
 $(H1 \setminus * H2) ==> (H1 \setminus * H2').$`

`Proof using.` *Admitted.*

□

Exercise: 1 star, standard, especially useful (`himpl_frame_lr`) The monotonicity property of the star operator w.r.t. entailment can also be stated in a symmetric fashion, as shown next. Prove this result. Hint: exploit the transitivity of entailment (`himpl_trans`) and the asymmetric monotonicity result (`himpl_frame_l`).

`Lemma himpl_frame_lr : ∀ H1 H1' H2 H2',
 $H1 ==> H1' \rightarrow$
 $H2 ==> H2' \rightarrow$
 $(H1 \setminus * H2) ==> (H1' \setminus * H2').$`

`Proof using.` *Admitted.*

□

40.2.4 Introduction and Elimination Rules w.r.t. Entailments

The rules for introducing and eliminating pure facts and existential quantifiers in entailments are essential. They are presented next.

Consider an entailment of the form $H ==> (\setminus [P] \setminus * H')$. To establish this entailment, one must prove that P is true, and that H entails H' .

Exercise: 2 stars, standard, especially useful (`himpl_hstar_hpure_r`). Prove the rule `himpl_hstar_hpure_r`. Hint: recall from `Hprop` the lemma `hstar_hpure_l`, which asserts the equality $(\setminus [P] \setminus * H) \text{ h} = (P \wedge H \text{ h})$.

`Lemma himpl_hstar_hpure_r : ∀ P H H',
 $P \rightarrow$
 $(H ==> H') \rightarrow$
 $H ==> (\setminus [P] \setminus * H').$`

`Proof using.` *Admitted.*

□

Reciprocally, consider an entailment of the form $(\setminus [P] \setminus * H) ==> H'$. To establish this entailment, one must prove that H entails H' under the assumption that P is true.

Indeed, the former proposition asserts that if a heap \mathbf{h} satisfies H and that P is true, then \mathbf{h} satisfies H' , while the latter asserts that if P is true, then if a heap \mathbf{h} satisfies H it also satisfies H' .

The “extraction rule for pure facts in the left of an entailment” captures the property that the pure fact $\setminus[P]$ can be extracted into the Coq context. It is stated as follows.

Exercise: 2 stars, standard, especially useful (himpl_hstar_hpure_l). Prove the rule `himpl_hstar_hpure_l`.

Lemma `himpl_hstar_hpure_l` : $\forall (P:\text{Prop}) (H H':\text{hprop}),$

$(P \rightarrow H ==> H') \rightarrow$

$(\setminus[P] \setminus * H) ==> H'.$

Proof using. *Admitted.*

□

Consider an entailment of the form $H ==> (\exists x, J x)$, where x has some type A and J has type $A \rightarrow \text{hprop}$. To establish this entailment, one must exhibit a value for x for which H entails $J x$.

Exercise: 2 stars, standard, especially useful (himpl_hexists_r). Prove the rule `himpl_hexists_r`.

Lemma `himpl_hexists_r` : $\forall A (x:A) H J,$

$(H ==> J x) \rightarrow$

$H ==> (\exists x, J x).$

Proof using. *Admitted.*

□

Reciprocally, consider an entailment $(\exists x, (J x)) ==> H$. To establish this entailment, one has to prove that, whatever the value of x , the predicate $J x$ entails H .

Indeed the former proposition asserts that if a heap \mathbf{h} satisfies $J x$ for some x , then \mathbf{h} satisfies H' , while the latter asserts that if, for some x , the predicate \mathbf{h} satisfies $J x$, then \mathbf{h} satisfies H' .

Observe how the existential quantifier on the left of the entailment becomes an universal quantifier outside of the arrow.

The “extraction rule for existentials in the left of an entailment” captures the property that existentials can be extracted into the Coq context. It is stated as follows.

Exercise: 2 stars, standard, especially useful (himpl_hexists_l). Prove the rule `himpl_hexists_l`.

Lemma `himpl_hexists_l` : $\forall (A:\text{Type}) (H:\text{hprop}) (J:A \rightarrow \text{hprop}),$

$(\forall x, J x ==> H) \rightarrow$

$(\exists x, J x) ==> H.$

Proof using. *Admitted.*

□

40.2.5 Extracting Information from Heap Predicates

We next present an example showing how entailments can be used to state lemmas allowing to extract information from particular heap predicates. We show that from a heap predicate of the form $(p \sim\sim> v1) \setminus^* (p \sim\sim> v2)$ describes two “disjoint” cells that are both “at location p”, one can extract a contradiction.

Indeed, such a state cannot exist. The underlying contraction is formally captured by the following entailment relation, which concludes **False**.

```
Lemma hstar_hsingl_same_loc : ∀ (p:loc) (v1 v2:val),
  (p ∼\sim> v1) \setminus^* (p ∼\sim> v2) ==> \[False].
```

The proof of this result exploits a result on finite maps. Essentially, the domain of a single singleton map that binds a location p to some value is the singleton set \set{p} , thus such a singleton map cannot be disjoint from another singleton map that binds the same location p .

Check disjoint_single_single_same_inv : forall (p:loc) (v1 v2:val), Fmap.disjoint (Fmap.single p v1) (Fmap.single p v2) -> False.

Using this lemma, we can prove `hstar_hsingl_same_loc` by unfolding the definition of `hstar` to reveal the contradiction on the disjointness assumption.

Proof using.

```
intros. unfold hsingl. intros h (h1&h2&E1&E2&D&E). false.
subst. applys Fmap.disjoint_single_single_same_inv D.
```

Qed.

More generally, a heap predicate of the form $H \setminus^* H$ is generally suspicious in Separation Logic. In (the simplest variant of) Separation Logic, such a predicate can only be satisfied if H covers no memory cell at all, that is, if H is a pure predicate of the form \set{P} for some proposition P .

40.2.6 Consequence, Frame, and their Combination

The rule of consequence in Separation Logic is similar to that in Hoare logic.

```
Parameter triple_conseq : ∀ t H Q H' Q',
  triple t H' Q' →
  H ==> H' →
  Q' ===> Q →
  triple t H Q.
```

Recall the frame rule introduced in the previous chapter.

```
Parameter triple_frame : ∀ t H Q H',
  triple t H Q →
  triple t (H \setminus^* H') (Q \setminus^+ H').
```

Observe that, stated as such, it is very unlikely for the frame rule to be applicable in practice, because the precondition must be exactly of the form $H \setminus^* H'$ and the postcondition

exactly of the form $Q \setminus^* H'$, for some H' . For example, the frame rule would not apply to a proof obligation of the form $\text{triple } t (H' \setminus^* H) (Q \setminus^* H')$, simply because $H' \setminus^* H$ does not match $H \setminus^* H'$.

This limitation of the frame rule can be addressed by combining the frame rule with the rule of consequence into a single rule, called the “consequence-frame rule”. This rule, shown below, enables deriving a triple from another triple, without any restriction on the exact shape of the precondition and postcondition of the two triples involved.

Lemma `triple_conseq_frame` : $\forall H2\ H1\ Q1\ t\ H\ Q,$

$$\begin{aligned} \text{triple } t\ H1\ Q1 &\rightarrow \\ H ==> H1 \setminus^* H2 &\rightarrow \\ Q1 \setminus^{*+} H2 ==> Q &\rightarrow \\ \text{triple } t\ H\ Q. \end{aligned}$$

Exercise: 1 star, standard, especially useful (`triple_conseq_frame`) Prove the combined consequence-frame rule.

Proof using. *Admitted.*

□

40.2.7 The Extraction Rules for Triples

A judgment of the form $\text{triple } t ([P] \setminus^* H) Q$ is equivalent to $P \rightarrow \text{triple } t\ H\ Q$. This structural rule is called `triple_hpure` and formalized as shown below. It captures the extraction of the pure facts out of the precondition of a triple, in a similar way as `himpl_hstar_hpure_l` for entailments.

Parameter `triple_hpure` : $\forall t\ (P:\text{Prop})\ H\ Q,$
 $(P \rightarrow \text{triple } t\ H\ Q) \rightarrow$
 $\text{triple } t\ (\setminus [P]\ \setminus^* H)\ Q.$

A judgment of the form $\text{triple } t (\exists x, J x) Q$ is equivalent to $\forall x, \text{triple } t (J x) Q$. This structural rule is called `triple_hexists` and formalized as shown below. It captures the extraction of an existential quantifier out of the precondition of a triple, in a similar way as `himpl_hexists_l` for entailments.

Parameter `triple_hexists` : $\forall t\ (A:\text{Type})\ (J:A \rightarrow \text{hprop})\ Q,$
 $(\forall x, \text{triple } t (J x) Q) \rightarrow$
 $\text{triple } t (\setminus \exists x, J x)\ Q.$

40.3 More Details

Module XSIMPLTACTIC.

Import LibSepReference.

Notation "'hprop'" := (Hprop.hprop).

40.3.1 Identifying True and False Entailments

Module CASESTUDY.

Implicit Types $p q : \text{loc}$.

Implicit Types $n m : \text{int}$.

End CASESTUDY.

40.3.2 Proving Entailments by Hand

Module ENTAILMENTPROOFS.

Implicit Types $p : \text{loc}$.

Implicit Types $n : \text{int}$.

Proving an entailment by hand is generally a tedious task. This is why most Separation Logic based framework include an automated tactic for simplifying entailments. In this course, the relevant tactic is named *xsimpl*. Further in this chapter, we describe by means of examples the behavior of this tactic. But in order to best appreciate what the tactic provides and best understand how it works, it is very useful to complete a few proofs by hand.

Exercise: 3 stars, standard, optional (himpl_example_1) Prove the example entailment below. Hint: exploit `hstar_comm`, `hstar_assoc`, or `hstar_comm_assoc` which combines the two, and exploit `himpl_frame_l` or `himpl_frame_r` to cancel out matching pieces.

```
Lemma himpl_example_1 : ∀ p1 p2 p3 p4,
  p1 ~~> 6 ∗ p2 ~~> 7 ∗ p3 ~~> 8 ∗ p4 ~~> 9
  ==> p4 ~~> 9 ∗ p3 ~~> 8 ∗ p2 ~~> 7 ∗ p1 ~~> 6.
```

Proof using. Admitted.

□

Exercise: 3 stars, standard, optional (himpl_example_2) Prove the example entailment below. Hint: use `himpl_hstar_hpure_r` to extract pure facts, once they appear at the head of the left-hand side of the entailment.

```
Lemma himpl_example_2 : ∀ p1 p2 p3 n,
  p1 ~~> 6 ∗ [n > 0] ∗ p2 ~~> 7 ∗ [n < 0]
  ==> p3 ~~> 8.
```

Proof using. Admitted.

□

Exercise: 3 stars, standard, optional (himpl_example_3) Prove the example entailment below. Hint: use lemmas among `himpl_hexists_r`, `himpl_hexists_l`, `himpl_hstar_hpure_r` and `himpl_hstar_hpure_r` to deal with pure facts and quantifiers.

```
Lemma himpl_example_3 : ∀ p,
```

```
\exists n, p ~~> n * \[n > 0]
==> \exists n, \[n > 1] * p ~~> (n-1).
```

Proof using. Admitted.

□

End ENTAILMENTPROOFS.

40.3.3 The *xsimpl* Tactic

Performing manual simplifications of entailments by hand is an extremely tedious task. Fortunately, it can be automated using specialized Coq tactic. This tactic, called *xsimpl*, applies to an entailment and implements a 3-step process:

1. extract pure facts and existential quantifiers from the LHS,
2. cancel out equal predicates occurring both in the LHS and RHS,
3. generate subgoals for the pure facts occurring in the RHS, and instantiate the existential quantifiers from the RHS (using either unification variables or user-provided hints).

These steps are detailed and illustrated next.

The tactic *xpull* is a degraded version of *xsimpl* that only performs the first step. We will also illustrate its usage.

xsimpl to Extract Pure Facts and Quantifiers in LHS

The first feature of *xsimpl* is its ability to extract the pure facts and the existential quantifiers from the left-hand side out into the Coq context.

In the example below, the pure fact $n > 0$ appears in the LHS. After calling *xsimpl*, this pure fact is turned into an hypothesis, which may be introduced with a name into the Coq context.

```
Lemma xsimpl_demo_lhs_hpure : \forall H1 H2 H3 H4 (n:int),
  H1 * H2 * \[n > 0] * H3 ==> H4.
```

Proof using.

```
  intros. xsimpl. intros Hn.
```

Abort.

In case the LHS includes a contradiction, such as the pure fact **False**, the goal gets solved immediately by *xsimpl*.

```
Lemma xsimpl_demo_lhs_hpure : \forall H1 H2 H3 H4,
  H1 * H2 * \[False] * H3 ==> H4.
```

Proof using.

```
  intros. xsimpl.
```

Qed.

The *xsimpl* tactic also extracts existential quantifier from the LHS. It turns them into universally quantified variables outside of the entailment relation, as illustrated through the following example.

```
Lemma xsimpl_demo_lhs_hexists : ∀ H1 H2 H3 H4 (p:loc),
  H1 \* ∃ (n:int), (p ~> n \* H2) \* H3 ==> H4.
```

Proof using.

```
  intros. xsimpl. intros n.
```

Abort.

A call to *xsimpl*, or to its degraded version *xpull*, extract at once all the pure facts and quantifiers from the LHS, as illustrated next.

```
Lemma xsimpl_demo_lhs_several : ∀ H1 H2 H3 H4 (p q:loc),
  H1 \* ∃ (n:int), (p ~> n \* \[n > 0] \* H2) \* \[p ≠ q] \* H3 ==> H4.
```

Proof using.

```
  intros.
```

```
  xsimpl. intros n Hn Hp.
```

Abort.

xsimpl to Cancel Out Heap Predicates from LHS and RHS

The second feature of *xsimpl* is its ability to cancel out similar heap predicates that occur on both sides of an entailment.

In the example below, *H2* occurs on both sides, so it is canceled out by *xsimpl*.

```
Lemma xsimpl_demo_cancel_one : ∀ H1 H2 H3 H4 H5 H6 H7,
  H1 \* H2 \* H3 \* H4 ==> H5 \* H6 \* H2 \* H7.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

xsimpl actually cancels out at once all the heap predicates that it can spot appearing on both sides. In the example below, *H2*, *H3*, and *H4* are canceled out.

```
Lemma xsimpl_demo_cancel_many : ∀ H1 H2 H3 H4 H5,
  H1 \* H2 \* H3 \* H4 ==> H4 \* H3 \* H5 \* H2.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

If all the pieces of heap predicate get canceled out, the remaining proof obligation is $\[] ==> \[]$. In this case, *xsimpl* automatically solves the goal by invoking the reflexivity property of entailment.

```
Lemma xsimpl_demo_cancel_all : ∀ H1 H2 H3 H4,
  H1 \* H2 \* H3 \* H4 ==> H4 \* H3 \* H1 \* H2.
```

Proof using.

```
  intros. xsimpl.
```

Qed.

xsimpl to Instantiate Pure Facts and Quantifiers in RHS

The third feature of *xsimpl* is its ability to extract pure facts from the RHS as separate subgoals, and to instantiate existential quantifiers from the RHS.

Let us first illustrate how it deals with pure facts. In the example below, the fact $n > 0$ gets spawned in a separated subgoal.

```
Lemma xsimpl_demo_rhs_hpure : ∀ H1 H2 H3 (n:int),
  H1 ==> H2 \* \[n > 0] \* H3.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

When it encounters an existential quantifier in the RHS, the *xsimpl* tactic introduces a unification variable denoted by a question mark, that is, an “evar”, in Coq terminology. In the example below, the *xsimpl* tactic turns $\exists n, .. p \sim\sim> n ..$ into $.. p \sim\sim> ?x ..$

```
Lemma xsimpl_demo_rhs_hexists : ∀ H1 H2 H3 H4 (p:loc),
  H1 ==> H2 \* \exists (n:int), (p \sim\sim> n \* H3) \* H4.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

The “evar” often gets subsequently instantiated as a result of a cancellation step. For example, in the example below, *xsimpl* instantiates the existentially quantified variable n as $?x$, then cancels out $p \sim\sim> ?x$ from the LHS against $p \sim\sim> 3$ on the right-hand-side, thereby unifying $?x$ with 3.

```
Lemma xsimpl_demo_rhs_hexists_unify : ∀ H1 H2 H3 H4 (p:loc),
  H1 \* (p \sim\sim> 3) ==> H2 \* \exists (n:int), (p \sim\sim> n \* H3) \* H4.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

The instantiation of the evar $?x$ can be observed if there is another occurrence of the same variable in the entailment. In the next example, which refines the previous one, observe how $n > 0$ becomes $3 > 0$.

```
Lemma xsimpl_demo_rhs_hexists_unify_view : ∀ H1 H2 H4 (p:loc),
  H1 \* (p \sim\sim> 3) ==> H2 \* \exists (n:int), (p \sim\sim> n \* \[n > 0]) \* H4.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

(Advanced.) In some cases, it is desirable to provide an explicit value for instantiating an existential quantifier that occurs in the RHS. The *xsimpl* tactic accepts arguments, which will be used to instantiate the existentials (on a first-match basis). The syntax is *xsimpl v1 .. vn*, or *xsimpl (» v1 .. vn)* in the case $n > 3$.

```
Lemma xsimpl_demo_rhs_hints : ∀ H1 (p q:loc),
  H1 ==> ∃ (n m:int), (p ~~> n ∗ q ~~> m).
```

Proof using.

```
  intros. xsimpl 3 4.
```

Abort.

(Advanced.) If two existential quantifiers quantify variables of the same type, it is possible to provide a value for only the second quantifier by passing as first argument to *xsimpl* the special value `_`. The following example shows how, on LHS of the form $\exists n m, \dots$, the tactic *xsimpl* `_` 4 instantiates `m` with 4 while leaving `n` as an unresolved evar.

```
Lemma xsimpl_demo_rhs_hints_evar : ∀ H1 (p q:loc),
  H1 ==> ∃ (n m:int), (p ~~> n ∗ q ~~> m).
```

Proof using.

```
  intros. xsimpl _ 4.
```

Abort.

xsimpl on Entailments Between Postconditions

The tactic *xsimpl* also applies on goals of the form $Q1 ==> Q2$.

For such goals, it unfolds the definition of $==>$ to reveal an entailment of the form $==>$, then invokes the *xsimpl* tactic.

```
Lemma qimpl_example_1 : ∀ (Q1 Q2:val→hprop) (H2 H3:hprop),
  Q1 ∗+ H2 ==> Q2 ∗+ H2 ∗+ H3.
```

Proof using. intros. *xsimpl*. intros r. Abort.

Example of Entailment Proofs using *xsimpl*

```
Lemma himpl_example_1 : ∀ (p:loc),
  p ~~> 3 ==>
  ∃ (n:int), p ~~> n.
```

Proof using. *xsimpl*. Qed.

```
Lemma himpl_example_2 : ∀ (p q:loc),
  p ~~> 3 ∗ q ~~> 3 ==>
  ∃ (n:int), p ~~> n ∗ q ~~> n.
```

Proof using. *xsimpl*. Qed.

```
Lemma himpl_example_4 : ∀ (p:loc),
  ∃ (n:int), p ~~> n ==>
  ∃ (m:int), p ~~> (m + 1).
```

Proof using.

```
  intros. xpull. intros n. xsimpl (n-1).
  replace (n-1+1) with n. { auto. } { math. }
```

Qed.

```
Lemma himpl_example_5 : ∀ (H:hprop),  
  \[False] ==> H.
```

Proof using. xsimpl. Qed.

40.3.4 The *xchange* Tactic

The tactic *xchange* is to entailment what *rewrite* is to equality.

Assume an entailment goal of the form $H1 \ast H2 \ast H3 ==> H4$. Assume an entailment assumption M , say $H2 ==> H2'$. Then *xchange M* turns the goal into $H1 \ast H2' \ast H3 ==> H4$, effectively replacing $H2$ with $H2'$.

```
Lemma xchange_demo_base : ∀ H1 H2 H2' H3 H4,  
  H2 ==> H2' →  
  H1 \ast H2 \ast H3 ==> H4.
```

Proof using.

```
  introv M. xchange M.
```

Abort.

The tactic *xchange* can also take as argument equalities. The tactic *xchange M* exploits the left-to-right direction of an equality M , whereas *xchange ← M* exploits the right-to-left direction .

```
Lemma xchange_demo_eq : ∀ H1 H2 H3 H4 H5,  
  H1 \ast H3 = H5 →  
  H1 \ast H2 \ast H3 ==> H4.
```

Proof using.

```
  introv M. xchange M.  
  xchange ← M.
```

Abort.

The tactic *xchange M* does accept a lemma or hypothesis M featuring universal quantifiers, as long as its conclusion is an equality or an entailment. In such case, *xchange M* instantiates M before attempting to perform a replacement.

```
Lemma xchange_demo_inst : ∀ H1 (J J':int→hprop) H3 H4,  
  (forall n, J n = J' (n+1)) →  
  H1 \ast J 3 \ast H3 ==> H4.
```

Proof using.

```
  introv M. xchange M.
```

Abort.

How does the *xchange* tactic work? Consider a goal of the form $H ==> H'$ and assume *xchange* is invoked with an hypothesis of type $H1 ==> H1'$ as argument. The tactic *xchange* should attempt to decompose H as the star of $H1$ and the rest of H , call it $H2$. If it succeeds, then the goal $H ==> H'$ can be rewritten as $H1 \ast H2 ==> H'$. To exploit the hypothesis

$H1 ==> H1'$, the tactic should replace the goal with the entailment $H1' \setminus * H2 ==> H'$. The lemma shown below captures this piece of reasoning implemented by the tactic *xchange*.

Exercise: 2 stars, standard, especially useful (xchange_lemma) Prove, without using the tactic *xchange*, the following lemma which captures the internal working of *xchange*.

Lemma `xchange_lemma : ∀ H1 H1' H H' H2, H1 ==> H1' →`

`H ==> H1 * H2 →`

`H1' * H2 ==> H' →`

`H ==> H'.`

Proof using. *Admitted*.

□

End XSIMPLTACTIC.

40.4 Optional Material

40.4.1 Proofs for the Separation Algebra

Module FUNDAMENTALPROOFS.

We next show the details of the proofs establishing the fundamental properties of the Separation Logic operators.

Note that all these results must be proved without help of the tactic *xsimpl*, because the implementation of the tactic *xsimpl* itself depends on these fundamental properties.

We begin with the frame property, which is the simplest to prove.

Exercise: 1 star, standard, especially useful (himpl_frame_l) Prove the frame property for entailment. Hint: unfold the definition of *hstar*.

Lemma `himpl_frame_l : ∀ H2 H1 H1', H1 ==> H1' →`

`(H1 * H2) ==> (H1' * H2).`

Proof using. *Admitted*.

□

Exercise: 1 star, standard, especially useful (himpl_frame_r) Prove the lemma *himpl_frame_r*, symmetric to *himpl_frame_l*.

Lemma `himpl_frame_r : ∀ H1 H2 H2', H2 ==> H2' →`

`(H1 * H2) ==> (H1 * H2').`

Proof using. *Admitted*.

□

The second simplest result is the extrusion property for existentials. To begin with, we exploit the antisymmetry of entailment to turn the equality into a conjunction of two entailments. Then, it is simply a matter of unfolding the definitions of `hexists`, `hstar` and `==>`.

By default, Coq does not display any of the parentheses written in the statement below, making the proof obligation somewhat confusing. This is a small price to pay in exchange for maximal flexibility in allowing the parsing of unparenthesized expressions such as $H1 \ast \exists x, H2$ and $\exists x, H1 \ast H2$.

As a result, for this proof and the subsequent ones from this section, you should consider activating the display of parentheses. In CoqIDE, use the “View” menu, check “Display Parentheses”. Alternatively, use the command “Set Printing Parentheses”, or even “Set Printing All”.

Lemma `hstar_hexists` : $\forall A (J:A \rightarrow \text{hprop}) H, (\exists x, J x) \ast H = \exists x, (J x) \ast H$.

Proof using.

```
intros. applys himpl_antisym.
{ intros h (h1&h2&M1&M2&D&U). destruct M1 as (x&M1). ∃ x h1 h2. }
{ intros h (x&M). destruct M as (h1&h2&M1&M2&D&U). ∃ h1 h2.
  splits¬. ∃ x. }
```

Qed.

There remains to establish the commutativity and associativity of the star operator, and the fact that the empty heap predicate is its neutral element. To establish these properties, we need to exploit a few basic facts about finite maps. We introduce them as we go along.

To prove the commutativity of the star operator, i.e. $H1 \ast H2 = H2 \ast H1$, it is sufficient to prove the entailment in one direction, e.g., $H1 \ast H2 ==> H2 \ast H1$. Indeed, the other other direction is symmetrical. The symmetry argument is captured by the following lemma, which we will exploit in the proof of `hstar_comm`.

Lemma `hprop_op_comm` : $\forall (op:\text{hprop} \rightarrow \text{hprop} \rightarrow \text{hprop}), (\forall H1 H2, op H1 H2 ==> op H2 H1) \rightarrow (\forall H1 H2, op H1 H2 = op H2 H1)$.

Proof using. `introv M. intros. applys himpl_antisym; applys M. Qed.`

To prove commutativity of star, we need to exploit the fact that the union of two finite maps with disjoint domains commutes. This fact is captured by the following lemma.

Check `Fmap.union_comm_of_disjoint` : forall $h1 h2$, `Fmap.disjoint h1 h2 -> h1 \cup h2 = h2 \cup h1.`

The commutativity result is then proved as follows. Observe the use of the lemma `hprop_op_comm`, which allows establishing the entailment in only one of the two directions.

Lemma `hstar_comm` : $\forall H1 H2, H1 \ast H2 = H2 \ast H1$.

Proof using.

```

applys hprop_op_comm. intros. intros h (h1&h2&M1&M2&D&U). ∃ h2 h1.
subst. splits¬. { rewrite Fmap.union_comm_of_disjoint; auto. }
Qed.

```

Exercise: 3 stars, standard, especially useful (hstar_hempty_l) Prove that the empty heap predicate is a neutral element for the star. You will need to exploit the fact that the union with an empty map is the identity, as captured by lemma *Fmap.union_empty_l*.

Check *Fmap.union_empty_l* : forall h, *Fmap.empty* \u h = h.

```

Lemma hstar_hempty_l : ∀ H,
  [] \* H = H.

```

Proof using. Admitted.

□

The lemma showing that *hempty* is a right neutral can be derived from the previous result (*hempty* is a left neutral) and commutativity.

```

Lemma hstar_hempty_r : ∀ H,
  H \* [] = H.

```

Proof using.

```
intros. rewrite hstar_comm. rewrite hstar_hempty_l. auto.
```

Qed.

Associativity of star is the most tedious result to derive. We need to exploit the associativity of union on finite maps, as well as lemmas about the disjointness of a map with the result of the union of two maps.

Check *Fmap.union_assoc* : forall h1 h2 h3, (h1 \u h2) \u h3 = h1 \u (h2 \u h3).

Check *Fmap.disjoint_union_eq_l* : forall h1 h2 h3, *Fmap.disjoint* (h2 \u h3) h1 = (*Fmap.disjoint* h1 h2 /\ *Fmap.disjoint* h1 h3).

Check *Fmap.disjoint_union_eq_r* : forall h1 h2 h3, *Fmap.disjoint* h1 (h2 \u h3) = (*Fmap.disjoint* h1 h2 /\ *Fmap.disjoint* h1 h3).

Exercise: 1 star, standard, optional (hstar_assoc) Complete the right-to-left part of the proof of associativity of the star operator.

```

Lemma hstar_assoc : ∀ H1 H2 H3,
  (H1 \* H2) \* H3 = H1 \* (H2 \* H3).

```

Proof using.

```
intros. applys himpl_antisym.
```

```
{ intros h (h'&h3&M1&M2&D&U). destruct M1 as (h1&h2&M3&M4&D'&U').
```

```
  subst h'. rewrite Fmap.disjoint_union_eq_l in D.
```

```
  ∃ h1 (h2 \u h3). splits.
```

```
  { applys M3. }
```

```
  { ∃ h2 h3. }
```

```
  { rewrite@Fmap.disjoint_union_eq_r. }
```

```
  { rewrite@Fmap.union_assoc in U. } }
```

Admitted.

□

End FUNDAMENTALPROOFS.

40.4.2 Proof of the Consequence Rule

Module PROVECONSEQUENCERULES.

Recall the statement of the rule of consequence.

Lemma triple_conseq' : $\forall t H Q H' Q',$
triple $t H' Q' \rightarrow$
 $H ==> H' \rightarrow$
 $Q' ===> Q \rightarrow$
triple $t H Q.$

A direct proof of triple_conseq goes through the low-level interpretation of Separation Logic triples in terms of heaps.

Proof using.

intros $M WH WQ.$ rewrite triple_iff_triple_lowlevel in *.
intros $h1 h2 D HH.$ forwards $(v \& h1' \& D' \& R \& HQ): M D.$ applys $WH HH.$
 $\exists v h1'. splits \neg.$ applys $WQ HQ.$

Qed.

An alternative proof consists of first establishing the consequence rule for the hoare judgment, then derive its generalization to the triple judgment of Separation Logic.

Exercise: 3 stars, standard, especially useful (hoare_conseq) Prove the consequence rule for Hoare triples.

Lemma hoare_conseq : $\forall t H Q H' Q',$
hoare $t H' Q' \rightarrow$
 $H ==> H' \rightarrow$
 $Q' ===> Q \rightarrow$
hoare $t H Q.$

Proof using. *Admitted.*

□

Exercise: 2 stars, standard, especially useful (triple_conseq) Prove the consequence rule by leveraging the lemma hoare_conseq. Hint: unfold the definition of triple, apply the lemma hoare_conseq with the appropriate arguments, then exploit himpl_frame_l to prove the entailment relations.

Lemma triple_conseq : $\forall t H Q H' Q',$
triple $t H' Q' \rightarrow$

$H ==> H' \rightarrow$
 $Q' ==> Q \rightarrow$
 $\text{triple } t \ H \ Q.$

Proof using. Admitted.

□

End PROVECONSEQUENCERULES.

40.4.3 Proof of the Extraction Rules for Triples

Module PROVEEXTRACTIONRULES.

Recall the extraction rule for pure facts.

Parameter $\text{triple_hpure}' : \forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{triple } t \ H \ Q) \rightarrow$
 $\text{triple } t (\setminus [P] \setminus * H) Q.$

To prove this lemma, we first establish corresponding result on `hoare`, then derive its version for `triple`.

Lemma $\text{hoare_hpure} : \forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{hoare } t \ H \ Q) \rightarrow$
 $\text{hoare } t (\setminus [P] \setminus * H) Q.$

Proof using.

```
intros M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).
subst. rewrite Fmap.union_empty_l. applys M HP M2.
```

Qed.

Lemma $\text{triple_hpure} : \forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{triple } t \ H \ Q) \rightarrow$
 $\text{triple } t (\setminus [P] \setminus * H) Q.$

Proof using.

```
intros M. unfold triple. intros H'.
rewrite hstar_assoc. applys hoare_hpure.
intros HP. applys M HP.
```

Qed.

Similarly, the extraction rule for existentials for `triple` can be derived from that for `hoare`.

Exercise: 2 stars, standard, especially useful (`triple_hexists`) Prove the extraction rule `triple_hexists`. Hint: start by stating and proving the corresponding lemma for `hoare` triples.

Lemma $\text{hoare_hexists} : \forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\forall x, \text{hoare } t (J x) Q) \rightarrow$
 $\text{hoare } t (\setminus \exists x, J x) Q.$

Proof using. *introv M. intros h (x&Hh). applys M Hh. Qed.*

Lemma triple_hexists : $\forall t \ (A:\text{Type}) \ (J:A \rightarrow \text{hprop}) \ Q,$

$(\forall x, \text{triple } t (J x) Q) \rightarrow$

$\text{triple } t (\exists x, J x) Q.$

Proof using. *Admitted.*

□

Remark: the rules for extracting existentials out of entailments and out of preconditions can be stated in a slightly more concise way by exploiting the combinator `hexists` rather than its associated notation $\exists x, H$, which stands for `hexists (fun x => H)`.

These formulation, shown below, tend to behave slightly better with respect to Coq unification, hence we use them in the CFML framework.

Parameter *hstar_hexists'* : $\forall A \ (J:A \rightarrow \text{hprop}) \ H,$
 $(\text{hexists } J) \ \backslash * \ H = \text{hexists } (J \ \backslash *+ \ H).$

Parameter *triple_hexists'* : $\forall t \ (A:\text{Type}) \ (J:A \rightarrow \text{hprop}) \ Q,$
 $(\forall x, \text{triple } t (J x) Q) \rightarrow$
 $\text{triple } t (\text{hexists } J) Q.$

Remark: in chapter `Hprop`, we observed that $\backslash[P]$ can be encoded as $\exists (p:P), \backslash[]$. When this encoding is used, the rule `triple_hpure` turns out to be a particular instance of the rule `triple_hexists`, as we prove next.

Parameter *hpure_encoding* : $\forall P,$
 $\backslash[P] = (\exists (p:P), \backslash[]).$

Lemma triple_hpure_from_triple_hexists : $\forall t \ (P:\text{Prop}) \ H \ Q,$
 $(P \rightarrow \text{triple } t H Q) \rightarrow$
 $\text{triple } t (\backslash[P] \ \backslash * \ H) Q.$

Proof using.

introv M. rewrite hpure_encoding.

rewrite hstar_hexists. applys triple_hexists. rewrite hstar_hempty_l. applys M.

Qed.

End PROVEEXTRACTIONRULES.

Rules for Naming Heaps

Thereafter, we write $= h$ for `fun h' => h' = h`, that is, the heap predicate that only accepts heaps exactly equal to `h`.

Exercise: 3 stars, standard, optional (`hexists_named_eq`) Prove that a heap predicate `H` is equivalent to the heap predicate which asserts that the heap is, for a heap `h` such that `H h`, exactly equal to `H`.

Hint: use `hstar_hpure_l` and `hexists_intro`, as well as the extraction rules `himpl_hexists_l` and `himpl_hstar_hpure_l`.

Lemma `hexists_named_eq` : $\forall H,$
 $H = (\exists h, \set{H}{h} \setminus * (= h)).$

Proof using. *Admitted.*

□

Exercise: 1 star, standard, optional (hoare_named_heap) Prove that the proposition `hoare t H Q` is equivalent to: for any heap h satisfying the precondition H , the Hoare triple whose precondition requires the input heap to be exactly equal to h , and whose postcondition is Q holds.

Lemma `hoare_named_heap` : $\forall t H Q,$
 $(\forall h, H h \rightarrow \text{hoare } t (= h) Q) \rightarrow$
 $\text{hoare } t H Q.$

Proof using. *Admitted.*

□

Exercise: 3 stars, standard, optional (triple_named_heap) Prove the counterpart of `hoare_named_heap` for Separation Logic triples.

It is possible to exploit the lemma `hoare_named_heap`, yet there exists a simpler, more direct proof that exploits the lemma `hexists_name_eq`, which is stated above.

Lemma `triple_named_heap` : $\forall t H Q,$
 $(\forall h, H h \rightarrow \text{triple } t (= h) Q) \rightarrow$
 $\text{triple } t H Q.$

Proof using. *Admitted.*

□

40.4.4 Alternative Structural Rule for Existentials

Module ALTERNATIVEEXISTENTIALRULE.

Traditional papers on Separation Logic do not include `triple_hexists`, but instead a rule called `triple_hexists2` that includes an existential quantifier both in the precondition and the postcondition.

As we show next, in the presence of the consequence rule, the two rules are equivalent.

The formulation of `triple_hexists` is not only more concise, it is also better suited for practical applications.

Lemma `triple_hexists2` : $\forall A (Hof:A \rightarrow \text{hprop}) (Qof:A \rightarrow \text{val} \rightarrow \text{hprop}) t,$
 $(\forall x, \text{triple } t (Hof x) (Qof x)) \rightarrow$
 $\text{triple } t (\exists x, Hof x) (\text{fun } v \Rightarrow \exists x, Qof x v).$

Proof using.

intros M .

applys `triple_hexists`. *intros* x .

```


$$\text{applys } \text{triple\_conseq } (M\ x).$$


$$\{ \text{applys } \text{himpl\_refl}. \}$$


$$\{ \text{intros } v. \text{applys } \text{himpl\_hexists\_r } x. \text{applys } \text{himpl\_refl}. \}$$


```

Qed.

Lemma triple_hexists_of_triple_hexists2 : $\forall t \ (A:\text{Type}) \ (Hof:A \rightarrow \text{hprop}) \ Q,$
 $(\forall x, \text{triple } t \ (Hof\ x) \ Q) \rightarrow$
 $\text{triple } t \ (\exists x, Hof\ x) \ Q.$

Proof using.

```

introv M.

$$\text{applys } \text{triple\_conseq } (\exists x, Hof\ x) (\text{fun } (v:\text{val}) \Rightarrow \exists (x:A), Q\ v).$$


$$\{ \text{applys } \text{triple\_hexists2}. \text{intros } x. \text{applys } M. \}$$


$$\{ \text{applys } \text{himpl\_refl}. \}$$


$$\{ \text{intros } v. \text{applys } \text{himpl\_hexists\_l}. \text{intros } _. \text{applys } \text{himpl\_refl}. \}$$


```

Qed.

End ALTERNATIVEEXISTENTIALRULE.

40.4.5 Historical Notes

Nearly every project that aims for practical program verification using Separation Logic features, in one way or another, some amount of tooling for automatically simplifying Separation Logic assertion.

The tactic used here, *xsiml*, was developed for the CFML tool. Its specification may be found in Appendix K from the paper: http://www.chargueraud.org/research/2020/seq_seplogic/seq_seplog.pdf though it makes sense to wait until chapter Wand for reading it.

Chapter 41

Library `SLF.Rules`

41.1 Rules: Reasoning Rules

Set Implicit Arguments.

This file imports `LibSepDirect.v` instead of `Hprop.v` and `Himpl.v`. The file `LibSepDirect.v` contains definitions that are essentially similar to those from `Hprop.v` and `Himpl.v`, yet with one main difference: `LibSepDirect` makes the definition of Separation Logic operators opaque.

As a result, one cannot unfold the definition of `hstar`, `hpure`, etc. To carry out reasoning, one must use the introduction and elimination lemmas (e.g. `hstar_intro`, `hstar_elim`). These lemmas enforce abstraction: they ensure that the proofs do not depend on the particular choice of the definitions used for constructing Separation Logic.

From `SLF` Require Export `LibSepReference`.

From `SLF` Require Basic.

41.2 First Pass

In the previous chapters, we have:

- introduced the key heap predicate operators,
- introduced the notion of Separation Logic triple,
- introduced the entailment relation,
- introduced the structural rules of Separation Logic.

We are now ready to present the other reasoning rules, which enable establishing properties of concrete programs.

These reasoning rules are proved correct with respect to the semantics of the programming language in which the programs are expressed. Thus, a necessary preliminary step is to

present the syntax and the semantics of a (toy) programming language, for which we aim to provide Separation Logic reasoning rules.

The present chapter is thus organized as follows:

- definition of the syntax of the language,
- definition of the semantics of the language,
- statements of the reasoning rules associated with each of the term constructions from the language,
- specification of the primitive operations of the language, in particular those associated with memory operations,
- review of the 4 structural rules introduced in prior chapters,
- examples of practical verification proofs.

The bonus section (optional) also includes:

- proofs of the reasoning rules associated with each term construct,
- proofs of the specification of the primitive operations.

41.2.1 Semantic of Terms

Module SYNTAXANDSEMANTICS.

Syntax

The syntax described next captures the “abstract syntax tree” of a programming language. It follows a presentation that distinguishes between closed values and terms. This presentation is intended to simplify the definition and evaluation of the substitution function: because values are always closed (i.e., no free variables in them), the substitution function never needs to traverse through values.

The grammar for values includes unit, boolean, integers, locations, functions, recursive functions, and primitive operations. For example, `val_int 3` denotes the integer value 3. The value `val_fun x t` denotes the function `fun x => t`, and the value `val_fix f x t` denotes the function `fix f x => t`, which is written `let rec f x = t in f` in OCaml syntax.

For conciseness, we include just a few primitive operations: `ref`, `get`, `set` and `free` for manipulating the mutable state, the operation `add` to illustrate a simple arithmetic operation, and the operation `div` to illustrate a partial operation.

Inductive `val` : Type :=
| `val_unit` : `val`

```

| val_bool : bool → val
| val_int : int → val
| val_loc : loc → val
| val_fun : var → trm → val
| val_fix : var → var → trm → val
| val_ref : val
| val_get : val
| val_set : val
| val_free : val
| val_add : val
| val_div : val

```

The grammar for terms includes values, variables, function definitions, recursive function definitions, function applications, sequences, let-bindings, and conditionals.

with **trm** : Type :=

```

| trm_val : val → trm
| trm_var : var → trm
| trm_fun : var → trm → trm
| trm_fix : var → var → trm → trm
| trm_app : trm → trm → trm
| trm_seq : trm → trm → trm
| trm_let : var → trm → trm → trm
| trm_if : trm → trm → trm → trm.

```

Note that **trm_fun** and **trm_fix** denote functions that may feature free variables, unlike **val_fun** and **val_fix** which denote closed values. The intention is that the evaluation of a **trm_fun** in the empty context produces a **val_fun** value. Likewise, a **trm_fix** eventually evaluates to a **val_fix**.

Several value constructors are declared as coercions, to enable more concise statements. For example, **val_loc** is declared as a coercion, so that a location **p** of type **loc** can be viewed as the value **val_loc p** where an expression of type **val** is expected. Likewise, a boolean **b** may be viewed as the value **val_bool b**, and an integer **n** may be viewed as the value **val_int n**.

Coercion **val_loc** : **loc** >-> **val**.

Coercion **val_bool** : **bool** >-> **val**.

Coercion **val_int** : **Z** >-> **val**.

State

The language we consider is an imperative language, with primitive functions for manipulating the state. Thus, the statement of the evaluation rules involve a memory state.

Recall from **Hprop** that a state is represented as a finite map from location to values. Finite maps are presented using the type **fmap**. Details of the construction of finite maps

are beyond the scope of this course; details may be found in the file *LibSepFmap.v*.

Definition state : Type := **fmap loc val**.

For technical reasons related to the internal representation of finite maps, to enable reading in a state, we need to justify that the grammar of values is inhabited. This property is captured by the following command, whose details are not relevant for understanding the rest of the chapter.

Instance Inhab_val : **Inhab val**.

Proof using. apply (**Inhab_of_val val_unit**). Qed.

Substitution

The semantics of the evaluation of function is described by means of a substitution function. The substitution function, written **subst y w t**, replaces all occurrences of a variable **y** with a value **w** inside a term **t**.

The substitution function is always the identity function on values, because our language only considers closed values. In other words, we define **subst y w (trm_val v) = (trm_val v)**.

The substitution function, when reaching a variable, performs a comparison between two variables. To that end, it exploits the comparison function **var_eq x y**, which produces a boolean value indicating whether **x** and **y** denote the same variable.

“Optional contents”: the remaining of this section describes further details about the substitution function that may be safely skipped over in first reading.

The substitution operation traverses all other language constructs in a structural manner. It takes care of avoiding “variable capture” when traversing binders: **subst y w t** does not recurse below the scope of binders whose name is equal to **y**. For example, the result of **subst y w (trm_let x t1 t2)** is defined as **trm_let x (subst y w t1) (if var_eq x y then t2 else (subst y w t2))**.

The auxiliary function *if_y_eq*, which appears in the definition of **subst** shown below, helps performing the factorizing the relevant checks that prevent variable capture.

```
Fixpoint subst (y:var) (w:val) (t:trm) : trm :=
  let aux t := subst y w t in
  let if_y_eq x t1 t2 := if var_eq x y then t1 else t2 in
  match t with
  | trm_val v => trm_val v
  | trm_var x => if_y_eq x (trm_val w) t
  | trm_fun x t1 => trm_fun x (if_y_eq x t1 (aux t1))
  | trm_fix f x t1 => trm_fix f x (if_y_eq f t1 (if_y_eq x t1 (aux t1)))
  | trm_app t1 t2 => trm_app (aux t1) (aux t2)
  | trm_seq t1 t2 => trm_seq (aux t1) (aux t2)
  | trm_let x t1 t2 => trm_let x (aux t1) (if_y_eq x t2 (aux t2))
  | trm_if t0 t1 t2 => trm_if (aux t0) (aux t1) (aux t2)
  end.
```

Implicit Types and Coercions

To improve the readability of the evaluation rules stated further, we take advantage of both implicit types and coercions.

The implicit types are defined as shown below. For example, the first command indicates that variables whose name begins with the letter 'b' are, by default, variables of type **bool**.

Implicit Types $b : \text{bool}$.

Implicit Types $v r : \text{val}$.

Implicit Types $t : \text{trm}$.

Implicit Types $s : \text{state}$.

We next introduce two key coercions. First, we declare **trm_val** as a coercion, so that, instead of writing **trm_val v**, we may write simply **v** wherever a term is expected.

Coercion $\text{trm_val} : \text{val} \rightarrow \text{trm}$.

Second, we declare **trm_app** as a “Funclass” coercion. This piece of magic enables us to write $t_1 t_2$ as a shorthand for **trm_app** $t_1 t_2$. The idea of associating **trm_app** as the “Funclass” coercion for the type **trm** is that if a term t_1 of type **trm** is applied like a function to an argument, then t_1 should be interpreted as **trm_app** t_1 .

Coercion $\text{trm_app} : \text{trm} \rightarrow \text{Funclass}$.

Interestingly, the “Funclass” coercion for **trm_app** can be iterated. The expression $t_1 t_2 t_3$ is parsed by Coq as $(t_1 t_2) t_3$. The first application $t_1 t_2$ is interpreted as **trm_app** $t_1 t_2$. This expression, which itself has type **trm**, is applied to t_3 . Hence, it is interpreted as **trm_app** $(\text{trm_app } t_1 t_2) t_3$, which indeed corresponds to a function applied to two arguments.

Big-Step Semantics

The semantics is presented in big-step style. This presentation makes it slightly easier to establish reasoning rules than with small-step reduction rules, because both the big-step judgment and a triple judgment describe complete execution, relating a term with the value that it produces.

The big-step evaluation judgment, written **eval s t s' v**, asserts that, starting from state **s**, the evaluation of the term **t** terminates in a state **s'**, producing an output value **v**.

For simplicity, we assume terms to be in “A-normal form”: the arguments of applications and of conditionals are restricted to variables and value. Such a requirement does not limit expressiveness, yet it simplifies the statement of the evaluation rules.

For example, if a source program includes a conditional **trm_if t0 t1 t2**, then it is required that **t0** be either a variable or a value. This is not a real restriction, because **trm_if t0 t1 t2** can always be encoded as **let x = t0 in if x then t1 else t2**.

The big-step judgment is inductively defined as follows.

Inductive **eval** : **state** → **trm** → **state** → **val** → **Prop** :=

1. **eval** for values and function definitions.

A value evaluates to itself. A term function evaluates to a value function. Likewise for a recursive function.

```
| eval_val : ∀ s v,
  eval s (trm_val v) s v
| eval_fun : ∀ s x t1,
  eval s (trm_fun x t1) s (val_fun x t1)
| eval_fix : ∀ s f x t1,
  eval s (trm_fix f x t1) s (val_fix f x t1)
```

2. eval for function applications.

The beta reduction rule asserts that $(\text{val_fun } x \text{ } t1) \text{ } v2$ evaluates to the same result as $\text{subst } x \text{ } v2 \text{ } t1$.

In the recursive case, $(\text{val_fix } f \text{ } x \text{ } t1) \text{ } v2$ evaluates to $\text{subst } x \text{ } v2 \text{ } (\text{subst } f \text{ } v1 \text{ } t1)$, where $v1$ denotes the recursive function itself, that is, $\text{val_fix } f \text{ } x \text{ } t1$.

```
| eval_app_fun : ∀ s1 s2 v1 v2 x t1 v,
  v1 = val_fun x t1 →
  eval s1 (subst x v2 t1) s2 v →
  eval s1 (trm_app v1 v2) s2 v
| eval_app_fix : ∀ s1 s2 v1 v2 f x t1 v,
  v1 = val_fix f x t1 →
  eval s1 (subst x v2 (subst f v1 t1)) s2 v →
  eval s1 (trm_app v1 v2) s2 v
```

3. eval for structural constructs.

A sequence $\text{trm_seq } t1 \text{ } t2$ first evaluates $t1$, taking the state from $s1$ to $s2$, drops the result of $t1$, then evaluates $t2$, taking the state from $s2$ to $s3$.

The let-binding $\text{trm_let } x \text{ } t1 \text{ } t2$ is similar, except that the variable x gets substituted for the result of $t1$ inside $t2$.

```
| eval_seq : ∀ s1 s2 s3 t1 t2 v1 v,
  eval s1 t1 s2 v1 →
  eval s2 t2 s3 v →
  eval s1 (trm_seq t1 t2) s3 v
| eval_let : ∀ s1 s2 s3 x t1 t2 v1 r,
  eval s1 t1 s2 v1 →
  eval s2 (subst x v1 t2) s3 r →
  eval s1 (trm_let x t1 t2) s3 r
```

4. eval for conditionals.

A conditional in a source program is assumed to be of the form `if t0 then t1 else t2`, where $t0$ is either a variable or a value. If it is a variable, then by the time it reaches an evaluation position, the variable must have been substituted by a value. Thus, the evaluation rule only considers the form `if v0 then t1 else t2`. The value $v0$ must be a boolean value,

otherwise evaluation gets stuck.

The term `trm_if (val_bool true) t1 t2` behaves like `t1`, whereas the term `trm_if (val_bool false) t1 t2` behaves like `t2`. This behavior is described by a single rule, leveraging Coq's "if" constructor to factor out the two cases.

```
| eval_if : ∀ s1 s2 b v t1 t2,
  eval s1 (if b then t1 else t2) s2 v →
  eval s1 (trm_if (val_bool b) t1 t2) s2 v
```

5. eval for primitive stateless operations.

For similar reasons as explained above, the behavior of applied primitive functions only need to be described for the case of value arguments.

An arithmetic operation expects integer arguments. The addition of `val_int n1` and `val_int n2` produces `val_int (n1 + n2)`.

The division operation, on the same arguments, produces the quotient `n1 / n2`, under the assumption that the divisor `n2` is non-zero. In other words, if a program performs a division by zero, then it cannot satisfy the `eval` judgment.

```
| eval_add : ∀ s n1 n2,
  eval s (val_add (val_int n1) (val_int n2)) s (val_int (n1 + n2))
| eval_div : ∀ s n1 n2,
  n2 ≠ 0 →
  eval s (val_div (val_int n1) (val_int n2)) s (val_int (Z.quot n1 n2))
```

6. eval for primitive operations on memory.

There remains to describe operations that act on the mutable store.

`val_ref v` allocates a fresh cell with contents `v`. The operation returns the location, written `p`, of the new cell. This location must not be previously in the domain of the store `s`.

`val_get (val_loc p)` reads the value in the store `s` at location `p`. The location must be bound to a value in the store, else evaluation is stuck.

`val_set (val_loc p) v` updates the store at a location `p` assumed to be bound in the store `s`. The operation modifies the store and returns the unit value.

`val_free (val_loc p)` deallocates the cell at location `p`.

```
| eval_ref : ∀ s v p,
  Fmap.indom s p →
  eval s (val_ref v) (Fmap.update s p v) (val_loc p)
| eval_get : ∀ s p,
  Fmap.indom s p →
  eval s (val_get (val_loc p)) s (Fmap.read s p)
| eval_set : ∀ s p v,
  Fmap.indom s p →
  eval s (val_set (val_loc p) v) (Fmap.update s p v) val_unit
| eval_free : ∀ s p,
  Fmap.indom s p →
```

```
eval s (val_free (val_loc p)) (Fmap.remove s p) val_unit.
```

End SYNTAXANDSEMANTICS.

Loading of Definitions from *Direct*

Throughout the rest of this file, we rely not on the definitions shown above, but on the definitions from *LibSepDirect.v*. The latter are slightly more general, yet completely equivalent to the ones presented above for the purpose of establishing the reasoning rules that we are interested in.

To reduce the clutter in the statement of lemmas, we associate default types to a number of common meta-variables.

```
Implicit Types x : var.  
Implicit Types b : bool.  
Implicit Types p : loc.  
Implicit Types n : int.  
Implicit Types v w r : val.  
Implicit Types t : trm.  
Implicit Types h : heap.  
Implicit Types s : state.  
Implicit Types H : hprop.  
Implicit Types Q : val → hprop.
```

41.2.2 Rules for Terms

We next present reasoning rule for terms. Most of these Separation Logic rules have a statement essentially identical to the statement of the corresponding Hoare Logic rule. The main difference lies in their interpretation: whereas Hoare Logic pre- and post-conditions describe the full state, a Separation Logic rule describes only a fraction of the mutable state.

Reasoning Rule for Sequences

Let us begin with the reasoning rule for sequences. The Separation Logic reasoning rule for a sequence $t_1; t_2$ is essentially the same as that from Hoare logic. The rule is:

$$\{H\} t_1 \{\text{fun } v \Rightarrow H_1\} \{H_1\} t_2 \{Q\}$$

$$\{H\} (t_1; t_2) \{Q\}$$

The Coq statement corresponding to the above rule is:

```
Parameter triple_seq : ∀ t1 t2 H Q H1,  
  triple t1 H (fun v ⇒ H1) →  
  triple t2 H1 Q →  
  triple (trm_seq t1 t2) H Q.
```

The variable v denotes the result of the evaluation of t_1 . For well-typed programs, this result would always be `val_unit`. Yet, because we here consider an untyped language, we do not bother adding the constraint $v = \text{val_unit}$. Instead, we simply treat the result of t_1 as a value irrelevant to the remaining of the evaluation.

Reasoning Rule for Let-Bindings

Next, we present the reasoning rule for let-bindings. Here again, there is nothing specific to Separation Logic, the rule would be exactly the same in Hoare Logic.

The reasoning rule for a let binding `let x = t1 in t2` could be stated, in informal writing, in the form:

$$\{H\} t_1 \{Q_1\} (\text{forall } x, \{Q_1 x\} t_2 \{Q\})$$

$$\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}$$

Yet, such a presentation makes a confusion between the x that denotes a program variable in `let x = t1 in t2`, and the x that denotes a value when quantified as $\forall x$.

The correct statement involves a substitution from the variable x to a value quantified as $\forall (v:\text{val})$.

$$\{H\} t_1 \{Q_1\} (\text{forall } v, \{Q_1 v\} (\text{subst } x v t_2) \{Q\})$$

$$\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}$$

The corresponding Coq statement is thus as follows.

```
Parameter triple_let :  $\forall x t_1 t_2 Q_1 H Q$ ,
  triple  $t_1 H Q_1 \rightarrow$ 
  ( $\forall v_1$ , triple  $(\text{subst } x v_1 t_2) (Q_1 v_1) Q \rightarrow$ 
  triple  $(\text{trm\_let } x t_1 t_2) H Q$ .
```

Reasoning Rule for Conditionals

The rule for a conditional is, again, exactly like in Hoare logic.

$$b = \text{true} \rightarrow \{H\} t_1 \{Q\} b = \text{false} \rightarrow \{H\} t_2 \{Q\}$$

$$\{H\} (\text{if } b \text{ then } t_1 \text{ in } t_2) \{Q\}$$

The corresponding Coq statement appears next.

```
Parameter triple_if_case :  $\forall b t_1 t_2 H Q$ ,
  ( $b = \text{true} \rightarrow$  triple  $t_1 H Q$ )  $\rightarrow$ 
  ( $b = \text{false} \rightarrow$  triple  $t_2 H Q$ )  $\rightarrow$ 
  triple  $(\text{trm\_if } (\text{val\_bool } b) t_1 t_2) H Q$ .
```

Note that the two premises may be factorized into a single one using Coq's conditional construct. Such an alternative statement is discussed further in this chapter.

Reasoning Rule for Values

The rule for a value v can be written as a triple with an empty precondition and a postcondition asserting that the result value r is equal to v , in the empty heap. Formally:

$$\{\square\} v \{ \text{fun } r \Rightarrow \lceil r = v \rceil \}$$

It is however more convenient in practice to work with a judgment whose conclusion is of the form $\{H\} v \{Q\}$, for an arbitrary H and Q . For this reason, we prefer the following rule for values.

$$H ==> Q v$$

$$\{H\} v \{Q\}$$

It may not be completely obvious at first sight why this alternative rule is equivalent to the former. We prove the equivalence further in this chapter.

The Coq statement of the rule for values is thus as follows.

```
Parameter triple_val : ∀ v H Q,
  H ==> Q v →
  triple (trm_val v) H Q.
```

Let us prove that the “minimalistic” rule $\{\square\} v \{ \text{fun } r \Rightarrow \lceil r = v \rceil \}$ is equivalent to *triple_val*.

Exercise: 1 star, standard, especially useful (*triple_val_minimal*) Prove that the alternative rule for values derivable from *triple_val*. Hint: use the tactic *xsimpl* to conclude the proof.

```
Lemma triple_val_minimal : ∀ v,
  triple (trm_val v) [] (fun r ⇒ \[r = v]).
```

Proof using. *Admitted*.

□

Exercise: 2 stars, standard, especially useful (*triple_val'*) More interestingly, prove that *triple_val* is derivable from *triple_val_minimal*. Hint: you will need to exploit the appropriate structural rule(s).

```
Lemma triple_val' : ∀ v H Q,
  H ==> Q v →
  triple (trm_val v) H Q.
```

Proof using. *Admitted*.

□

Exercise: 4 stars, standard, especially useful (*triple_let_val*) Consider a term of the form `let x = v1 in t2`, that is, where the argument of the let-binding is already a value. State and prove a reasoning rule of the form:

Lemma triple_let_val : forall x v1 t2 H Q, ... -> triple (trm_let x v1 t2) H Q.

Hint: you'll need to guess the intermediate postcondition Q_1 associated with the let-binding rule, and exploit the appropriate structural rules.

□

Reasoning Rule for Functions

In addition to the reasoning rule for values, we need reasoning rules for functions and recursive functions that appear as terms in the source program (as opposed to appearing as values).

A function definition $\text{trm_fun } x \ t1$, expressed as a subterm in a program, evaluates to a value, more precisely to $\text{val_fun } x \ t1$. Again, we could consider a rule with an empty precondition:

$$\{\square\} (\text{trm_fun } x \ t1) \{\text{fun } r \Rightarrow \langle r = \text{val_fun } x \ t1 \rangle\}$$

However, we prefer a conclusion of the form $\{H\} (\text{trm_fun } x \ t1) \{Q\}$. We thus consider the following rule, very similar to *triple_val*.

```
Parameter triple_fun : ∀ x t1 H Q,  
  H ==> Q (val_fun x t1) →  
  triple (trm_fun x t1) H Q.
```

The rule for recursive functions is similar. It is presented further in the file.

Last but not least, we need a reasoning rule to reason about a function application. Consider an application $\text{trm_app } v1 \ v2$. Assume $v1$ to be a function, that is, to be of the form $\text{val_fun } x \ t1$. Then, according to the beta-reduction rule, the semantics of $\text{trm_app } v1 \ v2$ is the same as that of $\text{subst } x \ v2 \ t1$. This reasoning rule is thus:

$$v1 = \text{val_fun } x \ t1 \{H\} (\text{subst } x \ v2 \ t1) \{Q\}$$

$$\{H\} (\text{trm_app } v1 \ v2) \{Q\}$$

The corresponding Coq statement is as shown below.

```
Parameter triple_app_fun : ∀ x v1 v2 t1 H Q,  
  v1 = val_fun x t1 →  
  triple (subst x v2 t1) H Q →  
  triple (trm_app v1 v2) H Q.
```

The generalization to the application of recursive functions is straightforward. It is discussed further in this chapter.

41.2.3 Specification of Primitive Operations

Before we can tackle verification of actual programs, there remains to present the specifications for the primitive operations. Let us begin with basic arithmetic operations: addition and division.

Specification of Arithmetic Primitive Operations

Consider a term of the form `val_add n1 n2`, which is short for `trm_app (trm_app (trm_val val_add) (val_int n1)) (val_int n2)`. Indeed, recall that `val_int` is declared as a coercion.

The addition operation may execute in an empty state. It does not modify the state, and returns the value `val_int (n1+n2)`.

In the specification shown below, the precondition is written `\[]` and the postcondition binds a return value `r` of type `val` specified to be equal to `val_int (n1+n2)`.

```
Parameter triple_add : ∀ n1 n2,
  triple (val_add n1 n2)
  \[]
  (fun r ⇒ \[r = val_int (n1 + n2)]).
```

The specification of the division operation `val_div n1 n2` is similar, yet with the extra requirement that the argument `n2` must be nonzero. This requirement `n2 ≠ 0` is a pure fact, which can be asserted as part of the precondition, as follows.

```
Parameter triple_div : ∀ n1 n2,
  triple (val_div n1 n2)
  \[n2 ≠ 0]
  (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Equivalently, the requirement `n2 ≠ 0` may be asserted as an hypothesis to the front of the triple judgment, in the form of a standard Coq hypothesis, as shown below.

```
Parameter triple_div' : ∀ n1 n2,
  n2 ≠ 0 →
  triple (val_div n1 n2)
  \[]
  (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

This latter presentation with pure facts such as `n2 ≠ 0` placed to the front of the triple turns out to be more practical to exploit in proofs. Hence, we always follow this style of presentation, and reserve the precondition for describing pieces of mutable state.

Specification of Primitive Operations Acting on Memory

There remains to describe the specification of operations on the heap.

Recall that `val_get` denotes the operation for reading a memory cell. A call of the form `val_get v'` executes safely if `v'` is of the form `val_loc p` for some location `p`, in a state that features a memory cell at location `p`, storing some contents `v`. Such a state is described as `p ~~> v`. The read operation returns a value `r` such that `r = v`, and the memory state of the operation remains unchanged. The specification of `val_get` is thus expressed as follows.

```
Parameter triple_get : ∀ v p,
  triple (val_get (val_loc p))
  (p ~~> v)
```

```
(fun r => \[r = v] \* (p ~~> v)).
```

Remark: `val_loc` is registered as a coercion, so `val_get (val_loc p)` could be written simply as `val_get p`, where `p` has type `loc`. We here chose to write `val_loc` explicitly for clarity.

Recall that `val_set` denotes the operation for writing a memory cell. A call of the form `val_set v' w` executes safely if `v'` is of the form `val_loc p` for some location `p`, in a state `p ~~> v`. The write operation updates this state to `p ~~> w`, and returns the unit value, which can be ignored. Hence, `val_set` is specified as follows.

```
Parameter triple_set : \forall w p v,
  triple (val_set (val_loc p) w)
    (p ~~> v)
  (fun _ => p ~~> w).
```

Recall that `val_ref` denotes the operation for allocating a cell with a given contents. A call to `val_ref v` does not depend on the contents of the existing state. It extends the state with a fresh singleton cell, at some location `p`, assigning it `v` as contents. The fresh cell is then described by the heap predicate `p ~~> v`. The evaluation of `val_ref v` produces the value `val_loc p`. Thus, if `r` denotes the result value, we have `r = val_loc p` for some `p`. In the corresponding specification shown below, observe how the location `p` is existentially quantified in the postcondition.

```
Parameter triple_ref : \forall v,
  triple (val_ref v)
    []
  (fun (r:\text{val}) => \exists (p:loc), \[r = val_loc p] \* p ~~> v).
```

Using the notation `funloc p => H` as a shorthand for `fun (r:\text{val}) => \exists (p:loc), \[r = val_loc p] * H`, the specification for `val_ref` becomes more concise.

```
Parameter triple_ref' : \forall v,
  triple (val_ref v)
    []
  (funloc p => p ~~> v).
```

Recall that `val_free` denotes the operation for deallocated a cell at a given address. A call of the form `val_free p` executes safely in a state `p ~~> v`. The operation leaves an empty state, and asserts that the return value, named `r`, is equal to unit.

```
Parameter triple_free : \forall p v,
  triple (val_free (val_loc p))
    (p ~~> v)
  (fun _ => []).
```

41.2.4 Review of the Structural Rules

Let us review the essential structural rules, which were introduced in the previous chapters. These structural rules are involved in the practical verification proofs carried out further in this chapter.

The frame rule asserts that the precondition and the postcondition can be extended together by an arbitrary heap predicate. Recall that the definition of `triple` was set up precisely to validate this frame rule, so in a sense it holds “by construction”.

```
Parameter triple_frame : ∀ t H Q H',
  triple t H Q →
  triple t (H \* H') (Q \*+ H').
```

The consequence rule allows to strengthen the precondition and weaken the postcondition.

```
Parameter triple_conseq : ∀ t H' Q' H Q,
  triple t H' Q' →
  H ==> H' →
  Q' ==> Q →
  triple t H Q.
```

In practice, it is most convenient to exploit a rule that combines both frame and consequence into a single rule, as stated next. (Note that this “combined structural rule” was proved as an exercise in chapter `Himpl`.)

```
Parameter triple_conseq_frame : ∀ H2 H1 Q1 t H Q,
  triple t H1 Q1 →
  H ==> H1 \* H2 →
  Q1 \*+ H2 ==> Q →
  triple t H Q.
```

The two extraction rules enable to extract pure facts and existentially quantified variables, from the precondition into the Coq context.

```
Parameter triple_hpure : ∀ t (P:Prop) H Q,
  (P → triple t H Q) →
  triple t (\[P] \* H) Q.
```

```
Parameter triple_hexists : ∀ t (A:Type) (J:A→hprop) Q,
  (forall (x:A), triple t (J x) Q) →
  triple t (\exists (x:A), J x) Q.
```

Exercise: 1 star, standard, optional (`triple_hpure'`) The extraction rule `triple_hpure` assumes a precondition of the form $\backslash[P] \ * H$. The variant rule `triple_hpure'`, stated below, assumes instead a precondition with only the pure part, i.e., of the form $\backslash[P]$. Prove that `triple_hpure'` is indeed a corollary of `triple_hpure`.

```
Lemma triple_hpure' : ∀ t (P:Prop) Q,
  (P → triple t \[] Q) →
  triple t \[P] Q.
```

Proof using. *Admitted.*

□

41.2.5 Verification Proof in Separation Logic

We have at hand all the necessary rules for carrying out actual verification proofs in Separation Logic. Let's do it!

Module EXAMPLEPROGRAMS.

Export *ProgramSyntax*.

Proof of *incr*

First, we consider the verification of the increment function, which is written in OCaml syntax as:

OCaml:

```
let incr p = p := !p + 1
```

Recall that, for simplicity, we assume programs to be written in “A-normal form”, that is, with all intermediate expressions named by a let-binding. Thereafter, we thus consider the following definition for the *incr*.

OCaml:

```
let incr p = let n = !p in let m = n+1 in p := m
```

Using the construct from our toy programming language, the definition of *incr* is written as shown below.

Definition *incr* : **val** :=

```
val_fun "p" (
  trm_let "n" (trm_app val_get (trm_var "p")) (
    trm_let "m" (trm_app (trm_app val_add
      (trm_var "n")) (val_int 1)) (
      trm_app (trm_app val_set (trm_var "p")) (trm_var "m")))).
```

Alternatively, using notation and coercions, the same program can be written as shown below.

Definition *incr'* : **val** :=

```
<{ fun 'p =>
  let 'n = ! 'p in
  let 'm = 'n + 1 in
  'p := 'm }>.
```

Let us check that the two definitions are indeed the same.

Lemma *incr_eq_incr'* :

incr = *incr'*.

Proof using. reflexivity. Qed.

Recall from the first chapter the specification of the increment function. Its precondition requires a singleton state of the form $p \sim\!> n$. Its postcondition describes a state of the form $p \sim\!> (n+1)$.

Lemma *triple_incr* : $\forall (p:\text{loc}) (n:\text{int})$,

```

triple (trm_app incr p)
  (p ~~> n)
  (fun _ => p ~~> (n+1)).

```

We next show a detailed proof for this specification. It exploits:

- the structural reasoning rules,
- the reasoning rules for terms,
- the specification of the primitive functions,
- the *ximpl* tactic for simplifying entailments.

Proof using.

```

intros. applys triple_app_fun. { reflexivity. } simpl.
applys triple_let.
{ apply triple_get. }

intros n'. simpl.
apply triple_hpure. intros →.
applys triple_let.
{ applys triple_conseq_frame.
  { applys triple_add. }
  { xsimpl. }
  { xsimpl. } }

intros m'. simpl.
apply triple_hpure. intros →.
{ applys triple_set. }

```

Qed.

Proof of succ_using_incr

Recall from Basic the function `succ_using_incr`.

```

Definition succ_using_incr : val :=
<{ fun 'n =>
  let 'r = val_ref 'n in
  incr 'r;
  let 'x = ! 'r in
  val_free 'r;
  'x }>.

```

Recall the specification of `succ_using_incr`.

```

Lemma triple_succ_using_incr : ∀ (n:int),

```

```

triple (trm_app succ_using_incr n)
  \[]
  (fun v => \[v = val_int (n+1)]).

```

Exercise: 4 stars, standard, especially useful (triple_succ_using_incr) Verify the function `triple_succ_using_incr`. Hint: follow the pattern of `triple_incr`. Hint: use *applys* `triple_seq` for reasoning about a sequence. Hint: use *applys* `triple_val` for reasoning about the final return value, namely `x`.

Proof using. *Admitted.*

□

Proof of factorec

Import Basic.Facto.

Recall from `Basic` the function `repeat_incr`.

OCaml:

```
let rec factorec n = if n <= 1 then 1 else n * factorec (n-1)
```

Definition factorec : val :=

```

<{ fix 'f 'n =>
  let 'b = ('n ≤ 1) in
  if 'b
    then 1
    else let 'x = 'n - 1 in
      let 'y = 'f 'x in
      'n × 'y }>.

```

Exercise: 4 stars, standard, especially useful (triple_factorec) Verify the function `factorec`. Hint: exploit `triple_app_fix` for reasoning about the recursive function. Hint: `triple_hpure'`, the corollary of `triple_hpure`, is helpful. Hint: exploit `triple_le` and `triple_sub` and `triple_mul` to reason about the behavior of the primitive operations involved. Hint: exploit *applys* `triple_if_case_if` as *C*. to reason about the conditional; alternatively, if using `triple_if_case`, you'll need to use the tactic `rew_bool_eq in *` to simplify, e.g., the expression `isTrue (m ≤ 1) = true`.

Lemma triple_factorec : ∀ n,

```

n ≥ 0 →
triple (factorec n)
  \[]
  (fun r => \[r = facto n]).
```

Proof using. *Admitted.*

□

End EXAMPLEPROGRAMS.

41.2.6 What's Next

The matter of the next chapter is to introduce additional technology to streamline the proof process, notably by:

- automating the application of the frame rule
- eliminating the need to manipulate program variables and substitutions during the verification proof.

The rest of this chapter is concerned with alternative statements for the reasoning rules, and with the proofs of the reasoning rules.

41.3 More Details

41.3.1 Alternative Specification Style for Pure Preconditions

Module DIVSPEC.

Recall the specification for division.

```
Parameter triple_div : ∀ n1 n2,
  n2 ≠ 0 →
  triple (val_div n1 n2)
  \ []
  (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Equivalently, we could place the requirement $n2 \neq 0$ in the precondition:

```
Parameter triple_div' : ∀ n1 n2,
  triple (val_div n1 n2)
  \[n2 ≠ 0]
  (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Let us formally prove that the two presentations are equivalent.

Exercise: 1 star, standard, especially useful (triple_div_from_triple_div') Prove *triple_div* by exploiting *triple_div'*. Hint: the key proof step is *applys* *triple_conseq*

```
Lemma triple_div_from_triple_div' : ∀ n1 n2,
  n2 ≠ 0 →
  triple (val_div n1 n2)
  \ []
  (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Proof using. Admitted.

□

Exercise: 2 stars, standard, especially useful (triple_div'_from_triple_div) Prove *triple_div'* by exploiting *triple_div*. Hint: the first key proof step is *applys triple_hpure*. Yet some preliminary rewriting is required for this tactic to apply. Hint: the second key proof step is *applys triple_conseq*.

```
Lemma triple_div'_from_triple_div : ∀ n1 n2,
  triple (val_div n1 n2)
  \[n2 ≠ 0]
  (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Proof using. Admitted.

□

As we said, placing pure preconditions outside of the triples makes it slightly more convenient to exploit specifications. For this reason, we adopt the style that precondition only contain the description of heap-allocated data structures.

End DIVSPEC.

41.3.2 The Combined let-frame Rule Rule

Module LETFRAME.

Recall the Separation Logic let rule.

```
Parameter triple_let : ∀ x t1 t2 Q1 H Q,
  triple t1 H Q1 →
  (forall v1, triple (subst x v1 t2) (Q1 v1) Q) →
  triple (trm_let x t1 t2) H Q.
```

At first sight, it seems that, to reason about `let x = t1 in t2` in a state described by precondition *H*, we need to first reason about *t1* in that same state. Yet, *t1* may well require only a subset of the state *H* to evaluate, and not all of *H*.

The “let-frame” rule combines the rule for let-bindings with the frame rule to make it more explicit that the precondition *H* may be decomposed in the form *H1 * H2*, where *H1* is the part needed by *t1*, and *H2* denotes the rest of the state. The part of the state covered by *H2* remains unmodified during the evaluation of *t1*, and appears as part of the precondition of *t2*.

The corresponding statement is as follows.

```
Lemma triple_let_frame : ∀ x t1 t2 Q1 H H1 H2 Q,
  triple t1 H1 Q1 →
  H ==> H1 \* H2 →
  (forall v1, triple (subst x v1 t2) (Q1 v1 \* H2) Q) →
  triple (trm_let x t1 t2) H Q.
```

Exercise: 3 stars, standard, especially useful (triple_let_frame) Prove the let-frame rule.

Proof using. *Admitted.*

□

End LETFRAME.

41.3.3 Proofs for the Rules for Terms

Module PROOFS.

The proofs for the Separation Logic reasoning rules all follow a similar pattern: first establish a corresponding rule for Hoare triples, then generalize it to a Separation Logic triple.

To establish a reasoning rule w.r.t. a Hoare triple, we reveal the definition expressed in terms of the big-step semantics.

Definition hoare ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : Prop := forall $s, H s \rightarrow \exists s' v, \text{eval } s t s' v \wedge Q v s'$.

Concretely, we consider a given initial state s satisfying the precondition, and we have to provide witnesses for the output value v and output state s' such that the reduction holds and the postcondition holds.

Then, to lift the reasoning rule from Hoare logic to Separation Logic, we reveal the definition of a Separation Logic triple.

Definition triple $t H Q := \forall H', \text{hoare } t (H \setminus^* H') (Q \setminus^+ H')$.

Recall that we already employed this two-step scheme in the previous chapter, e.g., to establish the consequence rule (*rule-conseq*).

Proof of *triple_val*

The big-step evaluation rule for values asserts that a value v evaluates to itself, without modification to the current state s .

Parameter eval_val : $\forall s v, \text{eval } s v s v$.

The Hoare version of the reasoning rule for values is as follows.

Lemma hoare_val : $\forall v H Q, H ==> Q v \rightarrow \text{hoare } (\text{trm_val } v) H Q$.

Proof using.

1. We unfold the definition of **hoare**. *introv M. intros s K0.*
 2. We provide the witnesses for the output value and heap. These witnesses are dictated by the statement of **eval_val**. $\exists s v. \text{splits}$.
 3. We invoke the big-step rule **eval_val** { *applys eval_val.* }
 4. We establish the postcondition, exploiting the entailment hypothesis. { *applys M. auto.* }
- Qed.

The Separation Logic version of the rule then follows.

```
Lemma triple_val : ∀ v H Q,
  H ==> Q v →
  triple (trm_val v) H Q.
```

Proof using.

1. We unfold the definition of `triple` to reveal a `hoare` judgment. *intros M. intros H'.*
2. We invoke the reasoning rule `hoare_val` that was just established. *applys hoare_val.*
3. We exploit the assumption and conclude using `xchange`. *xchange M.*

Qed.

Proof of `triple_seq`

The big-step evaluation rule for a sequence is given next.

```
Parameter eval_seq : ∀ s1 s2 s3 t1 t2 v1 v,
  eval s1 t1 s2 v1 →
  eval s2 t2 s3 v →
  eval s1 (trm_seq t1 t2) s3 v.
```

The Hoare triple version of the reasoning rule is proved as follows. This lemma, called `hoare_seq`, has the same statement as `triple_seq`, except with occurrences of `triple` replaced with `hoare`.

```
Lemma hoare_seq : ∀ t1 t2 H Q H1,
  hoare t1 H (fun v ⇒ H1) →
  hoare t2 H1 Q →
  hoare (trm_seq t1 t2) H Q.
```

Proof using.

1. We unfold the definition of `hoare`. Let $K0$ describe the initial state. *intros M1 M2. intros s K0. unfolds hoare.*
2. We exploit the first hypothesis to obtain information about the evaluation of the first subterm $t1$. The state before $t1$ executes is described by $K0$. The state after $t1$ executes is described by $K1$. *forwards (s1' & v1 & R1 & K1): (rm M1) K0.*
3. We exploit the second hypothesis to obtain information about the evaluation of the first subterm $t2$. The state before $t2$ executes is described by $K1$. The state after $t2$ executes is described by $K2$. *forwards (s2' & v2 & R2 & K2): (rm M2) K1.*
4. We provide witness for the output value and heap. They correspond to those produced by the evaluation of $t2$. $\exists s2' v2. \text{split.}$
5. We invoke the big-step rule. $\{ \text{applys eval_seq R1 R2.} \}$
6. We establish the final postcondition, which is directly inherited from the reasoning on $t2$. $\{ \text{apply K2.} \}$

Qed.

The Separation Logic reasoning rule is proved as follows.

```
Lemma triple_seq : ∀ t1 t2 H Q H1,
```

```

triple t1 H (fun v => H1) →
triple t2 H1 Q →
triple (trm_seq t1 t2) H Q.

```

Proof using.

1. We unfold the definition of `triple` to reveal a `hoare` judgment. *intros H'. unfolds triple.*
 2. We invoke the reasoning rule `hoare_seq` that we just established. *applys hoare_seq.*
 3. For the hypothesis on the first subterm `t1`, we can invoke directly our first hypothesis.
 $\{ \text{applys } M1. \}$
 $\{ \text{applys } M2. \}$
- Qed.

Proof of `triple_let`

Recall the big-step evaluation rule for a let-binding.

```

Parameter eval_let : ∀ s1 s2 s3 x t1 t2 v1 v,
  eval s1 t1 s2 v1 →
  eval s2 (subst x v1 t2) s3 v →
  eval s1 (trm_let x t1 t2) s3 v.

```

Exercise: 3 stars, standard, especially useful (`triple_let`) Following the same proof scheme as for `triple_seq`, establish the reasoning rule for `triple_let`, whose statement appears earlier in this file. Make sure to first state and prove the lemma `hoare_let`, which has the same statement as `triple_let` yet with occurrences of `triple` replaced with `hoare`.

□

41.3.4 Proofs for the Arithmetic Primitive Operations

Addition

Recall the evaluation rule for addition.

```

Parameter eval_add : ∀ s n1 n2,
  eval s (val_add (val_int n1) (val_int n2)) s (val_int (n1 + n2)).

```

In the proof, we will need to use the following result, established in the first chapter.

```

Parameter hstar_hpure_l : ∀ P H h,
  (\\[P] \\* H) h = (P ∧ H h).

```

As usual, we first establish a Hoare triple.

```

Lemma hoare_add : ∀ H n1 n2,
  hoare (val_add n1 n2)
  H

```

```
(fun r => \[r = val_int (n1 + n2)] \* H).
```

Proof using.

```
intros. intros s K0. ∃ s (val_int (n1 + n2)). split.
{ applys eval_add. }
{ rewrite hstar_hpure_l. split.
{ auto. }
{ applys K0. } }
```

Qed.

Deriving `triple_add` is then straightforward.

Lemma `triple_add` : $\forall n1\ n2,$

```
triple (val_add n1 n2)
\ []
(fun r => \[r = val_int (n1 + n2)]).
```

Proof using.

```
intros. intros H'. applys hoare_conseq.
{ applys hoare_add. } { xsimpl. } { xsimpl. auto. }
```

Qed.

Division

Recall the evaluation rule for division.

Parameter `eval_div'` : $\forall s\ n1\ n2,$

```
n2 ≠ 0 →
eval s (val_div (val_int n1) (val_int n2)) s (val_int (Z.quot n1 n2)).
```

Exercise: 3 stars, standard, optional (`triple_div`) Following the same proof scheme as for `triple_add`, establish the reasoning rule for `triple_div`. Make sure to first state and prove `hoare_div`, which is like `triple_div` except with `hoare` instead of `triple`.

□

41.3.5 Proofs for Primitive Operations Operating on the State

The proofs for establishing the Separation Logic reasoning rules for `ref`, `get` and `set` follow a similar proof pattern, that is, they go through the proofs of rules for Hoare triples.

Unlike before, however, the Hoare triples are not directly established with respect to the big-step evaluation rules. Instead, we start by proving corollaries to the big-step rules to reformulate them in a way that give already them a flavor of “Separation Logic”. Concretely, we reformulate the evaluation rules, which are expressed in terms of read and updates in finite maps, to be expressed instead entirely in terms of disjoint unions.

The introduction of these disjoint union operations then significantly eases the justification of the separating conjunctions that appear in the targeted Separation Logic triples.

In this section, the constructor *hval_val* appears. This constructor converts a “value” into a “heap value”. For the purpose, of this file, the two notion are identical. Yet, to allow for generalization to the semantics of allocation by blocks, we need to assume that states are finite maps from location to heap values. Heap values, of type *hval*, can be assumed to be defined by the following inductive data type.

```
Inductive hval : Type := | hval_val : val -> hval.
```

Read in a Reference

The big-step rule for *get p* requires that *p* be in the domain of the current state *s*, and asserts that the output value is the result of reading in *s* at location *p*.

```
Parameter eval_get : ∀ v s p,
  Fmap.indom s p →
  Fmap.read s p = v →
  eval s (val_get (val_loc p)) s v.
```

We reformulate this rule by isolating from the current state *s* the singleton heap made of the cell at location *p*, and let *s2* denote the rest of the heap. When the singleton heap is described as *Fmap.single p v*, then *v* is the result value returned by *get p*.

```
Lemma eval_get_sep : ∀ s s2 p v,
  s = Fmap.union (Fmap.single p v) s2 →
  eval s (val_get (val_loc p)) s v.
```

The proof of this lemma is of little interest. We show it only to demonstrate that it relies only a few basic facts related to finite maps.

Proof using.

```
intros v →. forwards Dv: Fmap.indom_single p v.
applys eval_get.
{ applys× Fmap.indom_union_l. }
{ rewrite× Fmap.read_union_l. rewrite× Fmap.read_single. }
```

Qed.

Our goal is to establish the triple:

triple (val_get *p*) (*p* \rightsquigarrow *v*) (fun *r* => $\setminus r = v \setminus^* (p \rightsquigarrow v)$).

Establishing this lemma requires us to reason about propositions of the form $(\setminus [P] \setminus^* H)$ *h* and $(p \rightsquigarrow v) h$. To that end, recall lemma *hstar_hpure_l*, which was already exploited in the proof of *triple_add*, and recall *hsingle_inv* from *Hprop*.

```
Parameter hsingl_inv: ∀ p v h,
  (p ∽> v) h →
  h = Fmap.single p v.
```

We establish the specification of *get* first w.r.t. to the *hoare* judgment.

```
Lemma hoare_get : ∀ H v p,
  hoare (val_get p)
```

```
((p ~~> v) \* H)
  (fun r => \[r = v] \* (p ~~> v) \* H).
```

Proof using.

```
intros. intros s K0.
exists s v. split.
{
  destruct K0 as (s1&s2&P1&P2&D&U).
  let E1: hsingle_inv P1. subst s1.
  applys eval_get_sep U.
}
{
  rewrite hstar_hpure_l. auto.
}
```

Qed.

Deriving the Separation Logic triple follows the usual pattern.

```
Lemma triple_get : ∀ v p,
  triple (val_get p)
    (p ~~> v)
    (fun r => \[r = v] \* (p ~~> v)).
```

Proof using.

```
intros. intros H'. applys hoare_conseq.
{
  applys hoare_get.
  {
    xsimpl.
    {
      xsimpl. auto.
    }
}
```

Qed.

Allocation of a Reference

Next, we consider the reasoning rule for operation *ref*, which involves a proof yet slightly more trickier than that for *get* and *set*.

The big-step evaluation rule for *ref* v extends the initial state s with an extra binding from p to v , for some fresh location p .

```
Parameter eval_ref : ∀ s v p,
  ↗ Fmap.indom s p →
  eval s (val_ref v) (Fmap.update s p v) (val_loc p).
```

Let us reformulate *eval_ref* to replace references to *Fmap.indom* and *Fmap.update* with references to *Fmap.single* and *Fmap.disjoint*. Concretely, *ref* v extends the state from $s1$ to $s1 \cup s2$, where $s2$ denotes the singleton heap *Fmap.single* $p v$, and with the requirement that *Fmap.disjoint* $s2 s1$, to capture freshness.

```
Lemma eval_ref_sep : ∀ s1 s2 v p,
  s2 = Fmap.single p v →
  Fmap.disjoint s2 s1 →
  eval s1 (val_ref v) (Fmap.union s2 s1) (val_loc p).
```

Proof using.

It is not needed to follow through this proof. *intros* $\rightarrow D$. *forwards* Dv : `Fmap.indom_single p v`.
`rewrite ← Fmap.update_eq_union_single.` *applys* \times `eval_ref`.
 $\{ \text{intros } N. \text{applys} \times \text{Fmap.disjoint_inv_not_indom_both } D N. \}$
Qed.

In order to apply the rules `eval_ref` or `eval_ref_sep`, we need to be able to synthesize fresh locations. The following lemma (from *LibSepFmap.v*) captures the existence, for any state s , of a (non-null) location p not already bound in s .

Parameter exists_fresh : $\forall s,$
 $\exists p, \neg \text{Fmap.indom } s p \wedge p \neq \text{null}.$

We reformulate the lemma above in a way that better matches the premise of the lemma `eval_ref_sep`, which we need to apply for establishing the specification of `ref`.

This reformulation, shown below, asserts that, for any h , there exists a non-null location p such that the singleton heap `Fmap.single p v` is disjoint from h .

Lemma single_fresh : $\forall h v,$
 $\exists p, \text{Fmap.disjoint } (\text{Fmap.single } p v) h.$

Proof using.

It is not needed to follow through this proof. *intros*. *forwards* ($p \& F \& N$): `exists_fresh h`.
 $\exists p.$ *applys* \times `Fmap.disjoint_single_of_not_indom`.
Qed.

The proof of the Hoare triple for `ref` is as follows.

Lemma hoare_ref : $\forall H v,$
`hoare (val_ref v)`
 H
 $(\text{fun } r \Rightarrow (\exists p, [r = \text{val_loc } p] \wedge p \sim v) \wedge H).$

Proof using.

intros. *intros* $s1 K0$.
forwards \times ($p \& D$): (`single_fresh s1 v`).
 $\exists ((\text{Fmap.single } p v) \wedge s1) (\text{val_loc } p).$ *split*.
 $\{$
 $\quad \text{applys eval_ref_sep } D.$ *auto*. $\}$
 $\{$
 $\quad \text{applys hstar_intro.}$
 $\quad \{ \exists p.$ *rewrite hstar_hpure_l*.
 $\quad \quad \text{split.} \{ \text{auto.} \} \{ \text{applys} \neg \text{hsingle_intro.} \} \}$
 $\quad \{ \text{applys } K0. \}$
 $\quad \{ \text{applys } D. \} \}$
Qed.

We then derive the Separation Logic triple as usual.

```

Lemma triple_ref : ∀ v,
  triple (val_ref v)
  \[]
  (funloc p ⇒ p ~~> v).

```

Proof using.

```

intros. intros H'. applys hoare_conseq.
{ applys hoare_ref. }
{ xsimpl. }
{ xsimpl. auto. }

```

Qed.

End PROOFS.

41.4 Optional Material

41.4.1 Reasoning Rules for Recursive Functions

This reasoning rules for functions immediately generalizes to recursive functions. A term describing a recursive function is written `trm_fix f x t1`, and the corresponding value is written `val_fix f x t1`.

```

Parameter triple_fix : ∀ f x t1 H Q,
H ==> Q (val_fix f x t1) →
triple (trm_fix f x t1) H Q.

```

The reasoning rule that corresponds to beta-reduction for a recursive function involves two substitutions: a first substitution for recursive occurrences of the function, followed with a second substitution for the argument provided to the call.

```

Parameter triple_app_fix : ∀ v1 v2 f x t1 H Q,
v1 = val_fix f x t1 →
triple (subst x v2 (subst f v1 t1)) H Q →
triple (trm_app v1 v2) H Q.

```

Other Proofs of Reasoning Rules for Terms

Module PROOFSCONTINUED.

Proof of `triple_fun` and `triple_fix`

The proofs for `triple_fun` and `triple_fix` are essentially identical to that of `triple_val`, so we do not include them here.

Proof of *triple_if*

Recall the reasoning rule for conditionals. Recall that this rule is stated by factorizing the premises.

```
Lemma eval_if : ∀ s1 s2 b v t1 t2,
  eval s1 (if b then t1 else t2) s2 v →
  eval s1 (trm_if b t1 t2) s2 v.
```

Proof using.

```
intros. case_if; applys eval_if; auto_false.
```

Qed.

The reasoning rule for conditional w.r.t. Hoare triples is as follows.

```
Lemma hoare_if_case : ∀ b t1 t2 H Q,
  (b = true → hoare t1 H Q) →
  (b = false → hoare t2 H Q) →
  hoare (trm_if b t1 t2) H Q.
```

Proof using.

```
introv M1 M2. intros s K0. destruct b.
{ forwards× (s1'&v1&R1&K1): (rm M1) K0.
  ∃ s1' v1. split×. { applys× eval_if. } }
{ forwards× (s1'&v1&R1&K1): (rm M2) K0.
  ∃ s1' v1. split×. { applys× eval_if. } }
```

Qed.

The corresponding Separation Logic reasoning rule is as follows.

```
Lemma triple_if_case : ∀ b t1 t2 H Q,
  (b = true → triple t1 H Q) →
  (b = false → triple t2 H Q) →
  triple (trm_if (val_bool b) t1 t2) H Q.
```

Proof using.

```
unfold triple. introv M1 M2. intros H'.
applys hoare_if_case; intros Eb.
{ applys× M1. }
{ applys× M2. }
```

Qed.

Observe that the above proofs contain a fair amount of duplication, due to the symmetry between the $b = \text{true}$ and $b = \text{false}$ branches.

If we state the reasoning rules using Coq's conditional just like it appears in the evaluation rule *eval_if*, we can better factorize the proof script.

```
Lemma hoare_if : ∀ (b:bool) t1 t2 H Q,
  hoare (if b then t1 else t2) H Q →
  hoare (trm_if b t1 t2) H Q.
```

Proof using.

$\text{introv } M1. \text{ intros } s K0.$
 $\text{forwards } (s' \& v \& R1 \& K1): (\text{rm } M1) K0.$
 $\exists s' v. \text{ split. } \{ \text{applys eval_if } R1. \} \{ \text{applys } K1. \}$
 Qed.

Lemma triple_if : $\forall b t1 t2 H Q,$
 $\text{triple } (\text{if } b \text{ then } t1 \text{ else } t2) H Q \rightarrow$
 $\text{triple } (\text{trm_if } (\text{val_bool } b) t1 t2) H Q.$

Proof using.

unfold triple. introv M1. intros H'. applys hoare_if. applys M1.

Qed.

Proof of *triple_app_fun*

The reasoning rule for an application asserts that the a pre- and poscondition hold for a beta-redex $(\text{val_fun } x \ t1) \ v2$ provided that they hold for the term $\text{subst } x \ v2 \ t1$.

This result follows directly from the big-step evaluation rule for applications.

Parameter eval_app_fun : $\forall s1 s2 v1 v2 x t1 v,$
 $v1 = \text{val_fun } x t1 \rightarrow$
 $\text{eval } s1 (\text{subst } x v2 t1) s2 v \rightarrow$
 $\text{eval } s1 (\text{trm_app } v1 v2) s2 v.$

Exercise: 2 stars, standard, optional (hoare_app_fun) Prove the lemma *hoare_app_fun*.

Lemma hoare_app_fun : $\forall v1 v2 x t1 H Q,$
 $v1 = \text{val_fun } x t1 \rightarrow$
 $\text{hoare } (\text{subst } x v2 t1) H Q \rightarrow$
 $\text{hoare } (\text{trm_app } v1 v2) H Q.$

Proof using. Admitted.

□

Exercise: 2 stars, standard, optional (triple_app_fun) Prove the lemma *triple_app_fun*.

Lemma triple_app_fun : $\forall x v1 v2 t1 H Q,$
 $v1 = \text{val_fun } x t1 \rightarrow$
 $\text{triple } (\text{subst } x v2 t1) H Q \rightarrow$
 $\text{triple } (\text{trm_app } v1 v2) H Q.$

Proof using. Admitted.

□

Deallocation of a Reference

Optional contents: this section may be safely skipped.

Last, we consider the reasoning rule for operation *free*. We leave this one as exercise.

Recall the big-step evaluation rule for *free p*.

```
Parameter eval_free : ∀ s p,
  Fmap.indom s p →
  eval s (val_set (val_loc p)) (Fmap.remove s p) val_unit.
```

Let us reformulate *eval_free* to replace references to *Fmap.indom* and *Fmap.remove* with references to *Fmap.single* and *Fmap.union* and *Fmap.disjoint*. The details are not essential, thus omitted.

```
Parameter eval_free_sep : ∀ s1 s2 v p,
  s1 = Fmap.union (Fmap.single p v) s2 →
  Fmap.disjoint (Fmap.single p v) s2 →
  eval s1 (val_free p) s2 val_unit.
```

Exercise: 3 stars, standard, optional (hoare_free) Prove the Hoare triple for the operation *free*. Hint: exploit the lemma *eval_free_sep*.

```
Lemma hoare_free : ∀ H p v,
  hoare (val_free (val_loc p))
    ((p ~~> v) \* H)
    (fun _ ⇒ H).
```

Proof using. Admitted.

□

Exercise: 1 star, standard, optional (triple_free) Derive from the Hoare triple for the operation *free* the corresponding Separation Logic triple. Hint: adapt the proof of lemma *triple_set*.

```
Lemma triple_free : ∀ p v,
  triple (val_free (val_loc p))
    (p ~~> v)
    (fun _ ⇒ []).
```

Proof using. Admitted.

□

Write in a Reference

The big-step evaluation rule for **set** *p v* updates the initial state *s* by re-binding the location *p* to the value *v*. The location *p* must already belong to the domain of *s*.

```
Parameter eval_set : ∀ m p v,
  Fmap.indom m p →
```

eval m (**val_set** (**val_loc** p) v) (**Fmap.update** m p v) **val_unit**.

As for *get*, we first reformulate this lemma, to replace references to *Fmap.indom* and *Fmap.update* with references to *Fmap.union*, *Fmap.single*, and *Fmap.disjoint*, to prepare for the introduction of separating conjunctions.

```
Lemma eval_set_sep : ∀ s1 s2 h2 p v1 v2,
  s1 = Fmap.union (Fmap.single p v1) h2 →
  s2 = Fmap.union (Fmap.single p v2) h2 →
  Fmap.disjoint (Fmap.single p v1) h2 →
  eval s1 (val_set (val_loc p) v2) s2 val_unit.
```

Proof using.

It is not needed to follow through this proof. *introv → → D. forwards Dv: Fmap.indom_single p v1.*

```
applys_eq eval_set.
{ rewrite× Fmap.update_union_l. fequals.
  rewrite× Fmap.update_single. }
{ applys× Fmap.indom_union_l. }
```

Qed.

The proof of the Hoare rule for **set** makes use of the following fact (from *LibSepFmap.v*) about *Fmap.disjoint*: when one of its arguments is a singleton map, the value stored in that singleton map is irrelevant.

Check *Fmap.disjoint_single_set* : forall p $v1$ $v2$ $h2$, *Fmap.disjoint* (*Fmap.single* p $v1$) $h2$ -> *Fmap.disjoint* (*Fmap.single* p $v2$) $h2$.

Exercise: 5 stars, standard, optional (hoare_set) Prove the lemma **hoare_set**. Hints:

- exploit the evaluation rule **eval_set_sep** presented above,
- exploit the lemma *Fmap.disjoint_single_set* presented above,
- to obtain an elegant proof, prefer invoking the lemmas **hsingle_intro**, **hsingle_inv**, **hstar_intro**, and **hstar_inv**, rather than unfolding the definitions of **hstar** and **hsingle**.

```
Lemma hoare_set : ∀ H v p v',
  hoare (val_set (val_loc p) v)
    ((p ~~> v') \* H)
    (fun _ ⇒ (p ~~> v) \* H).
```

Proof using. *Admitted.*

□

We then derive the Separation Logic triple as usual.

```
Lemma triple_set : ∀ w p v,
  triple (val_set (val_loc p) w)
    (p ~~> v)
```

```
(fun _ => p ~~> w).
```

Proof using.

```
intros. intros H'. applys hoare_conseq.  
{ applys hoare_set. }  
{ xsimpl. }  
{ xsimpl. }
```

Qed.

End PROOFSCONTINUED.

Proofs Revisited using the triple_of_hoare Lemma

Module PROOFSFACTORIZATION.

The proof that, e.g., `triple_add` is a consequence of `hoare_add` follows the same pattern as many other similar proofs, each time invoking the lemma `hoare_conseq`. Thus, we could attempt at factorizing this proof pattern. The following lemma corresponds to such an attempt.

Exercise: 2 stars, standard, optional (`triple_of_hoare`) Prove the lemma `triple_of_hoare` stated below.

```
Lemma triple_of_hoare : ∀ t H Q,  
  (∀ H', ∃ Q', hoare t (H \* H') Q'  
    ∧ Q' ==> Q \*+ H') →  
  triple t H Q.
```

Proof using. Admitted.

□

Exercise: 2 stars, standard, optional (`triple_add'`) Prove that `triple_add` is a consequence of `hoare_add` by exploiting `triple_of_hoare`.

```
Lemma triple_add' : ∀ n1 n2,  
  triple (val_add n1 n2)  
  ([]  
   (fun r => [r = val_int (n1 + n2)])).
```

Proof using. Admitted.

□

End PROOFSFACTORIZATION.

Triple for Terms with Same Semantics

Module PROOFSSAMESEMANTICS.

A general principle is that if t_1 has the same semantics as t_2 (w.r.t. the big-step evaluation judgment `eval`), then t_1 and t_2 satisfy the same triples.

Let us formalize this principle.

Two (closed) terms are semantically equivalent, written `trm_equiv t1 t2`, if two terms, when evaluated in the same state, produce the same output.

Definition `trm_equiv (t1 t2:trm) : Prop :=`
 $\forall s s' v, \mathbf{eval} s t1 s' v \leftrightarrow \mathbf{eval} s t2 s' v.$

Two terms that are equivalent satisfy the same Separation Logic triples (and the same Hoare triples).

Indeed, the definition of a Separation Logic triple directly depends on the notion of Hoare triple, and the latter directly depends on the semantics captured by the predicate `eval`.

Let us formalize the result in three steps.

`eval_like t1 t2` asserts that t_2 evaluates like t_1 . In particular, this relation hold whenever t_2 reduces in small-step to t_1 .

Definition `eval_like (t1 t2:trm) : Prop :=`
 $\forall s s' v, \mathbf{eval} s t1 s' v \rightarrow \mathbf{eval} s t2 s' v.$

For example `eval_like t (trm_let x t x)` holds, reflecting the fact that `let x = t in x` reduces in small-step to t .

Lemma `eval_like_eta_reduction : \forall (t:trm) (x:var),`
 $\mathbf{eval_like} t (\mathbf{trm_let} x t x).$

Proof using.

`intros R. applys eval_let R.`
`simpl. rewrite var_eq_spec. case_if. applys eval_val.`

Qed.

It turns out that the symmetric relation `eval_like (trm_let x t x) t` also holds: the term t does not exhibit more behaviors than those of `let x = t in x`.

Lemma `eval_like_eta_expansion : \forall (t:trm) (x:var),`
 $\mathbf{eval_like} (\mathbf{trm_let} x t x) t.$

Proof using.

`intros R. inverts R as. introv R1 R2.`
`simpl in R2. rewrite var_eq_spec in R2. case_if.`
`inverts R2. auto.`

Qed.

We deduce that a term t denotes a program equivalent to the program `let x = t in x`.

Lemma `trm_equiv_eta : \forall (t:trm) (x:var),`
 $\mathbf{trm_equiv} t (\mathbf{trm_let} x t x).$

Proof using.

`intros. intros s s' v. iff M.`
 $\{ \text{applys eval_like_eta_reduction } M. \}$
 $\{ \text{applys eval_like_eta_expansion } M. \}$

Qed.

If eval_like $t_1 t_2$, then any triple that holds for t_1 also holds for t_2 .

Lemma hoare_eval_like : $\forall t_1 t_2 H Q,$

eval_like $t_1 t_2 \rightarrow$

hoare $t_1 H Q \rightarrow$

hoare $t_2 H Q.$

Proof using.

introv E M1 K0. forwards ($s' \& v \& R1 \& K1$): M1 K0.

$\exists s' v.$ split. { applys E R1. } { applys K1. }

Qed.

Lemma triple_eval_like : $\forall t_1 t_2 H Q,$

eval_like $t_1 t_2 \rightarrow$

triple $t_1 H Q \rightarrow$

triple $t_2 H Q.$

Proof using.

introv E M1. intros H'. applys hoare_eval_like E. applys M1.

Qed.

It follows that if two terms are equivalent, then they admit the same triples.

Lemma triple_trm_equiv : $\forall t_1 t_2 H Q,$

trm_equiv $t_1 t_2 \rightarrow$

triple $t_1 H Q \leftrightarrow$ triple $t_2 H Q.$

Proof using.

introv E. unfolds trm_equiv. iff M.

{ applys triple_eval_like M. introv R. applys× E. }

{ applys triple_eval_like M. introv R. applys× E. }

Qed.

The reasoning rule triple_eval_like has a number of practical applications. One, show below, is to revisit the proof of triple_app_fun in a much more succinct way, by arguing that trm_app (val_fun x t1) v2 and subst x v2 t1 are equivalent terms, hence they admit the same behavior.

Lemma triple_app_fun : $\forall x v_1 v_2 t_1 H Q,$

$v_1 = \text{val_fun } x t_1 \rightarrow$

triple (subst x v2 t1) H Q \rightarrow

triple (trm_app v1 v2) H Q.

Proof using.

introv E M1. applys triple_eval_like M1.

introv R. applys eval_app_fun E R.

Qed.

Another application is the following rule, which allows to modify the parenthesis structure of a sequence.

Exercise: 3 stars, standard, optional (triple_trm_seq_assoc) Prove that the term $t1; (t2; t3)$ satisfies the same triples as the term $(t1;t2); t3$.

```
Lemma triple_trm_seq_assoc : ∀ t1 t2 t3 H Q,
  triple (trm_seq (trm_seq t1 t2) t3) H Q →
  triple (trm_seq t1 (trm_seq t2 t3)) H Q.
```

Proof using. Admitted.

□

Such a change in the parenthesis structure of a sequence can be helpful to apply the frame rule around $t1;t2$, for example.

Another useful application of the lemma `triple_eval_like` appears in chapter `Affine`, for proving the equivalence of two versions of the garbage collection rule.

End PROOFSSAMESEMANTICS.

41.4.2 Alternative Specification Style for Result Values.

Module MATCHSTYLE.

Recall the specification for the function `ref`.

```
Parameter triple_ref : ∀ v,
  triple (val_ref v)
  \[]
  (fun r ⇒ \exists p, \[r = val_loc p] \* p ~~> v).
```

Its postcondition could be equivalently stated by using, instead of an existential quantifier \exists , a pattern matching.

```
Parameter triple_ref' : ∀ v,
  triple (val_ref v)
  \[]
  (fun r ⇒ match r with
    | val_loc p ⇒ (p ~~> v)
    | _ ⇒ \[False]
  end).
```

However, the pattern-matching presentation is less readable and would be fairly cumbersome to work with in practice. Thus, we systematically prefer using existentials.

End MATCHSTYLE.

41.4.3 Historical Notes

Gordon 1989 (in Bib.v) presents the first mechanization of Hoare logic in a proof assistant, using the HOL tool. Gordon's pioneering work was followed by numerous formalizations of Hoare logic, targeting various programming languages.

The original presentation of Separation Logic (1999-2001) consists of a set of rules written down on paper. These rules were not formally described in a proof assistant. Nevertheless, mechanized presentation of Separation Logic emerged a few years later.

Yu, Hamid, and Shao 2003 (in Bib.v) present the CAP framework for the verification in Coq of assembly-level code. This framework exploits separation logic style specifications, with predicate for lists and list segments involving the separating conjunction operator.

In parallel, *Weber* 2004 (in Bib.v), advised by Nipkow, developed the first mechanization of the rules of Separation Logic for a while language, using the Isabelle/HOL tool. His presentation is quite close from the original, paper presentation.

Numerous mechanized presentations of Separation Logic, targeting various languages (assembly, C, core-Java, ML, etc.) and using various tools (Isabelle/HOL, Coq, PVS, HOL4, HOL). For a detailed list, as of 2020, we refer to Section 10.3 from the paper: http://www.chargueraud.org/research/2020/seq_seplogic/seq_seplogic.pdf.

Chapter 42

Library `SLF.WPsem`

42.1 WPsem: Semantics of Weakest Preconditions

Set Implicit Arguments.

From *SLF* Require Export Rules.

Implicit Types f : var.

Implicit Types b : **bool**.

Implicit Types v : val.

Implicit Types h : heap.

Implicit Types P : Prop.

Implicit Types H : hprop.

Implicit Types Q : val \rightarrow hprop.

42.2 First Pass

In previous chapters, we have introduced the notion of Separation Logic triple, written `triple t H Q`.

In this chapter, we introduce the notion of “weakest precondition” for Separation Logic triples, written `wp t Q`.

The intention is for `wp t Q` to be a heap predicate (of type `hprop`) such that $H ==> \text{wp } t \ Q$ if and only if `triple t H Q` holds.

The benefits of introducing weakest preconditions is two-fold:

- the use of `wp` greatly reduces the number of structural rules required, and thus reduces accordingly the number of tactics required for carrying out proofs in practice;
- the predicate `wp` will serve as guidelines for setting up in the next chapter a “characteristic formula generator”, which is the key ingredient at the heart of the implementation of the CFML tool.

This chapter presents:

- the notion of weakest precondition, as captured by `wp`,
- the reformulation of structural rules in `wp`-style,
- the reformulation of reasoning rules in `wp`-style,
- (optional) alternative, equivalent definitions for `wp`, and alternative proofs for deriving `wp`-style reasoning rules.

42.2.1 Notion of Weakest Precondition

We next introduce a function `wp`, called “weakest precondition”. Given a term `t` and a postcondition `Q`, the expression `wp t Q` denotes a heap predicate `wp t Q` such that, for any heap predicate `H`, the entailment `H ==> wp t Q` is equivalent to `triple t H Q`.

The notion of `wp` usually sounds fairly mysterious at first sight. It will make more sense when we describe the properties of `wp`.

`Parameter wp : trm → (val → hprop) → hprop.`

`Parameter wp_equiv : ∀ t H Q,`

$$(H ==> \text{wp } t \ Q) \leftrightarrow (\text{triple } t \ H \ Q).$$

The `wp t Q` is called “weakest precondition” for two reasons: because (1) it is a precondition, and (2) it is the weakest one, as we explain next.

First, `wp t Q` is always a “valid precondition” for a triple associated with the term `t` and the postcondition `Q`.

`Lemma wp_pre : ∀ t Q,`

$$\text{triple } t (\text{wp } t \ Q) \ Q.$$

`Proof using. intros. rewrite ← wp_equiv. applys himpl_refl. Qed.`

Second, `wp t Q` is the “weakest” of all valid preconditions for the term `t` and the postcondition `Q`, in the sense that any other valid precondition `H`, i.e. satisfying `triple t H Q`, is such that `H` entails `wp t Q`.

`Lemma wp_weakest : ∀ t H Q,`

$$\text{triple } t H Q \rightarrow$$

$$H ==> \text{wp } t \ Q.$$

`Proof using. introv M. rewrite wp_equiv. applys M. Qed.`

In other words, `wp t Q` is the “smallest” `H` satisfying `triple t H Q` with respect to the order on heap predicates induced by the entailment relation \Rightarrow .

There are several equivalent ways to define `wp`, as we show in the optional contents of this chapter. It turns out that the equivalence $(H ==> \text{wp } t \ Q) \leftrightarrow (\text{triple } t \ H \ Q)$ fully characterizes the predicate `wp`, and that it is all we need to carry out formal reasoning.

For this reason, we postpone to further on in this chapter the description of alternative, direct definitions for `wp`.

42.2.2 Structural Rules in Weakest-Precondition Style

We next present reformulations of the frame rule and of the rule of consequence in “weakest-precondition style”.

The Frame Rule

The frame rule for wp asserts that $(\text{wp } t \ Q) \setminus^* H$ entails $\text{wp } t (Q \setminus^{*+} H)$. This statement can be read as follows: if you own both a piece of state satisfying H and a piece of state in which the execution of t produces (a result and an output value satisfying) Q , then you own a piece of state in which the execution of t produces $Q \setminus^{*+} H$, that is, produces both Q and H .

```
Lemma wp_frame : ∀ t H Q,
  (wp t Q) \setminus^* H ==> wp t (Q \setminus^{*+} H).
```

The lemma is proved by exploiting the frame rule for triples and the equivalence that characterizes wp .

Proof using.

```
intros. rewrite wp_equiv.
applys triple_frame. rewrite × ← wp_equiv.
```

Qed.

The connection with the frame might not be totally obvious. Recall the frame rule for triples.

$\text{triple } t \ H1 \ Q \rightarrow \text{triple } t (H1 \setminus^* H) (Q \setminus^{*+} H)$

Let us replace the form $\text{triple } t \ H \ Q$ with the form $H ==> \text{wp } t \ Q$. We obtain the following statement.

```
Lemma wp_frame_trans : ∀ t H1 Q H,
  H1 ==> wp t Q →
  (H1 \setminus^* H) ==> wp t (Q \setminus^{*+} H).
```

If we exploit transitivity of entailment to eliminate $H1$, then we obtain exactly wp_frame , as illustrated by the proof script below.

Proof using. *introv M. xchange M. applys× wp_frame. Qed.*

The Rule of Consequence

The rule of consequence for wp materializes as a covariance property: it asserts that $\text{wp } t \ Q$ is covariant in Q . In other words, if one weakens Q , then one weakens $\text{wp } t \ Q$. The corresponding formal statement appears next.

```
Lemma wp_conseq : ∀ t Q1 Q2,
  Q1 ==> Q2 →
  wp t Q1 ==> wp t Q2.
```

Proof using.

introv M. rewrite wp_equiv. applys \times triple_conseq (wp t Q1) M. applys wp_pre.
Qed.

The connection with the rule of consequence is, again, not totally obvious. Recall the rule of consequence for triples.

$\text{triple t H1 Q1} \rightarrow \text{H2} ==> \text{H1} \rightarrow \text{Q1} ==> \text{Q2} \rightarrow \text{triple t H2 Q2}$

Let us replace the form triple t H Q with the form $H ==> \text{wp t Q}$. We obtain the following statement:

Lemma wp_conseq_trans : $\forall t H1 H2 Q1 Q2,$
 $H1 ==> \text{wp t Q1} \rightarrow$
 $H2 ==> \text{H1} \rightarrow$
 $Q1 ==> \text{Q2} \rightarrow$
 $H2 ==> \text{wp t Q2}.$

If we exploit transitivity of entailment to eliminate $H1$ and $H2$, then we obtain exactly wp_conseq , as illustrated below.

Proof using.

introv M WH WQ. xchange WH. xchange M. applys wp_conseq WQ.
Qed.

The Extraction Rules

The extraction rules `triple_hpure` and `triple_hexists` have no specific counterpart with the `wp` presentation. Indeed, in a weakest-precondition style presentation, the extraction rules for triples correspond exactly to the extraction rules for entailment.

To see why, consider for example the rule `triple_hpure`.

Parameter triple_hpure : $\forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{triple t H Q}) \rightarrow$
 $\text{triple t } (\setminus [P] \setminus * H) Q.$

Replacing the form triple t H Q with $H ==> \text{wp t Q}$ yields the following statement.

Lemma triple_hpure_with_wp : $\forall t H Q (P:\text{Prop}),$
 $(P \rightarrow (H ==> \text{wp t Q})) \rightarrow$
 $(\setminus [P] \setminus * H) ==> \text{wp t Q}.$

The above implication is just a special case of the extraction lemma for pure facts on the left on an entailment, named `himpl_hstar_hpure_l`, and whose statement is as follows.

$(P \rightarrow (H ==> H')) \rightarrow (\setminus P \setminus * H) ==> H'.$

Instantiating H' with wp t Q proves `triple_hpure_with_wp`.

Proof using. *introv M. applys himpl_hstar_hpure_l M. Qed.*

A similar reasoning applies to the extraction rule for existentials.

42.2.3 Reasoning Rules for Terms, in Weakest-Precondition Style

Rule for Values

Recall the rule *triple_val* which gives a reasoning rule for establishing a triple for a value *v*.

Parameter *triple_val* : $\forall v H Q,$
 $H \implies Q v \rightarrow$
 $\text{triple}(\text{trm_val } v) H Q.$

If we rewrite this rule in *wp* style, we obtain the rule below.

$H \implies Q v \rightarrow H \implies \text{wp}(\text{trm_val } v) Q.$

By exploiting transitivity of entailment, we can eliminate *H*. We obtain the following statement, which reads as follows: if you own a state satisfying *Q v*, then you own a state from which the evaluation of the value *v* produces *Q*.

Lemma *wp_val* : $\forall v Q,$
 $Q v \implies \text{wp}(\text{trm_val } v) Q.$

Proof using.

intros. rewrite *wp_equiv*. applys \times *triple_val*.

Qed.

We can verify that, when migrating to the *wp* presentation, we have not lost any expressiveness. To that end, we prove that *triple_val* is derivable from *wp_val*.

Lemma *triple_val_derived_from_wp_val* : $\forall v H Q,$
 $H \implies Q v \rightarrow$
 $\text{triple}(\text{trm_val } v) H Q.$

Proof using. introv *M*. rewrite \leftarrow *wp_equiv*. xchange *M*. applys *wp_val*. Qed.

Rule for Sequence

Recall the reasoning rule for a sequence *trm_seq t1 t2*.

Parameter *triple_seq* : $\forall t1 t2 H Q H1,$
 $\text{triple } t1 H (\text{fun } v \Rightarrow H1) \rightarrow$
 $\text{triple } t2 H1 Q \rightarrow$
 $\text{triple}(\text{trm_seq } t1 t2) H Q.$

Replacing *triple t H Q* with *H ==> wp t Q* throughout the rule gives the statement below.

$H \implies (\text{wp } t1) (\text{fun } v \Rightarrow H1) \rightarrow H1 \implies (\text{wp } t2) Q \rightarrow H \implies \text{wp}(\text{trm_seq } t1 t2) Q.$

This entailment holds for any *H* and *H1*. Let us specialize it to $H1 := (\text{wp } t2) Q$ and $H := (\text{wp } t1) (\text{fun } v \Rightarrow (\text{wp } t2) Q)$.

This leads us to the following statement, which reads as follows: if you own a state from which the evaluation of *t1* produces a state from which the evaluation of *t2* produces the postcondition *Q*, then you own a state from which the evaluation of the sequence *t1;t2* produces *Q*.

Lemma `wp_seq` : $\forall t1\ t2\ Q,$
 $\wp t1 (\text{fun } v \Rightarrow \wp t2 Q) ==> \wp (\text{trm_seq } t1\ t2) Q.$

Proof using.

```
intros. rewrite wp_equiv. applys triple_seq.
{ rewrite $\times$  ← wp_equiv. }
{ rewrite $\times$  ← wp_equiv. }
```

Qed.

Exercise: 2 stars, standard, optional (`triple_seq_from_wp_seq`) Check that `wp_seq` is just as expressive as `triple_seq`, by proving that `triple_seq` is derivable from `wp_seq` and from the structural rules for `wp` and/or the structural rules for `triple`.

Lemma `triple_seq_from_wp_seq` : $\forall t1\ t2\ H\ Q\ H1,$
 $\text{triple } t1\ H (\text{fun } v \Rightarrow H1) \rightarrow$
 $\text{triple } t2\ H1\ Q \rightarrow$
 $\text{triple } (\text{trm_seq } t1\ t2)\ H\ Q.$

Proof using. *Admitted.*

□

42.3 More Details

42.3.1 Other Reasoning Rules for Terms

Rule for Functions

Recall the reasoning rule for a term `trm_fun` $x\ t1$, which evaluates to the value `val_fun` $x\ t1$.

Parameter `triple_fun` : $\forall x\ t1\ H\ Q,$
 $H ==> Q (\text{val_fun } x\ t1) \rightarrow$
 $\text{triple } (\text{trm_fun } x\ t1)\ H\ Q.$

The rule for functions follow exactly the same pattern as for values.

Lemma `wp_fun` : $\forall x\ t\ Q,$
 $Q (\text{val_fun } x\ t) ==> \wp (\text{trm_fun } x\ t) Q.$
Proof using. `intros. rewrite wp_equiv. applys \times triple_fun.` **Qed.**

A similar rule holds for the evaluation of a recursive function.

Lemma `wp_fix` : $\forall f\ x\ t\ Q,$
 $Q (\text{val_fix } f\ x\ t) ==> \wp (\text{trm_fix } f\ x\ t) Q.$
Proof using. `intros. rewrite wp_equiv. applys \times triple_fix.` **Qed.**

Rule for Conditionals

Recall the reasoning rule for a term `triple_if` $b\ t1\ t2$.

```

Parameter triple_if : ∀ b t1 t2 H Q,
  triple (if b then t1 else t2) H Q →
  triple (trm_if (val_bool b) t1 t2) H Q.

```

Replacing `triple` using `wp` entailments yields:

$H \implies wp(\text{if } b \text{ then } t1 \text{ else } t2) Q \rightarrow H \implies wp(\text{trm_if}(\text{val_bool } b) t1 t2) Q.$

which simplifies by transitivity to:

$wp(\text{if } b \text{ then } t1 \text{ else } t2) Q \implies wp(\text{trm_if}(\text{val_bool } b) t1 t2) Q.$

This statement corresponds to the `wp`-style reasoning rule for conditionals. The proof appears next.

```

Lemma wp_if : ∀ b t1 t2 Q,
  wp(if b then t1 else t2) Q ==> wp(trm_if (val_bool b) t1 t2) Q.

```

Proof using.

```
intros. rewrite wp_equiv. applys triple_if. rewrite× ← wp_equiv.
```

Qed.

Rule for Let-Bindings

Recall the reasoning rule for a term `trm_let x t1 t2`.

```

Parameter triple_let : ∀ x t1 t2 Q1 H Q,
  triple t1 H Q1 →
  (forall v1, triple (subst x v1 t2) (Q1 v1) Q) →
  triple (trm_let x t1 t2) H Q.

```

The rule of `trm_let x t1 t2` is very similar to that for `trm_seq`, the only difference being the substitution of `x` by `v` in `t2`, where `v` denotes the result of `t1`.

```

Lemma wp_let : ∀ x t1 t2 Q,
  wp t1 (fun v1 => wp (subst x v1 t2) Q) ==> wp (trm_let x t1 t2) Q.

```

Proof using.

```
intros. rewrite wp_equiv. applys triple_let.
{ rewrite× ← wp_equiv. }
{ intros v. rewrite× ← wp_equiv. }
```

Qed.

Rule For Function Applications

Recall the reasoning rule for an application `(val_fun x t1) v2`.

```

Parameter triple_app_fun : ∀ x v1 v2 t1 H Q,
  v1 = val_fun x t1 →
  triple (subst x v2 t1) H Q →
  triple (trm_app v1 v2) H Q.

```

The corresponding `wp` rule is stated and proved next.

```

Lemma wp_app_fun : ∀ x v1 v2 t1 Q,

```

```

v1 = val_fun x t1 →
wp (subst x v2 t1) Q ==> wp (trm_app v1 v2) Q.

```

Proof using.

```

introv EQ1. rewrite wp_equiv. applys× triple_app_fun.
rewrite× ← wp_equiv.

```

Qed.

A similar rule holds for the application of a recursive function.

42.4 Optional Material

42.4.1 A Concrete Definition for Weakest Precondition

Module WPHIGHLEVEL.

The lemma `wp_equiv` captures the characteristic property of `wp`, that is, $(H ==> \text{wp } t Q) \leftrightarrow (\text{triple } t H Q)$.

However, it does not give evidence that there exists a predicate `wp` satisfying this equivalence. We next present one possible definition.

The idea is to define `wp t Q` as the predicate $\exists H, H \setminus^* \text{triple } t H Q$, which, reading literally, is satisfied by “any” heap predicate H which is a valid precondition for a triple for the term t and the postcondition Q .

Definition `wp (t:trm) (Q:val→hprop) : hprop :=`
 $\exists (H:hprop), H \setminus^* \text{triple } t H Q$.

First, let us prove that `wp t Q` is itself a valid precondition, in the sense that `triple t (wp t Q) Q` always holds (as asserted by the lemma `wp_pre`).

To establish this fact, we have to prove: `triple t (exists H, H \setminus^* [triple t H Q]) Q`.

Applying the extraction rule for existentials gives: $\forall H, \text{triple } t (H \setminus^* \text{triple } t H Q) Q$.

Applying the extraction rule for pure facts gives: $\forall H, (\text{triple } t H Q) \rightarrow (\text{triple } t H Q)$, which is true.

Second, let us demonstrate that the heap predicate `wp t Q` is entailed by any precondition H that satisfies `triple t H Q`, as asserted by the lemma `wp_weakest`.

Assume `triple t H Q`. Let us prove $H ==> \text{wp } t Q$, that is $H ==> \exists H, H \setminus^* \text{triple } t H Q$. Instantiating the H on the right-hand side as the H from the left-hand side suffices to satisfy the entailment.

Recall that the properties `wp_pre` and `wp_weakest` were derivable from the characteristic equivalence $\text{triple } t H Q \leftrightarrow H ==> \text{wp } Q$. Thus, to formalize the proofs of `wp_pre` and `wp_weakest`, all we have to do is to establish that equivalence.

Exercise: 2 stars, standard, especially useful (`wp_equiv`) Prove that the definition `wp_high` satisfies the characteristic equivalence for weakest preconditions.

Lemma `wp_equiv : ∀ t H Q,`

$$(H \Rightarrow wp t Q) \leftrightarrow (\text{triple } t H Q).$$

Proof using. Admitted.

□

End WPHIGHLEVEL.

42.4.2 Equivalence Between all Definitions Of `wp`

We next prove that the equivalence $(\text{triple } t H Q) \leftrightarrow (H \Rightarrow wp t Q)$ defines a unique predicate `wp`. In other words, all possible definitions of `wp` are equivalent to each another. Thus, it really does not matter which concrete definition of `wp` we consider: they are all equivalent.

Concretely, assume two predicates `wp1` and `wp2` to both satisfy the characteristic equivalence. We prove that they are equal.

```
Lemma wp_unique : ∀ wp1 wp2,
  (∀ t H Q, (triple t H Q) ↔ (H ==> wp1 t Q)) →
  (∀ t H Q, (triple t H Q) ↔ (H ==> wp2 t Q)) →
  wp1 = wp2.
```

Proof using.

```
intros M1 M2. applys fun_ext_2. intros t Q. applys himpl_antisym.
{ rewrite ← M2. rewrite M1. auto. }
{ rewrite ← M1. rewrite M2. auto. }
```

Qed.

Recall that both `wp_pre` and `wp_weakest` are derivable from `wp_equiv`. Let us also that, reciprocally, `wp_equiv` is derivable from the conjunction of `wp_pre` and `wp_weakest`.

In other words, the property of “being the weakest precondition” also uniquely characterizes the definition of `wp`.

```
Lemma wp_from_weakest_pre : ∀ wp',
  (∀ t Q, triple t (wp' t Q) Q) →
  (∀ t H Q, triple t H Q → H ==> wp' t Q) →
  (∀ t H Q, H ==> wp' t Q ↔ triple t H Q). Proof using.
intros M1 M2. iff M.
{ applys triple_conseq M1 M. auto. }
{ applys M2. auto. }
```

Qed.

42.4.3 An Alternative Definition for Weakest Precondition

Module WPLOWLEVEL.

The concrete definition for `wp` given above is expressed in terms of Separation Logic combinators. In contrast to this “high level” definition, there exists a more “low level” definition, expressed directly as a function over heaps.

In that alternative definition, the heap predicate $\text{wp } t \ Q$ is defined as a predicate that holds of a heap h if and only if the execution of t starting in exactly the heap h produces the post-condition Q .

Technically, $\text{wp } t \ Q$ can be defined as: $\text{fun } (h:\text{heap}) \Rightarrow \text{triple } t (\text{fun } h' \Rightarrow h' = h) \ Q$. In other words, the precondition requires the input heap to be exactly h .

Definition $\text{wp } (t:\text{trm}) (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} :=$
 $\text{fun } (h:\text{heap}) \Rightarrow \text{triple } t (\text{fun } h' \Rightarrow (h' = h)) \ Q.$

Exercise: 4 stars, standard, optional (wp_equiv_wp_low) Prove this alternative definition of wp also satisfies the characteristic equivalence $H ==> \text{wp } Q \leftrightarrow \text{triple } t H Q$. Hint: exploit the lemma `triple_named_heap` which was established as an exercise in the appendix of the chapter `Himpl.`)

Lemma $\text{wp_equiv_wp_low} : \forall t H Q,$
 $(H ==> \text{wp } t Q) \leftrightarrow (\text{triple } t H Q).$

Proof using. *Admitted.*

□

End WPLOWLEVEL.

42.4.4 Extraction Rule for Existentials

Recall the extraction rule for existentials.

Parameter $\text{triple_hexists} : \forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\forall x, \text{triple } t (J x) Q) \rightarrow$
 $\text{triple } t (\exists x, J x) Q.$

Replacing $\text{triple } t H Q$ with $H ==> \text{wp } t Q$ yields the lemma stated below.

Exercise: 1 star, standard, optional (triple_hexists_in_wp) Prove the extraction rule for existentials in wp style.

Lemma $\text{triple_hexists_in_wp} : \forall t Q A (J:A \rightarrow \text{hprop}),$
 $(\forall x, (J x ==> \text{wp } t Q)) \rightarrow$
 $(\exists x, J x) ==> \text{wp } t Q.$

Proof using. *Admitted.*

□

In other words, in the wp presentation, we do not need a specific extraction rule for existentials, because the extraction rule for entailment already does the job.

42.4.5 Combined Structural Rule

Recall the combined consequence-frame rule for `triple`.

```

Parameter triple_conseq_frame : ∀ H2 H1 Q1 t H Q,
  triple t H1 Q1 →
  H ==> H1 \* H2 →
  Q1 \*+ H2 ==> Q →
  triple t H Q.

```

Let us reformulate this rule using `wp`, replacing the form `triple t H Q` with the form `H ==> wp t Q`.

Exercise: 2 stars, standard, especially useful (`wp_conseq_frame_trans`) Prove the combined structural rule in `wp` style. Hint: exploit `wp_conseq_trans` and `wp_frame`.

```

Lemma wp_conseq_frame_trans : ∀ t H H1 H2 Q1 Q,
  H1 ==> wp t Q1 →
  H ==> H1 \* H2 →
  Q1 \*+ H2 ==> Q →
  H ==> wp t Q.

```

Proof using. *Admitted.*

□

The combined structural rule for `wp` can actually be stated in a more concise way, as follows. The rule reads as follows: if you own a state from which the execution of `t` produces (a result and a state satisfying) `Q1` and you own `H`, and if you can trade the combination of `Q1` and `H` against `Q2`, then you own a piece of state from which the execution of `t` produces `Q2`.

Exercise: 2 stars, standard, especially useful (`wp_conseq_frame`) Prove the concise version of the combined structural rule in `wp` style. Many proofs are possible.

```

Lemma wp_conseq_frame : ∀ t H Q1 Q2,
  Q1 \*+ H ==> Q2 →
  (wp t Q1) \* H ==> (wp t Q2).

```

Proof using. *Admitted.*

□

42.4.6 Alternative Statement of the Rule for Conditionals

Module WPIFALT.

We have established the following rule for reasoning about conditionals using `wp`.

```

Parameter wp_if : ∀ b t1 t2 Q,
  wp (if b then t1 else t2) Q ==> wp (trm_if b t1 t2) Q.

```

Equivalently, the rule may be stated with the conditional around the calls to `wp t1 Q` and `wp t2 Q`.

Exercise: 1 star, standard, optional (`wp_if'`) Prove the alternative statement of rule `wp_if`, either from `wp_if` or directly from `triple_if`.

Lemma `wp_if'` : $\forall b t1 t2 Q, (\text{if } b \text{ then } (\text{wp } t1 Q) \text{ else } (\text{wp } t2 Q)) ==> \text{wp } (\text{trm_if } b t1 t2) Q.$

Proof using. *Admitted.*

□

End WPIFALT.

42.4.7 Definition of `wp` Directly from `hoare`

Let's take a step back and look at our construction of Separation Logic so far.

1. We defined Hoare triples (`hoare`) with respect to the big-step judgment (`eval`).
2. We defined Separation Logic triples (`triple`) in terms of Hoare triples (`hoare`), through the definition: $\forall H', \text{hoare } t (H \setminus^* H') (Q \setminus^+ H')$.
3. We then defined Separation Logic weakest-preconditions (`wp`) in terms of Separation Logic triples (`triple`).

Through the construction, we established reasoning rules, first for Hoare triples (`hoare`), then for Separation Logic triples (`triple`), and finally for weakest-preconditions (`wp`).

One question that naturally arises is whether there is a more direct route to deriving reasoning rules for weakest preconditions. In other words, can we obtain the same end result through simpler proofs?

The notion of Hoare triple is a key abstraction that enables conduction further proofs without manipulating heaps (of type `heap`) explicitly. Experiments suggest that it is beneficial to introduce the Hoare logic layer. In other words, it is counterproductive to try an prove Separation Logic reasoning rules, whether for `triple` or for `wp`, directly with respect to the evaluation judgment `eval`.

Thus, the only question that remains is whether it would have some interest to derive the reasoning rules for weakest preconditions (`wp`) directly from the the reasoning rules for Hoare triples (`hoare`), that is, by bypassing the statement and proofs for the reasoning rules for Separation Logic triples (`triple`).

In what follows, we show that if one cares only for `wp`-style rules, then the route to deriving them straight from `hoare`-style rules may indeed be somewhat shorter.

Module `WPFROMHOARE`.

Recall the definition of `triple` in terms of `hoare`.

Definition `triple` ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : Prop := forall ($H':\text{hprop}$), `hoare` $t (H \setminus^* H') (Q \setminus^+ H')$.

In what follows, we conduct the proofs by assuming a concrete definition for `wp`, namely `wp_high`, which lends itself better to automated proofs.

Definition `wp` ($t:\text{trm}$) := fun ($Q:\text{val} \rightarrow \text{hprop}$) \Rightarrow
 $\exists H, H \setminus^* \setminus^* [\text{triple } t H Q].$

First, we check the equivalence between triple $t \ H \ Q$ and $H ==> \text{wp } t \ Q$. This proof is the same as `wp_equiv` from the module `WPHIGHLEVEL` given earlier in this chapter.

Lemma `wp_equiv` : $\forall t \ H \ Q,$
 $(H ==> \text{wp } t \ Q) \leftrightarrow (\text{triple } t \ H \ Q)$.

Proof using.

```
unfold wp. iff M.
{ applys× triple_conseq Q M.
  applys triple_hexists. intros H'.
  rewrite hstar_comm. applys× triple_hpure. }
{ xsimpl× H. }
```

Qed.

Second, we prove the consequence-frame rule associated with `wp`. It is the only structural rule that is needed for working with weakest preconditions.

Lemma `wp_conseq_frame` : $\forall t \ H \ Q1 \ Q2,$
 $Q1 \ \backslash*+ H ==> Q2 \rightarrow$
 $(\text{wp } t \ Q1) \ \backslash* H ==> (\text{wp } t \ Q2)$.

The proof leverages the consequence rule for `hoare` triples, and the frame property comes from the $\forall H'$ quantification baked in the definition of `triple`.

Proof using.

```
intros M. unfold wp. xpull. intros H' N. xsimpl (H' \* H).
unfolds triple. intros H''. specializes N (H \* H'').
applys hoare_conseq N. { xsimpl. } { xchange M. }
```

Qed.

Third and last, we establish reasoning rules for terms in `wp`-style directly from the corresponding rules for `hoare` triples.

The proof details are beyond the scope of this course. The point here is to show that the proofs are fairly concise.

Lemma `wp_val` : $\forall v \ Q,$
 $Q \ v ==> \text{wp } (\text{trm_val } v) \ Q$.

Proof using.

```
intros. unfold wp. xsimpl. intros H'. applys hoare_val. xsimpl.
```

Qed.

Lemma `wp_fun` : $\forall x \ t \ Q,$
 $Q \ (\text{val_fun } x \ t) ==> \text{wp } (\text{trm_fun } x \ t) \ Q$.

Proof using.

```
intros. unfold wp. xsimpl. intros H'. applys hoare_fun. xsimpl.
```

Qed.

Lemma `wp_fix` : $\forall f \ x \ t \ Q,$
 $Q \ (\text{val_fix } f \ x \ t) ==> \text{wp } (\text{trm_fix } f \ x \ t) \ Q$.

Proof using.

intros. unfold wp. xsimpl. intros H' . applys hoare_fix. xsimpl.
Qed.

Lemma wp_if : $\forall b t1 t2 Q$,
 $\text{wp}(\text{if } b \text{ then } t1 \text{ else } t2) Q \Rightarrow \text{wp}(\text{trm_if } b t1 t2) Q$.

Proof using.

intros. unfold wp. xsimpl. intros $H M H'$.
applys hoare_if. applys M .

Qed.

Lemma wp_app_fun : $\forall x v1 v2 t1 Q$,
 $v1 = \text{val_fun } x t1 \rightarrow$
 $\text{wp}(\text{subst } x v2 t1) Q \Rightarrow \text{wp}(\text{trm_app } v1 v2) Q$.

Proof using.

introv $EQ1$. unfold wp. xsimpl. intros $H' M$. intros H'' . applys hoare_app_fun.

Qed.

Lemma wp_app_fix : $\forall f x v1 v2 t1 Q$,
 $v1 = \text{val_fix } f x t1 \rightarrow$
 $\text{wp}(\text{subst } x v2 (\text{subst } f v1 t1)) Q \Rightarrow \text{wp}(\text{trm_app } v1 v2) Q$.

Proof using.

introv $EQ1$. unfold wp. xsimpl. intros $H' M$. intros H'' . applys hoare_app_fix.

Qed.

Exercise: 4 stars, standard, especially useful (wp_let) Prove wp-style rule for let bindings.

Lemma wp_let : $\forall x t1 t2 Q$,
 $\text{wp} t1 (\text{fun } v \Rightarrow \text{wp}(\text{subst } x v t2) Q) \Rightarrow \text{wp}(\text{trm_let } x t1 t2) Q$.

Proof using. Admitted.

□

Note: wp_seq admits essentially the same proof as wp_let , simply replacing `hoare_let` with `hoare_seq`, and removing the tactic `intros v`.

It is technically possible to bypass even the definition of `triple` and specify all functions directly using the predicate `wp`. However, using `triple` leads to better readability of specifications, thus it seems preferable to continue using that style for specifying functions. (See discussion in chapter `Wand`, appendix on “Texan triples”.)

End WPFROMHOARE.

42.4.8 Historical Notes

The idea of weakest precondition was introduced by Dijkstra 1975 (in Bib.v) in his seminal paper “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”.

Weakest preconditions provide a reformulation of Floyd-Hoare logic. Numerous practical verification tools leverage weakest preconditions, e.g. ESC/Java, Why3, Boogie, Spec

Chapter 43

Library `SLF.WPgen`

43.1 WPgen: Weakest Precondition Generator

```
Set Implicit Arguments.  
From SLF Require Import WPsem.  
  
Implicit Types f : var.  
Implicit Types b : bool.  
Implicit Types v : val.  
Implicit Types h : heap.  
Implicit Types P : Prop.  
Implicit Types H : hprop.  
Implicit Types Q : val → hprop.
```

43.2 First Pass

In the previous chapter, we have introduced a predicate called `wp` to describe the weakest precondition of a term *t* with respect to a given postcondition *Q*. The weakest precondition `wp` is defined by the equivalence: $H \implies wp\ t\ Q$ if and only if `triple t H Q`.

With respect to this “characterization of the semantics of `wp`”, we could prove “`wp`-style” reasoning rules. For example, the lemma `wp_seq` asserts: $wp\ t1\ (fun\ r\ \Rightarrow\ wp\ t2\ Q) \implies wp\ (trm_seq\ t1\ t2)\ Q$.

In this chapter, we introduce a function, called `wpgen`, to “effectively compute” the weakest precondition of a term. The value of `wpgen t` is defined recursively over the structure of the term *t*, and ultimately produces a formula that is logically equivalent to `wp t`.

The major difference between `wp` and `wpgen` is that, whereas `wp t` is a predicate that we can reason about by “applying” reasoning rules, `wpgen t` is a predicate that we can reason about simply by “unfolding” its definition. Moreover, `wp` and `wpgen` differs on the way they handle variable substitution. Consider, e.g., a let-binding. The `wp`-style reasoning rule for a let-binding introduces a substitution of the form `wp (subst x v t2)`, which the user must

simplify explicitly. On the contrary, when working with `wpgen`, all the substitutions get automatically simplified during the initial evaluation of `wpgen` on the source program; the end-user sees none of these substitutions.

At a high level, the introduction of `wpgen` is a key ingredient to smoothening the user-experience of conducting interactive proofs in Separation Logic. The matter of the present chapter is to show:

- how to define `wpgen t` as a recursive function that computes in Coq,
- how to integrate support for the frame rule in this recursive definition,
- how to carry out practical proofs using `wpgen`.

A bonus section explains how to establish the soundness theorem for `wpgen`.

The “first pass” section that comes next is fairly short. It only gives a bird-eye tour of the steps of the construction. Detailed explanations are provided in the main body of the chapter.

As first approximation, `wpgen t Q` is defined as a recursive function that pattern matches on its argument `t`, and produces the appropriate heap predicate in each case. The definitions somewhat mimic those of `wp`. For example, where the rule `wp_let` asserts the entailment, $\text{wp } t1 (\text{fun } v \Rightarrow \text{wp} (\text{subst } x v t2) Q) ==> \text{wp} (\text{trm_let } x t1 t2) Q$, the definition of `wpgen` is such that `wpgen (trm_let x t1 t2) Q` is, by definition, equal to `wpgen t1 (\text{fun } v \Rightarrow \text{wpgen} (\text{subst } x v t2) Q)`. One special case is that of applications. We define `wpgen (trm_app v1 v2)` as `wp (trm_app v1 v2)`, that is, we fall back onto the semantical definition of weakest precondition.

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v | trm_var x => \False | trm_app v1 v2 => wp t Q | trm_let x t1 t2 => wpgen t1 (\text{fun } v => wpgen (\text{subst } x v t2) Q) ... end.
```

From there, to obtain the actual definition of `wpgen`, we need to refine the above definition in four steps.

In a first step, we modify the definition in order to make it structurally recursive. Indeed, in the above the recursive call `wpgen (subst x v t2)` is not made to a strict subterm of `trm_let x t1 t2`, thus Coq refuse this definition as it stands.

To fix the issue, we change the definition to the form `wpgen E t Q`, where `E` denotes an association list bindings values to variables. The intention is that `wpgen E t Q` computes the weakest precondition for the term obtained by substituting all the bindings from `E` in `t`. (This term is described by the operation `isubst E t` in the chapter.)

The updated definition looks as follows. Observe how, when traversing `trm_let x t1 t2`, the context `E` gets extended as `(x,v)::E`. Observe also how, when reaching a variable `x`, a lookup for `x` into the context `E` is performed for recovering the value that, morally, should have been substituted for `x`.

```
Fixpoint wpgen (E:ctx) (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v | trm_var x => match lookup x E with | Some v => Q v | None => \False end |
```

`trm_app v1 v2 => wp (isubst E t) Q | trm_let x t1 t2 => wpgen E t1 (fun v => wpgen ((x,v)::E) t2 Q) ... end.`

In a second step, we slightly tweak the definition so as to swap the place where Q is taken as argument with the place where the pattern matching on t occurs. The idea is to make it obvious that `wpgen E t` can be computed without any knowledge of Q .

The type of `wpgen E t` is $(\text{val} \rightarrow \text{hprop}) \rightarrow \text{hprop}$, a type which we thereafter call `formula`.

Fixpoint `wpgen (E:ctx) (t:trm) : formula :=` match t with | `trm_val v => fun (Q:val->hprop) => Q v | trm_var x => fun (Q:val->hprop) => \text{match lookup } x \text{ in } E \text{ with } | \text{Some } v => Q v | \text{None } => \text{\color{red}\textbf{False}} \text{ end} | \text{trm_app } v1 v2 => \text{fun (Q:val->hprop) => wp (isubst } E t) Q | \text{trm_let } x t1 t2 => \text{fun (Q:val->hprop) => wpgen } E t1 (\text{fun } v => \text{wpgen } ((x,v)::E) t2 Q) \dots \text{end.}`

In a third step, we introduce auxiliary definitions to improve the readability of the output of calls to `wpgen`. For example, we let `wpgen_val v` be a shorthand for `fun (Q:val->hprop) => Q v`. Likewise, we let `wpgen_let F1 F2` be a shorthand for `fun (Q:val->hprop) => F1 (fun v => F2 Q)`. Using these auxiliary definitions, the definition of `wpgen` rewrites as follows.

Fixpoint `wpgen (E:ctx) (t:trm) : formula :=` match t with | `trm_val v => wpgen_val v | trm_var x => wpgen_var E x | trm_app t1 t2 => wp (isubst E t) | trm_let x t1 t2 => wpgen_let (wpgen E t1) (\text{fun } v => \text{wpgen } ((x,v)::E) t2) \dots \text{end.}`

Each of the auxiliary definitions introduced comes with a custom notation that enables a nice display of the output of `wpgen`. For example, we set up the notation `Let' v := F1 in F2` to stand for `wpgen_let F1 (fun v => F2)`. Thanks to this notation, the result of computing `wpgen` on a source term `Let x := t1 in t2` (of type `trm`) will be a formula displayed in the form `Let x := F1 in F2` (of type `formula`).

Thanks to these auxiliary definitions and pieces of notation, the formula that `wpgen` produces as output reads pretty much like the source term provided as input.

In a fourth step, we refine the definition of `wpgen` in order to equip it with inherent support for applications of the structural rules of the logic, namely the frame rule and the rule of consequence. To achieve this, we consider a well-crafted predicate called `mkstruct`, and insert it at the head of the output of every call to `wpgen`, including all its recursive calls. The definition of `wpgen` thus now admits the following structure.

Fixpoint `wpgen (E:ctx) (t:trm) : formula :=` `mkstruct (match t with | trm_val v => ... | ... end).`

Without entering the details, the predicate `mkstruct` is a function of type `formula → formula` that captures the essence of the wp-style consequence-frame structural rule. This rule, called `wp_conseq_frame` in the previous chapter, asserts: $Q1 \setminus^* H \implies Q2 \rightarrow (\text{wp } t \text{ } Q1) \setminus^* H \implies (\text{wp } t \text{ } Q2)$.

This concludes our little journey towards the definition of `wpgen`.

For conducting proofs in practice, there remains to state lemmas and define tactics to assist the user in the manipulation of the formula produced by `wpgen`. Ultimately, the end-user only manipulates CFML’s “x-tactics” (recall the first two chapters), without ever being required to understand how `wpgen` is defined.

In other words, the contents of this chapter reveals the details that we work very hard

to make completely invisible to the end user.

43.3 More Details

43.3.1 Definition of `wpgen` for Term Rules

`wpgen` computes a heap predicate that has the same meaning as `wp`. In essence, `wpgen` takes the form of a recursive function that, like `wp`, expects a term t and a postcondition Q , and produces a heap predicate. The function is recursively defined and its result is guided by the structure of the term t .

In essence, the definition of `wpgen` admits the following shape:

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => .. | trm_seq t1 t2 => .. | trm_let x t1 t2 => .. | trm_var x => .. | trm_app t1 t2 => .. | trm_fun x t1 => .. | trm_fix f x t1 => .. | trm_if v0 t1 t2 => .. end).
```

Our first goal is to figure out how to fill in the dots for each of the term constructors.

The intention that guides us for filling the dot is the soundness theorem for `wpgen`, which takes the following form:

$$\text{wpgen } t \ Q ==> \text{wp } t \ Q$$

This entailment asserts in particular that, if we are able to establish a statement of the form $H ==> \text{wpgen } t \ Q$, then we can derive from it $H ==> \text{wp } t \ Q$. The latter is also equivalent to `triple t H Q`. Thus, `wpgen` can be viewed as a practical tool to establish triples.

Remark: the main difference between `wpgen` and a “traditional” weakest precondition generator (as the reader might have seen for Hoare logic) is that here we compute the weakest precondition of a raw term, that is, a term not annotated with any invariant or specification.

Definition of `wpgen` for Values

Consider first the case of a value v . Recall the reasoning rule for values in weakest-precondition style.

```
Parameter wp_val : ∀ v Q,
  Q v ==> wp (trm_val v) Q.
```

The soundness theorem for `wpgen` requires the entailment `wpgen (trm_val v) Q ==> wp (trm_val v) Q` to hold.

To satisfy this entailment, according to the rule `wp_val`, it suffices to define `wpgen (trm_val v) Q` as $Q v$.

Concretely, we fill in the first dots as follows:

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v ...
```

Definition of `wpgen` for Functions

Consider the case of a function definition `trm_fun x t`. Recall that the `wp` reasoning rule for functions is very similar to that for values.

`Parameter wp_fun : ∀ x t1 Q,`

$$Q (\text{val_fun } x \ t1) \implies \text{wp} (\text{trm_fun } x \ t1) \ Q.$$

So, likewise, we can define `wpgen` for functions and for recursive functions as follows:

`Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_fun x t1 =>`

$$Q (\text{val_fun } x \ t1) \mid \text{trm_fix } f \ x \ t1 \implies Q (\text{val_fix } f \ x \ t1) \dots$$

An important observation is that we here do not attempt to recursively compute `wpgen` over the body of the function. This is something that we could do, and that we will see how to achieve further on, yet we postpone it for now because it is relatively technical. In practice, if the program features a local function definition, the user may explicitly request the computation of `wpgen` over the body of that function. Thus, the fact that we do not recursively traverse functions bodies does not harm expressiveness.

Definition of `wpgen` for Sequence

Recall the `wp` reasoning rule for a sequence `trm_seq t1 t2`.

`Parameter wp_seq : ∀ t1 t2 Q,`

$$\text{wp} \ t1 \ (\text{fun } v \Rightarrow \text{wp} \ t2 \ Q) \implies \text{wp} (\text{trm_seq } t1 \ t2) \ Q.$$

The intention is for `wpgen` to admit the same semantics as `wp`. We thus expect the definition of `wpgen (trm_seq t1 t2) Q` to have a similar shape as `wp t1 (fun v => wp t2 Q)`.

We therefore define `wpgen (trm_seq t1 t2) Q` as `wpgen t1 (fun v => wpgen t2 Q)`. The definition of `wpgen` thus gets refined as follows:

`Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_seq t1 t2 =>`

$$\text{wpgen} \ t1 \ (\text{fun } v \Rightarrow \text{wpgen} \ t2 \ Q) \dots$$

Definition of `wpgen` for Let-Bindings

The case of let bindings is similar to that of sequences, except that it involves a substitution. Recall the `wp` rule:

`Parameter wp_let : ∀ x t1 t2 Q,`

$$\text{wp} \ t1 \ (\text{fun } v \Rightarrow \text{wp} (\text{subst } x \ v \ t2) \ Q) \implies \text{wp} (\text{trm_let } x \ t1 \ t2) \ Q.$$

We fill in the dots as follows:

`Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_let x t1 t2 =>`

$$\text{wpgen} \ t1 \ (\text{fun } v \Rightarrow \text{wpgen} (\text{subst } x \ v \ t2) \ Q) \dots$$

One important observation to make at this point is that the function `wpgen` is no longer structurally recursive. Indeed, while the first recursive call to `wpgen` is invoked on `t1`, which is a strict subterm of `t`, the second call is invoked on `subst x v t2`, which is not a strict subterm of `t`.

It is technically possible to convince Coq that the function `wpgen` terminates, yet with great effort. Alternatively, we can circumvent the problem altogether by casting the function in a form that makes it structurally recursive. Concretely, we will see further on how to add as argument to `wpgen` a substitution context (written E) to delay the computation of substitutions until the leaves of the recursion.

Definition of `wpgen` for Variables

We have seen no reasoning rules for establishing a triple for a program variable, that is, to prove `triple (trm_var x) H Q`. Indeed, `trm_var x` is a stuck term: its execution does not produce an output.

While a source term may contain program variables, all these variables should get substituted away before the execution reaches them.

In the case of the function `wpgen`, a variable bound by let-binding get substituted while traversing that let-binding construct. Thus, if a free variable is reached by `wpgen`, it means that this variable was originally a dangling free variable, and therefore that the initial source term was invalid.

Although we have presented no reasoning rules for `triple (trm_var x) H Q` nor for $H \Rightarrow wp (trm_var x) Q$, we nevertheless have to provide some meaningful definition for `wpgen (trm_var x) Q`. This definition should capture the fact that this case must not happen. The heap predicate `\[False]` captures this intention perfectly well.

Fixpoint `wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_var x => \[False] ...`

Remark: the definition of `\[False]` translates the fact that, technically, we could have stated a Separation Logic rule for free variables, using `False` as a premise `\[False]` as precondition. There are three canonical ways of presenting this rule, they are shown next.

`Lemma wp_var : \forall x Q,`

`\[False] ==> wp (trm_var x) Q.`

`Proof using. intros. intros h Hh. destruct (hpure_inv Hh). false. Qed.`

`Lemma triple_var : \forall x H Q,`

`False ->`

`triple (trm_var x) H Q.`

`Proof using. intros. false. Qed.`

`Lemma triple_var' : \forall x Q,`

`triple (trm_var x) \[False] Q.`

`Proof using. intros. rewrite ← wp_equiv. applys wp_var. Qed.`

All these rules are correct, albeit totally useless.

Definition of `wpgen` for Function Applications

Consider an application in A-normal form, that is, an application of the form `trm_app v1 v2`.

We have seen `wp`-style rules to reason about the application of a known function, e.g. `trm_app (val_fun x t1) v2`. However, if `v1` is an abstract value (e.g., a Coq variable of type `val`), we have no reasoning rule at hand that applies.

Thus, we will simply define `wpgen (trm_app v1 v2) Q` as `wp (trm_app v1 v2) Q`. In other words, to define `wpgen` for a function application, we fall back to the semantic definition of `wp`. We thus extend the definition of `wpgen` as follows.

Fixpoint `wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_app v1 v2 => wp (trm_app v1 v2) Q ...`

As we carry out verification proofs in practice, when reaching an application we will face a goal of the form:

$H ==> wpgen (\text{trm_app } v1 v2) Q$

By revealing the definition of `wpgen` on applications, we get:

$H ==> wp (\text{trm_app } v1 v2) Q$

Then, by exploiting the equivalence with triples, we obtain:

$\text{triple} (\text{trm_app } v1 v2) H Q$

Such a proof obligation can be discharged by invoking a specification triple for the function `v1`.

In other words, by falling back to the semantics definition when reaching a function application, we allow the user to choose which specification lemma to exploit for reasoning about this particular function application.

Remark: we assume throughout the course that terms are written in A-normal form. Nevertheless, we need to define `wpgen` even on terms that are not in A-normal form. One possibility is to map all these terms to `\[False]`. In the specific case of an application of the form `trm_app t1 t2` where `t1` and `t2` are not both values, it is still correct to define `wpgen (trm_app t1 t2)` as `wp (trm_app t1 t2)`. So, we need not bother checking in the definition of `wpgen` that the arguments of `trm_app` are actually values.

Thus, the most concise definition for `wpgen` on applications is:

Fixpoint `wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_app t1 t2 => wp t Q ...`

Definition of `wpgen` for Conditionals

The last remaining case is that for conditionals. Recall the `wp`-style reasoning rule stated using a Coq conditional.

Parameter `wp_if : \forall (b:bool) t1 t2 Q,`

`(if b then (wp t1 Q) else (wp t2 Q)) ==> wp (\text{trm_if } (\text{val_bool } b) t1 t2) Q.`

We need to define `wpgen` for all conditionals in A-normal form, i.e., all terms of the form `trm_if (trm_val v0) t1 t2`, where `v0` could be a value of unknown shape. Typically, a program may feature a conditional `trm_if (trm_var x) t1 t2` that, after substitution for `x`, becomes `trm_if (trm_val v) t1 t2`, for some abstract `v` of type `val` that we might not yet know to be a boolean value.

Yet, the rule `wp_if` only applies when the first argument of `trm_if` is syntactically a boolean value `b`. To handle this mismatch, we set up `wpgen` to pattern-match a conditional as `trm_if t0 t1 t2`, and then express using a Separation Logic existential quantifier that there should exist a boolean `b` such that $t0 = \text{trm_val}(\text{val_bool } b)$.

This way, we delay the moment at which the argument of the conditional needs to be shown to be a boolean value. The formal definition is:

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_if t0 t1 t2 =>
\exists (b:bool), \t0 = \text{trm\_val}(\text{val\_bool } b) \^* (if b then (wpgen t1) Q else (wpgen t2) Q) ...
```

Summary of the Definition of `wpgen` for Term Rules

In summary, we have defined:

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v | trm_fun x t1 => Q (val_fun x t) | trm_fix f x t1 => Q (val_fix f x t) | trm_seq t1 t2 => wpgen t1 (fun v => wpgen t2 Q) | trm_let x t1 t2 => wpgen t1 (fun v => wpgen (subst x v t2) Q) | trm_var x => \False | trm_app v1 v2 => wp t Q | trm_if t0 t1 t2 => \exists (b:bool), \t0 = \text{trm\_val}(\text{val\_bool } b) \^* (if b then (wpgen t1) Q else (wpgen t2) Q) end.
```

As pointed out earlier, this definition is not structurally recursive and thus not accepted by Coq, due to the recursive call `wpgen (subst x v t2) Q`. Our next step is to fix this issue.

43.3.2 Computing with `wpgen`

`wpgen` is not structurally recursive because of the substitutions that takes places in-between recursive calls. To fix this, we are going to delay the substitutions until the leaves of the recursion.

To that end, we introduce a substitution context, written E , to record the substitutions that should have been performed.

Concretely, we modify the function to take the form `wpgen E t`, where E denotes a set of bindings from variables to values. The intention is that `wpgen E t` computes the weakest precondition for the term `isubst E t`, which denotes the result of substituting all bindings from E inside the term t .

Definition of Contexts and Operations on Them

The simplest way to define a context E is as an association list relating variables to values.

Definition `ctx` : Type := `list (var × val)`.

Before we explain how to revisit the definition of `wpgen` using contexts, we need to define the “iterated substitution” operation. This operation, written `isubst E t`, describes the substitution of all the bindings from E inside a term t .

The definition of iterated substitution is relatively standard: we traverse the term recursively and, when reaching a variable, we perform a lookup in the context E . We need to take care to respect variable shadowing. To that end, when traversing a binder that binds a variable x , we remove all occurrences of x that might exist in E .

The formal definition of `isubst` involves two auxiliary functions: `lookup` and `rem` on association lists.

The definition of the operation `lookup x E` on association lists is standard. It returns an option on a value.

```
Fixpoint lookup (x:var) (E:ctx) : option val :=
  match E with
  | nil => None
  | (y,v)::E1 => if var_eq x y
    then Some v
    else lookup x E1
  end.
```

The definition of the removal operation, written `rem x E`, is also standard. It returns a filtered context.

```
Fixpoint rem (x:var) (E:ctx) : ctx :=
  match E with
  | nil => nil
  | (y,v)::E1 =>
    let E1' := rem x E1 in
    if var_eq x y then E1' else (y,v)::E1'
  end.
```

The definition of the operation `isubst E t` can then be expressed as a recursive function over the term `t`. It invokes `lookup x E` when reaching a variable `x`. It invokes `rem x E` when traversing a binding on the name `x`.

```
Fixpoint isubst (E:ctx) (t:trm) : trm :=
  match t with
  | trm_val v =>
    v
  | trm_var x =>
    match lookup x E with
    | None => t
    | Some v => v
    end
  | trm_fun x t1 =>
    trm_fun x (isubst (rem x E) t1)
  | trm_fix f x t1 =>
    trm_fix f x (isubst (rem x (rem f E)) t1)
  | trm_app t1 t2 =>
    trm_app (isubst E t1) (isubst E t2)
  | trm_seq t1 t2 =>
    trm_seq (isubst E t1) (isubst E t2)
  | trm_let x t1 t2 =>
```

```

trm_let x (isubst E t1) (isubst (rem x E) t2)
| trm_if t0 t1 t2 =>
  trm_if (isubst E t0) (isubst E t1) (isubst E t2)
end.

```

Remark: it is also possible to define the substitution by iterating the unary substitution `subst` over the list of bindings from E . However, this alternative approach yields a less efficient function and leads to more complicated proofs.

In what follows, we present the definition of `wpgen E t` case by case. Throughout these definitions, recall that `wpgen E t` is interpreted as the weakest precondition of `isubst E t`.

wpgen: the Let-Binding Case

When `wpgen` traverses a let-binding, rather than eagerly performing a substitution, it simply extends the current context.

Concretely, a call to `wpgen E (trm_let x t1 t2)` triggers a recursive call to `wpgen ((x,v)::E) t2`. The corresponding definition is:

```

Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct (match t with ... | trm_let x t1 t2 => fun Q => (wpgen E t1) (fun v => wpgen ((x,v)::E) t2) ... ) end.

```

wpgen: the Variable Case

When `wpgen` reaches a variable, it lookups for a binding on the variable x inside the context E . Concretely, the evaluation of `wpgen E (trm_var x)` triggers a call to `lookup x E`.

If the context E binds the variable x to some value v , then the operation `lookup x E` returns `Some v`. In that case, `wpgen` returns the weakest precondition for that value v , that is, $Q v$.

Otherwise, if E does not bind x , the lookup operation returns `None`. In that case, `wpgen` returns $\backslash[\text{False}]$, which we have explained to be the weakest precondition for a stuck program.

```

Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct (match t with ... | trm_var x => fun Q => match lookup x E with | Some v => Q v | None => \False end ... ) end.

```

wpgen: the Application Case

In the previous definition of `wpgen` (the one without contexts), we argued that, in the case where t denotes an application, the result of `wpgen t Q` should be simply `wp t Q`.

In the definition of `wpgen` with contexts, the interpretation of `wpgen E t` is the weakest precondition of the term `isubst E t` (which denotes the result of substituting variables from E in t).

When t is an application, we thus define `wpgen E t` as the formula `fun Q => wp (isubst E t) Q`, which simplifies to `wp (isubst E t)` after eliminating the eta-expansion.

```

Fixpoint wpgen (t:trm) : formula := mkstruct (match t with ... | trm_app v1 v2 => wp (isubst E t) .. )

```

wpgen: the Function Definition Case

Consider the case where t is a function definition, for example $\text{trm_fun } x \ t1$. Here again, the formula $\text{wpgen } E \ t$ is interpreted as the weakest precondition of $\text{isubst } E \ t$.

By unfolding the definition of isubst in the case where t is $\text{trm_fun } x \ t1$, we obtain $\text{trm_fun } x \ (\text{isubst } (\text{rem } x \ E) \ t1)$.

The corresponding value is $\text{trm_val } x \ (\text{isubst } (\text{rem } x \ E) \ t1)$. The weakest precondition for that value is $\text{fun } Q \Rightarrow Q \ (\text{val_fun } x \ (\text{isubst } (\text{rem } x \ E) \ t1))$.

Thus, $\text{wpgen } E \ t$ handles functions, and recursive functions, as follows:

Fixpoint $\text{wpgen } (E:\text{ctx}) \ (t:\text{trm}) : \text{formula} := \text{mkstruct } (\text{match } t \ \text{with} \dots \mid \text{trm_fun } x \ t1 \Rightarrow \text{fun } Q \Rightarrow Q \ (\text{val_fun } x \ (\text{isubst } (\text{rem } x \ E) \ t1)) \mid \text{trm_fix } f \ x \ t1 \Rightarrow \text{fun } Q \Rightarrow Q \ (\text{val_fix } f \ x \ (\text{isubst } (\text{rem } x \ (\text{rem } f \ E)) \ t1)) \dots) \ \text{end.}$

wpgen: at Last, an Executable Function

Module WPGENEXEC1.

At last, we arrive to a definition of wpgen that type-checks in Coq, and that can be used to effectively compute weakest preconditions in Separation Logic.

```
Fixpoint wpgen (E:ctx) (t:trm) (Q:val→hprop) : hprop :=
  match t with
  | trm_val v ⇒ Q v
  | trm_fun x t1 ⇒ Q (val_fun x (isubst (rem x E) t1))
  | trm_fix f x t1 ⇒ Q (val_fix f x (isubst (rem x (rem f E)) t1))
  | trm_seq t1 t2 ⇒ wpgen E t1 (fun v ⇒ wpgen E t2 Q)
  | trm_let x t1 t2 ⇒ wpgen E t1 (fun v ⇒ wpgen ((x,v)::E) t2 Q)
  | trm_var x ⇒
    match lookup x E with
    | Some v ⇒ Q v
    | None ⇒ \[False]
    end
  | trm_app v1 v2 ⇒ wp (isubst E t) Q
  | trm_if t0 t1 t2 ⇒
    \exists (b:bool), \[t0 = trm_val (val_bool b)]
    \* (if b then (wpgen E t1) Q else (wpgen E t2) Q)
  end.
```

Compared with the presentation using the form $\text{wpgen } t$, the new presentation using the form $\text{wpgen } E \ t$ has the main benefits that it is structurally recursive, thus easy to define in Coq. Moreover, it is algorithmically more efficient in general, because it performs substitutions lazily rather than eagerly.

Let us state the soundness theorem and its corollary for establishing triples for functions.

Parameter $\text{wpgen_sound} : \forall E \ t \ Q,$
 $\text{wpgen } E \ t \ Q ==> \text{wp } (\text{isubst } E \ t) \ Q.$

The entailment above asserts in particular that if we can derive triple $t \ H \ Q$ by proving $H ==> \text{wpgen } t \ Q$.

A useful corollary combines the soundness theorem with the rule *triple_app_fun*, which allows establishing triples for functions. Recall the rule *triple_app_fun* from Rules.

```
Parameter triple_app_fun : ∀ x v1 v2 t1 H Q,
  v1 = val_fun x t1 →
  triple (subst x v2 t1) H Q →
  triple (trm_app v1 v2) H Q.
```

Reformulating the rule above into a rule for **wpgen** takes 3 steps.

First, we rewrite the premise $\text{triple} (\text{subst } x \ v2 \ t1) \ H \ Q$ using **wp**. It becomes $H ==> \text{wp} (\text{subst } x \ v2 \ t1) \ Q$.

Second, we observe that the term $\text{subst } x \ v2 \ t1$ is equal to $\text{isubst} ((x,v2)::\text{nil}) \ t1$. (This equality is captured by the lemma *subst_eq_isubst_one* proved in the bonus section of the chapter.) Thus, the heap predicate $\text{wp} (\text{subst } x \ v2 \ t1) \ Q$ is equivalent to $\text{wp} (\text{isubst} ((x,v2)::\text{nil}) \ t1)$.

Third, according to **wpgen_sound**, the predicate $\text{wp} (\text{isubst} ((x,v2)::\text{nil}) \ t1)$ is entailed by $\text{wpgen} ((x,v2)::\text{nil}) \ t1$. Thus, we can use the latter as premise in place of the former. We thereby obtain the following lemma.

```
Parameter triple_app_fun_from_wpgen : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wpgen ((x,v2)::nil) t1 Q →
  triple (trm_app v1 v2) H Q.
```

Executing **wpgen** on a Concrete Program

Import *ExamplePrograms*.

Let us exploit *triple_app_fun_from_wpgen* to demonstrate the computation of **wpgen** on a practical program.

Recall the function *incr* (defined in Rules), and its specification, whose statement appears below.

```
Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
    (p ~~> n)
    (fun _ ⇒ p ~~> (n+1)).
```

Proof using.

```
intros. applys triple_app_fun_from_wpgen. { reflexivity. }
simpl. Abort.
```

The goal takes the form $H ==> \text{wpgen } body \ Q$, where H denotes the precondition, Q the postcondition, and *body* the body of the function *incr*.

Observe the invocations of **wp** on the application of primitive operations.

Observe that the goal is nevertheless somewhat hard to relate to the original program.

In what follows, we explain how to remedy the situation, and set up `wpgen` is such a way that its output is human-readable, moreover resembles the original source code.

End WPGENEXEC1.

43.3.3 Optimizing the Readability of `wpgen` Output

To improve the readability of the formulae produced by `wpgen`, we take the following 3 steps:

- first, we modify the presentation of `wpgen` so that the `fun (Q:val→hprop) ⇒` appears insides the branches of the `match t with` rather than around it,
- second, we introduce one auxiliary definition for each branch of the `match t with`,
- third, we introduce one piece of notation for each of these auxiliary definitions.

Readability Step 1: Moving the Function below the Branches.

We distribute the `fun Q` into the branches of the `match t`. Concretely, we change from:

Fixpoint `wpgen (E:ctx) (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v | trm_fun x t1 => Q (val_fun x (isubst (rem x E) t1)) ... end.`
to the equivalent form:

Fixpoint `wpgen (E:ctx) (t:trm) : (val->hprop)->hprop := match t with | trm_val v => fun (Q:val->hprop) => Q v | trm_fun x t1 => fun (Q:val->hprop) => Q (val_fun x (isubst (rem x E) t1)) ... end.`

The result type of `wpgen E t` is `(val->hprop)->hprop`. Thereafter, we let `formula` be a shorthand for this type.

Definition `formula : Type := (val->hprop)->hprop.`

Readability Steps 2 and 3, Illustrated on the Case of Sequences

We introduce auxiliary definitions to denote the result of each of the branches of the `match t` construct. Concretely, we change from:

Fixpoint `wpgen (E:ctx) (t:trm) : formula := match t with ... | trm_seq t1 t2 => fun Q => (wpgen E t1) (fun v => wpgen E t2 Q) ... end.`
to the equivalent form:

Fixpoint `wpgen (E:ctx) (t:trm) : formula := match t with ... | trm_seq t1 t2 => wpgen_seq (wpgen E t1) (wpgen E t2) ... end.`
where `wpgen_seq` is defined as shown below.

Definition `wpgen_seq (F1 F2:formula) : formula := fun Q => F1 (fun v => F2 Q).`

Remark: above, `F1` and `F2` denote the results of the recursive calls, `wpgen E t1` and `wpgen E t2`, respectively.

With the above definitions, $\text{wpgen } E$ ($\text{trm_seq } t1 \ t2$) evaluates to wp_seq ($\text{wpgen } E \ t1$) ($\text{wpgen } E \ t2$).

Finally, we introduce a piece of notation for each case. In the case of the sequence, we set up the notation defined next to so that any formula of the form $\text{wpgen_seq } F1 \ F2$ gets displayed as $\text{Seq } F1 ; F2$.

```
Notation "'Seq' F1 ; F2" :=
  ((wpgen_seq F1 F2))
  (at level 68, right associativity,
  format "'[v] Seq '[' F1 ']' ';' '/' '[' F2 ']' ']').
```

Thanks to this notation, the wpgen of a sequence $t1 ; t2$ displays as $\text{Seq } F1 ; F2$ where $F1$ and $F2$ denote the wpgen of $t1$ and $t2$, respectively.

Readability Step 2: Auxiliary Definitions for other Constructs

We generalize the approach illustrated for sequences to every other term construct. The corresponding definitions are stated below. It is not required to understand the details in this subsection.

```
Definition wpgen_val (v:val) : formula := fun Q =>
  Q v.
```

```
Definition wpgen_let (F1:formula) (F2of:val → formula) : formula := fun Q =>
  F1 (fun v => F2of v Q).
```

```
Definition wpgen_if (t:trm) (F1 F2:formula) : formula := fun Q =>
  ∃ (b:bool), ∃ [t = trm_val (val_bool b)] ∗ (if b then F1 Q else F2 Q).
```

```
Definition wpgen_fail : formula := fun Q =>
  \[False].
```

```
Definition wpgen_var (E:ctx) (x:var) : formula :=
  match lookup x E with
  | None => wpgen_fail
  | Some v => wpgen_val v
  end.
```

The new definition of wpgen reads as follows.

Module WPGENEXEC2.

```
Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  match t with
  | trm_val v => wpgen_val v
  | trm_var x => wpgen_var E x
  | trm_fun x t1 => wpgen_val (val_fun x (isubst (rem x E) t1))
  | trm_fix f x t1 => wpgen_val (val_fix f x (isubst (rem x (rem f E)) t1))
  | trm_app t1 t2 => wp (isubst E t)
  | trm_seq t1 t2 => wpgen_seq (wpgen E t1) (wpgen E t2)
```

```

| trm_let x t1 t2 => wpgen_let (wpgen E t1) (fun v => wpgen ((x,v)::E) t2)
| trm_if t0 t1 t2 => wpgen_if (isubst E t0) (wpgen E t1) (wpgen E t2)
end.

```

This definition is, up to unfolding of the new intermediate definitions, totally equivalent to the previous one. We will prove the soundness result and its corollary further on.

```

Parameter wpgen_sound : ∀ E t Q,
  wpgen E t Q ==> wp (isubst E t) Q.

```

```

Parameter triple_app_fun_from_wpgen : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wpgen ((x,v2)::nil) t1 Q →
  triple (trm_app v1 v2) H Q.

```

End WPGENEXEC2.

Readability Step 3: Notation for Auxiliary Definitions

We generalize the notation introduced for sequences to every other term construct. The corresponding notation is defined below. It is not required to understand the details from this subsection.

To avoid conflicts with other existing notation, we write *Let'* and *If'* in place of *Let* and *If*.

Here again, it is not required to understand all the details.

Declare Scope wpgen_scope.

```

Notation "'Val' v" :=
  ((wpgen_val v))
  (at level 69) : wpgen_scope.

```

```

Notation "'Let'" x ':=' F1 'in' F2" :=
  ((wpgen_let F1 (fun x => F2)))
  (at level 69, x ident, right associativity,
  format "[v] ['] 'Let'" x ':=' F1 'in' ['] '/ ['] F2 ['] ']")
  : wpgen_scope.

```

```

Notation "'If'" b 'Then' F1 'Else' F2" :=
  ((wpgen_if b F1 F2))
  (at level 69) : wpgen_scope.

```

```

Notation "'Fail'" :=
  ((wpgen_fail))
  (at level 69) : wpgen_scope.

```

In addition, we introduce handy notation of the result of *wpgen t* where *t* denotes an application.

```

Notation "'App'" f v1 " :="
  ((wp (trm_app f v1)))

```

```
(at level 68, f, v1 at level 0) : wpigen_scope.
```

Test of `wpigen` with Notation.

Consider again the example of `incr`.

```
Module WPGENWITHNOTATION.
Import ExamplePrograms WpigenExec2.
Open Scope wpigen_scope.
```

```
Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
  (p ~~> n)
  (fun _ ⇒ p ~~> (n+1)).
```

Proof using.

```
intros. applys triple_app_fun_from_wpigen. { reflexivity. }
simpl. Abort.
```

Up to proper tabulation, alpha-renaming, and removal of parentheses (and dummy quotes after `Let` and `If`), the formula F reads as:

Let $n := \text{App} \text{ val_get } p$ in Let $m := \text{App} (\text{val_add } n) 1$ in $\text{App} (\text{val_set } p) m$

With the introduction of intermediate definitions for `wpigen` and the introduction of associated notations for each term construct, what we achieved is that the output of `wpigen` is, for any input term t , a human readable formula whose display closely resembles the syntax source code of the term t .

End WPGENWITHNOTATION.

43.3.4 Extension of `wpigen` to Handle Structural Rules

The definition of `wpigen` proposed so far integrates the logic of the reasoning rules for terms, however it lacks support for conveniently exploiting the structural rules of the logic.

We fix this next, by showing how to tweak the definition of `wpigen` in such a way that, by construction, it satisfies both:

- the frame rule, which asserts $(\text{wpigen } t \ Q) \setminus^* H ==> \text{wpigen } t \ (Q \setminus^{*+} H)$,
- and the rule of consequence, which asserts that $Q1 ==> Q2$ implies $\text{wpigen } t \ Q1 ==> \text{wpigen } t \ Q2$.

Introduction of `mkstruct` in the Definition of `wpigen`

Recall from the previous chapter the statement of the frame rule in `wp`-style.

```
Parameter wp_frame : ∀ t H Q,
  (wp t Q) \* H ==> wp t (Q \*+ H).
```

We would like `wpgen` to satisfy the same rule, so that we can exploit the frame rule while reasoning about a program using the heap predicate produced by `wpgen`.

With the definition of `wpgen` set up so far, it is possible to prove, for any concrete term t , that the frame property $(\text{wpgen } t \ Q) \setminus^* H \implies \text{wpgen } t \ (Q \setminus^{*+} H)$ holds. However, establishing this result requires an induction over the entire structure of the term t —a lot of tedious work.

Instead, we are going to tweak the definition of `wpgen` so as to produce, at every step of the recursion, a special token to capture the idea that whatever the details of the output predicate produced by `wpgen`, this predicate does satisfy the frame property.

We achieve this magic by introducing a predicate called `mkstruct`, and inserting it at the head of the output produced by `wpgen` (and all of its recursive invocation) as follows:

Fixpoint `wpgen` ($E:\text{ctx}$) ($t:\text{trm}$) : $(\text{val} \rightarrow \text{hprop}) \rightarrow \text{hprop} := \text{mkstruct} \ (\text{match } t \ \text{with} \ |\ \text{trm_val } v \Rightarrow \text{wpgen_val } v \dots \text{end})$.

The interest of the insertion of `mkstruct` above is that every result of a computation of `wpgen t` on a concrete term t is, by construction, of the form `mkstruct F` for some argument F .

There remains to investigate how `mkstruct` should be defined.

Properties of `mkstruct`

Module `MKSTRUCTPROP`.

Let us state the properties that `mkstruct` should satisfy.

Because `mkstruct` appears between the prototype and the `match` in the body of `wpgen`, the predicate `mkstruct` must have type `formula → formula`.

Parameter `mkstruct` : `formula → formula`.

There remains to find a suitable definition for `mkstruct` that enables the frame property and the consequence property. These properties can be stated by mimicking the rules `wp_frame` and `wp_conseq`.

Parameter `mkstruct_frame` : $\forall (F:\text{formula}) \ H \ Q,$
 $(\text{mkstruct } F \ Q) \setminus^* H \implies \text{mkstruct } F \ (Q \setminus^{*+} H)$.

Parameter `mkstruct_conseq` : $\forall (F:\text{formula}) \ Q1 \ Q2,$
 $Q1 \implies Q2 \rightarrow$
 $\text{mkstruct } F \ Q1 \implies \text{mkstruct } F \ Q2$.

In addition, it should be possible to erase `mkstruct` from the head of the output produced `wpgen t` when we do not need to apply any structural rule. In other words, we need to be able to prove $H \implies \text{mkstruct } F \ Q$ by proving $H \implies F \ Q$, for any H . This erasure property is captured by the following entailment.

Parameter `mkstruct_erase` : $\forall (F:\text{formula}) \ Q,$
 $F \ Q \implies \text{mkstruct } F \ Q$.

Moreover, `mkstruct` should be monotone with respect to the formula: if $F1$ is stronger than $F2$, then `mkstruct F1` should be stronger than `mkstruct F2`.

```
Parameter mkstruct_monotone : ∀ F1 F2 Q,
  (forall Q, F1 Q ==> F2 Q) →
  mkstruct F1 Q ==> mkstruct F2 Q.
```

End MKSTRUCTPROP.

Realization of `mkstruct`

Fortunately, it turns out that there exists a predicate `mkstruct` satisfying the four required properties. To begin with, let us just pull out of our hat a definition of `mkstruct` that works.

```
Definition mkstruct (F:formula) : formula :=
  fun (Q:val → hprop) => ∃ Q1 H, F Q1 ∗ H ∗ [Q1 ∗+ H ==> Q].
```

We postpone to a bonus section the discussion of why it works and of how one could have guessed this definition. Again, it really does not matter the details of the definition of `mkstruct`. All that matters is that `mkstruct` satisfies the three required properties: `mkstruct_frame`, `mkstruct_conseq`, and `mkstruct_erase`. Let us establish these properties for the definition considered. (Proof details can be skipped over.)

```
Lemma mkstruct_frame : ∀ F H Q,
  (mkstruct F Q) ∗ H ==> mkstruct F (Q ∗+ H).
```

Proof using.

```
  intros. unfold mkstruct. xpull; intros Q' H' M. xsimpl. xchange M.
```

Qed.

```
Lemma mkstruct_conseq : ∀ F Q1 Q2,
  Q1 ==> Q2 →
  mkstruct F Q1 ==> mkstruct F Q2.
```

Proof using.

```
  introv WQ. unfold mkstruct. xpull; intros Q' H' M.
  xsimpl. xchange M. xchange WQ.
```

Qed.

```
Lemma mkstruct_erase : ∀ F Q,
  F Q ==> mkstruct F Q.
```

Proof using. intros. unfold mkstruct. xsimpl. xsimpl. Qed.

```
Lemma mkstruct_monotone : ∀ F1 F2 Q,
  (forall Q, F1 Q ==> F2 Q) →
  mkstruct F1 Q ==> mkstruct F2 Q.
```

Proof using.

```
  introv WF. unfolds mkstruct. xpull. intros Q' H M.
  xchange WF. xsimpl Q'. applys M.
```

Qed.

An interesting property of `mkstruct` is it has no effect on a formula of the form `wp t`. Intuitively, such a formula already satisfies all the structural reasoning rules, hence adding `mkstruct` to it does not increase its expressive power.

```
Lemma mkstruct_wp : ∀ t,
  mkstruct (wp t) = (wp t).
```

Proof using.

```
intros. applys fun_ext_1. intros Q. applys himpl_antisym.
{ unfold mkstruct. xpull. intros H Q' M. applys wp_conseq_frame M. }
{ applys mkstruct_erase. }
```

Qed.

Another interesting property of `mkstruct` is its idempotence property, that is, it is such that `mkstruct (mkstruct F) = mkstruct F`.

Idempotence asserts that applying the predicate `mkstruct` more than once does not provide more expressiveness than applying it just once.

Intuitively, idempotence reflects in particular the fact that two nested applications of the rule of consequence can always be combined into a single application of that rule (due to the transitivity of entailment); and that, similarly, two nested applications of the frame rule can always be combined into a single application of that rule (framing on $H1$ then framing on $H2$ is equivalent to framing on $H1 \ast H2$).

Exercise: 3 stars, standard, especially useful (`mkstruct_idempotent`) Complete the proof of the idempotence of `mkstruct`. Hint: leverage `xpull` and `xsimpl`.

```
Lemma mkstruct_idempotent : ∀ F,
  mkstruct (mkstruct F) = mkstruct F.
```

Proof using.

```
intros. apply fun_ext_1. intros Q. applys himpl_antisym.
```

Admitted.

□

Definition of `wpgen` that Includes `mkstruct`

Our final definition of `wpgen` refines the previous one by inserting the `mkstruct` predicate to the front of the `match t with` construct.

```
Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  mkstruct (match t with
  | trm_val v ⇒ wpgen_val v
  | trm_var x ⇒ wpgen_var E x
  | trm_fun x t1 ⇒ wpgen_val (val_fun x (isubst (rem x E) t1))
  | trm_fix f x t1 ⇒ wpgen_val (val_fix f x (isubst (rem x (rem f E)) t1))
  | trm_app t1 t2 ⇒ wp (isubst E t)
  | trm_seq t1 t2 ⇒ wpgen_seq (wpgen E t1) (wpgen E t2))
```

```

| trm_let x t1 t2 => wpgen_let (wpgen E t1) (fun v => wpgen ((x,v)::E) t2)
| trm_if t0 t1 t2 => wpgen_if (isubst E t0) (wpgen E t1) (wpgen E t2)
end).

```

Again, we assert the soundness theorem and its corollary, and we postpone the proof.

```

Parameter wpgen_sound : ∀ E t Q,
  wpgen E t Q ==> wp (isubst E t) Q.

```

```

Parameter triple_app_fun_from_wpgen : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wpgen ((x,v2)::nil) t1 Q →
  triple (trm_app v1 v2) H Q.

```

Notation for `mkstruct` and `Test`

To avoid clutter in reading the output of `wpgen`, we introduce a lightweight shorthand to denote `mkstruct F`, allowing it to display simply as ‘`F`’.

```
Notation "“ F ” := (mkstruct F) (at level 10, format “ F ”) : wpgen_scope.
```

Let us test again the readability of the output of `wpgen` on a concrete example.

```
Module WPGENWITHMKSTRUCT.
```

```
Import ExamplePrograms.
```

```
Open Scope wpgen_scope.
```

```

Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
  (p ~~> n)
  (fun _ => p ~~> (n+1)).

```

```
Proof using.
```

```

  intros. applys triple_app_fun_from_wpgen. { reflexivity. }
  simpl.

```

```
Abort.
```

Up to proper tabulation, alpha-renaming, and removal of parentheses (and dummy quotes after `Let` and `If`), `F` reads as:

```
‘Let n := ‘(App val_get p) in ‘Let m := ‘(App (val_add n) 1) in ‘App (val_set p) m
```

```
End WPGENWITHMKSTRUCT.
```

43.3.5 Lemmas for Handling `wpgen` Goals

The last major step of the setup of our Separation Logic based verification framework consists of lemmas and tactics to assist in the processing of formulas produced by `wpgen`. For each term construct, and for `mkstruct`, we introduce a dedicated lemma, called “`x`-lemma”, to help with the elimination of the construct.

`xstruct_lemma` is a reformulation of `mkstruct_erase`.

Lemma `xstruct_lemma` : $\forall F H Q,$

$H ==> F Q \rightarrow$

$H ==> \text{mkstruct } F Q.$

Proof using. `introv M. xchange M. applys mkstruct_erase.` `Qed.`

`xval_lemma` reformulates the definition of `wpgen_val`. It just unfolds the definition.

Lemma `xval_lemma` : $\forall v H Q,$

$H ==> Q v \rightarrow$

$H ==> \text{wpgen_val } v Q.$

Proof using. `introv M. applys M.` `Qed.`

`xlet_lemma` reformulates the definition of `wpgen_let`. It just unfolds the definition.

Lemma `xlet_lemma` : $\forall H F1 F2of Q,$

$H ==> F1 (\text{fun } v \Rightarrow F2of v Q) \rightarrow$

$H ==> \text{wpgen_let } F1 F2of Q.$

Proof using. `introv M. xchange M.` `Qed.`

Likewise, `xseq_lemma` reformulates `wpgen_seq`.

Lemma `xseq_lemma` : $\forall H F1 F2 Q,$

$H ==> F1 (\text{fun } v \Rightarrow F2 Q) \rightarrow$

$H ==> \text{wpgen_seq } F1 F2 Q.$

Proof using. `introv M. xchange M.` `Qed.`

`xapp_lemma` applies to goals produced by `wpgen` on an application. In such cases, the proof obligation is of the form $H ==> \text{wp } t Q$.

`xapp_lemma` reformulates the frame-consequence rule, and states the premise of the rule using a `triple`, because triples are used for stating specification lemmas.

Lemma `xapp_lemma` : $\forall t Q1 H1 H2 H Q,$

$\text{triple } t H1 Q1 \rightarrow$

$H ==> H1 \text{ \texttt{*} } H2 \rightarrow$

$Q1 \text{ \texttt{*+} } H2 ==> Q \rightarrow$

$H ==> \text{wp } t Q.$

Proof using.

`introv M WH WQ. rewrite wp_equiv. applys \times triple_conseq_frame M.`

`Qed.`

`xwp_lemma` is another name for `triple_app_fun_from_wpgen`. It is used for establishing a triple for a function application.

Lemma `xwp_lemma` : $\forall v1 v2 x t1 H Q,$

$v1 = \text{val_fun } x t1 \rightarrow$

$H ==> \text{wpgen } ((x, v2) :: \text{nil}) t1 Q \rightarrow$

$\text{triple } (\text{trm_app } v1 v2) H Q.$

Proof using. `introv M1 M2.` `applys \times triple_app_fun_from_wpgen.` `Qed.`

43.3.6 An Example Proof

Let us illustrate how the “x-lemmas” help clarifying verification proof scripts.

Module PROOFSWITHXLEMMAS.

Import ExamplePrograms.

Open Scope wpigen_scope.

```
Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
  (p ~~> n)
  (fun _ ⇒ p ~~> (n+1)).
```

Proof using.

intros.

applys xwp_lemma. { reflexivity. }

simpl.

applys xstruct_lemma.

applys xlet_lemma.

applys xstruct_lemma.

applys xapp_lemma. { apply triple_get. } { xsimpl. }

xpull; intros ? →.

applys xstruct_lemma.

applys xlet_lemma.

applys xstruct_lemma.

applys xapp_lemma. { apply triple_add. } { xsimpl. }

xpull; intros ? →.

applys xstruct_lemma.

applys xapp_lemma. { apply triple_set. } { xsimpl. }

xsimpl.

Qed.

Exercise: 2 stars, standard, especially useful (triple_succ_using_incr_with_xlemmas)

Using x-lemmas, verify the proof of triple_succ_using_incr. (The proof was carried out using triples in chapter Rules.)

```
Lemma triple_succ_using_incr_with_xlemmas : ∀ (n:int),
  triple (trm_app succ_using_incr n)
  \[]
  (fun v ⇒ \[v = n+1]).
```

Proof using. Admitted.

□

End PROOFSWITHXLEMMAS.

43.3.7 Making Proof Scripts More Concise

For each x-lemma, we introduce a dedicated tactic to apply that lemma and perform the associated bookkeeping.

xstruct eliminates the leading `mkstruct`.

Tactic Notation "xstruct" :=

applys xstruct_lemma.

val, *xseq* and *xlet* invoke the corresponding x-lemmas, after calling *xstruct* if a leading `mkstruct` is in the way.

Tactic Notation "xstruct_if_needed" :=

 try match goal with $\vdash ?H \Rightarrow \text{mkstruct } ?F ?Q \Rightarrow \text{xstruct}$ end.

Tactic Notation "xval" :=

xstruct_if_needed; applys xval_lemma.

Tactic Notation "xlet" :=

xstruct_if_needed; applys xlet_lemma.

Tactic Notation "xseq" :=

xstruct_if_needed; applys xseq_lemma.

xapp_nosubst applies `xapp_lemma`, after calling *xseq* or *xlet* if applicable. (We will subsequently define *xapp* as an enhanced version of *xapp_nosubst* that is able to automatically perform substitutions.)

Tactic Notation "xseq_xlet_if_needed" :=

 try match goal with $\vdash ?H \Rightarrow \text{mkstruct } ?F ?Q \Rightarrow$

 match *F* with

 | `wpgen_seq` *?F1 ?F2* $\Rightarrow \text{xseq}$

 | `wpgen_let` *?F1 ?F2of* $\Rightarrow \text{xlet}$

 end end.

Tactic Notation "xapp_nosubst" constr(*E*) :=

xseq_xlet_if_needed; xstruct_if_needed;

applys xapp_lemma E; [xsimpl | xpull].

xwp applies `xwp_lemma`, then requests Coq to evaluate the `wpgen` function. (For technical reasons, we need to explicitly request the unfolding of `wpgen_var`.)

Tactic Notation "xwp" :=

intros; applys xwp_lemma;

 [`reflexivity`

 | `simpl; unfold wpgen_var; simpl`].

Let us revisit the previous proof scripts using x-tactics instead of x-lemmas. The reader may contemplate the gain in conciseness.

Module PROOFSWITHXTACTICS.

Import ExamplePrograms.

```
Open Scope wpigen_scope.
```

```
Lemma triple_incr : ∀ (p:loc) (n:int),  
  triple (trm_app incr p)  
  (p ~~> n)  
  (fun _ ⇒ p ~~> (n+1)).
```

Proof using.

```
xwp.  
xapp_nosubst triple_get. intros ? →.  
xapp_nosubst triple_add. intros ? →.  
xapp_nosubst triple_set.  
xsimpl.
```

Qed.

Exercise: 2 stars, standard, especially useful (triple_succ_using_incr_with_xtactics)
Using x-tactics, verify the proof of succ_using_incr.

```
Lemma triple_succ_using_incr_with_xtactics : ∀ (n:int),  
  triple (trm_app succ_using_incr n)  
  \[]  
  (fun v ⇒ \[v = n+1]).
```

Proof using. Admitted.

□

End PROOFSWITHXTACTICS.

43.3.8 Further Improvements to the *xapp* Tactic.

In this section, we describe two improvements to the *xapp* tactic.

The pattern *xapp_nosubst E. intros ? ->*. appears frequently in the above proofs. This pattern is typically useful whenever the specification *E* features a postcondition of the form *fun v ⇒ \[v = ..]* or of the form *fun v ⇒ \[v = ..] * ...*

Likewise, the pattern *xapp_nosubst E. intros ? p ->*. appears frequently. This pattern is typically useful whenever the specification *E* features a postcondition of the form *fun v ⇒ \exists p, \[v = ..] * ...*.

It therefore makes sense to introduce a tactic *xapp E* that, after calling *xapp_nosubst E*, attempts to invoke *intros ? →* or *intros ? x ->; revert x*. In the latter case, to ensure that the name *x* that we use does not modify the existing name of the bound variable, we play some Ltac hacks, captured by the tactic *xapp_try_subst*.

```
Tactic Notation "xapp_try_subst" :=  
try match goal with  
| ⊢ ∀ (r:val), (r = _) → _ ⇒ intros ? →  
| ⊢ ∀ (r:val), ∀ x, (r = _) → _ ⇒  
  let y := fresh x in intros ? y ->; revert y
```

```
end.
```

```
Tactic Notation "xapp" constr(E) :=
  xapp_nosubst E; xapp_try_subst.
```

Explicitly providing arguments to *xapp_nosubst* or *xapp* is very tedious. To avoid that effort, we can set up the tactics to automatically look up for the relevant specification.

To that end, we instrument `eauto` to exploit a database of already-established specification triples. This database, named `triple`, can be populated using the `Hint Resolve ... : triple` command, as illustrated below.

```
Hint Resolve triple_get triple_set triple_ref triple_free triple_add : triple.
```

The argument-free variants *xapp_subst* and *xapp* are implemented by invoking `eauto` with `triple` to retrieve the relevant specification.

The definition from *Direct* is slightly more powerful, in that it is also able to pick up an induction hypothesis from the context for instantiating the triple.

DISCLAIMER: the tactic *xapp* that leverages the `triple` database is not able to automatically apply specifications that feature a premise that `eauto` cannot solve. To exploit such specifications, one need to provide the specification explicitly (using *xapp E*), or to exploit a more complex hint mechanism (as done in CFML). (A poor-man's workaround consists in moving all the premises inside the precondition, however doing so harms readability.)

```
Tactic Notation "xapp_nosubst" :=
  xseq_xlet_if_needed; xstruct_if_needed;
  applys xapp_lemma; [ solve [ eauto with triple ] | xsimpl | xpull ].
```

```
Tactic Notation "xapp" :=
  xapp_nosubst; xapp_try_subst.
```

43.3.9 Demo of a Practical Proof using x-Tactics.

```
Module PROOFSTWITHADVANCEDXTACTICS.
```

```
Import ExamplePrograms.
```

```
Open Scope wpgen_scope.
```

The proof script for the verification of *incr* using the tactic *xapps* with implicit argument.

```
Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
  (p ~~> n)
  (fun _ => p ~~> (n+1)).
```

```
Proof using.
```

```
  xwp. xapp. xapp. xapp. xsimpl.
```

```
Qed.
```

In order to enable automatically exploiting the specification of `triple` in the verification of `succ_using_incr`, which includes a function call to `triple`, we add it to the hint database `triple`.

Hint Resolve triple_incr : triple.

Exercise: 2 stars, standard, especially useful (triple_succ_using_incr_with_xapps)
Using the improved x-tactics, verify the proof of succ_using_incr.

```
Lemma triple_succ_using_incr_with_xapps : ∀ (n:int),
  triple (trm_app succ_using_incr n)
  \ []
  (fun v ⇒ \[v = n+1]).
```

Proof using. Admitted.

□

In summary, thanks to `wpgen` and its associated x-tactics, we are able to verify concrete programs in Separation Logic with concise proof scripts.

End PROOFSWITHADVANCEDXTACTICS.

43.4 Optional Material

43.4.1 Tactics *xconseq* and *xframe*

The tactic *xconseq* enables weakening the current postcondition. Optionally, the tactic can be passed an explicit argument, using the syntax *xconseq Q*.

The tactic is implemented by applying the lemma `xconseq_lemma`, stated below.

Exercise: 1 star, standard, optional (xconseq_lemma) Prove the `xconseq_lemma`.

```
Lemma xconseq_lemma : ∀ Q1 Q2 H F,
  H ==> mkstruct F Q1 →
  Q1 ===> Q2 →
  H ==> mkstruct F Q2.
```

Proof using. Admitted.

□

```
Tactic Notation "xconseq" :=
  applys xconseq_lemma.
```

```
Tactic Notation "xconseq" constr(Q) :=
  applys xconseq_lemma Q.
```

The tactic *xframe* enables applying the frame rule on a formula produced by `wpgen`. The syntax *xframe H* is used to specify the heap predicate to keep, and the syntax *xframe_out H* is used to specify the heap predicate to frame out—everything else is kept.

Exercise: 2 stars, standard, optional (xframe_lemma) Prove the `xframe_lemma`. Exploit the properties of `mkstruct`; do not try to unfold the definition of `mkstruct`.

```
Lemma xframe_lemma : ∀ H1 H2 H Q Q1 F,
  H ==> H1 \* H2 →
  H1 ==> mkstruct F Q1 →
  Q1 \*+ H2 ==> Q →
  H ==> mkstruct F Q.
```

Proof using. Admitted.

□

```
Tactic Notation "xframe" constr(H) :=
  eapply (@xframe_lemma H); [ xsimpl || ].
```

```
Tactic Notation "xframe_out" constr(H) :=
  eapply (@xframe_lemma _ H); [ xsimpl || ].
```

Let's illustrate the use of `xframe` on an example.

```
Module PROOFSTHROUGHSTRUCTURALXTACTICS.
```

```
Import ExamplePrograms.
```

```
Open Scope wpgen_scope.
```

```
Lemma triple_incr_frame : ∀ (p q:loc) (n m:int),
  triple (trm_app incr p)
  (p ~~> n \* q ~~> m)
  (fun _ => (p ~~> (n+1)) \* (q ~~> m)).
```

Proof using.

xwp.

Instead of calling `xapp`, we put aside `q ~~> m` and focus on `p ~~> n`. *xframe* (`p ~~> n`). *xapp. xapp. xapp.*

Finally we check the check that the current state augmented with the framed predicate `q ~~> m` matches with the claimed postcondition. *xsimp.*

Qed.

```
End PROOFSTHROUGHSTRUCTURALXTACTICS.
```

43.4.2 Soundness Proof for `wpgen`

```
Module WPGENSOUND.
```

Introduction of the Predicate `formula_sound`

The soundness theorem that we aim to establish for `wpgen` is:

Parameter wpgen_sound : forall E t Q, wpgen E t Q ==> wp (isubst E t) Q.

Before we enter the details of the proof, let us reformulate the soundness statement of the soundness theorem in a way that will make proof obligations and induction hypotheses

easier to read. To that end, we introduce the predicate `formula_sound t F`, which asserts that F is a weakest-precondition style formula that entails `wp t`. Formally:

Definition `formula_sound (t:trm) (F:formula) : Prop :=`
 $\forall Q, F \ Q ==> \text{wp } t \ Q.$

Using `formula_sound`, the soundness theorem for `wpgen` reformulates as follows.

Parameter `wpgen_sound' : \forall E t,`
`formula_sound (isubst E t) (wpgen E t).`

Soundness for the Case of Sequences

Let us forget about the existence of `mkstruct` for a minute, that is, pretend that `wpgen` is defined without `mkstruct`.

In that setting, `wpgen E (trm_seq t1 t2)` evaluates to `wpgen_seq (wpgen E t1) (wpgen E t2)`.

Recall the definition of `wpgen_seq`.

Definition `wpgen_seq (F1 F2:formula) : formula := fun Q => F1 (fun v => F2 Q).`

Consider the soundness theorem `wpgen_sound` and let us specialize it to the particular case where t is of the form `trm_seq t1 t2`. The corresponding statement is:

Parameter `wpgen_sound_seq : \forall E t1 t2,`
`formula_sound (isubst E (trm_seq t1 t2)) (wpgen E (trm_seq t1 t2)).`

In that statement:

- `wpgen E (trm_seq t1 t2)` evaluates to `wpgen_seq (wpgen E t1) (wpgen E t2)`.
- `isubst E (trm_seq t1 t2)` evaluates to `trm_seq (isubst E t1) (isubst E t2)`.

Moreover, by induction hypothesis, we will be able to assume the soundness result to already hold for the subterms $t1$ and $t2$.

Thus, to establish the soundness of `wpgen` for sequences, we have to prove the following result:

Parameter `wpgen_sound_seq' : \forall E t1 t2,`
`formula_sound (isubst E t1) (wpgen E t1) →`
`formula_sound (isubst E t2) (wpgen E t2) →`
`formula_sound (trm_seq (isubst E t1) (isubst E t2))`
`(wpgen_seq (wpgen E t1) (wpgen E t2)).`

In the above statement, let us abstract `isubst E t1` as $t1'$ and `wpgen t1` as $F1$, and similarly `isubst E t2` as $t2'$ and `wpgen t2` as $F2$. The statement reformulates as:

Lemma `wpgen_seq_sound : \forall t1' t2' F1 F2,`
`formula_sound t1' F1 →`
`formula_sound t2' F2 →`
`formula_sound (trm_seq t1' t2') (wpgen_seq F1 F2).`

This statement captures the essence of the correctness of the definition of `wpgen_seq`. Let's prove it.

Proof using.

```

introv S1 S2.
unfolds formula_sound.
intros Q.
unfolds wpgen_seq.
applys himpl_trans.
2:{ applys wp_seq. }

{ applys himpl_trans.
  { applys S1. }
  { applys wp_conseq. intros v. applys S2. } }

```

Qed.

Soundness of `wpgen` for the Other Term Constructs

The proof for the other term constructs are shown below and will be used to set up the main induction. The reader may skip over the proof details.

Lemma `wpgen_val_sound` : $\forall v,$
 $\text{formula_sound}(\text{trm_val } v) (\text{wpgen_val } v).$

Proof using. `intros. intros Q. unfolds wpgen_val. applys wp_val.` Qed.

Lemma `wpgen_fun_val_sound` : $\forall x t,$
 $\text{formula_sound}(\text{trm_fun } x t) (\text{wpgen_val}(\text{val_fun } x t)).$

Proof using. `intros. intros Q. unfolds wpgen_val. applys wp_fun.` Qed.

Lemma `wpgen_fix_val_sound` : $\forall f x t,$
 $\text{formula_sound}(\text{trm_fix } f x t) (\text{wpgen_val}(\text{val_fix } f x t)).$

Proof using. `intros. intros Q. unfolds wpgen_val. applys wp_fix.` Qed.

Lemma `wpgen_let_sound` : $\forall F1 F2of x t1 t2,$
 $\text{formula_sound} t1 F1 \rightarrow$
 $(\forall v, \text{formula_sound}(\text{subst } x v t2) (F2of v)) \rightarrow$
 $\text{formula_sound}(\text{trm_let } x t1 t2) (\text{wpgen_let } F1 F2of).$

Proof using.

```

introv S1 S2. intros Q. unfolds wpgen_let. applys himpl_trans wp_let.
applys himpl_trans S1. applys wp_conseq. intros v. applys S2.

```

Qed.

Lemma `wpgen_if_sound` : $\forall F1 F2 t0 t1 t2,$
 $\text{formula_sound} t1 F1 \rightarrow$
 $\text{formula_sound} t2 F2 \rightarrow$
 $\text{formula_sound}(\text{trm_if } t0 t1 t2) (\text{wpgen_if } t0 F1 F2).$

Proof using.

```

intros S1 S2. intros Q. unfold wpgen_if. xpull. intros b →.
applys himpl_trans wp_if. case_if. { applys S1. } { applys S2. }
Qed.

```

The formula `wpgen_fail` is a sound formula not just for a variable `trm_var x`, but in fact for any term `t`. Indeed, the entailment $\backslash[\text{False}] ==> \text{wp } t \ Q$ is always true. Hence the general statement for `wpgen_fail` that appears next.

```

Lemma wpgen_fail_sound : ∀ t,
  formula_sound t wpgen_fail.

```

```

Proof using. intros. intros Q. unfold wpgen_fail. xpull. Qed.

```

The formula `wp t` is a sound formula for a term `t`, not just when `t` is an application, but for any term `t`. Hence the general statement for `wp` that appears next.

```

Lemma wp_sound : ∀ t,
  formula_sound t (wp t).

```

```

Proof using. intros. intros Q. applys himpl_refl. Qed.

```

Soundness of `mkstruct`

To carry out the soundness proof for `wpgen`, we also need to justify that the addition of `mkstruct` to the head of every call to `wpgen` preserves the entailment $\text{wpgen } t \ Q ==> \text{wp } t \ Q$.

Consider a term `t`. The result of `wpgen t` takes the form `mkstruct F`, where `F` denotes the main pattern matching on `t` that occurs in the definition of `wpgen`.

Our goal is to show that if the formula `F` is a valid weakest precondition for `t`, then so is `mkstruct F`. One way to prove this result is to exploit the fact that, when a formula `F` is of the form `wp t`, applying `mkstruct` does not alter its meaning (recall lemma `mkstruct_wp`).

```

Lemma mkstruct_sound : ∀ t F,
  formula_sound t F →
  formula_sound t (mkstruct F).

```

```

Proof using.

```

```

  introv M. unfolds formula_sound. intros Q.
  rewrite ← mkstruct_wp. applys mkstruct_monotone. applys M.

```

```

Qed.

```

Another, lower-level proof for the same result reveals the definition of `mkstruct` and exploits the consequence-frame rule for `wp`.

```

Lemma mkstruct_sound' : ∀ t F,
  formula_sound t F →
  formula_sound t (mkstruct F).

```

```

Proof using.

```

```

  introv M. unfolds formula_sound.
  intros Q.
  unfolds mkstruct.

```

```

 $x\text{simpl}; \text{intros } Q' H N.$ 
 $\text{lets } M': M \ Q'. \text{xchange } M'.$ 
 $\text{applys wp\_conseq\_frame. applys } N.$ 
Qed.

```

Lemmas Capturing Properties of Iterated Substitutions

In the proof of soundness for `wpgen`, we need to exploit two basic properties of the iterated substitution function `isubst`.

The first property asserts that the substitution for the empty context is the identity. This result is needed to cleanly state the final statement of the soundness theorem.

```

Parameter isubst_nil :  $\forall t,$ 
isubst nil  $t = t.$ 

```

The second property asserts that the substitution for a context $(x,v)::E$ is the same as the substitution for the context `rem x E` followed with the substitution of x by v using the basic substitution function `subst`. This second property is needed to handle the case of let-bindings.

```

Parameter isubst_rem :  $\forall x v E t,$ 
isubst  $((x,v)::E) t = \text{subst } x v (\text{isubst } (\text{rem } x E) t).$ 

```

The proofs for these two lemmas is technical and of limited interest. They can be found in appendix near the end of this chapter.

Main Induction for the Soundness Proof of `wpgen`

We prove the soundness of `wpgen E t` by structural induction on t .

As explained previously, the soundness lemma is stated with help of the predicate `formula_sound`, in the form: `formula_sound (isubst E t) (wpgen t)`.

For each term construct, the proof case consists of two steps:

- first, invoke the lemma `mkstruct_sound` to justify soundness of the leading `mkstruct` produced by `wpgen`,
- second, invoke the the soundness lemma specific to that term construct, e.g. `wpgen_seq_sound` for sequences.

```

Lemma wpgen_sound_induct :  $\forall E t,$ 
formula_sound  $(\text{isubst } E t) (\text{wpgen } E t).$ 

```

Proof using.

```

intros. gen E. induction t; intros; simpl;
applys mkstruct_sound.
{ applys wpgen_val_sound. }
{ rename v into x. unfold wpgen_var. case_eq (lookup x E).

```

```

{ intros v EQ. applys wpgen_val_sound. }
{ intros N. applys wpgen_fail_sound. } }
{ applys wpgen_fun_val_sound. }
{ applys wpgen_fix_val_sound. }
{ applys wp_sound. }
{ applys wpgen_seq_sound. { applys IHt1. } { applys IHt2. } }
{ rename v into x. applys wpgen_let_sound.
  { applys IHt1. }
  { intros v. rewrite ← isubst_rem. applys IHt2. } }
{ applys wpgen_if_sound. { applys IHt2. } { applys IHt3. } }
Qed.

```

Statement of Soundness of `wpgen` for Closed Terms

For a closed term t , the context E is instantiated as `nil`, and `isubst nil t` simplifies to t . We obtain the main soundness statement for `wpgen`.

Lemma `wpgen_sound` : $\forall t Q$,
 $wpgen \text{ nil } t Q \implies wp t Q$.

Proof using.

`introsv M. lets N: (wpgen_sound nil t). rewrite isubst_nil in N. applys × N.`

Qed.

A corollary can be derived for deriving a triple of the form `triple t H Q` from `wpgen nil t`.

Lemma `triple_of_wpgen` : $\forall t H Q$,
 $H \implies wpgen \text{ nil } t Q \rightarrow$
 $\text{triple } t H Q$.

Proof using.

`introsv M. rewrite ← wp_equiv. xchange M. applys wpgen_sound.`

Qed.

The lemma `triple_app_fun_from_wpgen`, used by the tactic `xwp`, is a variant of `wpgen_of_triple` specialized for establishing a triple for a function application. (Recall that, in essence, this lemma is a reformulation of the rule `triple_app_fun`.)

Lemma `triple_app_fun_from_wpgen` : $\forall v1 v2 x t1 H Q$,
 $v1 = \text{val_fun } x t1 \rightarrow$
 $H \implies wpgen ((x, v2) :: \text{nil}) t1 Q \rightarrow$
 $\text{triple } (\text{trm_app } v1 v2) H Q$.

Proof using.

`introsv M1 M2. applys triple_app_fun M1.`
`asserts_rewrite (subst x v2 t1 = isubst ((x, v2) :: \text{nil}) t1).`
`{ rewrite isubst_rem. rewrite × isubst_nil. }`
`rewrite ← wp_equiv. xchange M2. applys wpgen_sound_induct.`

Qed.

The lemma *triple_app_fix_from_wpgen* is a variant of the above lemma suited for recursive functions. Note that the proof exploits a lemma called *isubst_rem_2* which is an immediate generalization of the lemma *isubst_rem*. The proof of *isubst_rem_2* appears near the bottom of this file.

```
Lemma triple_app_fix_from_wpgen : ∀ v1 v2 f x t1 H Q,
  v1 = val_fix f x t1 →
  H ==> wpgen ((f, v1) :: (x, v2) :: nil) t1 Q →
  triple (trm_app v1 v2) H Q.
```

Proof using.

```
intros M1 M2. applys triple_app_fix M1.
asserts_rewrite (subst x v2 (subst f v1 t1)
  = isubst ((f, v1) :: (x, v2) :: nil) t1).
{ rewrite isubst_rem_2. rewrite× isubst_nil. }
rewrite ← wp_equiv. xchange M2. applys wpgen_sound_induct.
```

Qed.

End WPGENSOUND.

43.4.3 Guessing the Definition of `mkstruct`

Module MKSTRUCTGUESS.

How could we have possibly guessed the definition of `mkstruct`?

Recall that we observed, in an exercise, that the definition of `mkstruct` satisfies the idempotence property:

Lemma `mkstruct_idempotence` : forall F mkstruct (mkstruct F) = mkstruct F

This idempotence property reflects in particular the fact that consecutive applications of the frame rule can be combined into a single application of this rule, and likewise for the rule of consequence. Since it seems to make some sense for `mkstruct` to be idempotent, let us pretend that this property is a requirement for `mkstruct`.

In other words, assume that we are searching for a predicate satisfying 4 properties: `mkstruct_frame`, `mkstruct_conseq`, `mkstruct_erase`, and `mkstruct_idempotence`.

The following reasoning steps can lead to figure out a definition of `mkstruct` that satisfies these properties.

Recall the statement of `mkstruct_frame` and of `mkstruct_conseq`.

```
Parameter mkstruct_frame : ∀ F H Q,
  (mkstruct F Q) \* H ==> mkstruct F (Q \*+ H).
```

```
Parameter mkstruct_conseq : ∀ F Q1 Q2,
  Q1 ==> Q2 →
  mkstruct F Q1 ==> mkstruct F Q2.
```

The two rules can be combined into a single one as follows (similarly to the way it is done for `wp_conseq_frame`).

```

Parameter mkstruct_conseq_frame : ∀ F Q1 Q2 H,
  Q1 \*+ H ==> Q2 →
  H \* (mkstruct F Q1) ==> mkstruct F Q2.

```

By the idempotence property *mkstruct_idempotence*, `mkstruct F` should be equal to `mkstruct (mkstruct F)`. Let's exploit this equality for the second occurrence of `mkstruct F`.

```

Parameter mkstruct_conseq_idempotence : ∀ F Q1 Q2 H,
  Q1 \*+ H ==> Q2 →
  H \* (mkstruct F Q1) ==> mkstruct (mkstruct F) Q2.

```

Now, let's replace `mkstruct F` with F' . Doing so results in the statement shown below, which gives a sufficient condition for the statement *mkstruct_conseq_idempotence* to hold.

```

Parameter mkstruct_conseq_idempotence_generalized : ∀ F' Q1 Q2 H,
  Q1 \*+ H ==> Q2 →
  H \* (F' Q1) ==> mkstruct F' Q2.

```

We can reformulate the above statement as an introduction rule by merging the hypothesis into the left-hand side of the entailment from the conclusion. We thereby obtain an introduction lemma for `mkstruct`.

```

Parameter mkstruct_introduction : ∀ F' Q2,
  ∃ Q1 H, \[Q1 \*+ H ==> Q2] \* H \* (F' Q1) ==> mkstruct F' Q2.

```

For this entailment to hold, because entailment is a reflexive relation, it is sufficient to define `mkstruct F' Q2`, that is, the right-hand side of the entailment, as equal to the contents of the left-hand side.

```

Definition mkstruct (F':formula) : formula :=
  fun (Q2:val→hprop) ⇒ ∃ Q1 H, \[Q1 \*+ H ==> Q2] \* H \* (F' Q1).

```

As we have proved earlier in this chapter, this definition indeed satisfies the 4 desired properties: `mkstruct_frame`, `mkstruct_conseq`, `mkstruct_erase`, and `mkstruct_idempotence`.

End MKSTRUCTGUESS.

43.4.4 Proof of Properties of Iterated Substitution

Module ISUBSTPROP.

Open Scope liblist_scope.

Implicit Types E : ctx.

Recall that `isubst E t` denotes the multi-substitution in the term t of all bindings form the association list E .

The soundness proof for `wpgen` and the proof of its corollary *triple_app_fun_from_wpgen* rely on two key properties of iterated substitutions, captured by the lemmas called `isubst_nil` and `isubst_rem`, which we state and prove next.

`isubst nil t = t`

```
subst x v (isubst (rem x E) t) = isubst ((x,v)::E) t
```

The first lemma is straightforward by induction.

```
Lemma isubst_nil : ∀ t,  
  isubst nil t = t.
```

Proof using.

```
  intros t. induction t; simpl; f_equal.
```

Qed.

The second lemma is much more involved to prove.

We introduce the notion of “two equivalent contexts” E_1 and E_2 , and argue that substitution for two equivalent contexts yields the same result.

We then introduce the notion of “contexts with disjoint domains”, and argue that if E_1 and E_2 are disjoint then $\text{isubst } (E_1 ++ E_2) t = \text{isubst } E_1 (\text{isubst } E_2 t)$.

Before we start, we describe the tactic *case_var*, which helps with the case_analyses on variable equalities, and we prove an auxiliary lemma that describes the result of a lookup on a context from which a binding has been removed. It is defined in file *LibSepVar.v* as:

Tactic Notation “*case_var*” := repeat rewrite var_eq_spec in *; repeat *case_if*.

The tactic *case_var** is a shorthand for *case_var* followed with automation (essentially, *eauto*).

On key auxiliary lemma relates *subst* and *isubst*.

```
Lemma subst_eq_isubst_one : ∀ x v t,  
  subst x v t = isubst ((x, v)::nil) t.
```

Proof using.

```
  intros. induction t; simpl.  
  { f_equal. }  
  { case_var*. }  
  { f_equal. case_var*. { rewrite* isubst_nil. } }  
  { f_equal. case_var; try case_var; simpl; try case_var;  
    try rewrite isubst_nil; auto. }  
  { f_equal*. }  
  { f_equal*. }  
  { f_equal*. case_var*. { rewrite* isubst_nil. } }  
  { f_equal*. }
```

Qed.

A lemma about the lookup in a removal.

```
Lemma lookup_rem : ∀ x y E,  
  lookup x (rem y E) = if var_eq x y then None else lookup x E.
```

Proof using.

```
  intros. induction E as [(z, v) E].  
  { simpl. case_var*. }  
  { simpl. case_var*; simpl; case_var*. }
```

Qed.

A lemma about the removal over an append.

```
Lemma rem_app : ∀ x E1 E2,  
  rem x (E1 ++ E2) = rem x E1 ++ rem x E2.
```

Proof using.

```
  intros. induction E1 as [(y,w) E1']; rew_list; simpl. { auto. }  
  { case_var. { rew_list. fequals. } }
```

Qed.

The definition of equivalent contexts.

```
Definition ctx_equiv E1 E2 :=  
  ∀ x, lookup x E1 = lookup x E2.
```

The fact that removal preserves equivalence.

```
Lemma ctx_equiv_rem : ∀ x E1 E2,  
  ctx_equiv E1 E2 →  
  ctx_equiv (rem x E1) (rem x E2).
```

Proof using.

```
  introv M. unfolds ctx_equiv. intros y.  
  do 2 rewrite lookup_rem. case_var.
```

Qed.

The fact that substitution for equivalent contexts yields equal results.

```
Lemma isubst_ctx_equiv : ∀ t E1 E2,  
  ctx_equiv E1 E2 →  
  isubst E1 t = isubst E2 t.
```

Proof using.

```
  hint ctx_equiv_rem.  
  intros t. induction t; introv EQ; simpl; fequals.  
  { rewrite EQ. auto. }
```

Qed.

The definition of disjoint contexts.

```
Definition ctx_disjoint E1 E2 :=  
  ∀ x v1 v2, lookup x E1 = Some v1 → lookup x E2 = Some v2 → False.
```

Removal preserves disjointness.

```
Lemma ctx_disjoint_rem : ∀ x E1 E2,  
  ctx_disjoint E1 E2 →  
  ctx_disjoint (rem x E1) (rem x E2).
```

Proof using.

```
  introv D. intros y v1 v2 K1 K2. rewrite lookup_rem in *.  
  case_var. applys D K1 K2.
```

Qed.

Lookup in the concatenation of two disjoint contexts

```

Lemma lookup_app : ∀ x E1 E2,
  lookup x (E1 ++ E2) = match lookup x E1 with
    | None ⇒ lookup x E2
    | Some v ⇒ Some v
  end.

```

Proof using.

```

intros. induction E1 as [(y,w) E1]; rew_list; simpl; intros.
{ auto. }
{ case_var. }

```

Qed.

The key induction shows that `isubst` distributes over context concatenation.

```

Lemma isubst_app : ∀ t E1 E2,
  isubst (E1 ++ E2) t = isubst E2 (isubst E1 t).

```

Proof using.

```

hint ctx_disjoint_rem.
intros t. induction t; simpl; intros.
{ fequals. }
{ rename v into x. rewrite $\times$  lookup_app.
  case_eq (lookup x E1); introv K1; case_eq (lookup x E2); introv K2; auto.
  { simpl. rewrite $\times$  K2. }
  { simpl. rewrite $\times$  K2. } }
{ fequals. rewrite $\times$  rem_app. }
{ fequals. do 2 rewrite $\times$  rem_app. }
{ fequals $\times$ . }
{ fequals $\times$ . }
{ fequals $\times$ . { rewrite $\times$  rem_app. } }
{ fequals $\times$ . }

```

Qed.

The next lemma asserts that the concatenation order is irrelevant in a substitution if the contexts have disjoint domains.

```

Lemma isubst_app_swap : ∀ t E1 E2,
  ctx_disjoint E1 E2 →
  isubst (E1 ++ E2) t = isubst (E2 ++ E1) t.

```

Proof using.

```

intros D. applys isubst_ctx_equiv. applys $\times$  ctx_disjoint_equiv_app.

```

Qed.

We are ready to derive the desired property of `isubst`.

```

Lemma isubst_rem : ∀ x v E t,
  isubst ((x, v)::E) t = subst x v (isubst (rem x E) t).

```

Proof using.

```

intros. rewrite subst_eq_isubst_one. rewrite ← isubst_app.

```

```

rewrite isubst_app_swap.
{ applys isubst_ctx_equiv. intros y. rew_list. simpl. case_var×.
  { rewrite lookup_rem. case_var×. } }
{ intros y v1 v2 K1 K2. simpls. case_var.
  { subst. rewrite lookup_rem in K1. case_var×. } }
Qed.

```

We also prove the variant lemma which is useful for handling recursive functions.

```

Lemma isubst_rem_2 : ∀ f x vf vx E t,
  isubst ((f , vf) :: (x , vx) :: E) t
  = subst x vx (subst f vf (isubst (rem x (rem f E)) t)).
Proof using.

```

```

intros. do 2 rewrite subst_eq_isubst_one. do 2 rewrite ← isubst_app.
rewrite isubst_app_swap.
{ applys isubst_ctx_equiv. intros y. rew_list. simpl.
  do 2 rewrite lookup_rem. case_var×. }
{ intros y v1 v2 K1 K2. rew_listx in *. simpls.
  do 2 rewrite lookup_rem in K1. case_var. }

```

Qed.

End ISUBSTPROP.

43.4.5 Historical Notes

Many verification tools based on Hoare Logic are based on a weakest-precondition generator. Typically, a tool takes as input a source code annotated with specifications and invariants, and produces a logical formula that entails the correctness of the program. This logical formula is typically expressed in first-order logic, and is discharged using automated tools such as SMT solvers.

In contrast, the weakest-precondition generator presented in this chapter applies to unannotated code. It thus produces a logical formula that does not depend at all on the specifications and invariants. Such formula, which can be constructed in a systematic manner, is called a “characteristic formula” in the literature. In general, a characteristic formula provides not just a sound but also a complete description of the semantics of a program. Discussion of completeness is beyond the scope of this course.

The notion of characteristic formula was introduced work by *Hennessy and Milner* 1985 (in Bib.v) on process calculi. It was first applied to program logic by *Honda, Berger, and Yoshida* 2006 (in Bib.v). It was then applied to Separation Logic in the PhD work of *Charguéraud* 2010 (in Bib.v), which resulted in the CFML tool. CFML 1.0 used an external tool that produced characteristic formulae in the form of Coq axioms. Later work by *Guéneau, Myreen, Kumar and Norrish* 2017 (in Bib.v) showed how the characteristic formulae could be produced together with proofs justifying their correctness.

In Charguéraud’s PhD and in Guéneau et al.’s work, characteristic formulae were slightly

more complicated than those presented here, because they did not leverage the weakest-precondition approach, which streamlines the presentation.

Compared with Hoare Logic, one key aspect of characteristic formulae for Separation Logic is the need to support the frame rule. In this chapter, the predicate `mkstruct` introduced at every node of the output of `wpgen` serves that purpose. The definition of `wpgen` as stated in this chapter will probably be the matter of a publication in 2021.

Chapter 44

Library `SLF.Wand`

44.1 Wand: The Magic Wand Operator

Set Implicit Arguments.

From *SLF* Require Import LibSepReference.

From *SLF* Require Repr.

Close Scope *trm_scope*.

Implicit Types h : heap.

Implicit Types P : Prop.

Implicit Types H : hprop.

Implicit Types Q : val \rightarrow hprop.

44.2 First Pass

This chapter introduces an additional Separation Logic operator, called the “magic wand”, and written $H1 \text{ \textbackslash -}^* H2$.

This operator has multiple use:

- it helps reformulate the consequence-frame rule in an improved manner, through a rule called the “ramified frame rule”,
- it helps stating the weakest-preconditions of a number of languages constructs in a concise manner,
- it can be useful to state specification for certain data structures.

This chapter is organized as follows:

- definition and properties of the magic wand operator,
- generalization of the magic wand to postconditions,

- statement and benefits of the ramified frame rule,
- statement of the ramified frame rule in weakest-precondition style,
- generalized definition of `wpgen` that recurses in local functions.

The addition and bonus section includes further discussion, including:

- presentation of alternative, equivalent definitions of the magic wand,
- statement and proofs of additional properties of the magic wand,
- a revised definition of `mkstruct` using the magic wand.
- “Texan triples”, which express function specifications using the magic wand instead of using triples,
- two other operators, disjunction and non-separating conjunction, so as to complete the presentation of all Separation Logic operators.

44.2.1 Intuition for the Magic Wand

The magic wand operation $H1 \setminus^* H2$, to be read “H1 wand H2”, defines a heap predicate such that, if we extend it with $H1$, we obtain $H2$. Formally, the following entailment holds:

$$H1 \setminus^* (H1 \setminus^* H2) ==> H2.$$

Intuitively, if one can think of the star $H1 \setminus^* H2$ as the addition $H1 + H2$, then one can think of $H1 \setminus^* H2$ as the subtraction $-H1 + H2$. The entailment stated above essentially captures the idea that $(-H1 + H2) + H1$ simplifies to $H2$.

Note, however, that the operation $H1 \setminus^* H2$ only makes sense if $H1$ describes a piece of heap that “can” be subtracted from $H2$. Otherwise, the predicate $H1 \setminus^* H2$ characterizes a heap that cannot possibly exist. Informally speaking, $H1$ must somehow be a subset of $H2$ for the subtraction $-H1 + H2$ to make sense.

44.2.2 Definition of the Magic Wand

Module WANDDEF.

Technically, $H1 \setminus^* H2$ holds of a heap h if, for any heap h' disjoint from h and that satisfies $H1$, the union of h and h' satisfies $H2$.

The operator `hwand`, which implements the notation $H1 \setminus^* H2$, may thus be defined as follows.

```
Definition hwand' (H1 H2:hprop) : hprop :=
  fun h => ∀ h', Fmap.disjoint h h' → H1 h' → H2 (h \u h').
```

The definition above is perfectly fine, however it is more practical to use an alternative, equivalent definition of `hwand`, expressed in terms of previously introduced Separation Logic operators.

The alternative definition asserts that $H1 \setminus\ast H2$ corresponds to some heap predicate, called $H0$, such that $H0$ starred with $H1$ yields $H2$. In other words, $H0$ is such that $(H1 \setminus\ast H0) ==> H2$. In the definition of `hwand` shown below, observe how $H0$ is existentially quantified.

```
Definition hwand (H1 H2:hprop) : hprop :=  
  ∃ H0, H0 \* \[ H1 \* H0 ==> H2 ].
```

Notation " $H1 \setminus\ast H2$ " := (`hwand` $H1 H2$) (at level 43, right associativity).

As we establish further in this file, one can prove that `hwand` and `hwand'` define the same operator.

The reason we prefer taking `hwand` as definition rather than `hwand'` is that it enables us to establish all the properties of the magic wand by exploiting the tactic `xsimpl`, conducting all the reasoning at the level of `hprop` rather than having to work with explicit heaps of type `heap`.

44.2.3 Characteristic Property of the Magic Wand

The magic wand is not so easy to make sense of, at first. Reading its introduction and elimination rules may help further appreciate its meaning.

The operator $H1 \setminus\ast H2$ satisfies the following equivalence. Informally speaking, think of $H0 = -H1 + H2$ and $H1 + H0 = H2$ being equivalent.

```
Lemma hwand_equiv : ∀ H0 H1 H2,  
  (H0 ==> H1 \setminus\ast H2) ↔ (H1 \* H0 ==> H2).
```

Proof using.

```
unfold hwand. iff M.  
{ xchange M. intros H N. xchange N. }  
{ xsimpl H0. xchange M. }
```

Qed.

It turns out that the magic wand operator is uniquely defined by the equivalence $(H0 ==> H1 \setminus\ast H2) \leftrightarrow (H1 * H0 ==> H2)$. In other words, as we establish further on, any operator that satisfies the above equivalence for all arguments is provably equal to `hwand`.

The right-to-left direction of the equivalence is an introduction rule: it tells what needs to be proved for constructing a magic wand $H1 \setminus\ast H2$ from a state $H0$. What needs to be proved to establish $H0 ==> (H1 \setminus\ast H2)$ is that $H0$, when starred with $H1$, yields $H2$.

```
Lemma himpl_hwand_r : ∀ H0 H1 H2,  
  (H1 \* H0) ==> H2 →  
  H0 ==> (H1 \setminus\ast H2).
```

Proof using. introv M. applys hwand_equiv. applys M. Qed.

The left-to-right direction of the equivalence is an elimination rule: it tells what can be deduced from an entailment $H0 ==> (H1 \text{-}* H2)$. What can be deduced from this entailment is that if $H0$ is starred with $H1$, then $H2$ can be recovered.

`Lemma himpl_hwand_r_inv : ∀ H0 H1 H2,`

$H0 ==> (H1 \text{-}* H2) \rightarrow$

$(H1 \text{*} H0) ==> H2.$

`Proof using. introv M. applys hwand_equiv. applys M. Qed.`

This elimination rule can be equivalently reformulated in the following form, which makes clearer that $H1 \text{-}* H2$, when starred with $H1$, yields $H2$.

`Lemma hwand_cancel : ∀ H1 H2,`

$H1 \text{*} (H1 \text{-}* H2) ==> H2.$

`Proof using. intros. applys himpl_hwand_r_inv. applys himpl_refl. Qed.`

Arguments hwand_cancel : clear implicits.

Exercise: 3 stars, standard, especially useful (hwand_inv) Prove the following inversion lemma for `hwand`. This lemma essentially captures the fact that `hwand` entails its alternative definition `hwand'`.

`Lemma hwand_inv : ∀ h1 h2 H1 H2,`

$(H1 \text{-}* H2) \text{ h2} \rightarrow$

$H1 \text{ h1} \rightarrow$

`Fmap.disjoint h1 h2 →`

$H2 (h1 \cup h2).$

`Proof using. Admitted.`

□

44.2.4 Magic Wand for Postconditions

In what follows, we generalize the magic wand to operate on postconditions, introducing a heap predicate of the form $Q1 \text{-}* Q2$, of type `hprop`. Note that the entailment between two postconditions produces a heap predicate, and not a postcondition.

The definition follows exactly the same pattern as `hwand`: it quantifies some heap predicate $H0$ such that $H0$ starred with $Q1$ yields $Q2$.

`Definition qwand (Q1 Q2:val→hprop) : hprop :=`

$\exists H0, H0 \text{*} \forall [Q1 \text{*+} H0 ==> Q2].$

`Notation "Q1 \text{-}* Q2" := (qwand Q1 Q2) (at level 43).`

The operator `qwand` satisfies essentially the same properties as `hwand`. Let us begin with the associated equivalence rule, which captures both the introduction and the elimination rule.

`Lemma qwand_equiv : ∀ H Q1 Q2,`

$$H ==> (Q1 \setminus-* Q2) \leftrightarrow (Q1 \setminus*+ H) ==> Q2.$$

Proof using.

```
unfold qwand. iff M.
{ intros v. xchange M. intros H4 N. xchange N. }
{ xsimpl H. xchange M. }
```

Qed.

The cancellation rule follows.

```
Lemma qwand_cancel : \forall Q1 Q2,
  Q1 \setminus*+ (Q1 \setminus-* Q2) ==> Q2.
```

Proof using. intros. rewrite ← qwand_equiv. applys qimpl_refl. Qed.

An interesting property of `qwand` is the fact that we can specialize $Q1 \setminus-* Q2$ to $(Q1 \vee \setminus-* (Q2 v))$, for any value v .

```
Lemma qwand_specialize : \forall (v:val) (Q1 Q2:val → hprop),
  (Q1 \setminus-* Q2) ==> (Q1 v \setminus-* Q2 v).
```

Proof using.

```
intros. unfold qwand, hwand. xpull. intros H0 M.
xsimpl H0. xchange M.
```

Qed.

44.2.5 Frame Expressed with `hwand`: the Ramified Frame Rule

Recall the consequence-frame rule, which is pervasively used for example by the tactic `xapp` for reasoning about applications.

```
Parameter triple_conseq_frame : \forall H2 H1 Q1 t H Q,
  triple t H1 Q1 →
  H ==> H1 \setminus* H2 →
  Q1 \setminus*+ H2 ==> Q →
  triple t H Q.
```

This rule suffers from a practical issue, which we illustrate in details on a concrete example further on. For now, let us just attempt to describe the issue at a high-level.

In short, the problem stems from the fact that we need to instantiate $H2$ for applying the rule. Providing $H2$ by hand is not practical, thus we need to infer it. The value of $H2$ can be computed as the subtraction of H minus $H1$. The resulting value may then exploited in the last premise for constructing $Q1 \setminus*+ H2$. This transfer of information via $H2$ from one subgoal to another can be obtained by introducing an “evar” (Coq unification variable) for $H2$. However this approach does not work well in the cases where H contains existential quantifiers. Indeed, such existential quantifiers are typically first extracted out of the entailment $H ==> H1 \setminus* H2$ by the tactic `xsimpl`. However, these existentially quantified variables are not in the scope of $H2$, hence the instantiation of the evar associated with $H2$ typically fails.

The “ramified frame rule” exploits the magic wand operator to circumvent the problem, by merging the two premises $H ==> H1 \ast H2$ and $Q1 \ast+ H2 ==> Q$ into a single premise that no longer mentions $H2$.

This replacement premise is $H ==> H1 \ast (Q1 \dashv\ast Q)$. To understand where it comes from, observe first that the second premise $Q1 \ast+ H2 ==> Q$ is equivalent to $H2 ==> (Q1 \dashv\ast Q)$. By replacing $H2$ with $Q1 \dashv\ast Q$ inside the first premise $H ==> H1 \ast H2$, we obtain the new premise $H ==> H1 \ast (Q1 \dashv\ast Q)$.

This merging of the two entailments leads us to the statement of the “ramified frame rule” shown below.

Lemma triple_ramified_frame : $\forall H1 Q1 t H Q,$

```
triple t H1 Q1 →  
H ==> H1 \ast (Q1 \dashv\ast Q) →  
triple t H Q.
```

Proof using.

```
intros M W. applys triple_conseq_frame (Q1 \dashv\ast Q) M.  
{ applys W. } { applys qwand_cancel. }
```

Qed.

Reciprocally, we can prove that the ramified frame rule entails the consequence-frame rule. Hence, the ramified frame rule has the same expressive power as the consequence-frame rule.

Lemma triple_conseq_frame_of_ramified_frame : $\forall H2 H1 Q1 t H Q,$

```
triple t H1 Q1 →  
H ==> H1 \ast H2 →  
Q1 \ast+ H2 ==> Q →  
triple t H Q.
```

Proof using.

```
intros M WH WQ. applys triple_ramified_frame M.  
xchange WH. xsimpl. rewrite qwand_equiv. applys WQ.
```

Qed.

44.2.6 Ramified Frame Rule in Weakest-Precondition Style

The ramified frame rule, which we have just stated for triples, features a counterpart expressed in weakest-precondition style (**wp**).

In what follows, we present the “wp ramified rule”, named **wp_ramified**. This rule admits a concise statement and subsumes all other structural rules of Separation Logic. Its statement is as follows.

$(\text{wp } t \text{ Q1}) \ast (Q1 \dashv\ast Q2) ==> (\text{wp } t \text{ Q2}).$

To see where this statement comes from, recall from the chapter **WPsem** the rule named **wp_conseq_frame**, which combines the consequence rule and the frame rule for **wp**.

Parameter wp_conseq_frame : $\forall t H Q1 Q2,$

$Q1 \setminus *+ H ==> Q2 \rightarrow$
 $(\text{wp } t \ Q1) \setminus * H ==> (\text{wp } t \ Q2).$

Let us reformulate this rule using a magic wand. The premise $Q1 \setminus *+ H ==> Q2$ can be rewritten as $H ==> (Q1 \setminus -* Q2)$. By replacing H with $Q1 \setminus -* Q2$ in the conclusion of `wp_conseq_frame`, we obtain the ramified rule for `wp`.

Lemma `wp_ramified` : $\forall t \ Q1 \ Q2,$
 $(\text{wp } t \ Q1) \setminus * (Q1 \setminus -* Q2) ==> (\text{wp } t \ Q2).$

Proof using. `intros.` `applys wp_conseq_frame.` `applys qwand_cancel.` `Qed.`

Exercise: 3 stars, standard, especially useful (`wp_conseq_frame_of_wp_ramified`)
Prove that `wp_conseq_frame` is derivable from `wp_ramified`. To that end, prove the statement of `wp_conseq_frame` by using only `wp_ramified`, the characteristic property of the magic wand `qwand_equiv`, and properties of the entailment relation.

Lemma `wp_conseq_frame_of_wp_ramified` : $\forall t \ H \ Q1 \ Q2,$
 $Q1 \setminus *+ H ==> Q2 \rightarrow$
 $(\text{wp } t \ Q1) \setminus * H ==> (\text{wp } t \ Q2).$

Proof using. `Admitted.`

□

The following reformulation of `wp_ramified` can be more practical to exploit in practice, because it applies to any goal of the form $H ==> \text{wp } t \ Q$.

Lemma `wp_ramified_trans` : $\forall t \ H \ Q1 \ Q2,$
 $H ==> (\text{wp } t \ Q1) \setminus * (Q1 \setminus -* Q2) \rightarrow$
 $H ==> (\text{wp } t \ Q2).$

Proof using. `introv M. xchange M.` `applys wp_ramified.` `Qed.`

End WANDDEF.

44.2.7 Automation with `xsimpl` for `hwand` Expressions

One can extend the tactic `xsimpl` to recognize the magic wand, and automatically perform a number of obvious simplifications. This extension is implemented in the file `LibSepSimpl`, which exports the tactic `xsimpl` illustrated in this section.

Module `XSIMPLDEMO`.

`xsimpl` is able to spot a magic wand that cancels out. For example, if an iterated separating conjunction includes both $H2 \setminus -* H3$ and $H2$, then these two heap predicates can be merged, leaving just $H3$.

Lemma `xsimpl_demo_hwand_cancel` : $\forall H1 \ H2 \ H3 \ H4 \ H5,$
 $H1 \setminus * (H2 \setminus -* H3) \setminus * H4 \setminus * H2 ==> H5.$

Proof using. `intros.` `xsimpl.` `Abort.`

xsimpl is able to simplify uncurried magic wands. For example, if an iterated separating conjunction includes $(H1 \setminus * H2 \setminus * H3) \setminus -* H4$ and $H2$, the two predicates can be merged, leaving $(H1 \setminus * H3) \setminus -* H4$.

Lemma xsimpl_demo_hwand_cancel_partial : $\forall H1 H2 H3 H4 H5 H6,$

$$((H1 \setminus * H2 \setminus * H3) \setminus -* H4) \setminus * H5 \setminus * H2 ==> H6.$$

Proof using. intros. *xsimpl*. Abort.

xsimpl automatically applies the introduction rule `himpl_hwand_r` when the right-hand-side, after prior simplification, reduces to just a magic wand. In the example below, $H1$ is first cancelled out from both sides, then $H3$ is moved from the RHS to the LHS.

Lemma xsimpl_demo_himpl_hwand_r : $\forall H1 H2 H3 H4 H5,$

$$H1 \setminus * H2 ==> H1 \setminus * (H3 \setminus -* (H4 \setminus * H5)).$$

Proof using. intros. *xsimpl*. Abort.

xsimpl can iterate a number of simplifications involving different magic wands.

Lemma xsimpl_demo_hwand_iter : $\forall H1 H2 H3 H4 H5,$

$$H1 \setminus * H2 \setminus * ((H1 \setminus * H3) \setminus -* (H4 \setminus -* H5)) \setminus * H4 ==> ((H2 \setminus -* H3) \setminus -* H5).$$

Proof using. intros. *xsimpl*. Qed.

xsimpl is also able to deal with the magic wand for postconditions. In particular, it is able to merge $Q1 \setminus -* Q2$ with $Q1 \vee$, leaving $Q2 \vee$.

Lemma xsimpl_demo_qwand_cancel : $\forall v (Q1 Q2:\text{val} \rightarrow \text{hprop}) H1 H2,$

$$(Q1 \setminus -* Q2) \setminus * H1 \setminus * (Q1 v) ==> H2.$$

Proof using. intros. *xsimpl*. Abort.

End XSIMPLDEMO.

44.2.8 Evaluation of `wpgen` Recursively in Locally Defined Functions

Module WPGENREC.

Implicit Types $vx vf : \text{val}$.

Recall from chapter WPgen the original definition of `wpgen`, that is, before numerous refactoring. It admitted the following shape.

Fixpoint wpgen ($t:\text{trm}$) ($Q:\text{val} \rightarrow \text{hprop}$) : $\text{hprop} := \text{match } t \text{ with } | \text{trm_val } v ==> Q v | \text{trm_fun } x \text{ t1 ==> } Q (\text{val_fun } x \text{ t1}) | \text{trm_fix } f x \text{ t1 ==> } Q (\text{val_fix } f x \text{ t1}) \dots \text{end.}$

This definition of `wpgen t Q` does not recurse inside the body of functions that occur in the argument t . Instead, it treats such locally defined functions just like values.

Not processing local functions does not limit expressiveness, because it is always possible for the user to manually invoke `wpgen` for each locally defined function, during the verification proof.

Nevertheless, it is much more satisfying and much more practical to set up `wpgen` so that it recursively computes the weakest precondition of all the local functions that it encounters during its evaluation.

In what follows, we show how such a version of `wpgen` can be set up. We begin with the case of non-recursive functions of the form `trm_fun x t1`, then generalize the definition to the slightly more complex case of recursive functions of the form `trm_fix f x t1`. In both cases, the function `wpgen` will get recursively invoked on the body `t1` of the function, in a context extended with the appropriate bindings.

The new definition of `wpgen` will take the shape shown below, for well-suited definitions of `wpgen_fun` and `wpgen_fix` yet to be introduced. In the code snippet below, `vx` denotes a value to which the function may be applied, and `vf` denotes the value associated with the function itself, this value being used in particular in the case of recursive calls.

```
Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct match t with | trm_val v => wpgen_val v | trm_fun x t1 => wpgen_fun (fun vx => wpgen ((x,vx)::E) t1) | trm_fix f x t1 => wpgen_fix (fun vf vx => wpgen ((f,vf)::(x,vx)::E) t1) ... end.
```

1. Treatment of Non-Recursive Functions

For simplicity, let us assume for now the substitution context E to be empty, and let us ignore the presence of the predicate `mkstruct`. Our goal task is to provide a definition for `wpgen (trm_fun x t1) Q`, expressed in terms of `wpgen t1`.

Let vf denote the function `val_fun x t1`, which the term `trm_fun x t1` evaluates to. The heap predicate `wpgen (trm_fun x t1) Q` should assert that that the postcondition Q holds of the result value vf , i.e., that $Q\ vf$ should hold.

Rather than specifying that vf is equal to `val_fun x t1` as we were doing previously, we would like to specify that vf denotes a function whose body admits `wpgen t1` as weakest precondition. This information no longer exposes the syntax of the term `t1`, and is nevertheless sufficient for the user to reason about the behavior of the function vf .

To that end, we define the heap predicate `wpgen (trm_fun x t1) Q` to be of the form $\forall vf, \exists [P\ vf] \dashv^* Q\ vf$ for a carefully crafted predicate P that describes the behavior of the function by means of its weakest precondition. P is defined further on.

The universal quantification on vf is intended to hide away from the user the fact that vf actually denotes `val_fun x t1`. It would be correct to replace $\forall vf, \dots$ with `let vf := val_fun x t1 in ...`, yet we are purposely trying to abstract away from the syntax of `t1`, hence the universal quantification of vf .

In the heap predicate $\forall vf, \exists [P\ vf] \dashv^* Q\ vf$, the magic wand features a left-hand side of the form $\exists [P\ vf]$ is intended to provide to the user the knowledge of the weakest precondition of the body `t1` of the function, in such a way that the user is able to derive the specification aimed for the local function vf .

Concretely, the proposition $P\ vf$ should enable the user establishing properties of applications of the form `trm_app vf vx`, where vf denotes the function and vx denotes an argument to which the function is applied.

To figure out the details of the statement of P , it is useful to recall from chapter WPgen the statement of the lemma `triple_app_fun_from_wpgen`, which we used for reasoning about top-level functions. Its statement appears below—variables were renamed to better match the current context.

```

Parameter triple_app_fun_from_wpgen : ∀ vf vx x t1 H' Q',
  vf = val_fun x t1 →
  H' ==> wpgen ((x,vx)::nil) t1 Q' →
  triple (trm_app vf vx) H' Q'.

```

The lemma above enables establishing a triple for an application `trm_app vf vx` with precondition H' and postcondition Q' , from the premise $H' \Rightarrow wgen((x,vx)::nil) t1 Q'$.

It therefore makes sense, in our definition of the predicate `wpgen(trm_fun x t1) Q`, which we said would take the form $\forall vf, \exists [P vf] \dashv^* Q vf$, to define $P vf$ as:

forall vx H' Q', ($H' \Rightarrow wpgen((x,vx)::nil) t1 Q'$) \rightarrow triple (trm_app vf vx) H' Q'

This proposition can be slightly simplified, by using `wp` instead of `triple`, allowing to eliminate H' . We thus define $P vf$ as:

forall vx H', wpgen ((x,vx)::nil) t1 Q' ==> wp (trm_app vf vx) Q'

Overall, the definition of `wpgen E t` is as follows. Note that the occurrence of `nil` is replaced with E to account for the case of a nonempty context.

Fixpoint `wpgen (E:ctx) (t:trm) : formula := mkstruct match t with ... | trm_fun x t1 => fun Q => let P vf := (forall vx H', wpgen ((x,vx)::nil) t1 Q' ==> wp (trm_app vf vx) Q') in \forall vf, \P vf \dashv^* Q vf ... end.`

The actual definition of `wpgen` exploits an intermediate definition called `wpgen_fun`, as shown below:

Fixpoint `wpgen (E:ctx) (t:trm) : formula := mkstruct match t with ... | trm_fun x t1 => wpgen_fun (fun vx => wpgen ((x,vx)::E) t1) ... end.`

where `wpgen_fun` is defined as follows:

```

Definition wpgen_fun (Fof:val → formula) : formula := fun Q ⇒
  ∀ vf, \[∀ vx Q', Fof vx Q' ==> wp (trm_app vf vx) Q'] \dashv^* Q vf.

```

The soundness lemma for this construct follows from the `wp`-style reasoning rule for applications, called `wp_app_fun`, introduced in chapter WPsem. It is not needed to follow at this stage through the details of the proof. (The proof involves lemmas about \forall and about \dashv^* that are stated and proved only further on in this chapter.)

```

Lemma wpgen_fun_sound : ∀ x t1 Fof,
  (forall vx, formula_sound (subst x vx t1) (Fof vx)) →
  formula_sound (trm_fun x t1) (wpgen_fun Fof).

```

Proof using.

```

  introv M. intros Q. unfolds wpgen_fun. applys himpl_hforall_l (val_fun x t1).
  xchange hwand_hpure_l.
  { intros. applys himpl_trans_r. { applys× wp_app_fun. } { applys× M. } }
  { applys wp_fun. }
Qed.

```

When we carry out the proof of soundness for the new version of `wpgen` that features `wpgen_fun`, we obtain the following new proof obligation. (To see it, play the proof of lemma `wpgen_sound`, in file *LibSepDirect.v*.)

```

Lemma wpgen_fun_proof_obligation : ∀ E x t1,

```

$$\begin{aligned}
 (\forall E, \text{formula_sound}(\text{isubst } E t1) (\text{wpgen } E t1)) \rightarrow \\
 \text{formula_sound}(\text{trm_fun } x (\text{isubst}(\text{rem } x E) t1)) \\
 (\text{wpgen_fun}(\text{fun } v \Rightarrow \text{wpgen}((x, v) :: E) t1)).
 \end{aligned}$$

The proof exploits the lemma `wpgen_fun_sound` that we just established, as well as a substitution lemma, the same as the one used to justify the soundness of the `wpgen` of a let-binding.

Proof using.

```

introv IH. applys wpgen_fun_sound.
{ intros vx. rewrite ← isubst_rem. applys IH. }

```

Qed.

2. Treatment of Recursive Functions

The formula produced by `wpgen` for a recursive function `trm_fix f x t1` is almost the same as for a non-recursive function, the main difference being that we need to add a binding in the context to associate the name `f` of the recursive function with the value `vf` that corresponds to the recursive function.

Here again, the heap predicate `wpgen(trm_fun x t1) Q` will be of the form $\forall vf, \exists P [P vf] \dashv^* Q vf$.

To figure out the details of the statement of `P`, recall from WPgen the statement of `triple_app_fix_from_wpgen`, which is useful for reasoning about top-level recursive functions.

Parameter `triple_app_fix_from_wpgen` : $\forall vf vx f x t1 H' Q'$,

$$\begin{aligned}
 vf = \text{val_fix } f x t1 \rightarrow \\
 H' \implies \text{wpgen}((f, vf) :: (x, vx) :: \text{nil}) t1 Q' \rightarrow \\
 \text{triple}(\text{trm_app } vf vx) H' Q'.
 \end{aligned}$$

It therefore makes sense to define `P vf` as:

forall vx H' Q', $(H' \implies \text{wpgen}((f, vf) :: (x, vx) :: \text{nil}) t1 Q') \rightarrow \text{triple}(\text{trm_app } vf vx) H' Q'$

which can be rewritten as:

forall vx H', $\text{wpgen}((f, vf) :: (x, vx) :: \text{nil}) t1 Q' \implies \text{wp}(\text{trm_app } vf vx) Q'$

We thus consider:

Fixpoint `wpgen(E:ctx)(t:trm) : formula := mkstruct match t with | .. | trm_fix f x t1`
 $\Rightarrow \text{wpgen_fix}(\text{fun } vf v \Rightarrow \text{wpgen}((f, vf) :: (x, v) :: E) t1) | .. \text{ end}$
 with the following definition for `wpgen_fix`.

Definition `wpgen_fix(Fof:val → val → formula) : formula := fun Q ⇒`
 $\forall vf, \exists P [P vf] \dashv^* Q vf$.

The soundness lemma for `wpgen_fix` is very similar to that of `wpgen_fun`. Again, it is not needed to follow through the details of this proof.

Lemma `wpgen_fix_sound : ∀ f x t1 Fof,`
 $(\forall vf vx, \text{formula_sound}(\text{subst } x vx (\text{subst } f vf t1)) (Fof vf vx)) \rightarrow$
 $\text{formula_sound}(\text{trm_fix } f x t1) (\text{wpgen_fix } Fof)$.

Proof using.

```
intros M. intros Q. unfolds wpgen_fix.
applys himpl_hforall_l (val_fix f x t1). xchange hwand_hpure_l.
{ intros. applys himpl_trans_r. { applys× wp_app_fix. } { applys× M. } }
{ applys wp_fix. }
```

Qed.

The proof of soundness of `wpgen` involves the following proof obligation to handle the case of recursive functions. (To see it, play the proof of lemma `wpgen_sound`, in file *LibSepDirect.v*.)

```
Lemma wpgen_fix_proof_obligation : ∀ E f x t1,
(∀ E, formula_sound (isubst E t1) (wpgen E t1)) →
formula_sound (trm_fix f x (isubst (rem x (rem f E)) t1))
(wpgen_fix (fun vf vx ⇒ wpgen ((f , vf) :: (x , vx) :: E) t1)).
```

Proof using.

```
intros IH. applys wpgen_fix_sound.
{ intros vf vx. rewrite ← isubst_rem_2. applys IH. }
```

Qed.

Here again, we introduce a piece of notation for `wpgen_fix`. We let *Fix' f x := F* stand for `wpgen_fix (fun f x ⇒ F)`.

```
Notation "'Fix'" f x ':=' F1 := 
((wpgen_fix (fun f x ⇒ F1)))
(at level 69, f ident, x ident, right associativity,
format "[v] 'Fix' f x ':=' F1 ']']").
```

Remark: similarly to *xfun*, one could devise a *xfix* tactic. We omit the details.

3. Final Definition of `wpgen`, with Processing a Local Functions

The final definition of `wpgen` appears below.

```
Fixpoint wpgen (E:ctx) (t:trm) : formula :=
mkstruct match t with
| trm_val v ⇒ wpgen_val v
| trm_var x ⇒ wpgen_var E x
| trm_fun x t1 ⇒ wpgen_fun (fun v ⇒ wpgen ((x , v) :: E) t1)
| trm_fix f x t1 ⇒ wpgen_fix (fun vf v ⇒ wpgen ((f , vf) :: (x , v) :: E) t1)
| trm_if t0 t1 t2 ⇒ wpgen_if t0 (wpgen E t1) (wpgen E t2)
| trm_seq t1 t2 ⇒ wpgen_seq (wpgen E t1) (wpgen E t2)
| trm_let x t1 t2 ⇒ wpgen_let (wpgen E t1) (fun v ⇒ wpgen ((x , v) :: E) t2)
| trm_app t1 t2 ⇒ wp (isubst E t)
end.
```

The full soundness proof appears in file *LibSepDirect*, lemma `wpgen_sound`.

4. Notation and Tactic to Handle `wpgen_fun` and `wpgen_fix`

Like for other auxiliary functions associated with `wpgen`, we introduce a custom notation for `wpgen_fun`. Here, we let $\text{Fun}' x := F$ stand for `wpgen_fun (fun x => F)`.

```
Notation "'Fun" x ':=' F1 :=  
  ((wpgen_fun (fun x => F1)))  
  (at level 69, x ident, right associativity,  
  format "[v] ['] 'Fun" x ':=' F1 ['] ']").
```

Also, like for other language constructs, we introduce a custom tactic for `wpgen_fun`. It is called `xfun`, and helps the user to process a local function definition in the course of a verification script.

The tactic `xfun` can be invoked either with or without providing a specification for the local function.

First, we describe the tactic `xfun S`, where `S` describes the specification of the local function. A typical call is of the form `xfun (fun (f:val) => ∀ ... , triple (f ..) ...)`.

The tactic `xfun S` generates two subgoals. The first one requires the user to establish the specification `S` for the function whose body admits the weakest precondition `Fof`. The second one requires the user to prove that the rest of the program is correct, in a context where the local function can be assumed to satisfy the specification `S`.

The definition of `xfun S` appears next. It is not required to understand the details. An example use case appears further on.

```
Lemma xfun_spec_lemma : ∀ (S:val→Prop) H Q Fof,  
  (∀ vf,  
   (∀ vx H' Q', (H' ==> Fof vx Q') → triple (trm_app vf vx) H' Q') →  
    S vf) →  
  (∀ vf, S vf → (H ==> Q vf)) →  
  H ==> wpgen_fun Fof Q.
```

Proof using.

```
  introv M1 M2. unfold wpgen_fun. xsimpl. intros vf N.  
  applys M2. applys M1. introv K. rewrite ← wp_equiv. xchange K. applys N.
```

Qed.

```
Tactic Notation "xfun" constr(S) :=  
  xseq_xlet_if_needed; xstruct_if_needed; applys xfun_spec_lemma S.
```

Second, we describe the tactic `xfun` without argument. It applies to a goal of the form $H ==> \text{wpgen_fun } Fof \ Q$. The tactic `xfun` simply makes available an hypothesis about the local function. The user may subsequently exploit this hypothesis for reasoning about a call to that function, just like if the code of the function was inlined at its call site. The use of `xfun` without argument is usually relevant only for local functions that are invoked exactly once (as is often the case for functions passed to higher-order iterators). Again, an example use case appears further on.

```
Lemma xfun_nospec_lemma : ∀ H Q Fof,
```

```


$$\begin{aligned}
& (\forall vf, \\
& \quad (\forall vx H' Q', (H' ==> Fof vx Q') \rightarrow \text{triple}(\text{trm\_app} vf vx) H' Q') \rightarrow \\
& \quad (H ==> Q vf)) \rightarrow \\
& H ==> \text{wpgen\_fun } Fof Q.
\end{aligned}$$


```

Proof using.

```

introv M. unfold wpgen_fun. xsimpl. intros vf N. applys M.
introv K. rewrite ← wp_equiv. xchange K. applys N.

```

Qed.

```

Tactic Notation "xfun" :=
xseq_xlet_if_needed; xstruct_if_needed; applys xfun_nospec_lemma.

```

A generalization of *xfun* that handles recursive functions may be defined following exactly the same pattern.

This completes our presentation of a version of *wpgen* that recursively processes the local definition of non-recursive functions. A practical example is presented next.

5. Example Computation of *wpgen* in Presence of a Local Function

In the example that follows, we assume all the set up from WPgen to be reproduced with the definition of *wpgen* that leverages *wpgen_fun* and *wpgen_fix*. This set up is formalized in full in the file *LibSepDirect*.

Import *DemoPrograms*.

Consider the following example program, which involves a local function definition, then two successive calls to that local function.

```

Definition myfun : val :=
<{ fun 'p =>
  let 'f = (fun_ 'u => incr 'p) in
  'f ();
  'f () }>.

```

We first illustrate a call to *xfun* with an explicit specification. Here, the function *f* is specified as incrementing the reference *p*. Observe that the body function of the function *f* is verified only once. The reasoning about the two calls to the function *f* that appears in the code are done with respect to the specification that we provide as argument to *xfun* at the moment of the definition of *f*.

```

Lemma triple_myfun : ∀ (p:loc) (n:int),
  triple (trm_app myfun p)
  (p ~~> n)
  (fun _ => p ~~> (n+2)).

```

Proof using.

```

xwp.
xfun (fun (f:val) => ∀ (m:int),
  triple (f()))

```

```

(p ~~> m)
  (fun _ => p ~~> (m+1))); intros f Hf.
{ intros. applys Hf. clear Hf. xapp. xsimpl. }
xapp. xapp. replace (n+1+1) with (n+2); [|math]. xsimpl.
Qed.

```

We next illustrate a call to *xfun* without argument. The “generic specification” that comes as hypothesis about the local function is a proposition that describes the behavior of the function in terms of the weakest-precondition of its body.

When reasoning about a call to the function, one can invoke this generic specification. The effect is equivalent to that of inlining the code of the function at its call site.

Here, there are two calls to the function. We will thus have to exploit twice the generic specification of *f*, which corresponds to its body *incr p*. We will therefore have to reason twice about the increment function.

```

Lemma triple_myfun' : ∀ (p:loc) (n:int),
  triple (trm_app myfun p)
    (p ~~> n)
    (fun _ => p ~~> (n+2)).

```

Proof using.

```

xwp.
xfun; intros f Hf.
xapp. xapp. xapp. xapp. replace (n+1+1) with (n+2); [|math]. xsimpl.

```

Qed.

End WPGENREC.

44.3 More Details

44.3.1 Benefits of the Ramified Rule over the Consequence-Frame Rule

Earlier on, we sketched an argument claiming that the consequence-frame rule is not very well suited for carrying out proofs in practice, due to issues with working with evars for instantiating the heap predicate *H2* in the rule. Let us come back to this point and describe the issue in depth on a concrete example, and show how the ramified frame rule smoothly handles that same example.

Module WANDBENEFITS.

Import *WandDef*.

Recall the consequence-frame rule.

```

Parameter triple_conseq_frame : ∀ H2 H1 Q1 t H Q,
  triple t H1 Q1 →
  H ==> H1 \* H2 →

```

$Q1 \setminus *+ H2 ==> Q \rightarrow$
 $\text{triple } t H Q.$

One practical caveat with this rule is that we must resolve $H2$, which corresponds to the difference between H and $H1$. In practice, providing $H2$ explicitly is extremely tedious. The alternative is to leave $H2$ as an evar, and count on the fact that the tactic $xsimpl$, when applied to $H ==> H1 \setminus * H2$, will correctly instantiate $H2$.

This approach works in simple cases, but fails in particular in the case where H contains an existential quantifier. For a concrete example, consider the specification of the function ref , which allocates a reference.

```
Parameter triple_ref : ∀ (v:val),
  triple (val_ref v)
  \ []
  (funloc p ⇒ p ~~> v).
```

Assume that wish to derive the following triple, which extends both the precondition and the postcondition of the above specification $triple_ref$ with the heap predicate $\exists l' v', l' ~~> v'$. This predicate describes the existence of some, totally unspecified, reference cell. It is a bit artificial but illustrates well the issue.

```
Lemma triple_ref_extended : ∀ (v:val),
  triple (val_ref v)
  (\exists l' v', l' ~~> v')
  (funloc p ⇒ p ~~> v \* \exists l' v', l' ~~> v').
```

Let us prove that this specification is derivable from the original one, namely $triple_ref$.

Proof using.

```
intros. applys triple_conseq_frame.
{ applys triple_ref. }
{
  xsimpl.
```

Abort.

Now, let us apply the ramified frame rule to carry out the same proof, and observe how the problem does not show up.

```
Lemma triple_ref_extended' : ∀ (v:val),
  triple (val_ref v)
  (\exists l' v', l' ~~> v')
  (funloc p ⇒ p ~~> v \* \exists l' v', l' ~~> v').
```

Proof using.

```
intros. applys triple_ramified_frame.
{ applys triple_ref. }
{
  xsimpl.
  intros l' v'. rewrite qwand_equiv. xsimpl. auto.
}
```

Qed.

End WANDBENEFITS.

44.3.2 Properties of `hwand`

Module WANDPROPERTIES.

Import `WandDef`.

We next present the most important properties of $H1 \setminus-* H2$.

Thereafter, the tactic `xsimpl` is accessible, but in a form that does not provide support for the magic wand.

The actual tactic would trivially solve many of these lemmas, but using it would be cheating because the implementation of `xsimpl` relies on several of these lemmas.

Structural Properties of `hwand`

The operator $H1 \setminus-* H2$ is contravariant in $H1$ and covariant in $H2$, similarly to the implication operator \rightarrow .

Lemma `hwand_himpl` : $\forall H1 H1' H2 H2',$
 $H1' ==> H1 \rightarrow$
 $H2 ==> H2' \rightarrow$
 $(H1 \setminus-* H2) ==> (H1' \setminus-* H2').$

Proof using.

`introv M1 M2. applys himpl_hwand_r. xchange M1.`
`xchange (hwand_cancel H1 H2). applys M2.`

Qed.

Two predicates $H1 \setminus-* H2$ and $H2 \setminus-* H3$ may simplify to $H1 \setminus-* H3$. This simplification is reminiscent of the arithmetic operation $(-H1 + H2) + (-H2 + H3) = (-H1 + H3)$.

Lemma `hwand_trans_elim` : $\forall H1 H2 H3,$
 $(H1 \setminus-* H2) \setminus-* (H2 \setminus-* H3) ==> (H1 \setminus-* H3).$

Proof using.

`intros. applys himpl_hwand_r. xchange (hwand_cancel H1 H2).`

Qed.

The predicate $H \setminus-* H$ holds of the empty heap. Intuitively, one can rewrite 0 as $-H + H$.

Lemma `himpl_hempty_hwand_same` : $\forall H,$
 $\setminus[] ==> (H \setminus-* H).$

Proof using. `intros. apply himpl_hwand_r. xsimpl.` Qed.

Tempting Yet False Properties for `hwand`

The reciprocal entailment $(H \setminus-* H) ==> \setminus[]$ is false, however. For a counterexample, instantiate H as `fun h => True`, or, equivalently, $\exists H', H'$. A singleton heap does satisfy $H \setminus-* H$, although it clearly does not satisfy the empty predicate $\setminus[]$.

```

Lemma himpl_hwand_same_hempty_counterexample : ∀ p v,
  let H := (∃ H', H') in
  (p ~~> v) ==> (H \-* H).
Proof using. intros. subst H. rewrite hwand_equiv. xsimpl. Qed.

```

In technical terms, $H \setminus\ast H$ characterizes the empty heap only in the case where H is “precise”, that is, when it describes a heap of a specific shape. In the above counterexample, H is clearly not precise, because it is satisfied by heaps of any shape. The notion of “preciseness” can be defined formally, yet it is out of the scope of this course.

As another tempting yet false property of the magic wand, consider the reciprocal entailment to the cancellation lemma, that is, $H2 ==> H1 \ast (H1 \setminus\ast H2)$. It does not hold in general.

As counter-example, consider $H2 = \[]$ and $H1 = \![\text{False}]$. We can prove that the empty heap does not satisfies $\![\text{False}] \ast (\![\text{False}] \setminus\ast \[])$.

```

Lemma not_himpl_hwand_r_inv_reciprocal : ∃ H1 H2 ,
  ¬ (H2 ==> H1 \ast (H1 \setminus\ast H2)).

```

Proof using.

```

  ∃ \![\text{False}] \[]. intros N. forwards K: N (Fmap.empty:heap).
  applys hempty_intro. rewrite hstar_hpure_l in K. destruct K. auto.

```

Qed.

More generally, one has to be suspicious of any entailment that introduces wands “out of nowhere”.

The entailment `hwand_trans_elim`: $(H1 \setminus\ast H2) \ast (H2 \setminus\ast H3) ==> (H1 \setminus\ast H3)$ is correct because, intuitively, the left-hand-side captures that $H1 \leq H2$ and that $H1 \leq H3$ for some vaguely defined notion of \leq as “being a subset of”. From that, we can derive $H1 \leq H3$, and justify that the right-hand-side makes sense.

On the contrary, the reciprocal entailment $(H1 \setminus\ast H3) ==> (H1 \setminus\ast H2) \ast (H2 \setminus\ast H3)$ is certainly false, because from $H1 \leq H3$ there is no way to justify $H1 \leq H2$ nor $H2 \leq H3$.

Interaction of `hwand` with `hempty` and `hpure`

The heap predicate $\[] \setminus\ast H$ is equivalent to H .

```

Lemma hwand_hempty_l : ∀ H,
  (\[] \setminus\ast H) = H.

```

Proof using.

```

  intros. unfold hwand. xsimpl.
  { intros H0 M. xchange M. }
  { xsimpl. }

```

Qed.

The lemma above shows that the empty predicate $\[]$ can be removed from the LHS of a magic wand.

More generally, a pure predicate $\setminus[P]$ can be removed from the LHS of a magic wand, as long as P is true. Formally:

Lemma `hwand_hpure_l` : $\forall P H,$

$P \rightarrow$

$(\setminus[P] \setminus-* H) = H.$

Proof using.

introv HP. unfold hwand. xsimpl.

{ *intros H0 M. xchange M. applys HP.* }

{ *xpull. auto.* }

Qed.

Reciprocally, to prove that a heap satisfies $\setminus[P] \setminus-* H$, it suffices to prove that this heap satisfies H under the assumption that P is true. Formally:

Lemma `himpl_hwand_hpure_r` : $\forall H1 H2 P,$

$(P \rightarrow H1 ==> H2) \rightarrow$

$H1 ==> (\setminus[P] \setminus-* H2).$

Proof using. *introv M. applys himpl_hwand_r. xsimpl. applys M.* **Qed.**

Exercise: 2 stars, standard, optional (himpl_hwand_hpure_lr) Prove that $\setminus[P1 \rightarrow P2]$ entails $\setminus[P1] \setminus-* \setminus[P2]$.

Lemma `himpl_hwand_hpure_lr` : $\forall (P1 P2:\text{Prop}),$

$\setminus[P1 \rightarrow P2] ==> (\setminus[P1] \setminus-* \setminus[P2]).$

Proof using. *Admitted.*

□

Interaction of `hwand` with `hstar`

An interesting property is that arguments on the LHS of a magic wand can equivalently be “curried” or “uncurried”, just like a function of type “ $(A * B) \rightarrow C$ ” is equivalent to a function of type “ $A \rightarrow B \rightarrow C$ ”.

The heap predicates $(H1 \setminus* H2) \setminus-* H3$ and $H1 \setminus-* (H2 \setminus-* H3)$ and $H2 \setminus-* (H1 \setminus-* H3)$ are all equivalent. Intuitively, they all describe the predicate $H3$ with the missing pieces $H1$ and $H2$.

The equivalence between the uncurried form $(H1 \setminus* H2) \setminus-* H3$ and the curried form $H1 \setminus-* (H2 \setminus-* H3)$ is formalized by the lemma shown below. The third form, $H2 \setminus-* (H1 \setminus-* H3)$, follows from the commutativity property $H1 \setminus* H2 = H2 \setminus* H1$.

Lemma `hwand_curry_eq` : $\forall H1 H2 H3,$

$(H1 \setminus* H2) \setminus-* H3 = H1 \setminus-* (H2 \setminus-* H3).$

Proof using.

intros. applys himpl_antisym.

{ *apply himpl_hwand_r. apply himpl_hwand_r.* }

xchange (hwand_cancel (H1 \setminus H2) H3). }*

```
{ apply himpl_hwand_r. xchange (hwand_cancel H1 (H2 \-* H3)).
  xchange (hwand_cancel H2 H3). }
```

Qed.

Another interesting property is that the RHS of a magic wand can absorb resources that the magic wand is starred with.

Concretely, from $(H1 \-* H2) \-* H3$, one can get the predicate $H3$ to be absorbed by the $H2$ in the magic wand, yielding $H1 \-* (H2 \-* H3)$.

One way to read this: “if you own $H3$ and, when given $H1$ you own $H2$, then, when given $H1$, you own both $H2$ and $H3$.”

Lemma hstar_hwand : $\forall H1 H2 H3,$
 $(H1 \-* H2) \-* H3 ==> H1 \-* (H2 \-* H3).$

Proof using.

```
intros. applys himpl_hwand_r. xsimpl. xchange (hwand_cancel H1 H2).
```

Qed.

Remark: the reciprocal entailment is false: it is not possible to extract a heap predicate out of the LHS of an entailment. Indeed, that heap predicate might not exist if it is mentioned in the RHS of the magic wand.

Exercise: 1 star, standard, especially useful (himpl_hwand_hstar_same_r) Prove that $H1$ entails $H2 \-* (H2 \-* H1)$.

Lemma himpl_hwand_hstar_same_r : $\forall H1 H2,$
 $H1 ==> (H2 \-* (H2 \-* H1)).$

Proof using. Admitted.

□

Exercise: 2 stars, standard, especially useful (hwand_cancel_part) Prove that $H1 \-* ((H1 \-* H2) \-* H3)$ simplifies to $H2 \-* H3$.

Lemma hwand_cancel_part : $\forall H1 H2 H3,$
 $H1 \-* ((H1 \-* H2) \-* H3) ==> (H2 \-* H3).$

Proof using. Admitted.

□

End WANDPROPERTIES.

44.4 Optional Material

44.4.1 Equivalence Between Alternative Definitions of the Magic Wand

Module HWANDEQUIV.

Implicit Type $op : \text{hprop} \rightarrow \text{hprop} \rightarrow \text{hprop}$.

In what follows we prove the equivalence between the three characterizations of hwand $H1 \text{ H2}$ that we have presented:

1. The definition hwand' , expressed directly in terms of heaps: $\text{fun } h \Rightarrow \forall h', \text{Fmap.disjoint } h \ h' \rightarrow H1 \ h' \rightarrow H2 (h' \setminus\!\! \cup h)$
2. The definition hwand , expressed in terms of existing operators: $\exists H0, H0 \ \forall^* \ [(H1 \ \forall^* H0) ==> H2]$
3. The characterization via the equivalence hwand_equiv : $\forall H0 \ H1 \ H2, (H0 ==> H1 \ \forall^* H2) \leftrightarrow (H1 \ \forall^* H0 ==> H2)$.
4. The characterization via the pair of the introduction rule himpl_hwand_r and the elimination rule hwand_cancel .

To prove the 4-way equivalence, we first prove the equivalence between (1) and (2), then prove the equivalence between (2) and (3), and finally the equivalence between (3) and (4).

Definition $\text{hwand_characterization_1} (op:\text{hprop} \rightarrow \text{hprop} \rightarrow \text{hprop}) :=$

$$op = (\text{fun } H1 \ H2 \Rightarrow (\text{fun } h \Rightarrow \forall h', \text{Fmap.disjoint } h \ h' \rightarrow H1 \ h' \rightarrow H2 (h' \setminus\!\! \cup h))).$$

Definition $\text{hwand_characterization_2} (op:\text{hprop} \rightarrow \text{hprop} \rightarrow \text{hprop}) :=$

$$op = (\text{fun } H1 \ H2 \Rightarrow \exists H0, H0 \ \forall^* \ [H1 \ \forall^* H0 ==> H2]).$$

Definition $\text{hwand_characterization_3} (op:\text{hprop} \rightarrow \text{hprop} \rightarrow \text{hprop}) :=$

$$\forall H0 \ H1 \ H2, (H0 ==> op \ H1 \ H2) \leftrightarrow (H1 \ \forall^* H0 ==> H2).$$

Definition $\text{hwand_characterization_4} (op:\text{hprop} \rightarrow \text{hprop} \rightarrow \text{hprop}) :=$

$$(\forall H0 \ H1 \ H2, (H1 \ \forall^* H0 ==> H2) \rightarrow (H0 ==> op \ H1 \ H2)) \\ \wedge (\forall H1 \ H2, (H1 \ \forall^* (op \ H1 \ H2) ==> H2)).$$

Lemma $\text{hwand_characterization_1_eq_2} :$

$$\text{hwand_characterization_1} = \text{hwand_characterization_2}.$$

Proof using.

```

applys pred_ext_1. intros op.
unfold hwand_characterization_1, hwand_characterization_2.
asserts K: (\forall A B, A = B → (op = A ↔ op = B)).
{ intros. iff; subst×. } apply K; clear K.
apply pred_ext_3. intros H1 H2 h. iff M.
{ ∃ (=h). rewrite hstar_hpure_r. split.
  { auto. }
  { intros h3 K3. rewrite hstar_comm in K3.
    destruct K3 as (h1&h2&K1&K2&D&U). subst h1 h3.
    rewrites (» union_comm_of_disjoint D). applys M D K2. } }
{
  intros h1 D K1. destruct M as (H0&M).
  destruct M as (h0&h2&K0&K2&D'&U).
  lets (N&E): hpure_inv (rm K2). subst h h2.
  rewrite Fmap.union_empty_r in *.
}

```

```

    applys N. applys hstar_intro K1 K0. applys disjoint_sym D. }
Qed.
```

Lemma hwand_characterization_2_eq_3 :
 $\text{hwand_characterization_2} = \text{hwand_characterization_3}$.

Proof using.

```

applys pred_ext_1. intros op.
unfold hwand_characterization_2, hwand_characterization_3. iff K.
{ subst. intros. iff M.
  { xchange M. intros H3 N. xchange N. }
  { xsimpl H0. xchange M. } }
{ apply fun_ext_2. intros H1 H2. apply himpl_antisym.
  { lets (M&_): (K (op H1 H2) H1 H2). xsimpl (op H1 H2).
    applys M. applys himpl_refl. }
  { xsimpl. intros H0 M. rewrite K. applys M. } }
```

Qed.

Lemma hwand_characterization_3_eq_4 :
 $\text{hwand_characterization_3} = \text{hwand_characterization_4}$.

Proof using.

```

applys pred_ext_1. intros op.
unfold hwand_characterization_3, hwand_characterization_4. iff K.
{ split.
  { introv M. apply ← K. apply M. }
  { intros. apply K. auto. } }
{ destruct K as (K1&K2). intros. split.
  { introv M. xchange M. xchange (K2 H1 H2). }
  { introv M. applys K1. applys M. } }
```

Qed.

End HWANDEQUIV.

44.4.2 Operator **hforall**

Module NEWQWAND.
Export WandDef.

In the beginning of this chapter, we defined **qwand** following the pattern of **hwand**, as $\exists H0, H0 \setminus^* \setminus [Q1 \setminus^+ H0 ==> Q2]$. An alternative approach consists of defining **qwand** in terms of **hwand**.

This alternative definition involves the universal quantifier for heap predicates, written $\forall x, H$. The universal quantifier is the counterpart of the existential quantifier $\exists x, H$.

Using the \forall quantifier, we may define $Q1 \setminus^* Q2$ as the heap predicate $\forall v, (Q1 v) \setminus^* (Q2 v)$.

Let us first formalize the definition of the universal quantifier on **hprop**. Technically, a

heap predicate of the form $\forall x, H$ stands for `hforall (fun x => H)`, where the definition of `hforall` follows the exact same pattern as for `hexists`. The definition shown below is somewhat technical—details may be safely skipped over.

Definition `hforall (A : Type) (J : A → hprop) : hprop :=`
`fun h => ∀ x, J x h.`

Notation "`'forall' x1 .. xn , H`" :=
`(hforall (fun x1 => .. (hforall (fun xn => H)) ..))`
`(at level 39, x1 binder, H at level 50, right associativity,`
`format "'['forall' '/ 'x1 .. xn , '/ 'H ']'"`.

The introduction and elimination rule for `hforall` are as follows.

Lemma `hforall_intro : ∀ A (J:A→hprop) h,`
`(∀ x, J x h) →`
`(hforall J) h.`

Proof using. `introv M. applys× M. Qed.`

Lemma `hforall_inv : ∀ A (J:A→hprop) h,`
`(hforall J) h →`
`∀ x, J x h.`

Proof using. `introv M. applys× M. Qed.`

The introduction rule in an entailment for \forall appears below. To prove that a heap satisfies $\forall x, J x$, one must show that, for any x , this heap satisfies $J x$.

Lemma `himpl_hforall_r : ∀ A (J:A→hprop) H,`
`(∀ x, H ==> J x) →`
`H ==> (\forall x, J x).`

Proof using. `introv M. intros h K x. apply¬ M. Qed.`

The elimination rule in an entailment for \forall appears below. Assuming a heap satisfies $\forall x, J x$, one can derive that the same heap satisfies $J v$ for any desired value v .

Lemma `hforall_specialize : ∀ A (v:A) (J:A→hprop),`
`(\forall x, J x) ==> (J v).`

Proof using. `intros. intros h K. apply× K. Qed.`

This last lemma can equivalently be formulated in the following way, which makes it easier to apply in some cases.

Lemma `himpl_hforall_l : ∀ A (v:A) (J:A→hprop) H,`
`J v ==> H →`
`(\forall x, J x) ==> H.`

Proof using. `introv M. applys himpl_trans M. applys hforall_specialize. Qed.`

Universal quantifiers that appear in the precondition of a triple may be specialized, just like those in the left-hand side of an entailment.

Lemma `triple_hforall : ∀ A (v:A) t (J:A→hprop) Q,`
`triple t (J v) Q →`

`triple t ($\forall x, J x$) Q.`

`Proof.`

```
introv M. applys triple_conseq M.
{ applys hforall_specialize. }
{ applys qimpl_refl. }
```

`Qed.`

44.4.3 Alternative Definition of `qwand`

`Declare Scope qwand_scope.`

`Open Scope qwand_scope.`

We are now ready to state the alternative definition of $Q1 \setminus\ast Q2$, as the heap predicate $\forall v, (Q1 v) \setminus\ast (Q2 v)$.

`Definition qwand (Q1 Q2:val → hprop) : hprop :=`
 $\forall v, (Q1 v) \setminus\ast (Q2 v).$

`Notation "Q1 \setminus\ast Q2" := (qwand Q1 Q2) (at level 43) : qwand_scope.`

Let us establish the properties of the new definition of `qwand`.

We begin by the specialization lemma, which asserts that $Q1 \setminus\ast Q2$ can be specialized to $(Q1 v) \setminus\ast (Q2 v)$ for any value v . This result is a direct consequence of the corresponding specialization property of \forall .

`Lemma qwand_specialize : ∀ (v:val) (Q1 Q2:val → hprop),`
 $(Q1 \setminus\ast Q2) ==> (Q1 v \setminus\ast Q2 v).$

`Proof using.`

`intros. unfold qwand. applys himpl_hforall_l v. xsimpl.`

`Qed.`

We next prove the equivalence rule.

`Lemma qwand_equiv : ∀ H Q1 Q2,`
 $H ==> (Q1 \setminus\ast Q2) \leftrightarrow (Q1 \setminus\ast H) ==> Q2.$

`Proof using.`

```
intros. iff M.
{ intros x. xchange M. xchange (qwand_specialize x).
  xchange (hwand_cancel (Q1 x)). }
{ applys himpl_hforall_r. intros x. applys himpl_hwand_r.
  xchange (M x). }
```

`Qed.`

The cancellation rule follows.

`Lemma qwand_cancel : ∀ Q1 Q2,`
 $Q1 \setminus\ast (Q1 \setminus\ast Q2) ==> Q2.$

`Proof using. intros. rewrite ← qwand_equiv. applys qimpl_refl. Qed.`

Like $H1 \setminus-* H2$, the operation $Q1 \setminus-* Q2$ is contravariant in $Q1$ and covariant in $Q2$.

Lemma `qwand_himpl` : $\forall Q1 Q1' Q2 Q2',$

$$\begin{aligned} Q1' &\implies Q1 \rightarrow \\ Q2 &\implies Q2' \rightarrow \\ (Q1 \setminus-* Q2) &\implies (Q1' \setminus-* Q2'). \end{aligned}$$

Proof using.

```
intros M1 M2. rewrite qwand_equiv. intros x.
xchange (qwand_specialize x). xchange M1.
xchange (hwand_cancel (Q1 x)). xchange M2.
```

Qed.

Like $H1 \setminus-* H2$, the operation $Q1 \setminus-* Q2$ can absorb in its RHS resources to which it is starred.

Lemma `hstar_qwand` : $\forall Q1 Q2 H,$
 $(Q1 \setminus-* Q2) \ast H \implies Q1 \setminus-* (Q2 \ast H).$

Proof using.

```
intros. rewrite qwand_equiv. xchange (@qwand_cancel Q1).
```

Qed.

Exercise: 1 star, standard, especially useful (himpl_qwand_hstar_same_r) Prove that H entails $Q \setminus-* (Q \ast H)$.

Lemma `himpl_qwand_hstar_same_r` : $\forall H Q,$
 $H \implies Q \setminus-* (Q \ast H).$

Proof using. *Admitted.*

□

Exercise: 2 stars, standard, optional (qwand_cancel_part) Prove that $H \ast ((Q1 \ast H) \setminus-* Q2)$ simplifies to $Q1 \setminus-* Q2$. Hint: use *xchange*.

Lemma `qwand_cancel_part` : $\forall H Q1 Q2,$
 $H \ast ((Q1 \ast H) \setminus-* Q2) \implies (Q1 \setminus-* Q2).$

Proof using. *Admitted.*

□

44.4.4 Equivalence between Alternative Definitions of the Magic Wand

for Postconditions

Module `QWANDEQUIV`.

Implicit Type `op` : $(\text{val} \rightarrow \text{hprop}) \rightarrow (\text{val} \rightarrow \text{hprop}) \rightarrow \text{hprop}$.

In what follows we prove the equivalence between five equivalent characterizations of *qwand* $H1 H2$:

1. The definition expressed directly in terms of heaps: $\text{fun } h \Rightarrow \forall v h', \text{Fmap.disjoint } h h' \rightarrow Q1 \vee h' \rightarrow Q2 \vee (h \setminus u h')$
2. The definition `qwand`, expressed in terms of existing operators: $\exists H0, H0 \setminus^* \llbracket (Q1 \setminus^*+ H0) ==> Q2 \rrbracket$
3. The definition expressed using the universal quantifier: $\forall v, (Q1 \vee) \setminus^* (Q2 \vee)$
4. The characterization via the equivalence `hwand_equiv`: $\forall H0 H1 H2, (H0 ==> H1 \setminus^* H2) \leftrightarrow (H1 \setminus^* H0 ==> H2)$.
5. The characterization via the pair of the introduction rule `himpl_qwand_r` and the elimination rule `qwand_cancel`.

The proof are essentially identical to the equivalence proofs for `hwand`, except for definition (3), which is specific to `qwand`.

Definition `qwand_characterization_1 op` :=

$$op = (\text{fun } Q1 Q2 \Rightarrow (\text{fun } h \Rightarrow \forall v h', \text{Fmap.disjoint } h h' \rightarrow Q1 \vee h' \rightarrow Q2 \vee (h \setminus u h'))).$$

Definition `qwand_characterization_2 op` :=

$$op = (\text{fun } Q1 Q2 \Rightarrow \exists H0, H0 \setminus^* \llbracket Q1 \setminus^*+ H0 ==> Q2 \rrbracket).$$

Definition `qwand_characterization_3 op` :=

$$op = (\text{fun } Q1 Q2 \Rightarrow \forall v, (Q1 \vee) \setminus^* (Q2 \vee)).$$

Definition `qwand_characterization_4 op` :=

$$\forall H0 Q1 Q2, (H0 ==> op Q1 Q2) \leftrightarrow (Q1 \setminus^*+ H0 ==> Q2).$$

Definition `qwand_characterization_5 op` :=

$$\begin{aligned} & (\forall H0 Q1 Q2, (Q1 \setminus^*+ H0 ==> Q2) \rightarrow (H0 ==> op Q1 Q2)) \\ & \wedge (\forall Q1 Q2, (Q1 \setminus^* (op Q1 Q2) ==> Q2)). \end{aligned}$$

Lemma `hwand_characterization_1_eq_2` :

$$\text{qwand_characterization_1} = \text{qwand_characterization_2}.$$

Proof using.

```

applys pred_ext_1. intros op.
unfold qwand_characterization_1, qwand_characterization_2.
asserts K: (\forall A B, A = B → (op = A ↔ op = B)).
{ intros. iff; subst×. } apply K; clear K.
apply pred_ext_3. intros Q1 Q2 h. iff M.
{ ∃ (=h). rewrite hstar_hpure_r. split.
  { auto. }
  { intros v h3 K3. rewrite hstar_comm in K3.
    destruct K3 as (h1&h2&K1&K2&D&U). subst h1 h3. applys M D K2. } }
{ intros v h1 D K1. destruct M as (H0&M).
  destruct M as (h0&h2&K0&K2&D'&U).
  lets (N&E): hpure_inv (rm K2). subst h h2.
  rewrite Fmap.union_empty_r in *.
  applys N. rewrite hstar_comm. applys hstar_intro K0 K1 D. }

```

Qed.

```

Lemma qwand_characterization_2_eq_3 :
  qwand_characterization_2 = qwand_characterization_3.

```

Proof using.

```

  applys pred_ext_1. intros op.
  unfold qwand_characterization_2, qwand_characterization_3.
  asserts K: ( $\forall A B, A = B \rightarrow (op = A \leftrightarrow op = B)$ ).
  { intros. iff; subst x. } apply K; clear K.
  apply fun_ext_2. intros Q1 Q2. apply himpl_antisym.
  { xpull. intros H0 M. applys himpl_hforall_r. intros v.
    rewrite hwand_equiv. xchange M. }
  { xsimpl (qwand Q1 Q2). applys qwand_cancel. }

```

Qed.

```

Lemma qwand_characterization_2_eq_4 :
  qwand_characterization_2 = qwand_characterization_4.

```

Proof using.

```

  applys pred_ext_1. intros op.
  unfold qwand_characterization_2, qwand_characterization_4. iff K.
  { subst. intros. iff M.
    { xchange M. intros v H3 N. xchange N. }
    { xsimpl H0. xchange M. } }
  { apply fun_ext_2. intros H1 H2. apply himpl_antisym.
    { lets (M&_): (K (op H1 H2) H1 H2). xsimpl (op H1 H2).
      applys M. applys himpl_refl. }
    { xsimpl. intros H0 M. rewrite K. applys M. } }

```

Qed.

```

Lemma qwand_characterization_4_eq_5 :
  qwand_characterization_4 = qwand_characterization_5.

```

Proof using.

```

  applys pred_ext_1. intros op.
  unfold qwand_characterization_4, qwand_characterization_5. iff K.
  { split.
    { introv M. apply  $\leftarrow K$ . apply M. }
    { intros. apply K. auto. } }
  { destruct K as (K1&K2). intros. split.
    { introv M. xchange M. xchange (K2 Q1 Q2). }
    { introv M. applys K1. applys M. } }

```

Qed.

End QWANDEQUIV.

44.4.5 Simplified Definition of `mkstruct`

The definition of `mkstruct` can be simplified using the magic wand operator for postconditions.

Recall the definition of `mkstruct` from chapter WPgen.

```
Definition mkstruct' (F:formula) : formula :=
  fun (Q:val → hprop) ⇒ ∃ Q1 H, F Q1 ∗ H ∗ [Q1 ∗+ H ==> Q].
```

Observe that the fragment $\exists H, H ∗ [Q1 ∗+ H ==> Q]$ is equivalent to $Q1 ∗ Q$. This observation leads to the following more concise reformulation of the definition of `mkstruct`.

```
Definition mkstruct (F:formula) : formula :=
  fun Q ⇒ ∃ Q1, F Q1 ∗ (Q1 ∗ Q).
```

Let us prove, for this revised definition, that `mkstruct` satisfies the three required properties (recall WPgen): `mkstruct_erase`, `mkstruct_frame`, and `mkstruct_conseq`. In these proofs, we assume the revised definition of `qwand` expressed in terms of `hwand` and `hforall`.

```
Lemma mkstruct_erase : ∀ F Q,
  F Q ==> mkstruct F Q.
```

Proof using.

```
intros. unfold mkstruct. xsimpl Q. rewrite qwand_equiv. xsimpl.
```

Qed.

```
Lemma mkstruct_conseq : ∀ F Q1 Q2,
  Q1 ==> Q2 →
  mkstruct F Q1 ==> mkstruct F Q2.
```

Proof using.

```
introt WQ. unfold mkstruct. xpull. intros Q'. xsimpl Q'.
  rewrite qwand_equiv. xchange qwand_cancel. xchange WQ.
```

Qed.

```
Lemma mkstruct_frame : ∀ F H Q,
  (mkstruct F Q) ∗ H ==> mkstruct F (Q ∗+ H).
```

Proof using.

```
intros. unfold mkstruct. xpull. intros Q'. xsimpl Q'.
  rewrite qwand_equiv. xchange qwand_cancel.
```

Qed.

End NEWQWAND.

44.4.6 Texan Triples

Module TEXANTRIPLES.

In this section, we assume a version of `xsimpl` that handles the magic wand. Note that `hforall` and other operators are set “Opaque”, their definitions cannot be unfolded.

Implicit Types $v w : \text{val}$.

Implicit Types $p : \text{loc}$.

1. Example of Texan Triples

In this section, we show that specification triples can be presented in a different style using weakest preconditions.

Consider for example the specification triple for allocation.

```
Parameter triple_ref : ∀ v,
  triple (val_ref v)
  \[]
  (funloc p ⇒ p ∼> v).
```

This specification can be equivalently reformulated in the following form.

```
Parameter wp_ref : ∀ Q v,
  \[] \* (\forall p, p ∼> v \-* Q (val_loc p)) ==> wp (val_ref v) Q.
```

Above, we purposely left the empty heap predicate to the front to indicate where the precondition, if it were not empty, would go in the reformulation.

In what follows, we describe the chain of transformation that can take us from the triple form to the `wp` form, and establish the reciprocal. We then formalize the general pattern for translating a triple into a “texan triple” (i.e., the `wp`-based specification).

By replacing `triple t H Q` with $H ==> \text{wp } t \text{ } Q$, the specification `triple_ref` can be reformulated as follows.

```
Lemma wp_ref_0 : ∀ v,
  \[] ==> wp (val_ref v) (funloc p ⇒ p ∼> v).
```

`Proof using.` `intros.` `rewrite wp_equiv.` `applys triple_ref.` `Qed.`

We wish to cast the RHS in the form $\text{wp } (\text{val_ref } v) \text{ } Q$ for an abstract variable Q . To that end, we reformulate the above statement by including a magic wand relating the current postcondition, which is $(\text{funloc } p \Rightarrow p \sim> v)$, and Q .

```
Lemma wp_ref_1 : ∀ Q v,
  ((funloc p ⇒ p ∼> v) \-* Q) ==> wp (val_ref v) Q.
```

`Proof using.` `intros.` `xchange (wp_ref_0 v).` `applys wp_ramified.` `Qed.`

This statement can be made slightly more readable by unfolding the definition of the magic wand for postconditions, as shown next.

```
Lemma wp_ref_2 : ∀ Q v,
  (\forall r, (\exists p, \[r = val_loc p] \* p ∼> v) \-* Q r)
  ==> wp (val_ref v) Q.
```

`Proof using.` `intros.` `applys himpl_trans wp_ref_1.` `xsimpl.` `Qed.`

Interestingly, the variable `r`, which is equal to `val_loc p`, can now be substituted away, further increasing readability. We obtain the specification of `val_ref` in “Texan triple style”.

```
Lemma wp_ref_3 : ∀ Q v,
  (\forall p, (p ∼> v) \-* Q (val_loc p)) ==> wp (val_ref v) Q.
```

`Proof using.`

`intros.` `applys himpl_trans wp_ref_2.` `xsimpl.` `intros ? p →.`

xchange (*hforall_specialize p*).
 Qed.

2. The General Pattern

In practice, specification triples can (pretty much) all be casted in the form: *triple t H (fun r => ∃ x1 x2, \[r = v] * H'*. In such a specification:

- the value *v* may depend on the *xi*,
- the heap predicate *H'* may depend on *r* and the *xi*,
- the number of existentials *xi* may vary, possibly be zero,
- the equality *\r = v* may be removed if no pure fact is needed about *r*.

Such a specification triple of the form *triple t H (fun r => ∃ x1 x2, \[r = v] * H'* can be be reformulated as the Texan triple: *(\forall x1 x2, H \-* Q v) ==> wp t Q*.

We next formalize the equivalence between the two presentations, for the specific case where the specification involves a single auxiliary variable, called *x*. The statement below makes it explicit that *v* may depend on *x*, and that *H* may depend on *r* and *x*.

Lemma texan_triple_equiv : $\forall t H A (Hof:\text{val} \rightarrow A \rightarrow \text{hprop}) (vof:A \rightarrow \text{val}),$
 $(\text{triple } t H (\text{fun } r \Rightarrow \exists x, \lceil r = vof x \rceil * Hof r x))$
 $\leftrightarrow (\forall Q, H * (\forall x, Hof (vof x) x \-* Q (vof x)) ==> wp t Q).$

Proof using.

```
intros. rewrite ← wp_equiv. iff M.
{ intros Q. xchange M. applys wp_ramified_trans.
  xsimpl. intros r x →.
  xchange (hforall_specialize x). }
{ applys himpl_trans M. xsimpl. }
```

Qed.

3. Other Examples

Section WpSpecRef.

The wp-style specification of *ref*, *get* and *set* follow.

Lemma wp_get : $\forall v p Q,$
 $(p \rightsquigarrow v) * (p \rightsquigarrow v \-* Q v) ==> wp (\text{val_get } p) Q.$

Proof using.

```
intros. rewrite wp_equiv. applys triple_conseq_frame.
{ applys triple_get. } { applys himpl_refl. } { xsimpl. intros ? →. auto. }
```

Qed.

Lemma wp_set : $\forall v w p Q,$

$(p \rightsquigarrow v) \text{ \textbackslash *} (\forall r, p \rightsquigarrow w \text{ \textbackslash }-* Q r) ==> \text{wp} (\text{val_set } p w) Q.$

Proof using.

```
intros. rewrite wp_equiv. applys triple_conseq_frame.  
{ applys triple_set. } { applys himpl_refl. }  
{ intros r. xchange (hforall_specialize r). }
```

Qed.

Lemma wp_free : $\forall v p Q,$

$(p \rightsquigarrow v) \text{ \textbackslash *} (\forall r, Q r) ==> \text{wp} (\text{val_free } p) Q.$

Proof using.

```
intros. rewrite wp_equiv. applys triple_conseq_frame.  
{ applys triple_free. } { applys himpl_refl. }  
{ intros r. xchange (hforall_specialize r). }
```

Qed.

Alternatively, we can advertize that `set` and `free` output the unit value.

Parameter `triple_set'` : $\forall w p v,$

```
triple (val_set p w)  
(p \rightsquigarrow v)  
(fun r => \[r = val_unit] \text{ \textbackslash *} p \rightsquigarrow w).
```

Parameter `triple_free'` : $\forall p v,$

```
triple (val_free p)  
(p \rightsquigarrow v)  
(fun r => \[r = val_unit]).
```

Lemma `wp_set'` : $\forall v w p Q,$

$(p \rightsquigarrow v) \text{ \textbackslash *} (p \rightsquigarrow w \text{ \textbackslash }-* Q \text{ val_unit}) ==> \text{wp} (\text{val_set } p w) Q.$

Proof using.

```
intros. rewrite wp_equiv. applys triple_conseq_frame.  
{ applys triple_set'. }  
{ applys himpl_refl. }  
{ xsimpl. intros ? => auto. }
```

Qed.

Lemma `wp_free'` : $\forall v w p Q,$

$(p \rightsquigarrow v) \text{ \textbackslash *} (Q \text{ val_unit}) ==> \text{wp} (\text{val_free } p) Q.$

Proof using.

```
intros. rewrite wp_equiv. applys triple_conseq_frame.  
{ applys triple_free'. }  
{ applys himpl_refl. }  
{ xsimpl. intros ? => auto. }
```

Qed.

End WpSpecRef.

4. Equivalent expressiveness

Let's show that the specification `wp_ref_3` is as strong as the original specification `triple_ref`.

```
Lemma triple_ref_of_wp_ref_3 : ∀ v,
  triple (val_ref v)
  \[]
  (funloc p ⇒ p ∼> v).
```

Proof using.

```
intros. rewrite ← wp_equiv.
applys himpl_trans; [] applys wp_ref_3 ].
xsimpl ×.
```

Qed.

Likewise for the other three operations: the triple-based specification is derivable from the wp-based specification.

```
Lemma triple_get_of_wp_get : ∀ v p,
  triple (val_get p)
  (p ∼> v)
  (fun r ⇒ \[r = v] \* (p ∼> v)).
```

Proof using.

```
intros. rewrite ← wp_equiv.
applys himpl_trans; [] applys wp_get ].
xsimpl ×.
```

Qed.

```
Lemma triple_set : ∀ w p v,
  triple (val_set (val_loc p) v)
  (p ∼> w)
  (fun _ ⇒ p ∼> v).
```

Proof using.

```
intros. rewrite ← wp_equiv.
applys himpl_trans; [] applys wp_set ].
xsimpl ×.
```

Qed.

```
Lemma triple_free : ∀ p v,
  triple (val_free (val_loc p))
  (p ∼> v)
  (fun _ ⇒ \[]).
```

Proof using.

```
intros. rewrite ← wp_equiv.
applys himpl_trans; [] applys wp_free ].
xsimpl ×.
```

Qed.

5. Exercise

Let us put to practice the use of a Texan triple on a different example. Recall the function *incr* and its specification (from *Hprop.v*).

Parameter *incr* : **val**.

```
Parameter triple_incr : ∀ (p:loc) (n:int),
  triple (incr p)
  (p ~~> n)
  (fun v ⇒ \[v = val_unit] \* (p ~~> (n+1))).
```

Exercise: 3 stars, standard, especially useful (*wp_incr*) State a Texan triple for *incr* as a lemma called *wp_incr*, then prove this lemma from *triple_incr*.

Hint: the proof is a bit easier by first turning the *wp* into a *triple* and then reasoning about triples, compared to working on the *wp* form.

□

End TEXANTRIPLES.

44.4.7 Direct Proof of **wp_ramified** Directly from Hoare Triples

Module WPFROMHOARE.

Import *NewQwand*.

Recall from the last section of the chapter **WPsem** that it can be interesting to define *wp*-style rules directly from the **hoare** rules, so as to bypass the statements and proofs of rules for triples.

In the first part of this chapter, we proved that the rule **wp_ramified** is derivable from the consequence-frame rule for triples. Let us now show how to prove the rule **wp_ramified** directly from the rules of Hoare logic.

Lemma *wp_ramified* : ∀ t Q1 Q2,

$$(\text{wp } t Q1) \text{ * } (Q1 \dashv\ast Q2) \implies (\text{wp } t Q2).$$

Proof using.

```
intros. unfold wp. xpull. intros H M.
xsimpl (H \* (Q1 \dashv\ast Q2)). intros H'.
applys hoare_conseq M. { xsimpl. }
intros r. xchange (qwand_specialize r). xsimpl.
rewrite hstar_comm. applys hwand_cancel.
```

Qed.

End WPFROMHOARE.

44.4.8 Conjunction and Disjunction Operators on `hprop`

The disjunction and the (non-separating) conjunction are two other Separation Logic operators. They are not so useful in practice, because they can be trivially encoded using Coq conditional construct, or using Coq pattern matching. Nevertheless, these two operators can prove useful in specific contexts. We present them also for the sake of completeness.

`Module CONJDISJ.`

`Import NewQwand.`

Definition of `hor`

The heap predicate `hor H1 H2` lifts the disjunction operator $P1 \vee P2$ from `Prop` to `hprop`.

Concretely, the heap predicate `hor H1 H2` describes a heap that satisfies $H1$ or satisfies $H2$ (possibly both).

The heap predicate `hor` admits a direct definition as a function over heaps.

Definition `hor'` ($H1\ H2 : \text{hprop}$) : `hprop` :=

`fun h => H1 h \vee H2 h.`

An alternative definition leverages the \exists quantifier. The definition, shown below, reads as follows: “there exists an unspecified boolean value `b` such that if `b` is true then $H1$ holds, else if `b` is false then $H2$ holds”.

The benefits of this definition is that the proof of its properties can be established without manipulating heaps explicitly.

Definition `hor` ($H1\ H2 : \text{hprop}$) : `hprop` :=

`$\exists (b:\text{bool})$, if b then $H1$ else $H2$.`

Exercise: 3 stars, standard, optional (`hor_eq_hor'`) Prove the equivalence of the definitions `hor` and `hor'`.

Lemma `hor_eq_hor'` :

`hor = hor'.`

Proof using. *Admitted.*

□

The introduction and elimination rules for `hor` are as follows.

- If $H1$ holds, then “ $H1$ or $H2$ ” holds.
- Symmetrically, if $H2$ holds, then “ $H1$ or $H2$ ” holds.
- Reciprocally, if “ $H1$ or $H2$ ” holds, then one can perform a case analysis on whether it is $H1$ or $H2$ that holds. Concretely, to show that “ $H1$ or $H2$ ” entails $H3$, one must show both that $H1$ entails $H3$ and that $H2$ entails $H3$.

Lemma `himpl_hor_r_r` : $\forall H1 H2,$

$H1 \Rightarrow\! hor H1 H2.$

Proof using. `intros.` *unfolds* `hor`. $\exists \times \text{true}$. `Qed.`

Lemma `himpl_hor_r_l` : $\forall H1 H2,$

$H2 \Rightarrow\! hor H1 H2.$

Proof using. `intros.` *unfolds* `hor`. $\exists \times \text{false}$. `Qed.`

In practice, these two rules are easier to exploit when combined with a transitivity step.

Lemma `himpl_hor_r_r_trans` : $\forall H1 H2 H3,$

$H3 \Rightarrow\! H1 \rightarrow$

$H3 \Rightarrow\! hor H1 H2.$

Proof using. `introv W.` *applys* `himpl_trans` *W.* *applys* `himpl_hor_r_r`. `Qed.`

Lemma `himpl_hor_r_l_trans` : $\forall H1 H2 H3,$

$H3 \Rightarrow\! H2 \rightarrow$

$H3 \Rightarrow\! hor H1 H2.$

Proof using. `introv W.` *applys* `himpl_trans` *W.* *applys* `himpl_hor_r_l`. `Qed.`

The elimination rule is stated as follows.

Lemma `himpl_hor_l` : $\forall H1 H2 H3,$

$H1 \Rightarrow\! H3 \rightarrow$

$H2 \Rightarrow\! H3 \rightarrow$

$hor H1 H2 \Rightarrow\! H3.$

Proof using.

`introv M1 M2.` *unfolds* `hor`. *applys* `himpl_hexists_l`. `intros b.` *case_if* \times .

`Qed.`

The operator `hor` is commutative. To establish this property, it is handy to exploit the following lemma, called `if_neg`, which swaps the two branches of a conditional by negating the boolean condition.

Lemma `if_neg` : $\forall (b:\text{bool}) A (X Y:A),$

$(\text{if } b \text{ then } X \text{ else } Y) = (\text{if neg } b \text{ then } Y \text{ else } X).$

Proof using. `intros.` *case_if* \times . `Qed.`

Exercise: 2 stars, standard, especially useful (`hor_comm`) Prove that `hor` is a symmetric operator. Hint: exploit `if_neg` and `hprop_op_comm` (from chapter `Himpl`).

Lemma `hor_comm` : $\forall H1 H2,$

$hor H1 H2 = hor H2 H1.$

Proof using. *Admitted.*

□

Module `HOREEXAMPLE`.

Import `Repr`.

Implicit Types $q : \text{loc}$.

Recall from chapter `Repr` the definition of `MList`, and the two lemmas `MList_nil` and `MList_cons` that reformulates that definition.

Exercise: 4 stars, standard, especially useful (hor_comm) Prove that `MList` can be characterized by a disjunction expressed using `hor` as shown below.

Lemma `MList_using_hor` : $\forall L p, \text{MList } L p =$

$$\begin{aligned} & \text{hor} (\setminus [L = \text{nil} \wedge p = \text{null}]) \\ & (\setminus \exists x q L', \setminus [L = x :: L']) \\ & \quad \setminus * (p \rightsquigarrow \{ \text{head} := x; \text{tail} := q \}) \\ & \quad \setminus * (\text{MList } L' q). \end{aligned}$$

Proof using. Admitted.

□

End HOREXAMPLE.

Definition of hand

The heap predicate `hand` $H1 H2$ lifts the disjunction operator $P1 \wedge P2$ from `Prop` to `hprop`.

Concretely, the heap predicate `hand` $H1 H2$ describes a heap that satisfies $H1$ and at the same time satisfies $H2$.

The heap predicate `hand` admits a direct definition as a function over heaps.

Definition `hand'` ($H1 H2 : \text{hprop}$) : `hprop` :=
`fun h => H1 h \wedge H2 h.`

An alternative definition leverages the \forall quantifier. The definition, shown below, reads as follows: “for any boolean value `b`, if `b` is true then $H1$ should hold, and if `b` is false then $H2$ should hold”.

Definition `hand` ($H1 H2 : \text{hprop}$) : `hprop` :=
 $\forall (b:\text{bool}), \text{if } b \text{ then } H1 \text{ else } H2.$

Exercise: 2 stars, standard, especially useful (hand_eq_hand') Prove the equivalence of the definitions `hand` and `hand'`.

Lemma `hand_eq_hand'` :
`hand = hand'.`

Proof using. Admitted.

□

The introduction and elimination rules for `hand` are as follows.

- If “ $H1$ and $H2$ ” holds, then in particular $H1$ holds.
- Symmetrically, if “ $H1$ and $H2$ ” holds, then in particular $H2$ holds.

- Reciprocally, to prove that a heap predicate $H3$ entails “ $H1$ and $H2$ ”, one must prove that $H3$ entails $H1$, and that $H3$ satisfies $H2$.

`Lemma himpl_hand_l_r : ∀ H1 H2,`
 `hand H1 H2 ==> H1.`

`Proof using. intros. unfolds hand. applys× himpl_hforall_l true. Qed.`

`Lemma himpl_hand_l_l : ∀ H1 H2,`
 `hand H1 H2 ==> H2.`

`Proof using. intros. unfolds hand. applys× himpl_hforall_l false. Qed.`

`Lemma himpl_hand_r : ∀ H1 H2 H3,`
 `H3 ==> H1 →`
 `H3 ==> H2 →`
 `H3 ==> hand H1 H2.`

`Proof using. introv M1 M2 Hh. intros b. case_if×. Qed.`

Exercise: 1 star, standard, especially useful (hand_comm) Prove that `hand` is a symmetric operator. Hint: use `if_neg` and `hprop_op_comm`.

`Lemma hand_comm : ∀ H1 H2,`
 `hand H1 H2 = hand H2 H1.`

`Proof using. Admitted.`

□

`End CONJDISJ.`

44.4.9 Summary of All Separation Logic Operators

Module SUMMARYHPROP.

The core operators are defined as functions over heaps.

`Definition hempty : hprop :=`
 `fun h ⇒ (h = Fmap.empty).`

`Definition hsingle (p:loc) (v:val) : hprop :=`
 `fun h ⇒ (h = Fmap.single p v).`

`Definition hstar (H1 H2 : hprop) : hprop :=`
 `fun h ⇒ ∃ h1 h2, H1 h1`
 `∧ H2 h2`
 `∧ Fmap.disjoint h1 h2`
 `∧ h = Fmap.union h1 h2.`

`Definition hexists A (J:A→hprop) : hprop :=`
 `fun h ⇒ ∃ x, J x h.`

`Definition hforall (A : Type) (J : A → hprop) : hprop :=`

```
fun h ⇒ ∀ x, J x h.
```

The remaining operators can be defined either as functions over heaps, or as derived definitions expressed in terms of the core operators defined above.

Direct definition for the remaining operators.

```
Module REAMININGOPERATORSDIRECT.
```

```
Definition hpure (P:Prop) : hprop :=
  fun h ⇒ (h = Fmap.empty) ∧ P.
```

```
Definition hwand (H1 H2:hprop) : hprop :=
  fun h ⇒ ∀ h', Fmap.disjoint h h' → H1 h' → H2 (h \u h').
```

```
Definition qwand (Q1 Q2:val→hprop) : hprop :=
  fun h ⇒ ∀ v h', Fmap.disjoint h h' → Q1 v h' → Q2 v (h \u h').
```

```
Definition hor (H1 H2 : hprop) : hprop :=
  fun h ⇒ H1 h ∨ H2 h.
```

```
Definition hand (H1 H2 : hprop) : hprop :=
  fun h ⇒ H1 h ∧ H2 h.
```

```
End REAMININGOPERATORSDIRECT.
```

Alternative definitions for the same operators, expressed in terms of the core operators.

```
Module REAMININGOPERATORSDERIVED.
```

```
Definition hpure (P:Prop) : hprop :=
  \exists (p:P), \[].
```

```
Definition hwand (H1 H2 : hprop) : hprop :=
  \exists H0, H0 \* \[ (H1 \* H0) ==> H2 ].
```

```
Definition qwand (Q1 Q2 : val→hprop) : hprop :=
  \forall v, (Q1 v) \-* (Q2 v).
```

```
Definition qwand' (Q1 Q2 : val→hprop) : hprop :=
  \exists H0, H0 \* \[ (Q1 \*+ H0) ===> Q2].
```

```
Definition hand (H1 H2 : hprop) : hprop :=
  \forall (b:bool), if b then H1 else H2.
```

```
Definition hor (H1 H2 : hprop) : hprop :=
  \exists (b:bool), if b then H1 else H2.
```

```
End REAMININGOPERATORSDERIVED.
```

In practice, it saves a lot of effort to use the derived definitions, because using derived definitions all the properties of these definitions can be established with the help of the *xsimpl* tactic, through reasoning taking place exclusively at the level of *hprop*.

```
End SUMMARYHPROP.
```

44.4.10 Historical Notes

The magic wand is an operator that was introduced in the very first days of Separation Logic. From a logical perspective, it makes total sense to have it. From a practical perspective, however, it was not always entirely obvious how the magic wand could simplify specifications and proofs.

Experience with CFML 1.0 shows that it is possible to develop an entire verification frameworks and verify thousands of lines of advanced data structures and algorithms without ever involving the magic wand operator.

The magic wand, however, reveals its interest when exploited (1) in the ramified frame rule, and (2) in weakest-precondition style reasoning rules.

The idea of the ramified frame rule was introduced by *Krishnaswami, Birkedal, and Aldrich 2010* (in Bib.v). Its general statement, as formulated in the present chapter, was proposed by *Hobor and Villard 2013* (in Bib.v). Developers of the tools VST and Iris have advertised for the interest of this rule. The ramified frame rule was integrated in CFML 2.0 in 2018.

Chapter 45

Library `SLF.Affine`

45.1 Affine: Affine Separation Logic

Set Implicit Arguments.

From *SLF* Require Rules.

From *SLF* Require Export LibSepReference.

Implicit Types f : var.

Implicit Types b : **bool**.

Implicit Types v : **val**.

Implicit Types h : heap.

Implicit Types P : Prop.

Implicit Types H : hprop.

Implicit Types Q : **val** → hprop.

45.2 First Pass

The Separation Logic framework that we have constructed is well-suited for a language with explicit deallocation, but cannot be used as such for a language equipped with a garbage collector.

As we pointed out in the chapter `Basic`, there is no rule in our basic Separation Logic that allows discarding a heap predicate from the precondition or the postcondition.

In this chapter, we explain how to generalize the Separation Logic framework to support a “discard rule”, which one may invoke to discard heap predicates from the precondition or from the postcondition.

The framework extended with the discard rule corresponds to an “affine” logic, where heap predicates may be freely discarded, as opposed to a “linear” logic, where heap predicates cannot be thrown away.

This chapter is organized as follows:

- first, we recall the example illustrating the limitation of a logic without a discard rule, for a garbage-collected language;
- second, we present several versions of the “discard rule”;
- third, we show how to refine the definition of Separation Logic triples in such a way that the discard rules are satisfied.

Although in the present course we consider a language for which it is desirable that any heap predicate can be discarded, we will present general definitions allowing to fine-tune which heap predicates can be discarded and which cannot be discarded by the user. We argue further on why such a fine-tuning may be interesting.

45.2.1 Motivation for the Discard Rule

Module MOTIVATINGEXAMPLE.

Export DemoPrograms.

Let us recall the example of the function `succ_using_incr_attempt` from chapter Basic. This function allocates a reference with contents `n`, then increments that reference, and finally returning its contents, that is, `n+1`. Let us revisit this example, with this time the intention of establishing for it a postcondition that does not leak the existence of a left-over reference cell.

```
Definition succ_using_incr :=
  <{ fun 'n =>
    let 'p = ref 'n in
      incr 'p;
      ! 'p }>.
```

In the framework developed so far, the heap predicate describing the reference cell allocated by the function cannot be discarded, because the code considered does not include a deallocation operation. Thus, we are forced to include in the postcondition the description of a left-over reference with a heap predicate, e.g., $\exists p, p \sim\sim> (n+1)$, or $\exists p m, p \sim\sim> m$.

```
Lemma triple_succ_using_incr : ∀ (n:int),
  triple (succ_using_incr n)
  \[]
  (fun r ⇒ \[r = n+1] \* \exists p, (p \sim\sim> (n+1))).
```

Proof using.

`xwp. xapp. intros p. xapp. xapp. xsimpl. auto.`

Qed.

If we try to prove a specification that does not mention the left-over reference, then we get stuck with a proof obligation of the form $p \sim\sim> (n+1) ==> []$.

```
Lemma triple_succ_using_incr' : ∀ (n:int),
```

```

triple (succ_using_incr n)
  []
  (fun r => \[r = n+1]).
```

Proof using.

```
xwp. xapp. intros p. xapp. xapp. xsimpl. { auto. }
```

Abort.

This situation is desirable in a programming language with explicit deallocation, because it ensures that the code written by the programmer is not missing any deallocation operation. However, it is ill-suited for a programming language equipped with a garbage collector that deallocates data automatically.

In this chapter, we present an “affine” version of Separation Logic, where the above function `succ_using_incr` does admits the simple postcondition `fun r => \[r = n+1]`, i.e., that needs not mention the left-over reference in the postcondition.

End MOTIVATINGEXAMPLE.

45.2.2 Statement of the Discard Rule

There are several ways to state the “discard rule”. Let us begin with two rules: one that discards a heap predicate H' from the postcondition, and one that discards a heap predicate H' from the precondition.

The first rule, named *triple_hany_post*, asserts that an arbitrary heap predicate H' can be dropped from the postcondition, simplifying it from $Q \setminus * H'$ to Q .

```
Parameter triple_hany_post : ∀ t H H' Q,
  triple t H (Q \setminus * H') →
  triple t H Q.
```

Let us show that, using the rule *triple_hany_post*, we can derive the desired specification for the motivating example from the specification that mentions the left-over postcondition.

Module MOTIVATINGEXAMPLESOLVED.

Export MotivatingExample.

```
Lemma triple_succ_using_incr' : ∀ (n:int),
  triple (succ_using_incr n)
    []
    (fun r => \[r = n+1]).
```

Proof using.

```
intros. applys triple_hany_post. applys triple_succ_using_incr.
```

Qed.

End MOTIVATINGEXAMPLESOLVED.

A symmetric rule, named *triple_hany_pre*, asserts that an arbitrary heap predicate H' can be dropped from the precondition, simplifying it from $H \setminus * H'$ to H .

```
Parameter triple_hany_pre : ∀ t H H' Q,
```

```

triple t H Q →
triple t (H \* H') Q.

```

Observe the difference between the two rules. In *triple_hany_post*, the discarded predicate H' appears in the premise, reflecting the fact that we discard it after the evaluation of the term t . On the contrary, in *triple_hany_pre*, the discarded predicate H' appears in the conclusion, reflecting the fact that we discard it before the evaluation of t .

The two rules *triple_hany_pre* and *triple_hany_post* can be derived from each other. As we will establish further on, the rule *triple_hany_pre* is derivable from *triple_hany_post*, by a simple application of the frame rule. Reciprocally, *triple_hany_post* is derivable from *triple_hany_pre*, however the proof is more involved (it appears in the bonus section).

45.2.3 Fine-grained Control on Collectable Predicates

As suggested in the introduction, it may be useful to constrain the discard rule in such a way that it can be used to discard only certain types of heap predicates, and not arbitrary heap predicates.

For example, even in a programming language featuring a garbage collector, it may be useful to ensure that every file handle opened eventually gets closed, or that every lock acquired is eventually released. File handles and locks are example of resources that may be described in Separation Logic, yet that one should not be allowed to discard freely.

As another example, consider the extension of Separation Logic with “time credits”, which are used for complexity analysis. In such a setting, it is desirable to be able to throw away a positive number of credits to reflect for over-approximations in the analysis. However, the logic must forbid discarding predicates that capture a negative number of credits, otherwise soundness would be compromised.

The idea is to restrict the discard rules so that only predicates satisfying a predicate called `haffine` may get discarded. The two discard rules will thus feature an extra premise requiring `haffine H'`, where H' denotes the heap predicate being discarded.

Module PREVIEW.

```

Parameter haffine : hprop → Prop.

Parameter triple_haffine_post : ∀ t H H' Q,
  haffine H' →
  triple t H (Q \*+ H') →
  triple t H Q.

Parameter triple_haffine_pre : ∀ t H H' Q,
  haffine H' →
  triple t H Q →
  triple t (H \* H') Q.

```

End PREVIEW.

To constraint the discard rule and allow fine-tuning of which heap predicates may be

thrown away, we introduce the notions of “affine heap” and of “affine heap predicates”, captured by the judgments $\text{heap_affine } h$ and $\text{haffine } H$, respectively.

The definition of $\text{heap_affine } h$ is left abstract for the moment. We will show two extreme instantiations: one that leads to a logic where all predicates are affine (i.e. can be freely discarded), one one that leads to a logic where all predicates are linear (i.e. none can be freely discarded, like in our previous set up).

45.2.4 Definition of heap_affine and of haffine

Concretely, the notion of “affine heap” is characterize by the abstract predicate named heap_affine , which is a predicate over heaps.

`Parameter heap_affine : heap → Prop.`

This predicate heap_affine is assumed to satisfy two properties: it holds of the empty heap, and it is stable by (disjoint) union of heaps.

`Parameter heap_affine_empty :`

`heap_affine Fmap.empty.`

`Parameter heap_affine_union : ∀ h1 h2,`

`heap_affine h1 →`

`heap_affine h2 →`

`Fmap.disjoint h1 h2 →`

`heap_affine (Fmap.union h1 h2).`

The predicate $\text{haffine } H$ captures the notion of “affine heap predicate”. A heap predicate is affine iff it only holds of affine heaps.

`Definition haffine (H:hprop) : Prop :=`

`∀ h, H h → heap_affine h.`

The predicate haffine distributes in a natural way on each of the operators of Separation Logic: the combination of affine heap predicates yields affine heap predicates. In particular:

- $\set{\cdot}$ and \set{P} , which describes empty heaps, can always be discarded;
- $H1 \set{*} H2$ can be discarded if both $H1$ and $H2$ can be discarded;
- $\exists x, H$ and $\forall x, H$ can be discarded if H can be discarded for any x .

`Lemma haffine_hempty :`

`haffine \[].`

`Proof using.`

`introv K. lets E: hempty_inv K. subst. applys heap_affine_empty.`

`Qed.`

`Lemma haffine_hpure : ∀ P,`

`haffine \[P].`

Proof using.

```
intros. intros h K. lets (HP&M): hpure_inv K.  
subst. applys heap_affine_empty.
```

Qed.

Lemma haffine_hstar : $\forall H1 H2,$
 $\text{haffine } H1 \rightarrow$
 $\text{haffine } H2 \rightarrow$
 $\text{haffine } (H1 \setminus * H2).$

Proof using.

```
introv M1 M2 K. lets (h1&h2&K1&K2&D&U): hstar_inv K.  
subst. applys× heap_affine_union.
```

Qed.

Lemma haffine_hexists : $\forall A (J:A \rightarrow \text{hprop}),$
 $(\forall x, \text{haffine } (J x)) \rightarrow$
 $\text{haffine } (\exists x, (J x)).$

Proof using. introv F1 (x&Hx). applys× F1. Qed.

Lemma haffine_hforall : $\forall A \{Inhab\ A\} (J:A \rightarrow \text{hprop}),$
 $(\forall x, \text{haffine } (J x)) \rightarrow$
 $\text{haffine } (\forall x, (J x)).$

Proof using.

```
introv IA F1 Hx. lets N: hforall_inv Hx.  
applys× F1 (arbitrary (A:=A)).
```

Qed.

Note, in the last rule above, that the type A must be inhabited for this rule to make sense. In practice, the \forall quantifier is always invoked on inhabited types, so this is a benign restriction.

In addition, $\text{haffine } ([P] \setminus * H)$ should simplify to $\text{haffine } H$ under the hypothesis P . Indeed, if a heap h satisfies $[P] \setminus * H$, then it must be the case that the proposition P is true. Formally:

Lemma haffine_hstar_hpure_l : $\forall P H,$
 $(P \rightarrow \text{haffine } H) \rightarrow$
 $\text{haffine } ([P] \setminus * H).$

Proof using. introv M. intros h K. rewrite hstar_hpure_l in K. applys× M. Qed.

45.2.5 Definition of the “Affine Top” Heap Predicates

We next introduce a new heap predicate, called “affine top” and written $\setminus GC$, that is very handy for describing “the possibility to discard a heap predicate”. We use this predicate to reformulate the discard rules in a more concise and more usable manner.

This predicate is written $\setminus GC$ and named hgc in the formalization. It holds of any affine heap.

It can be equivalently defined as `fun h => heap_affine h`, or as $\exists H, \setminus[\text{affine } H] \setminus^* H$. The latter formulation is expressed in terms of existing Separation Logic operators, hence it is easier to manipulate in proofs using the `ximpl` tactic.

Definition `hgc : hprop :=`
 $\lambda H, \setminus[\text{affine } H] \setminus^* H.$

Declare Scope hgc_scope.

Open Scope hgc_scope.

Notation " $\setminus\text{GC}$ " := (`hgc`) : `hgc_scope`.

The introduction lemmas asserts that $\setminus\text{GC } h$ holds when `h` satisfies `heap_affine`.

Exercise: 2 stars, standard, especially useful (triple_frame) Prove that the affine heap predicate holds of any affine heap.

Lemma `hgc_intro : ∀ h,`
 $\text{heap_affine } h \rightarrow$
 $\setminus\text{GC } h.$

Proof using. *Admitted.*

□

The elimination lemma asserts the reciprocal.

Exercise: 2 stars, standard, optional (hgc_inv) Prove the elimination lemma for $\setminus\text{GC}$ expressed using `heap_affine`.

Lemma `hgc_inv : ∀ h,`
 $\setminus\text{GC } h \rightarrow$
 $\text{heap_affine } h.$

Proof using. *Admitted.*

□

Together, the introduction and the elimination rule justify the fact that `hgc` could equivalently have been defined as `fun h => heap_affine h`.

Definition `hgc' : hprop :=`
`fun h => heap_affine h.`

Lemma `hgc_eq_hgc' :`
 $\text{hgc} = \text{hgc}'.$

Proof using.

`intros. applys himpl_antisym.`
 $\{ \text{intros } h M. \text{applys } \times \text{hgc_inv}. \}$
 $\{ \text{intros } h M. \text{applys } \times \text{hgc_intro}. \}$

Qed.

45.2.6 Properties of the $\setminus GC$ Predicate

One fundamental property that appears necessary in many of the soundness proofs is the following lemma, which asserts that two occurrences of $\setminus GC$ can be collapsed into just one.

Lemma `hstar_hgc_hgc` :

$$(\setminus GC \setminus * \setminus GC) = \setminus GC.$$

Proof using.

`unfold hgc. applys himpl_antisym.`

`{ xpull. intros H1 K1 H2 K2. xsimpl (H1 \setminus * H2). applys× haffine_hstar. }`

`{ xpull. intros H K. xsimpl H \[]. auto. applys haffine_hempty. }`

Qed.

Another useful property is that the heap predicate $\setminus GC$ itself satisfies `haffine`. Indeed, $\setminus GC$ denotes some heap H such that H is affine; Thus, by essence, it denotes an affine heap predicate.

Lemma `haffine_hgc` :

`haffine \setminus GC.`

Proof using.

`applys haffine_hexists. intros H. applys haffine_hstar_hpure_l. auto.`

Qed.

The process of exploiting the $\setminus GC$ to “absorb” affine heap predicates is captured by the following lemma, which asserts that a heap predicate H entails $\setminus GC$ whenever H is affine.

Lemma `himpl_hgc_r` : $\forall H,$

$H \rightarrow$

$H ==> \setminus GC.$

Proof using. `introv M. intros h K. applys hgc_intro. applys M K. Qed.`

In particular, the empty heap predicate `\[]` entails $\setminus GC$, because the empty heap predicate is affine (recall lemma `haffine_hempty`).

Lemma `hempty_himpl_hgc` :

`\[] ==> \setminus GC.`

Proof using. `applys himpl_hgc_r. applys haffine_hempty. Qed.`

Using the predicate $\setminus GC$, we can reformulate the constrained discard rule `triple_haffine_post` as follows.

Parameter `triple_hgc_post` : $\forall t H Q,$

$\text{triple } t H (Q \setminus * \setminus GC) \rightarrow$

$\text{triple } t H Q.$

Not only is this rule more concise than `triple_haffine_post`, it also has the benefits that the piece of heap discarded, previously described by H' , no longer needs to be provided upfront at the moment of applying the rule. It may be provided further on in the reasoning, for example in an entailment, by instantiating the existential quantifier carried into the definition of $\setminus GC$.

45.2.7 Instantiation of *heap_affine* for a Fully Affine Logic

Module FULLYAFFINELOGIC.

To set up a fully affine logic, we consider a definition of *heap_affine* that holds of any heap.

```
Parameter heap_affine_def : ∀ h,
  heap_affine h = True.
```

It is trivial to check that *heap_affine* satisfies the required distribution properties on the empty heap and the union of heaps.

```
Lemma heap_affine_empty :
  heap_affine Fmap.empty.
Proof using. rewrite× heap_affine_def. Qed.
```

```
Lemma heap_affine_union : ∀ h1 h2,
  heap_affine h1 →
  heap_affine h2 →
  Fmap.disjoint h1 h2 →
  heap_affine (Fmap.union h1 h2).
```

```
Proof using. intros. rewrite× heap_affine_def. Qed.
```

With that instantiation, *haffine* holds of any heap predicate.

```
Lemma haffine_equiv : ∀ H,
  (haffine H) ↔ True.
```

Proof using.

```
intros. iff M.
{ auto. }
{ unfold haffine. intros. rewrite× heap_affine_def. }
```

Qed.

With that instantiation, the affine top predicate $\setminus GC$ is equivalent to the top predicate *htop*, defined as `fun h => True` or, equivalently, as $\exists H, H$.

```
Definition htop : hprop :=
  ∃ H, H.
```

Lemma hgc_eq_htop : hgc = htop.

Proof using.

```
unfold hgc, htop. applys himpl_antisym.
{ xsimpl. }
{ xsimpl. intros. rewrite× haffine_equiv. }
```

Qed.

End FULLYAFFINELOGIC.

45.2.8 Instantiation of *heap_affine* for a Fully Linear Logic

Module FULLYLINEARLOGIC.

To set up a fully affine logic, we consider a definition of *heap_affine* that holds only of empty heaps.

```
Parameter heap_affine_def : ∀ h,
  heap_affine h = (h = Fmap.empty).
```

Again, it is not hard to check that *heap_affine* satisfies the required distributivity properties.

```
Lemma heap_affine_empty :
  heap_affine Fmap.empty.
Proof using. rewrite× heap_affine_def. Qed.
```

```
Lemma heap_affine_union : ∀ h1 h2,
  heap_affine h1 →
  heap_affine h2 →
  Fmap.disjoint h1 h2 →
  heap_affine (Fmap.union h1 h2).
```

Proof using.

```
  introv K1 K2 D. rewrite heap_affine_def in *.
  subst. rewrite× Fmap.union_empty_r.
```

Qed.

The predicate *haffine* H is equivalent to $H ==> \[]$, that is, it characterizes heap predicates that hold of the empty heap.

```
Lemma haffine_equiv : ∀ H,
  haffine H ↔ (H ==> \[]).
```

Proof using.

```
  intros. unfold haffine. iff M.
  { intros h K. specializes M K. rewrite heap_affine_def in M.
    subst. applys hempty_intro. }
  { intros h K. specializes M K. rewrite heap_affine_def.
    applys hempty_inv M. }
```

Qed.

With that instantiation, the affine top predicate $\setminus GC$ is equivalent to the empty heap predicate *hempty*.

```
Lemma hgc_eq_hempty : hgc = hempty.
```

Proof using.

```
  unfold hgc. applys himpl_antisym.
  { xpull. introv N. rewrite× haffine_equiv in N. }
  { xsimpl \[]. applys haffine_hempty. }
```

Qed.

End FULLYLINEARLOGIC.

45.2.9 Refined Definition of Separation Logic Triples

Module NEWTRIPLES.

Thereafter, we purposely leave the definition of `haffine` abstract, for the sake of generality.

In what follows, we explain how to refine the notion of Separation Logic triple so as to accomodate the discard rule.

Recall the definition of triple for a linear logic.

Definition triple ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : Prop := forall ($H':\text{hprop}$), hoare t ($H \setminus* H'$) ($Q \setminus*+ H'$).

The discard rule *triple-htop-post* asserts that postconditions may be freely extended with the $\setminus GC$ predicate. To support this rule, it suffices to modify the definition of triple to include the predicate $\setminus GC$ in the postcondition of the underlying Hoare triple, as follows.

Definition triple ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : Prop :=
 $\forall (H':\text{hprop}), \text{hoare } t (H \setminus* H') (Q \setminus*+ H' \setminus*+ \setminus GC).$

Observe that this definition of triple is strictly more general than the previous one. Indeed, as explained earlier on, when considering the fully linear instantiation of `haffine`, the predicate $\setminus GC$ is equivalent to the empty predicate $\setminus []$. In this case, the occurence of $\setminus GC$ that appears in the definition of triple can be replaced with $\setminus []$, yielding a definition equivalent to our original definition of triple.

For the updated definition of triple using $\setminus GC$, one can prove that:

- all the existing reasoning rules of Separation Logic remain sound;
- the discard rules *triple-htop-post*, *triple-haffine-hpost* and *triple-haffine-hpre* can be proved sound.

45.2.10 Soundness of the Existing Rules

Let us update the soundness proof for the other structural rules.

Exercise: 3 stars, standard, especially useful (triple_frame) Prove the frame rule for the definition of triple that includes $\setminus GC$. Hint: unfold the definition of triple but not that of `hoare`, then exploit lemma `hoare_conseq` and conclude using the tactic `xsiml`.

Lemma triple_frame : $\forall t H Q H',$
 $\quad \text{triple } t H Q \rightarrow$
 $\quad \text{triple } t (H \setminus* H') (Q \setminus*+ H').$

Proof using. Admitted.

□

Exercise: 3 stars, standard, optional (triple_conseq) Prove the frame rule for the definition of triple that includes $\setminus GC$. Hint: follow the same approach as in the proof of triple_frame, and leverage the tactic *xchange* to conclude.

```
Lemma triple_conseq : ∀ t H' Q' H Q,
  triple t H' Q' →
  H ==> H' →
  Q' ===> Q →
  triple t H Q.
```

Proof using. Admitted.

□

The extraction rules remain valid as well.

```
Lemma triple_hpure : ∀ t (P:Prop) H Q,
  (P → triple t H Q) →
  triple t (¬[P] ∗ H) Q.
```

Proof using.

```
intros M. unfolds triple. intros H'.
rewrite hstar_assoc. applys× hoare_hpure.
```

Qed.

```
Lemma triple_hexists : ∀ t (A:Type) (J:A→hprop) Q,
  (∀ x, triple t (J x) Q) →
  triple t (¬∃ x, J x) Q.
```

Proof using.

```
intros M. unfolds triple. intros H'.
rewrite hstar_hexists. applys× hoare_hexists.
```

Qed.

The standard reasoning rules of Separation Logic can be derived for the revised notion of Separation Logic triple, the one which includes $\setminus GC$, following essentially the same proofs as for the original Separation Logic triples. The main difference is that one sometimes needs to invoke the lemma *hstar_hgc_hgc* for collapsing two $\setminus GC$ into a single one.

In what follows, we present just one representative example of such proofs, namely the reasoning rule for sequences. This proof is similar to that of lemma *triple_seq* from chapter Rules.

```
Lemma triple_seq : ∀ t1 t2 H Q H1,
  triple t1 H (fun v => H1) →
  triple t2 H1 Q →
  triple (trm_seq t1 t2) H Q.
```

Proof using.

```
intros M1 M2. intros H'. unfolds triple.
applys hoare_seq.
{ applys M1. }
{ applys hoare_conseq. { applys M2. } { xsimpl. } }
```

$\{ \text{xchanges hstar_hgc_hgc.} \}$
 Qed.

45.2.11 Soundness of the Discard Rules

Module Export GCRULES.

Let us first establish the soundness of the discard rule *triple_htop_post*.

Exercise: 3 stars, standard, especially useful (*triple_hgc_post*) Prove *triple_h_post* with respect to the refined definition of triple that includes $\setminus GC$ in the postcondition. Hint: exploit *hoare_conseq*, then exploit *hstar_hgc_hgc*, with help of the tactics *xchange* and *xsimpl*.

Lemma *triple_hgc_post* : $\forall t H Q,$
 $\text{triple } t H (Q \setminus *+ \setminus GC) \rightarrow$
 $\text{triple } t H Q.$

Proof using. Admitted.

□

EX1! (*triple_haffine_post*) Prove that *triple_haffine_post* is derivable from *triple_hgc_post*.

Hint: unfold the definition of $\setminus GC$ using *unfold hgc*.

Lemma *triple_haffine_post* : $\forall t H H' Q,$
 $\text{haffine } H' \rightarrow$
 $\text{triple } t H (Q \setminus *+ H') \rightarrow$
 $\text{triple } t H Q.$

Proof using. Admitted.

□

EX1? (*triple_hgc_post_from_triple_haffine_post*) Reciprocally, prove that *triple_hgc_post* is derivable from *triple_haffine_post*.

Lemma *triple_hgc_post_from_triple_haffine_post* : $\forall t H Q,$
 $\text{triple } t H (Q \setminus *+ \setminus GC) \rightarrow$
 $\text{triple } t H Q.$

Proof using. Admitted.

□

EX1? (*triple_haffine_pre*) Prove that *triple_haffine_pre* is derivable from *triple_hgc_post*.

Hint: exploit the frame rule, and leverage *triple_hgc_post* either directly or by invoking its corollary *triple_haffine_post*.

Lemma *triple_haffine_pre* : $\forall t H H' Q,$
 $\text{haffine } H' \rightarrow$
 $\text{triple } t H Q \rightarrow$
 $\text{triple } t (H \setminus * H') Q.$

Proof using. Admitted.

□

EX1? (*triple_conseq_frame_hgc*) Prove the combined structural rule *triple_conseq_frame_hgc*, which extends *triple_conseq_frame* with the discard rule, replacing $Q1 \setminus^* H2 \implies Q$ with $Q1 \setminus^* H2 \implies Q \setminus^* \setminus GC$. Hint: invoke *triple_conseq*, *triple_frame* and *triple_hgc_post* in the appropriate order.

```
Lemma triple_conseq_frame_hgc : ∀ H2 H1 Q1 t H Q,
  triple t H1 Q1 →
  H ==> H1 \* H2 →
  Q1 \*+ H2 ==> Q \*+ \GC →
  triple t H Q.
```

Proof using. Admitted.

□

EX1? (*triple_ramified_frame_hgc*) Prove the following generalization of the ramified frame rule that includes the discard rule. Hint: it is a corollary of *triple_conseq_frame_hgc*. Take inspiration from the proof of *triple_ramified_frame* in chapter Wand.

```
Lemma triple_ramified_frame_hgc : ∀ H1 Q1 t H Q,
  triple t H1 Q1 →
  H ==> H1 \* (Q1 \-* (Q \*+ \GC)) →
  triple t H Q.
```

Proof using. Admitted.

□

End GCRULES.

45.2.12 Discard Rules in WP Style

Let us update the definition of *wp* to use the new definition of *triple*.

```
Definition wp (t:trm) (Q:val → hprop) : hprop :=
  ∃ (H:hprop), H \* \[triple t H Q].
```

Recall the characteristic equivalence of *wp*.

```
Lemma wp_equiv : ∀ t H Q,
  (H ==> wp t Q) ↔ (triple t H Q).
```

Proof using.

```
unfold wp. iff M.
{ applys × triple_conseq Q M.
  applys triple_hexists. intros H'.
  rewrite hstar_comm. applys × triple_hpure. }
{ xsimpl × H. }
```

Qed.

In weakest precondition style, the discard rule *triple_hgc_post* translates into the entailment $\text{wp } t (Q \setminus^* \setminus GC) \implies \text{wp } t Q$, as we prove next.

EX1? (wp_hgc_post) Prove the discard rule in wp-style. Hint: exploit `wp_equiv` and `triple_hgc_post`.

Lemma `wp_hgc_post` : $\forall t Q$,
 $\text{wp } t (Q \setminus *+ \setminus GC) \implies \text{wp } t Q$.

Proof using. *Admitted.*

□

Likewise, the wp-style presentation of the rule `triple_hgc_pre` takes the following form.

Lemma `wp_haffine_pre` : $\forall t H Q$,
 $\text{haffine } H \rightarrow$
 $(\text{wp } t Q) \setminus * H \implies \text{wp } t Q$.

Proof using.

`intros K. rewrite wp_equiv. applys triple_haffine_pre.`
`{ applys K. } { rewrite \leftarrow wp_equiv. }`

Qed.

The revised presentation of the wp-style ramified frame rule includes an extra $\setminus GC$ predicate. This rule captures at once all the structural properties of Separation Logic, including the discard rule.

Lemma `wp_ramified` : $\forall Q1 Q2 t$,
 $(\text{wp } t Q1) \setminus * (Q1 \setminus -* (Q2 \setminus *+ \setminus GC)) \implies (\text{wp } t Q2)$.

For a change, let us present below a direct proof for this lemma, that is, not reusing the structural rules associated with triples.

Proof using.

`intros. unfold wp. xpull ;=> H M.`
`xsimpl (H \setminus * (Q1 \setminus -* Q2 \setminus *+ \setminus GC)).`
`unfolds triple. intros H'.`
`applys hoare_conseq (M ((Q1 \setminus -* Q2 \setminus *+ \setminus GC) \setminus * H')).`
`{ xsimpl. } { xchange hstar_hgc_hgc. xsimpl. }`

Qed.

45.2.13 Exploiting the Discard Rule in Proofs

In a practical verification proof, there are two useful ways to discard heap predicates that are no longer needed:

- either by invoking `triple_haffine_pre` to remove a specific predicate from the current state, i.e., the precondition;
- or by invoking `triple_hgc_post` to add a $\setminus GC$ into the current postcondition and allow subsequent removal of any predicate that may be left-over in the final entailment justifying that the final state satisfies the postcondition.

Eager removal of predicates from the current state is never necessary: one can always be lazy and postpone the application of the discard rule until the last step of reasoning.

In practical, it usually suffices to anticipate, right from the beginning of the verification proof, the possibility of discarding heap predicates from the final state.

To that end, we apply the rule *triple_hgc_post* as very first step of the proof to extend the postcondition with a $\setminus GC$ predicate, which will be used to absorb all the garbage left-over at the end of the proof.

We implement this strategy in a systematic manner by integrating directly the rule *triple_hgc_post* into the tactic *xwp*, which sets up the verification proof by computing the characteristic formula. To that end, we generalize the lemma *xwp_lemma*, which the tactic *xwp* applies. Its original statement is as follows.

```
Parameter xwp_lemma : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wp gen ((x, v2)::nil) t1 Q →
  triple (trm_app v1 v2) H Q.
```

Its generalized form extends the postcondition to which the formula computed by *wp gen* is applied from Q to $Q \setminus *+ \setminus GC$, as shown below.

```
Lemma xwp_lemma' : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wp gen ((x, v2)::nil) t1 (Q \*+ \GC) →
  triple (trm_app v1 v2) H Q.
```

Proof using. *intros E M. applys triple_hgc_post. applys xwp_lemma. Qed.*

Let us update the tactic *xwp* to exploit the lemma *xwp_lemma'* instead of *xwp_lemma*.

```
Tactic Notation "xwp" :=
  intros; applys xwp_lemma';
  [ reflexivity | simpl; unfold wp gen var; simpl ].
```

45.2.14 Example Proof Involving Discarded Heap Predicates

Using the updated version of *xwp*, let us revisite the proof of our motivating example *succ_using_incr* in a fully affine logic, i.e., a logical where any predicate can be discarded.

Module MOTIVATINGEXAMPLEWITHUPDATEDXWP.

Export MotivatingExample.

Assume a fully affine logic.

```
Parameter haffine_hany : ∀ (H:hprop),
  haffine H.
```

Observe in the proof below the $\setminus GC$ introduced in the postcondition by the call to *xwp*.

```
Lemma triple_succ_using_incr : ∀ (n:int),
  triple (succ_using_incr n)
  \[]
```

```
(fun r => \[r = n+1]).
```

Proof using.

```
xwp. xapp. intros r. xapp. xapp. xsimpl. { auto. }
```

```
applys himpl_hgc_r. applys haffine_hany.
```

Qed.

We will show further on how to automate the work from the last line of the proof above, by setting up *xsimpl* to automatically resolve goals of the form $H ==> \text{GC}$.

```
End MOTIVATINGEXAMPLEWITHUPDATEDXWP.
```

45.3 More Details

45.3.1 Revised Definition of `mkstruct`

Recall the definition `mkstruct`, as stated in the file `Wand`.

```
Definition mkstruct (F:formula) : formula := fun Q => \exists Q', (F Q') \* (Q' \-* Q).
```

This definition can be generalized to handle not just the consequence and the frame rule, but also the discard rule.

To that end, we augment `mkstruct` with an additional GC , as follows.

```
Definition mkstruct (F:formula) : formula :=
  fun Q => \exists Q', F Q' \* (Q' \-* (Q \*+ \GC)).
```

Let us prove that this revised definition of `mkstruct` does satisfy the `wp`-style statement of the discard rule, which is stated in a way similar to `wp_hgc_post`.

```
Lemma mkstruct_hgc : \forall Q F,
  mkstruct F (Q \*+ \GC) ==> mkstruct F Q.
```

Proof using.

```
intros. unfold mkstruct. set (X := hgc) at 3. replace X with (\GC \* \GC).
  { xsimpl. } { subst X. apply hstar_hgc_hgc. }
```

Qed.

Besides, let us prove that the revised definition of `mkstruct` still satisfies the three originally required properties: erasure, consequence, and frame.

Remark: the proofs shown below exploit a version of *xsimpl* that handles the magic wand but provides no built-in support for the GC predicate.

```
Lemma mkstruct_erase : \forall F Q,
  F Q ==> mkstruct F Q.
```

Proof using.

```
intros. unfold mkstruct. xsimpl Q. apply himpl_hgc_r. apply haffine_hempty.
```

Qed.

```
Lemma mkstruct_conseq : \forall F Q1 Q2,
```

```


$$Q1 \implies Q2 \rightarrow$$


$$\text{mkstruct } F \ Q1 \implies \text{mkstruct } F \ Q2.$$


```

Proof using.

```
intros WQ. unfold mkstruct. xpull. intros Q'. xsimpl Q'. xchange WQ.
```

Qed.

Lemma `mkstruct_frame` : $\forall F H Q,$
 $(\text{mkstruct } F \ Q) \setminus * H \implies \text{mkstruct } F \ (Q \setminus *+ H).$

Proof using.

```
intros. unfold mkstruct. xpull. intros Q'. xsimpl Q'.
```

Qed.

EX2? (`mkstruct_haffine_post`) Prove the reformulation of `triple_haffine_post` adapted to `mkstruct`, for discarding an affine piece of postcondition. Hint: `haffine` is an opaque definition at this stage; the assumption `haffine` needs to be exploited using the lemma `himpl_hgc_r`.

Lemma `mkstruct_haffine_post` : $\forall H Q F,$
 $\text{haffine } H \rightarrow$
 $\text{mkstruct } F \ (Q \setminus *+ H) \implies \text{mkstruct } F \ Q.$

Proof using. Admitted.

□

EX2? (`mkstruct_haffine_pre`) Prove the reformulation of `triple_haffine_pre` adapted to `mkstruct`, for discarding an affine piece of postcondition.

Lemma `mkstruct_haffine_pre` : $\forall H Q F,$
 $\text{haffine } H \rightarrow$
 $(\text{mkstruct } F \ Q) \setminus * H \implies \text{mkstruct } F \ Q.$

Proof using. Admitted.

□

End NEWTRIPLES.

45.3.2 The Tactic `xaffine`, and Behavior of `xsimpl` on $\setminus GC$

Module XAFFINE.

The tactic `xaffine` applies to a goal of the form `haffine H`. The tactic simplifies the goal using all the distributivity rules associated with `haffine`. Ultimately, it invokes `eauto` with `haffine`, which can leverage knowledge specific to the definition of `haffine` from the Separation Logic set up at hand.

Create HintDb `haffine`.

```

Tactic Notation "xaffine" :=
repeat match goal with ⊢ haffine ?H =>
  match H with
  | (hempty) => apply haffine_hempty

```

```

| (hpure _) => apply haffine_hpure
| (hstar _ _) => apply haffine_hstar
| (hexists _) => apply haffine_hexists
| (hforall _) => apply haffine_hforall
| (hgc) => apply haffine_hgc
| _ => eauto with haffine
end
end.

```

Lemma *xaffine_demo* : $\forall H1 H2 H3,$

haffine $H1 \rightarrow$

haffine $H3 \rightarrow$

haffine ($H1 \setminus * H2 \setminus * H3$).

Proof using. *introv K1 KJ. xaffine. Abort.*

End XAFFINE.

Module *XSIMPLEXTENDED*.

Import *LibSepReference*.

The tactic *xsimpl* is extended with support for simplifying goals of the form $H ==> \setminus GC$ into *haffine* H , using lemma *himpl_hgc_r*. For example, *xsimpl* can simplify the goal $H1 \setminus * H2 ==> H2 \setminus * \setminus GC$ into just *haffine* $H1$.

Lemma *xsimpl_xgc_demo* : $\forall H1 H2,$

$H1 \setminus * H2 ==> H2 \setminus * \setminus GC$.

Proof using. *intros. xsimpl. Abort.*

In addition, *xsimpl* invokes the tactic *xaffine* to simplify side-conditions of the form *haffine* H . For example, *xsimpl* can prove the following lemma.

Lemma *xsimpl_xaffine_demo* : $\forall H1 H2,$

haffine $H1 \rightarrow$

$H1 \setminus * H2 ==> H2 \setminus * \setminus GC$.

Proof using. *introv K1. xsimpl. Qed.*

End XSIMPLEXTENDED.

45.3.3 The Proof Tactics for Applying the Discard Rules

Module *XGC*.

Import *LibSepReference*.

To present the discard tactics *xgc*, *xc_keep* and *xgc_post*, we import the definitions from *LibSepDirect* but assume that the definition of *mkstruct* is like the one presented in the present file, that is, including the $\setminus GC$ when defining *mkstruct F* as *fun Q => $\exists Q'$, $F Q' \setminus * (Q' \setminus * (Q \setminus *+ \setminus GC))$* .

As argued earlier on, with this definition, *mkstruct* satisfies the following discard rule.

```
Parameter mkstruct_hgc : ∀ Q F,
  mkstruct F (Q \*+ \GC) ==> mkstruct F Q.
```

The tactic *xgc H* removes *H* from the precondition (i.e. from the current state), in the course of a proof exploiting a formula produced by *wpgen*.

More precisely, the tactic invokes the following variant of the rule *triple_haffine_pre*, which allows to leverage *xsimpl* for computing the heap predicate *H2* that remains after a predicate *H1* is removed from a precondition *H*, through the entailment $H ==> H1 \ast H2$.

```
Lemma xgc_lemma: ∀ H1 H2 H F Q,
  H ==> H1 \ast H2 →
  haffine H1 →
  H2 ==> mkstruct F Q →
  H ==> mkstruct F Q.
```

Proof using.

```
introv WH K M. xchange WH. xchange M.
applys himpl_trans mkstruct_frame.
applys himpl_trans mkstruct_hgc.
applys mkstruct_conseq. xsimpl.
```

Qed.

```
Tactic Notation "xgc" constr(H) :=
  eapply (@xgc_lemma H); [ xsimpl | xaffine | ].
```

```
Lemma xgc_demo : ∀ H1 H2 H3 F Q,
  haffine H2 →
  (H1 \ast H2 \ast H3) ==> mkstruct F Q.
```

Proof using. *introv K2. xgc H2. Abort.*

The tactic *xgc_keep H* is a variant of *xgc* that enables to discard everything but *H* from the precondition.

The implementation of the tactic leverages the same lemma *xgc_lemma*, only providing *H2* instead of *H1*.

```
Tactic Notation "xgc_keep" constr(H) :=
  eapply (@xgc_lemma _ H); [ xsimpl | xaffine | ].
```

```
Lemma xgc_keep_demo : ∀ H1 H2 H3 F Q,
  haffine H1 →
  haffine H3 →
  (H1 \ast H2 \ast H3) ==> mkstruct F Q.
```

Proof using. *introv K1 K3. xgc_keep H2. Abort.*

The tactic *xgc_post* simply extends the postcondition with a *\GC*, to enable subsequent discarding of heap predicates in the final entailment.

```
Lemma xgc_post_lemma : ∀ H Q F,
  H ==> mkstruct F (Q \*+ \GC) →
  H ==> mkstruct F Q.
```

Proof using. *introv M. xchange M. applys mkstruct_hgc. Qed.*

Tactic Notation "xgc_post" :=
 apply xgc_post_lemma.

Lemma xgc_keep_demo : $\forall H1 H2 H3 F Q,$
 haffine $H1 \rightarrow$
 haffine $H3 \rightarrow$
 $H1 ==> \text{mkstruct } F (Q \setminus *+ H2 \setminus *+ H3) \rightarrow$
 $H1 ==> \text{mkstruct } F Q.$

Proof using.

introv K1 K3 M. xgc_post. xchange M. applys mkstruct_conseq. xsimpl.

Abort.

End XGC.

45.4 Optional Material

45.4.1 Alternative Statement for Distribution of **haffine** on Quantifiers

Module HAFFINEQUANTIFIERS.

Recall the lemmas **haffine_hexists** and **haffine_hforall**.

Lemma **haffine_hexists** : forall A (J:A->hprop), (forall x, haffine (J x)) -> haffine ($\setminus \exists x, (J x)$).

Lemma **haffine_hforall** : forall A '{Inhab A} (J:A->hprop), (forall x, haffine (J x)) -> haffine ($\setminus \forall x, (J x)$).

They can be reformulated in a more concise way, as explained next.

First, to smoothly handle the distribution on the quantifiers, let us extend the notion of “affinity” to postconditions. The predicate **haffine_post J** asserts that **haffine** holds of $J x$ for any x .

Definition **haffine_post** (A:Type) (J:A->hprop) : Prop :=
 $\forall x, \text{haffine } (J x).$

The rules then reformulate as follows.

Lemma **haffine_hexists** : $\forall A (J:A \rightarrow \text{hprop}),$
 haffine_post $J \rightarrow$
 haffine (**hexists** J).

Proof using. *introv F1 (x&Hx). applys F1. Qed.*

Lemma **haffine_hforall** : $\forall A \{'\text{Inhab } A\} (J:A \rightarrow \text{hprop}),$
 haffine_post $J \rightarrow$
 haffine (**hforall** J).

Proof using.

introv IA F1 Hx. lets N: hforall_inv Hx. applys \times F1 (arbitrary (A:=A)).
 Qed.

End HAFFINEQUANTIFIERS.

45.4.2 Pre and Post Rules

Module FROMPRETOPOSTGC.

Import Rules ProofsSameSemantics.

Earlier on, we proved that *triple_hgc_pre* is derivable from *triple_hgc_post*, through a simple application of the frame rule.

We wrote that, reciprocally, the rule *triple_hgc_post* is derivable from *triple_hgc_pre*, yet with a slightly more involved proof. Let us present this proof.

In other word, assume *triple_hgc_pre*, and let us prove the result *triple_hgc_post*.

Parameter *triple_hgc_pre* : $\forall t H Q,$
 $\text{triple } t H Q \rightarrow$
 $\text{triple } t (H \setminus * \setminus \text{GC}) Q.$

Lemma *triple_hgc_post* : $\forall t H Q,$
 $\text{triple } t H (Q \setminus *+ \setminus \text{GC}) \rightarrow$
 $\text{triple } t H Q.$

The key idea of the proof is that a term *t* admits the same behavior as *let x = t in x*. Thus, to simulate discarding a predicate from the postcondition of *t*, one can invoke the discard rule on the precondition of the variable *x* that appears at the end of *let x = t in x*.

To formalize this idea, recall from Rules the lemma *eval_like_eta_expansion* which asserts the equivalence of *t* and *let x = t in x*, and recall the lemma *triple_eval_like*, which asserts that two equivalent terms satisfy the same triples.

Proof using.

```
introv M. lets E: eval_like_eta_expansion t "x".
applys triple_eval_like E. applys triple_let.
{ applys M. }
{ intros v. simpl. applys triple_hgc_pre. applys triple_val. auto. }
```

Qed.

End FROMPRETOPOSTGC.

45.4.3 Low-level Definition of Refined Triples

Module LOWLEVEL.
 Import NewTriples.

Consider the updated definition of *triple* introduced in this chapter.

Definition *triple* (*t:trm*) (*H:hprop*) (*Q:val \rightarrow hprop*) : Prop :=

$\forall (H':\text{hprop}), \text{hoare } t (H \setminus* H') (Q \setminus*+ H' \setminus*+ \setminus\text{GC}).$

In chapter `Hprop`, we presented an alternative definition for Separation Logic triples, called `triple_lowlevel`, expressed directly in terms of heaps.

Definition `triple_lowlevel (t:trm) (H:hprop) (Q:val->hprop) : Prop := forall h1 h2, Fmap.disjoint h1 h2 -> H h1 -> exists h1' v, Fmap.disjoint h1' h2 /\ eval (h1 \u h2) t (h1' \u h2) v /\ Q v h1'.`

In what follows, we explain how to generalize this definition to match our revised definition of `triple`, and thereby obtain a direct definition expressed in terms of heap, that does not depend on the definition of `hstar` nor that of $\setminus\text{GC}$.

Let us aim instead for a direct definition, entirely expressed in terms of union of heaps. To that end, we need to introduce an additional piece of state to describe the piece of the final heap covered by the $\setminus\text{GC}$ predicate.

We will need to describe the disjointness of the 3 pieces of heap that describe the final state. To that end, we exploit the auxiliary definition `Fmap.disjoint_3 h1 h2 h3`, which asserts that the three arguments denote pairwise disjoint heaps. It is defined in the module `FMAP` as follows.

Definition `disjoint_3 (h1 h2 h3:heap) : Prop := disjoint h1 h2 /\ disjoint h2 h3 /\ disjoint h1 h3.`

We then formulate `triple_lowlevel` using a final state of the form $h1' \setminus\text{u} h2 \setminus\text{u} h3'$ instead of just $h1' \setminus\text{u} h2$. There, $h3'$ denotes the piece of the final state covered by the $\setminus\text{GC}$ heap predicate. This piece of state is an affine heap, as captured by `heap_affine h3'`.

Definition `triple_lowlevel (t:trm) (H:hprop) (Q:val->hprop) : Prop :=`

```

 $\forall h1 h2,$ 
 $\text{Fmap.disjoint } h1 h2 \rightarrow$ 
 $H h1 \rightarrow$ 
 $\exists h1' h3' v,$ 
 $\text{Fmap.disjoint\_3 } h1' h2 h3'$ 
 $\wedge \text{heap\_affine } h3'$ 
 $\wedge \text{eval } (h1 \setminus\text{u} h2) t (h1' \setminus\text{u} h2 \setminus\text{u} h3') v$ 
 $\wedge Q v h1'.$ 

```

One can prove the equivalence of `triple` and `triple_lowlevel` following a similar proof pattern as previously.

Lemma `triple_eq_triple_lowlevel :`

`triple = triple_lowlevel.`

Proof using.

```

applys pred_ext_3. intros t H Q.
unfold triple, triple_lowlevel, hoare. iff M.
{ introv D P1.
forwards \neg (h' \& v \& R1 \& R2): M (=h2) (h1 \u h2). { apply \times hstar_intro. }
destruct R2 as (h2' \& h1'' \& N0 \& N1 \& N2 \& N3).
destruct N0 as (h1' \& h3' \& T0 \& T1 \& T2 \& T3). subst.

```

```

 $\exists h1' h1'' v. \text{splits} \times.$ 
{ rew_disjoint. auto. }
{ applys hgc_inv N1. }
{ applys_eq  $\times$  R1. } }

{ introv (h1 $\&$ h2 $\&$ N1 $\&$ N2 $\&$ D $\&$ U). }
forwards  $\neg$  (h1' $\&$ h3' $\&$ v $\&$ R1 $\&$ R2 $\&$ R3 $\&$ R4): M h1 h2.
 $\exists (h1' \setminus_u h3' \setminus_u h2) v. \text{splits} \times.$ 
{ applys_eq  $\times$  R3. }
{ subst. rewrite hstar_assoc. apply  $\times$  hstar_intro.
rewrite hstar_comm. applys  $\times$  hstar_intro. applys  $\times$  hgc_intro. } }

Qed.
End LowLEVEL.

```

45.4.4 Historical Notes

The seminal presentation of Separation Logic concerns a linear logic, for a programming language with explicit deallocation. More recent works on Separation Logic for ML-style languages, equipped with a garbage collector, consider affine logics. For example, the original presentation of the Iris framework provides an affine entailment, for which $H ==> \[]$ is always true. Follow-up work on Iris provides encodings for supporting linear resources, i.e., resources that are not allowed to be “dropped on the floor”.

This chapter gives a presentation of Separation Logic featuring a customizable predicate `haffine` for controlling which resources should be treated as affine, and which ones should be treated as linear. This direct approach to controlling linearity was introduced in the context of CFML, in work by *Guéneau, Jourdan, Charguéraud, and Pottier 2019* (in Bib.v)

Chapter 46

Library SLF.Struct

46.1 Struct: Arrays and Records

```
Set Implicit Arguments.  
From SLF Require Import LibSepReference LibSepTLCbuffer.  
Hint Rewrite conseq_cons' : rew_listx.  
  
Implicit Types P : Prop.  
Implicit Types H : hprop.  
Implicit Types Q : val → hprop.  
Implicit Type p q : loc.  
Implicit Type k : nat.  
Implicit Type i n : int.  
Implicit Type v : val.  
Implicit Type L : list val.  
Implicit Types z : nat.
```

46.2 First Pass

This chapter introduces support for reasoning about arrays and records.

In the first part of this chapter, we present the representation predicates for these structures, and present the statement of the specifications associated with operations on arrays and records.

In the second part of this chapter, we show how these specifications can be realized, with respect to a memory model that exposes a representation of headers for allocated blocks. More precisely, we show how to implement array and record operations using a pointer arithmetic primitive operation, and we establish the correctness of the specifications with respect to the semantics of the allocation, deallocation, and pointer arithmetic operations.

The memory model that we consider is somewhat artificial, in the sense that it does not perfectly match the memory model of an existing language—it lies somewhere between the

memory model of C and that of OCaml. Nevertheless, this memory model has the benefits of its simplicity, and it suffices to illustrate formal proofs involving block headers and pointer arithmetics.

46.2.1 Representation of a Set of Consecutive Cells

The cells from an array of length k can be represented as a range of k consecutive cells, starting at some location p . In other words, the array cells have addresses from p inclusive to $p+k$ exclusive.

The heap predicate $\text{hcells } L \ p$ represents a consecutive set of cells starting at location p and whose elements are described by the list L . The length of the list L corresponds to the length of the array.

On paper, we could write something along the lines of: $\{x \text{ at index } i \text{ in } L\} \{ (p+i) \sim > x\}$.

In Coq, we define the predicate $\text{hcells } L \ p$ by recursion on the list L , with the pointer p incremented by one unit at each cell, as follows.

```
Fixpoint hcells (L:list val) (p:loc) : hprop :=
  match L with
  | nil ⇒ []
  | x::L' ⇒ (p ~~> x) ∗ (hcells L' (p+1)%nat)
  end.
```

The description of a set of consecutive cells can be split in two parts, at an arbitrary index. Concretely, if we have $\text{hcells } (L1 ++ L2) \ p$, then we can separate the left part $\text{hcells } L1 \ p$ from the right part $\text{hcells } L2 \ q$, where the address q is equal to $p + \text{length } L1$. Reciprocally, the two parts can be merged back into the original form at any time.

```
Parameter hcells_concat_eq : ∀ p L1 L2,
  hcells (L1 ++ L2) p = (hcells L1 p ∗ hcells L2 (length L1 + p)%nat).
```

This “splitting lemma for arrays” is useful for carrying out local reasoning on arrays. For example, in the recursive quicksort algorithm, the specification requires a description of the segment to be sorted; the representation of this segment is split so that subsegments can be provided for reasoning about the recursive calls. One thereby obtains for free the fact that the cells outside of the targeted segment remain unmodified.

46.2.2 Representation of an Array with a Block Header

An array consists of a “header”, and of the description of its cells. The header is a heap predicate that describes the length of the array.

- In OCaml, the header consists of a physical memory cell at the head of the array. This cell may be queried to obtain the length of the array.

- In C, the header is a logical notion. It describes the length of the allocated block, and it is used in Separation Logic to specify the behavior of the deallocation function, which can only be called on the address of an allocated block, deallocating the full block at once.

In this course, we follow the OCaml view, with physical headers.

The predicate *hheader* k p asserts the existence of an allocated block at location p , such that the contents of the block is made of k cells, not counting the header cell. For the moment, we leave its definition abstract.

Parameter *hheader* : $\forall (k:\text{nat}) (p:\text{loc}), \text{hprop}.$

The heap predicate *hheader* k p should capture the information that p is not null. Indeed blocks cannot be allocated at the null location.

Parameter *hheader_not_null* : $\forall p k,$
 $\text{hheader } k p ==> \text{hheader } k p \setminus [p \neq \text{null}].$

An array is described by the predicate *harray* L p , where the list L describes the contents of the cells. This heap predicate covers both the header, which describes a block of length equal to *length* L , and the contents of the cells, described by *hcells* L ($p+1$).

Note that $p+1$ corresponds to the address of the first cell of the array, located immediately past the header cell that sits at location p .

Definition *harray* ($L:\text{list val}$) $(p:\text{loc}) : \text{hprop} :=$
 $\text{hheader} (\text{length } L) p \setminus \text{hcells } L (p+1)\%{\text{nat}}.$

46.2.3 Specification of Allocation

The primitive operation *val_alloc* k allocates a block made of k consecutive cells.

Parameter *val_alloc* : **prim**.

The operation *val_alloc* k is specified as producing an array whose cells contain the special “uninitialized value”, written *val_uninit*. The assume *val_uninit* to be part of the grammar of values.

Check *val_uninit* : *val*.

More precisely, the postcondition of *val_alloc* k is of the form *funloc* $p \Rightarrow \text{harray } L p$, where the list L is defined as the repetition of k times the value *val_uninit*. This list is written *LibList.make* k *val_uninit*.

Parameter *triple_alloc_nat* : $\forall k,$
 $\text{triple} (\text{val_alloc } k)$
 $\quad \setminus []$
 $\quad (\text{funloc } p \Rightarrow \text{harray} (\text{LibList.make } k \text{ val_uninit}) p).$

In practice, the operation *val_alloc* is applied to a non-negative integer, which might not necessarily be syntactically a natural number. Hence, the following lemma, which specifies *val_alloc* n for $n \geq 0$, is more handy to use.

```

Parameter triple_alloc : ∀ n,
  n ≥ 0 →
  triple (val_alloc n)
  []
  (funloc p ⇒ harray (LibList.make (abs n) val_uninit) p).

```

The specification above turns out to be often unnecessarily precise. For most applications, it is sufficient for the postcondition to describe the array as `harray L p` for some unspecified list `L` of length `n`. This weaker specification is stated and proved next.

```

Parameter triple_alloc_array : ∀ n,
  n ≥ 0 →
  triple (val_alloc n)
  []
  (funloc p ⇒ ∃ L, [n = length L] ∗ harray L p).

```

Remark: in OCaml, one must provide an initialization value explicitly, so there is no such thing as `val_uninit`; in JavaScript, `val_uninit` is called *undefined*; in Java, arrays are initialized with zeros; in C, uninitialized data should not be read. In that language, one would typically implement this policy by restricting the evaluation rule for the read operation, adding a premise of the form `v ≠ val_uninit` to ensure that uninitialized values cannot be read.

46.2.4 Specification of the Deallocation

The deallocation operation `val_dealloc p` deallocates the block allocated at location `p`.

Parameter `val_dealloc : prim`.

The specification of `val_dealloc p` features a precondition that requires an array of the form `harray L p`, and an empty postcondition.

```

Parameter triple_dealloc : ∀ L p,
  triple (val_dealloc p)
  (harray L p)
  (fun _ ⇒ []).

```

Observe that the `harray L p` predicate includes a header of the form `hheader k p`, ensuring that a block can be deallocated only once.

46.2.5 Specification of Array Operations

The operation `val_array_get p i` returns the contents of the `i`-th cell of the array at location `p`.

Parameter `val_array_get : val`.

The specification of `val_array_get` is as follows. The precondition describes the array in the form `harray L p`, with a premise that requires the index `i` to be in the valid range, that is, between zero (inclusive) and the length of `L` (exclusive). The postcondition asserts that

the result value is `nth (abs i) L`, and mentions the unmodified array, still described by `harray L p`.

```
Parameter triple_array_get : ∀ L p i,
  0 ≤ i < length L →
  triple (val_array_get p i)
  (harray L p)
  (fun r ⇒ \[r = LibList.nth (abs i) L] \* harray L p).
```

The operation `val_array_set p i v` updates the contents of the i -th cell of the array at location `p`.

Parameter `val_array_set` : `val`.

The specification of `val_array_set` admits the same precondition as `val_array_get`, with `harray L p` and the constraint $0 \leq i < \text{length } L$. Its postcondition describes the updated array using a predicate of the form `harray L' p`, where L' corresponds to `update (abs i) v L`.

```
Parameter triple_array_set : ∀ L p i v,
  0 ≤ i < length L →
  triple (val_array_set p i v)
  (harray L p)
  (fun _ ⇒ harray (LibList.update (abs i) v L) p).
```

The operation `val_array_length p` returns the length of the array allocated at location `p`.

Parameter `val_array_length` : `val`.

There are two useful specifications for `val_array_length`. The first one operates with the heap predicate `harray L p`. The return value is the length of the list L .

```
Parameter triple_array_length : ∀ L p,
  triple (val_array_length p)
  (harray L p)
  (fun r ⇒ \[r = length L] \* harray L p).
```

The second specification for `val_array_length` operates only on the header of the array. This small-footprint specification is useful to read the length of an array whose cells are described by separated segments, that is, after `hcells_concat_eq` has been exploited.

```
Parameter triple_array_length_header : ∀ k p,
  triple (val_array_length p)
  (hheader k p)
  (fun r ⇒ \[r = (k:int)] \* hheader k p).
```

Hint Resolve `triple_array_get` `triple_array_set` `triple_array_length` : `triple`.

46.2.6 Representation of Individual Records Fields

A record can be represented just like an array, with the field names corresponding to offsets in that array.

We let **field** denote the type of field names, an alias for **nat**.

Definition **field** : Type := **nat**.

For example, consider a mutable list cell allocated at location **p**. It is represented in memory as:

- a header at location **p**, storing the number of fields, that is, the value 2;
- a cell at location **p+1**, storing the contents of the head field,
- a cell at location **p+2**, storing the contents of the tail field.

Concretely, the record can be represented by the heap predicate: $(hheader\ 2\ p)\ \backslash^*\ ((p+1)\ \sim>\ x)\ \backslash^*\ ((p+2)\ \sim>\ q)$.

To avoid exposing pointer arithmetic to the end-user, we introduce the “record field” notation $p'.k\ \sim>\ v$ to denote $(p+1+k)\ \sim>\ v$.

For example, with the definition of the field offsets **head** := 0 and **tail** := 1, the same record as before can be represented as: $(hheader\ 2\ p)\ \backslash^*\ (p'.head\ \sim>\ x)\ \backslash^*\ (p'.tail\ \sim>\ q)$.

Definition **hfield** (*p:loc*) (*k:field*) (*v:val*) : **hprop** :=
 $(p+1+k)\%nat\ \sim>\ v$.

Notation "p'.k'> v" := (**hfield** *p k v*)
(at level 32, *k* at level 0, no associativity,
format "p'.k'> v").

46.2.7 Representation of Records

Describing, e.g., a list cell record in the form $(hheader\ 2\ p)\ \backslash^*\ (p'.head\ \sim>\ x)\ \backslash^*\ (p'.tail\ \sim>\ q)$ in particularly verbose and cumbersome to manipulate.

To improve the situation, we next introduce generic representation predicate for records that allows to describe the same list cell much more concisely, as $p\ \sim>\{ head := x; tail := q \}$.

It what follows, we show how to implement this notation by introducing the heap predicates **hfields** and **hrecords**. We then represent the specifications of record operations with respect to those predicates.

A record field is described as the pair of a field and a value stored in this field.

Definition **hrecord_field** : Type := (**field** × **val**).

A record consists of a list of fields.

Definition **hrecord_fields** : Type := **list** **hrecord_field**.

We let the meta-variable *kvs* denote such lists.

Implicit Types *kvs* : **hrecord_fields**.

A list cell with head field **x** and tail field **q** is represented by the list **(head,x)::(tail,q)::nil**. To support the syntax '{ head := x; tail := q }', we introduce the following notation.

```

Notation "{ k1 := v1 }" :=
  ((k1 , (v1:val))::nil)
  (at level 0, k1 at level 0, only parsing).

Notation "{ k1 := v1 ; k2 := v2 }" :=
  ((k1 , (v1:val))::(k2 , (v2:val))::nil)
  (at level 0, k1, k2 at level 0, only parsing).

Notation "{ k1 := v1 ; k2 := v2 ; k3 := v3 }" :=
  ((k1 , (v1:val))::(k2 , (v2:val))::(k3 , (v3:val))::nil)
  (at level 0, k1, k2, k3 at level 0, only parsing).

```

```

Notation "{ k1 := v1 }" :=
  ((k1 , v1)::nil)
  (at level 0, k1 at level 0, only printing).

Notation "{ k1 := v1 ; k2 := v2 }" :=
  ((k1 , v1)::(k2 , v2)::nil)
  (at level 0, k1, k2 at level 0, only printing).

```

```

Notation "{ k1 := v1 ; k2 := v2 ; k3 := v3 }" :=
  ((k1 , v1)::(k2 , v2)::(k3 , v3)::nil)
  (at level 0, k1, k2, k3 at level 0, only printing).

```

`Open Scope val_scope.`

The heap predicate `hfields kvs p` asserts that, at location `p`, one finds the representation of the fields described by the list `kvs`.

The predicate `hfields kvs p` is defined recursively on the list `kvs`. If `kvs` is empty, the predicate describes the empty heap predicate. Otherwise, it describes a first field, at offset `k` and with contents `v`, as the predicate `p^.k ~~> v`, and it describes the remaining fields recursively.

```

Fixpoint hfields (kvs:hrecord_fields) (p:loc) : hprop :=
  match kvs with
  | nil => []
  | (k , v)::kvs' => (p^.k ~~> v) \* (hfields kvs' p)
  end.

```

The heap predicate `hrecord kvs p` describes a record: it covers not just all the fields of the record, but also the header.

The cells are described by `hfields kvs p`, and the header is described by `hheader z p`, where `nb` should be such that the keys in the list `kvs` are between 0 inclusive and `nb` exclusive.

A permissive definition of `hrecord kvs p` would allow the fields from the list `kvs` to be permuted arbitrarily. Yet, to avoid complications related to permutations, we make in this course the simplifying assumptions that fields are always listed in the order of their associated offsets.

The auxiliary predicate `maps_all_fields z kvs` asserts that the keys from the association list `kvs` correspond exactly to the sequence made of the first `nb` natural numbers, that is, 0;

$1; \dots; nb-1$.

```
Definition maps_all_fields (nb:nat) (kvs:hrecord_fields) : Prop :=  
  LibList.map fst kvs = nat_seq 0 nb.
```

The predicate `hrecord kvs p` exploits `maps_all_fields z kvs` to relate the value `nb` stored in the header with the association list `kvs` that describes the contents of the fields.

```
Definition hrecord (kvs:hrecord_fields) (p:loc) : hprop :=  
  \exists z, hheader z p \* hfields kvs p \* \[maps_all_fields z kvs].
```

The heap predicate `hrecord kvs p` captures in particular the invariant that the location `p` is not null.

```
Lemma hrecord_not_null : \forall p kvs,  
  hrecord kvs p ==> hrecord kvs p \* \[p \neq null].
```

Proof using.

```
  intros. unfold hrecord. xpull. intros z M.  
  xchanges \* hheader_not_null.
```

Qed.

We introduce the notation `p $\sim\sim\sim >$ kvs` for `hrecord kvs p`, allowing to write, e.g., `p $\sim\sim\sim >$ { head := x; tail := q }`.

```
Notation "p ' $\sim\sim\sim >$ ' kvs" := (hrecord kvs p)  
(at level 32).
```

46.2.8 Example with Mutable Linked Lists

Recall the definition of the representation predicate `MList`, which was introduced in Basic.

```
Definition head : field := 0%nat.
```

```
Definition tail : field := 1%nat.
```

```
Fixpoint MList (L:list val) (p:loc) : hprop :=  
  match L with  
  | nil => \[p = null]  
  | x::L' => \exists q, (p  $\sim\sim\sim >$  { head := x; tail := q }) \* (MList L' q)  
  end.
```

Recall the statement of the lemma `MList_if`, which reformulates the definition of `MList` with a case analysis on `p = null`.

Observe the use of the lemma `hrecrod_not_null` to exploit the fact that a record cannot be allocated at the `null` location.

```
Lemma MList_if : \forall (p:loc) (L:list val),  
  (MList L p)  
 ==> (If p = null  
    then \[L = nil]  
    else \exists x q L', \[L = x::L']
```

```
\* (p ``> '{ head := x ; tail := q }) \* (MList L' q)).
```

Proof using.

```
intros. destruct L as [|x L']; simpl.
{ xpull. intros M. case_if. xsimpl×. }
{ xpull. intros q. xchange hrecord_not_null. intros N.
  case_if. xsimpl×. }
```

Qed.

Global Opaque MList.

46.2.9 Reading in Record Fields

Declare Scope *trm_scope_ext*.

The read operation is described by an expression of the form `val_get_field k p`, where `k` denotes a field name, and where `p` denotes the location of a record. Technically, `val_get_field k` is a value, and this value is applied to the pointer `p`. Hence, `val_get_field` has type `field → val`.

Parameter `val_get_field` : `field → val`.

The read operation `val_get_field k p` is abbreviated as `p'.k`.

```
Notation "t1 ``.` k" :=
  (val_get_field k t1)
  (in custom trm at level 56, k at level 0, format "t1 ``.` k" )
  : trm_scope_ext.
```

The operation `val_get_field k p` can be specified at three levels.

First, its small-footprint specification operates at the level of a single field, described by `p'.k ``> v`. The specification is very similar to that of `val_get`. The return value is exactly `v`.

```
Parameter triple_get_field : ∀ p k v,
  triple (val_get_field k p)
  (p'.k ``> v)
  (fun r ⇒ \[r = v] \* (p'.k ``> v)).
```

Second, this operation can be specified with respect to a list of fields, described in the form `hfields kvs p`. To that end, we introduce a function called `hfields_lookup` for extracting the value `v` associated with a field `k` in a list of record fields `kvs`.

The operation `hfields_lookup k kvs` returns a result of type `option val`, because we cannot presume that the field `k` occurs in `kvs`, even though it is always the case in practice.

```
Fixpoint hfields_lookup (k:field) (kvs:hrecord_fields) : option val :=
  match kvs with
  | nil ⇒ None
  | (ki, vi)::kvs' ⇒ if Nat.eq_dec k ki
    then Some vi
    else hfields_lookup k kvs'
```

```

    else hfields_lookup k kvs'
end.
```

Under the assumption that `hfields_lookup k kvs` returns `Some v`, the read operation `val_get_field k p` is specified to return `v`. The corresponding specification appears below.

```
Parameter triple_get_field_hfields : ∀ kvs p k v,
  hfields_lookup k kvs = Some v →
  triple (val_get_field k p)
  (hfields kvs p)
  (fun r ⇒ \[r = v] \* hfields kvs p).
```

Third and last, the read operation can be specified with respect to the predicate `hrecord kvs p`, describing the full record, including its header. The specification is similar to the previous one.

```
Parameter triple_get_field_hrecord : ∀ kvs p k v,
  hfields_lookup k kvs = Some v →
  triple (val_get_field k p)
  (hrecord kvs p)
  (fun r ⇒ \[r = v] \* hrecord kvs p).
```

46.2.10 Writing in Record Fields

The write operation is described by an expression of the form `val_set_field k p v`, where `k` denotes a field name, and where `p` denotes the location of a record, and `v` is the new value for the field.

Parameter `val_set_field` : field → val.

The write operation `val_get_field k p v` is abbreviated as `Set p'.k' := v`.

```
Notation "t1 `.` k `:=` t2" :=
  (val_set_field k t1 t2)
  (in custom trm at level 56, k at level 0, format "t1 `.` k `:=` t2")
  : trm_scope_ext.
```

Like for the read operation, the write operation can be specified at three levels. First, at the level of an individual field.

```
Parameter triple_set_field : ∀ v p k v',
  triple (val_set_field k p v)
  (p'.k ~~> v')
  (fun _ ⇒ p'.k ~~> v).
```

Then, at the level of `hfields` and `hrecord`. To that end, we introduce an auxiliary function called `hrecord_update` for computing the updated list of fields following an write operation. Concretely, `hrecord_update k w kvs` replaces the contents of the field named `k` with the value `w`. It returns some description `kvs'` of the record fields, provided the update operation

succeeded, i.e., provided that the field k on which the update is to be performed actually occurs in the list kvs .

```
Fixpoint hfields_update (k:field) (v:val) (kvs:hrecord_fields)
  : option hrecord_fields :=
  match kvs with
  | nil => None
  | (ki, vi)::kvs' => if Nat.eq_dec k ki
    then Some ((k, v)::kvs')
    else match hfields_update k v kvs' with
      | None => None
      | Some LR => Some ((ki, vi)::LR)
    end
  end.
```

The specification in terms of `hfields` is as follows.

```
Parameter triple_set_field_hfields : ∀ kvs kvs' k p v,
  hfields_update k v kvs = Some kvs' →
  triple (val_set_field k p v)
  (hfields kvs p)
  (fun _ => hfields kvs' p).
```

The specification in terms of `hrecord` is similar.

```
Parameter triple_set_field_hrecord : ∀ kvs kvs' k p v,
  hfields_update k v kvs = Some kvs' →
  triple (val_set_field k p v)
  (hrecord kvs p)
  (fun _ => hrecord kvs' p).
```

46.2.11 Allocation of Records

Because records are internally described like arrays, records may be allocated and deallocated using the operations `val_alloc` and `val_dealloc`, just like for arrays. That said, it is useful to express derived specifications for these two operations stated in terms of representation predicate `hrecord`, which describes a full record in terms of the list of its fields.

For allocation, one needs to provide, at some point, the fields names for the record being allocated. These fields names may be described by a list of field names of type `list field`.

This list, written ks , should be equivalent to a list of consecutive natural numbers $0 :: 1 :: \dots :: n-1$, where n denotes the number of fields. The interest of introducing the list ks is to provide readable names in place of numbers.

The operation `val_alloc_hrecord ks` is implemented by invoking `val_alloc` on the length of ks .

```
Definition val_alloc_hrecord (ks:list field) : trm :=
  val_alloc (length ks).
```

The specification of `val_alloc_hrecord` ks involves an empty precondition and a postcondition of the form `hrecord kvs p`, where the list kvs maps the fields names from ks to the value `val_uninit`. The premise expressed in terms of `nat_seq` ensures that the list ks contains consecutive offsets starting from zero.

In the statement below, `LibListExec.length` is a variant of `LibList.length` that computes in Coq (using `simpl` or `reflexivity`). Likewise for `LibListExec.map`, which is equivalent to `LibList.map`.

```
Parameter triple_alloc_hrecord : ∀ ks,
  ks = nat_seq 0 (LibListExec.length ks) →
  triple (val_alloc_hrecord ks)
  []
  (funloc p ⇒ hrecord (LibListExec.map (fun k ⇒ (k, val_uninit)) ks) p).
```

`Hint Resolve triple_alloc_hrecord : triple.`

For example, the allocation of a list cell is specified as follows.

```
Lemma triple_alloc_mcons :
  triple (val_alloc_hrecord (head::tail::nil))
  []
  (funloc p ⇒ p ~~~> '{ head := val_uninit ; tail := val_uninit }).
```

`Proof using.` *applys* \times `triple_alloc_hrecord`. `Qed.`

46.2.12 Deallocation of Records

Deallocation of a record, written `val_dealloc_hrecord p`, is implemented as `val_dealloc p`.

`Definition val_dealloc_hrecord : val :=`
 `val_dealloc.`

The specification of this operation simply requires as precondition the full record description, in the form `hrecord kvs p`, and yields the empty postcondition.

```
Parameter triple_dealloc_hrecord : ∀ kvs p,
  triple (val_dealloc_hrecord p)
  (hrecord kvs p)
  (fun _ ⇒ []).
```

`Hint Resolve triple_dealloc_hrecord : triple.`

To improve readability, we introduce the notation `Delete p` for record deallocation.

```
Notation "'delete'" :=
  (trm_val val_dealloc_hrecord)
  (in custom trm at level 0) : trm_scope_ext.
```

For example, the following corollary to `triple_dealloc_hrecord` may be used to reason about the deallocation of a list cell.

`Lemma triple_dealloc_mcons : ∀ p x q,`

```

triple (val_dealloc_hrecord p)
  (p ~~~> '{ head := x ; tail := q })
  (fun _ => []).

```

Proof using. intros. *applys* \times *triple_dealloc_hrecord*. Qed.

46.2.13 Combined Record Allocation and Initialization

It is often useful to allocate a record and immediately initialize its fields. To that end, we introduce the operation *val_new_hrecord*, which applies to a list of fields and to values for these fields.

This operation can be defined in an arity-generic way, yet, to avoid technicalities, we only present its specialization for arity 2.

```

Module RECORDINIT.
Import ProgramSyntax.
Open Scope trm_scope_ext.

```

In the definition below, the expression in braces *val_alloc_hrecord* (*k1::k2::nil*) refers to a Coq term, embedded in the syntax of program terms.

```

Definition val_new_hrecord_2 (k1:field) (k2:field) : val :=
<{ fun 'x1 'x2 =>
  let 'p = {val_alloc_hrecord (k1::k2::nil)} in
  'p'.k1 := 'x1;
  'p'.k2 := 'x2;
  'p }>.

```

To improve readability, we introduce notation to allow writing, e.g., '{ head := x; tail := q }' for the allocation and initialization of a list cell.

```

Notation "'{ k1 := v1 ; k2 := v2 }'" :=
(val_new_hrecord_2 k1 k2 v1 v2)
(in custom trm at level 65,
 k1, k2 at level 0,
 v1, v2 at level 65) : trm_scope_ext.

```

This operation is specified as follows.

```

Lemma triple_new_hrecord_2 : ∀ k1 k2 v1 v2,
  k1 = 0%nat →
  k2 = 1%nat →
  triple <{ '{ k1 := v1 ; k2 := v2 } }>
  [] (funloc p => p ~~~> '{ k1 := v1 ; k2 := v2 }).

```

Proof using.

```

intros → →. xwp. xapp triple_alloc_hrecord. { auto. } intros p. simpl.
xapp triple_set_field_hrecord. { reflexivity. }
xapp triple_set_field_hrecord. { reflexivity. }

```

xval. xsimpl \times .

Qed.

For example, the operation `mcons` \times `q` allocates a list cell with head value `x` and tail pointer `q`.

```
Definition mcons : val :=  
  val_new_hrecord_2 head tail.
```

```
Lemma triple_mcons :  $\forall (x q:\text{val})$ ,  
  triple (mcons x q)  
  \[]  
  (funloc p  $\Rightarrow$  p  $\rightsquigarrow$  '{ head := x ; tail := q }).
```

Proof using. intros. *applys* `xsimpl` `triple_new_hrecord_2`. Qed.

End RECORDINIT.

46.3 More Details

46.3.1 Extending *xapp* to Support Record Access Operations

The tactic *xapp* can be refined to automatically invoke the lemmas `triple_get_field_hrecord` and `triple_set_field_hrecord`, which involve preconditions of the form `hfields_lookup k kvs = Some v` and `hfields_update k v kvs = Some ks`, respectively.

The auxiliary lemmas reformulate the specification triples in weakest-precondition form. The premise takes the form $H ==> \exists kvs, (\text{hrecord } kvs p) \text{ * match } ... \text{ with } \text{Some } ... \Rightarrow ...$

This presentation enables using *xsimpl* to extract the description of the record, named `kvs`, before evaluating the `lookup` or `update` function for producing the suitable postcondition.

```
Lemma xapp_get_field_lemma :  $\forall H k p Q$ ,  
  H ==>  $\exists kvs, (\text{hrecord } kvs p) \text{ * }$   
  match hfields_lookup k kvs with  
  | None  $\Rightarrow$  \[False]  
  | Some v  $\Rightarrow$  ((fun r  $\Rightarrow$  \[r = v] \text{ * hrecord } kvs p) \text{ -* protect } Q) end  $\rightarrow$   
  H ==> wp (val_get_field k p) Q.
```

Proof using.

```
introv N. xchange N. intros kvs. cases (hfields_lookup k kvs).  
{ rewrite wp_equiv. applys triple_conseq_frame triple_get_field_hrecord.  
  xsimpl. intros r  $\rightarrow$ . xchange (qwand_specialize v). rewrite xpull. }  
Qed.
```

```
Lemma xapp_set_field_lemma :  $\forall H k p v Q$ ,  
  H ==>  $\exists kvs, (\text{hrecord } kvs p) \text{ * }$ 
```

```
  match hfields_update k v kvs with  
  | None  $\Rightarrow$  \[False]
```

```

| Some  $kvs'$   $\Rightarrow$  ((fun _  $\Rightarrow$  hrecord  $kvs' p$ ) \(*\!*\) protect  $Q$ ) end  $\rightarrow$ 
 $H ==> \text{wp}(\text{val\_set\_field } k p v) Q.$ 

```

Proof using.

```

introv N. xchange N. intros  $kvs$ . cases (hfields_update  $k v kvs$ ).
{ rewrite wp_equiv. applys  $\times$  triple_conseq_frame triple_set_field_hrecord.
  xsimpl. intros r. xchange (qwand_specialize r).
  { xpull. }
  { xpull. }
}
```

Qed.

```

Ltac xapp_nosubst_for_records tt ::= 
  first [ applys xapp_set_field_lemma; xsimpl; simpl; xapp_simpl
    | applys xapp_get_field_lemma; xsimpl; simpl; xapp_simpl ].
```

46.3.2 Deallocation Function for Lists

Recall that our Separation Logic set up enforces that all allocated data eventually gets properly deallocated. In what follows, we present a function for recursively deallocating an entire mutable list.

Module LISTDEALLOC.

Import ProgramSyntax RecordInit.

Open Scope trm_scope_ext.

The operation `mfree_list` deallocates all the cells in a given list. It is implemented as a recursive function that invokes `mfree_cell` on every cell that it traverses.

OCaml:

```
let rec mfree_list p = if p != null then begin let q = p.tail in delete p; mfree_list q end
```

Definition `mfree_list` : val :=

```

<{ fix 'f 'p =>
  let 'b = ('p != null) in
  if 'b then
    let 'q = 'p'.tail in
    delete 'p;
    'f 'q
  end }>.
```

The precondition of `mfree_list p` requires a full list `MList L p`. The postcondition is empty: the entire list is destroyed.

Exercise: 3 stars, standard, especially useful (`Triple_mfree_list`) Verify the function `mfree_list`. Hint: the overall pattern of the proof follows that used for the function `triple_mc当地`, from chapter Basic.

```

Lemma triple_mfree_list :  $\forall L p,$ 
  triple (mfree_list p)
  ( $\text{MList } L p$ )
```

```
(fun _ => []).
```

Proof using. Admitted.

□

End LISTDEALLOC.

46.4 Optional Material

The aim of this bonus section is to show how to establish the specifications presented in this chapter.

To that end, we consider an extended language, featuring the operations *val_alloc* and *val_dealloc*, which operates on blocks of cells.

Module REALIZATION.

46.4.1 Refined Source Language

We assume that every allocated block features a “header cell”, represented explicitly in the memory state.

To describe this header cell, we introduce a special value, written *val_header* k, where k denotes the length of the block that follows the header.

Parameter *val_header* : *nat* → *val*.

Note that values of the form *val_header* k should never be introduced by source terms. They are only introduced in heaps by the evaluation of *val_alloc*, and they should never leak in program terms.

The operation *val_alloc* k is specified as shown below. Starting from a state *sa*, it produces a state *sb* \u *sa* (i.e., the union of *sb* and *sa*), where *sb* consists of consecutive of k+1 consecutive cells. The head cell stores the special value *val_header* k, while the k following cells store the special value *val_uninit*, describing uninitialized cells.

```
Parameter eval_alloc : ∀ k n sa sb p,
  sb = Fmap.conseq (val_header k :: LibList.make k val_uninit) p →
  n = nat_to_Z k →
  p ≠ null →
  Fmap.disjoint sa sb →
  eval sa (val_alloc (val_int n)) (sb \u sa) (val_loc p).
```

The operation *val_dealloc* p is specified as shown below. Assume a state of the form *sb* \u *sa*, where *sb* consists of consecutive of k+1 consecutive cells. Assume the first of these cells to store the special value *val_header* k. Then, the deallocation operation removes the block described by *sb*, and leaves the state *sa*.

```
Parameter eval_dealloc : ∀ k vs sa sb p,
  sb = Fmap.conseq (val_header k :: vs) p →
  k = LibList.length vs →
```

```
Fmap.disjoint sa sb →
eval (sb \u sa) (val_dealloc (val_loc p)) sa val_unit.
```

Rather than extending the language with primitive operations for reading and writing in array cells and record fields, we simply include a pointer arithmetic operation, named *val_ptr_add*, and use it to encode all other access operations.

Parameter *val_ptr_add* : **prim**.

The operation *val_ptr* $p\ n$ applies to a pointer p and an integer n , and returns the address $p+n$.

```
Parameter eval_ptr_add : ∀ p1 p2 n s,
(p2:int) = p1 + n →
eval s (val_ptr_add (val_loc p1) (val_int n)) s (val_loc p2).
```

Note that the specification above allows the integer n to be negative, as long as $p+n$ is nonnegative. That said, thereafter we only apply *eval_ptr_add* to nonnegative arguments.

We also introduce the primitive operation *val_length* p , which returns the number stored in the header block at address p . This operation is useful to implement the function *val_array_length*, which reads the length of an array.

Parameter *val_length* : **prim**.

```
Parameter eval_length : ∀ s p k,
Fmap.indom s p →
(val_header k) = Fmap.read s p →
eval s (val_length (val_loc p)) s (val_int k).
```

46.4.2 Realization of *hheader*

The heap predicate *hheader* $k\ p$ describes a cell at location whose contents is the special value *val_header* k , and with the invariant that p is not null.

hheader $k\ p$ is defined as $p \sim\sim> (\text{val_header } k) \setminus^* \setminus [p \neq \text{null}]$.

Parameter *hheader_def* :

```
hheader = (fun (k:nat) (p:loc) ⇒ p \sim\sim> (val_header k) \setminus^* \setminus [p \neq \text{null}]).
```

Like other heap predicates, the definition of *hheader* is associated with an introduction and an elimination lemma.

```
Lemma hheader_intro : ∀ p k,
p ≠ null →
(hheader k p) (Fmap.single p (val_header k)).
```

Proof using.

```
introtv N. rewrite hheader_def. rewrite hstar_hpure_r.
split×. applys hsingle_intro.
```

Qed.

```
Lemma hheader_inv: ∀ h p k,
```

$(\text{hheader } k \ p) \ h \rightarrow$
 $h = \text{Fmap.single } p (\text{val_header } k) \wedge p \neq \text{null}.$

Proof using.

`introv E. rewrite hheader_def in E. rewrite hstar_hpure_r in E.
split×.`

Qed.

The heap predicate `hheader k p` captures the invariant $p \neq \text{null}$.

Lemma `hheader_not_null` : $\forall p \ k,$
 $\text{hheader } k \ p ==> \text{hheader } k \ p \setminus [p \neq \text{null}]$.

Proof using. `intros. rewrite hheader_def. xsimpl×. Qed.`

The definition of `hheader` is meant to show that one can prove all the specifications axiomatized so far. Note, however, that this definition should not be revealed to the end user. In other words, it should be made “strongly opaque”. (Technically, this could be achieved by means of a functor in Coq.)

Otherwise, the user could exploit the `val_set` operation to update the contents of a header field, replacing $p \rightsquigarrow (\text{val_header } k)$ with $p \rightsquigarrow v$ for another value v , thereby compromising the invariants of the memory model.

46.4.3 Introduction and Elimination Lemmas for `hcells` and `harray`

The heap predicates `hcells` and `harray` have their introduction and elimination lemmas stated as follows. These lemmas are useful for establishing the specifications of the allocation and of the deallocation operations.

Lemma `hcells_intro` : $\forall L \ p,$
 $(\text{hcells } L \ p) (\text{Fmap.conseq } L \ p).$

Proof using.

```
intros L. induction L as [|L']; intros; rew_listx.
{ applys hempty_intro. }
{ simpl. applys hstar_intro.
  { applys× hsingl_intro. }
  { applys IHL. }
  { applys Fmap.disjoint_single_conseq. left. math. } }
```

Qed.

Lemma `hcells_inv` : $\forall p \ L \ h,$
 $\text{hcells } L \ p \ h \rightarrow$
 $h = \text{Fmap.conseq } L \ p.$

Proof using.

```
introv N. gen p h. induction L as [|x L'];
  intros; rew_listx; simpls.
{ applys hempty_inv N. }
{ lets (h1&h2&N1&N2&N3&->): hstar_inv N. fequal.
```

```
{ applys hsingle_inv N1. }
{ applys IHL' N2. } }
```

Qed.

Lemma harray_intro : $\forall k p L,$
 $k = \text{length } L \rightarrow$
 $p \neq \text{null} \rightarrow$
 $(\text{harray } L p) (\text{Fmap.conseq} (\text{val_header } k :: L) p).$

Proof using.

```
intros E n. unfold harray. rew_listx. applys hstar_intro.
{ subst k. applys× hheader_intro. }
{ applys hcells_intro. }
{ applys disjoint_single_conseq. left. math. }
```

Qed.

Lemma harray_inv : $\forall p L h,$
 $(\text{harray } L p) h \rightarrow$
 $h = (\text{Fmap.conseq} (\text{val_header} (\text{length } L) :: L) p) \wedge p \neq \text{null}.$

Proof using.

```
intros N. unfolds harray. rew_listx.
lets (h1&h2&N1&N2&N3&&->): hstar_inv (rm N).
lets (N4&Np): hheader_inv (rm N1).
lets N2': hcells_inv (rm N2). subst×.
```

Qed.

46.4.4 Proving the Specification of Allocation and Deallocation

Following the usual pattern, we first establish a reasoning rule for allocation at the level of Hoare logic.

Lemma hoare_alloc_nat : $\forall H k,$
 $\text{hoare} (\text{val_alloc } k)$
 H
 $(\text{funloc } p \Rightarrow \text{harray} (\text{LibList.make } k \text{ val_uninit}) p \setminus* H).$

Proof using.

```
intros. intros h Hh. sets L: (LibList.make k val_uninit).
sets L': (val_header k :: L).
forwards¬ (p&Dp&Np): (Fmap.conseq_fresh null h L').
 $\exists ((\text{Fmap.conseq } L' p) \setminus u h) (\text{val\_loc } p).$  split.
{ applys¬ (@eval_alloc k). }
{ applys hexists_intro p. rewrite hstar_hpure_l. split×.
  { applys× hstar_intro. applys× harray_intro.
    subst L. rew_listx×. } }
```

Qed.

We then derive the Separation Logic reasoning rule.

```

Lemma triple_alloc_nat : ∀ k,
  triple (val_alloc k)
  \[]
  (funloc p ⇒ harray (LibList.make k val_uninit) p).

```

Proof using.

```

intros. intros H'. applys hoare_conseq.
{ applys hoare_alloc_nat H'. } { xsimpl. } { xsimpl×. }

```

Qed.

The two corollaries to `triple_alloc_nat` follow. The first one applies to an integer argument, as opposed to a natural number.

```

Lemma triple_alloc : ∀ n,
  n ≥ 0 →
  triple (val_alloc n)
  \[]
  (funloc p ⇒ harray (LibList.make (abs n) val_uninit) p).

```

Proof using.

```

introv N. rewrite ← (@abs_nonneg n) at 1; [|auto].
xapp triple_alloc_nat. xsimpl×.

```

Qed.

The second corollary weakens the postcondition by not specifying the contents of the allocated cells.

```

Lemma triple_alloc_array : ∀ n,
  n ≥ 0 →
  triple (val_alloc n)
  \[]
  (funloc p ⇒ ∃ L, \[n = length L] \* harray L p).

```

Proof using.

```

introv N. xapp triple_alloc. { auto. }
{ xpull. intros p. xsimpl×. { rewrite length_make. rewrite× abs_nonneg. } }

```

Qed.

We also establish the specification of deallocation, first w.r.t. Hoare triples, then w.r.t. Separation Logic triples.

```

Lemma hoare_dealloc : ∀ H L p,
  hoare (val_dealloc p)
  (harray L p \* H)
  (fun _ ⇒ H).

```

Proof using.

```

intros. intros h Hh. destruct Hh as (h1&h2&N1&N2&N3&N4). subst h.
∃ h2 val_unit. split; [|auto].
applys× eval_dealloc L. applys harray_inv N1.

```

Qed.

```

Lemma triple_dealloc : ∀ L p,
  triple (val_dealloc p)
    (harray L p)
    (fun _ => []).

```

Proof using.

```

intros. intros H'. applys hoare_conseq.
{ applys hoare_dealloc H'. } { xsimpl. } { xsimpl. }
Qed.

```

46.4.5 Splitting Lemmas for hcells

The description of the cells of an array can be split in pieces, allowing to describe only portions of the array.

```

Lemma hcells_nil_eq : ∀ p,
  hcells nil p = [].

```

Proof using. auto. Qed.

```

Lemma hcells_cons_eq : ∀ p x L,
  hcells (x :: L) p = (p ~~> x) \* hcells L (p + 1)%nat.

```

Proof using. intros. simpl. xsimpl. Qed.

```

Lemma hcells_one_eq : ∀ p x L,
  hcells (x :: nil) p = (p ~~> x).

```

Proof using. intros. rewrite hcells_cons_eq, hcells_nil_eq. xsimpl. Qed.

```

Lemma hcells_concat_eq : ∀ p L1 L2,
  hcells (L1 ++ L2) p = (hcells L1 p \* hcells L2 (length L1 + p)%nat).

```

Proof using.

```

intros. gen p. induction L1; intros; rew_list; simpl.
{ xsimpl. }
{ rewrite IHL1. math_rewrite (length L1 + (p + 1)) = S (length L1 + p))%nat.
  xsimpl. }

```

Qed.

In order to reason about the read or write operation on a specific cell, we need to isolate this cell from the other cells of the array. Then, after the operation, we need to fold back to the view on the entire array.

The isolation process is captured in a general way by the following “focus lemma”. It reads as follows. Assume $\text{hcells } L \ p$ to initially describe a segment of an array. Then, the k -th cell from this segment can be isolated as a predicate $(p+k) ~~> v$, where v denotes the k -th item of L , that is $\text{LibList.nth } k \ L$.

What remains of the heap can be described using the magic wand operator as $((p+k) ~~> v) \dashv^* (\text{hcells } L \ p)$, which captures the idea that when providing back the cell at location $p+k$, one regains the ownership of the original segment.

The following statement describes a focus lemma.

```

Parameter hcells_focus_read : ∀ k p v L,
  k < length L →
  v = LibList.nth k L →
  hcells L p ==>
    ((p+k)%nat ~~> v)
  ∗ ((p+k)%nat ~~> v ∗ hcells L p).

```

The above lemma is, however, limited to read operations. Indeed, it imposes the cell $(p+k) \sim\sim v$ to be merged back into the array segment, without modification to the original contents v .

The lemma can be generalized into a form that takes into account the possibility of folding back the array segment with a modified contents for the cell at $p+k$, described by $(p+k) \sim\sim w$, for any value w . The updated segment gets described as `update k w L`.

```

Lemma hcells_focus : ∀ k p L,
  k < length L →
  hcells L p ==>
    ((p+k)%nat ~~> LibList.nth k L)
  ∗ (∀ w, ((p+k)%nat ~~> w) ∗ hcells (LibList.update k w L) p).

```

Proof using.

```

intros E. gen k p. induction L as [|x L']; rew_list; intros.
{ false. math. }
{ simpl. rewrite nth_cons_case. case_if.
  { subst. math_rewrite (p + 0 = p)%nat. xsimpl. intros w.
    rewrite update_zero. rewrite hcells_cons_eq. xsimpl. }
  { forwards R: IHL' (k-1)%nat (p+1)%nat. { math. }
    math_rewrite ((p+1)+(k-1) = p+k)%nat in R. xchange (rm R).
    xsimpl. intros w. xchange (hforall_specialize w).
    rewrite update_cons_pos; [|math]. rewrite hcells_cons_eq. xsimpl. } }

```

Qed.

The above focus lemma immediately extends to a full array described in the form `harray L p`.

```

Lemma harray_focus : ∀ k p L,
  k < length L →
  harray L p ==>
    ((p+1+k)%nat ~~> LibList.nth k L)
  ∗ (∀ w, ((p+1+k)%nat ~~> w) ∗ harray (LibList.update k w L) p).

```

Proof using.

```

intros E. unfolds harray. xchanges (» hcells_focus E). intros w.
xchange (hforall_specialize w). xsimpl. rewrite length_update.

```

Qed.

46.4.6 Specification of Pointer Arithmetic

The operation `val_ptr_add p n` adds an integer `n` to the address `p`. This operation is used for encoding array and record operations by accessing cells at specific offsets.

The specification of `val_ptr_add` directly reformulates the evaluation rule. It is established following the usual pattern for primitive operations.

```
Lemma hoare_ptr_add : ∀ p n H,
  p + n ≥ 0 →
  hoare (val_ptr_add p n)
  H
  (fun r ⇒ \[r = val_loc (abs (p + n))] \* H).
```

Proof using.

```
intros N. intros s K0. ∃ s (val_loc (abs (p + n))). split.
{ applys eval_ptr_add. rewrite abs_nonneg; math. }
{ rewrite× hstar_hpure_l. }
```

Qed.

```
Lemma triple_ptr_add : ∀ p n,
  p + n ≥ 0 →
  triple (val_ptr_add p n)
  []
  (fun r ⇒ \[r = val_loc (abs (p + n))]).
```

Proof using.

```
intros N. unfold triple. intros H'.
applys× hoare_conseq hoare_ptr_add; xsimpl×.
```

Qed.

The following lemma specializes the specification for the case where the argument `n` is equal to a natural number `k`. This reformulation avoids the `abs` function, and is more practical for the encodings that we consider further in the subsequent sections.

```
Lemma triple_ptr_add_nat : ∀ p k,
  triple (val_ptr_add p k)
  []
  (fun r ⇒ \[r = val_loc (p+k)%nat]).
```

Proof using.

```
intros. applys triple_conseq triple_ptr_add. { math. } { xsimpl. }
{ xsimpl. intros. subst. fequals.
  applys eq_nat_of_eq_int. rewrite abs_nonneg; math. }
```

Qed.

46.4.7 Specification of the `length` Operation to Read the Header

To establish `triple_length`, we follow the same proof pattern as for `triple_get`.

```
Lemma eval_length_sep : ∀ s s2 p k,
```

```
s = Fmap.union (Fmap.single p (val_header k)) s2 →
eval s (val_length (val_loc p)) s (val_int k).
```

Proof using.

```
intros. forwards Dv: Fmap.indom_single p (val_header k).
applys eval_length.
{ applys Fmap.indom_union_l. }
{ rewrite Fmap.read_union_l. rewrite Fmap.read_single. }
```

Qed.

```
Lemma hoare_length : ∀ H k p,
hoare (val_length p)
((hheader k p) \* H)
(fun r ⇒ \[r = val_int k] \* (hheader k p) \* H).
```

Proof using.

```
intros. intros s K0. ∃ s (val_int k). split.
{ destruct K0 as (s1&s2&P1&P2&D&U). lets (E1&N): hheader_inv P1.
  subst s1. applys eval_length_sep U. }
{ rewrite hstar_hpure_l. }
```

Qed.

```
Lemma triple_length : ∀ k p,
triple (val_length p)
(hheader k p)
(fun r ⇒ \[r = val_int k] \* hheader k p).
```

Proof using.

```
intros. unfold triple. intros H'. applys hoare_conseq hoare_length; xsimpl.
```

Qed.

Hint Resolve triple_length : triple.

46.4.8 Encoding of Array Operations using Pointer Arithmetic

An access to the i -th cell of an array at location p can be encoded as an access to the cell at location $p+i+1$.

Module Export ARRAYACCESSDEF.

Import ProgramSyntax.

Open Scope wp_scope.

Let's first state and prove two auxiliary arithmetic lemmas that are needed in the proofs below.

```
Lemma abs_lt_inbound : ∀ i k,
0 ≤ i < nat_to_Z k →
(abs i < k).
```

Proof using.

```
intros N. apply lt_nat_of_lt_int. rewrite abs_nonneg; math.
```

Qed.

```
Lemma succ_int_plus_abs : ∀ p i,
  i ≥ 0 →
  ((p + 1 + abs i) = abs (nat_to_Z p + (i + 1)))%nat.
```

Proof using.

```
  introv N. rewrite abs_nat_plus_nonneg; [|math].
  math_rewrite (i+1 = 1 + i).
  rewrite ← succ_abs_eq_abs_one_plus; math.
```

Qed.

The length operation on an array, written `val_array_length` p , is encoded as `val_length` p , where `val_length` is the primitive operation for reading the contents of a header.

Definition `val_array_length` : `val` := `val_length`.

```
Lemma triple_array_length_header : ∀ k p,
  triple (val_array_length p)
  (hheader k p)
  (fun r ⇒ \[r = k] \* hheader k p).
```

Proof using. intros. applys triple_length. Qed.

```
Lemma triple_array_length : ∀ L p,
  triple (val_array_length p)
  (harray L p)
  (fun r ⇒ \[r = length L] \* harray L p).
```

Proof using.

```
  intros. unfold harray. applys triple_conseq_frame triple_length.
  { xsimpl. } { xsimpl. auto. }
```

Qed.

The get operation on an array, written `val_array_get` p i , is encoded as `val_get` ($p+i+1$).

```
Definition val_array_get : val :=
<{ fun 'p 'i ⇒
  let 'j = 'i + 1 in
  let 'n = val_ptr_add 'p 'j in
  val_get 'n }>.
```

```
Lemma triple_array_get : ∀ p i v L,
  0 ≤ i < length L →
  LibList.nth (abs i) L = v →
  triple (val_array_get p i)
  (harray L p)
  (fun r ⇒ \[r = v] \* harray L p).
```

Proof using.

```
  introv N E. xwp. xapp. xapp triple_ptr_add. { math. }
  xchange (@harray_focus (abs i) p L).
```

```

{ rew_listx. applys× abs_lt_inbound. }
sets w: (LibList.nth (abs i) L). rewrite succ_int_plus_abs; [|math].
xapp triple_get. xchange (hforall_specialize w). subst w.
rewrite update_nth_same. rewrite ← E. xsimpl×.
{ rew_listx. applys× abs_lt_inbound. }

```

Qed.

The set operation on an array, written `val_array_set p i v`, is encoded as `val_set (p+i+1) v`.

```

Definition val_array_set : val :=
<{ fun 'p 'i 'v =>
  let 'j = 'i + 1 in
  let 'n = val_ptr_add 'p 'j in
  val_set 'n 'v }>.

```

```

Lemma triple_array_set : ∀ p i v L,
0 ≤ i < length L →
triple (val_array_set p i v)
(harray L p)
(fun _ ⇒ harray (LibList.update (abs i) v L) p).

```

Proof using.

```

introv R. xwp. xpull. xapp. xapp triple_ptr_add. { math. }
xchange (@harray_focus (abs i) p L). { applys× abs_lt_inbound. }
rewrite succ_int_plus_abs; [|math].
xapp triple_set. auto. xchange (hforall_specialize v).

```

Qed.

End ARRAYACCESSDEF.

46.4.9 Encoding of Record Operations using Pointer Arithmetic

An access to the k-th field of a record at location p can be encoded as an access to the cell at location $p+k+1$.

Module Export FIELDACCESSDEF.

Import ProgramSyntax.

The get operation on a field, written $p'.k$, is encoded as `val_get (p+k+1)`.

```

Definition val_get_field (k:field) : val :=
<{ fun 'p =>
  let 'q = val_ptr_add 'p {nat_to_Z (k+1)} in
  val_get 'q }>.

```

```

Notation "t1 `.` f" :=
(val_get_field f t1)
(in custom trm at level 56, f at level 0, format "t1 `.` f").

```

```

Lemma triple_get_field : ∀ p k v,
  triple ((val_get_field k) p)
    (p'.k ~~> v)
    (fun r => \[r = v] \* (p'.k ~~> v)).

```

Proof using.

```

xwp. xapp triple_ptr_add_nat. unfold hfield.
math_rewrite (p+1+k = p+(k+1))%nat.
xapp triple_get. xsimpl×.

```

Qed.

The set operation on a field, written `Set p'.k := v`, is encoded as `val_set (p+k+1) v`.

Definition `val_set_field (k:field) : val :=`

```

<{ fun 'p 'v =>
  let 'q = val_ptr_add 'p {nat_to_Z (k+1)} in
  val_set 'q 'v }>.

```

```

Lemma triple_set_field : ∀ v1 p k v2,
  triple ((val_set_field k) p v2)
    (p'.k ~~> v1)
    (fun _ => p'.k ~~> v2).

```

Proof using.

```

intros. xwp. xapp triple_ptr_add_nat. unfold hfield.
math_rewrite (p+1+k = p+(k+1))%nat.
xapp triple_set. xsimpl×.

```

Qed.

```

Notation "t1 `.` f `:=` t2" :=
  (val_set_field f t1 t2)
  (in custom trm at level 56, f at level 0, format "t1 `.` f `:=` t2").

```

End FIELDACCESSDEF.

46.4.10 Specification of Record Operations w.r.t. `hfields` and `hrecord`

The specifications `triple_get_field` and `triple_set_field` established above correspond to small-footprint specifications, with preconditions mentioning a single field. Corollaries to these specifications can be established with preconditions mentioning a list of fields (predicate `hfields`), or a full record (predicate `hrecord`).

Get operation on a list of fields

```

Lemma triple_get_field_hfields : ∀ kvs p k v,
  hfields_lookup k kvs = Some v →
  triple (val_get_field k p)
    (hfields kvs p)
    (fun r => \[r = v] \* hfields kvs p).

```

Proof using.

```

intros L. induction L as [| [ki vi] L']; simpl; introv E.
{ inverts E. }
{ case_if.
  { inverts E. subst ki. applys triple_conseq_frame.
    { applys triple_get_field. } { xsimpl. } { xsimplx. } }
  { applys triple_conseq_frame.
    { applys IH $L'$  E. }
    { xsimpl. }
    { xsimplx. } } }

```

Qed.

Get operation on a record

```

Lemma triple_get_field_hrecord : ∀ kvs p k v,
  hfields_lookup k kvs = Some v →
  triple (val_get_field k p)
  (p ~~~> kvs)
  (fun r ⇒ \[r = v] \* p ~~~> kvs).

```

Proof using.

```

introv M. unfold hrecord. xtriple. xpull. intros z Hz.
xapp (» triple_get_field_hfields M). xsimplx.

```

Qed.

Set operation on a list of fields

```

Lemma triple_set_field_hfields : ∀ kvs kvs' k p v,
  hfields_update k v kvs = Some kvs' →
  triple (val_set_field k p v)
  (hfields kvs p)
  (fun _ ⇒ hfields kvs' p).

```

Proof using.

```

intros kvs. induction kvs as [| [ki vi] kvs']; simpl; introv E.
{ inverts E. }
{ case_if.
  { inverts E. subst ki. applys triple_conseq_frame.
    { applys triple_set_field. } { xsimpl. } { xsimplx. } }
  { cases (hfields_update k v kvs') as C2; tryfalse. inverts E.
    applys triple_conseq_frame. { applys IH $kvs'$  C2. }
    { xsimpl. } { simpl. xsimplx. } } }

```

Qed.

Auxiliary lemma about `hfields_update`, showing that the update operation preserves the names of the fields.

```

Lemma hfields_update_preserves_fields : ∀ kvs kvs' k v,
  hfields_update k v kvs = Some kvs' →
  LibList.map fst kvs' = LibList.map fst kvs.

```

Proof using.

```
intros kvs. induction kvs as [[ki vi] kvs1]; simpl; introv E.
{ introv _ H. inverts H. }
{ case_if.
  { inverts E. rew_listx. subst. fequals. }
  { cases (hfields_update k v kvs1).
    { inverts E. rew_listx. fequals. }
    { inverts E. } } }
```

Qed.

Lemma hfields_update_preserves_maps_all_fields : $\forall kvs\ kvs' z k v,$
 $\text{hfields_update } k v kvs = \text{Some } kvs' \rightarrow$
 $\text{maps_all_fields } z kvs = \text{maps_all_fields } z kvs'.$

Proof using.

```
introv M. unfold maps_all_fields. extens.
rewrites  $\times (\gg \text{hfields\_update\_preserves\_fields } M).$ 
```

Qed.

Set operation on a record

Lemma triple_set_field_hrecord : $\forall kvs\ kvs' k p v,$
 $\text{hfields_update } k v kvs = \text{Some } kvs' \rightarrow$
 $\text{triple } (\text{val_set_field } k p v)$
 $(p \rightsquigarrow kvs)$
 $(\text{fun } _ \Rightarrow p \rightsquigarrow kvs').$

Proof using.

```
introv M. unfold hrecord. xtriple. xpull. intros z Hz.
xapp ( $\gg \text{triple\_set\_field\_hfields } M).$  xsimpl.
rewrites  $\times \leftarrow (\gg \text{hfields\_update\_preserves\_maps\_all\_fields } z M).$ 
```

Qed.

46.4.11 Specification of Record Allocation and Deallocation

Recall that record allocation, written `val_alloc_hrecord ks`, is encoded as `val_alloc (length ks)`, and that record deallocation, written `val_dealloc_hrecord p`, is encoded as `val_dealloc p`.

The operations `val_alloc` and `val_dealloc` have already been specified, via lemmas `triple_alloc` and `triple_dealloc`.

In this section, we show that the specifications for `val_alloc_hrecord` and `val_dealloc_hrecord` can be derived from those.

The proofs are not completely trivial because we need to convert between the view of a list of consecutive cells, as captured by the predicate `hcells`, and the view of a list of fields, as captured by the predicate `hfields`.

To that end, the lemma `hfields_eq_hcells` asserts the equality between `hfields kvs p` and `hcells L (p+1)` under a suitable relation between the list of values `L` and the description of

the fields kvs : the values from the association list kvs must correspond to the list L , and the keys from the association list kvs must correspond to the `length L` first natural numbers.

The proof is carried out by induction, on the following statement.

```
Lemma hfields_eq_hcells_ind : ∀ L p kvs o,
  LibList.map fst kvs = nat_seq o (length L) →
  L = LibList.map snd kvs →
  hfields kvs p = hcells L (p+1+o)%nat.
```

`Proof using.`

```
intros L. induction L as [|v L']; introv M E; rew_listx in *;
destruct kvs as [|[k v'] kvs']; tryfalse; rew_listx in *.
{ auto. }
{ simpls. inverts M as M'. inverts E as E'. rew_listx in *.
unfold hfield. fequals.
math_rewrite (p+1+o+1=p+1+(o+1))%nat. applys× IHL'.
math_rewrite× (o+1=S o)%nat. }
```

`Qed.`

The statement of the lemma `hfields_eq_hcells` involves the predicate `maps_all_fields`, which appears in the definition of `hrecord`.

```
Lemma hfields_eq_hcells : ∀ z L p kvs,
  maps_all_fields z kvs →
  z = length L →
  L = LibList.map snd kvs →
  hfields kvs p = hcells L (p+1)%nat.
```

`Proof using.`

```
introv M HL E. subst z. rewrites (» hfields_eq_hcells_ind M E).
fequals. unfold loc. math.
```

`Qed.`

The following lemma converts from the `harray` view to the `hrecord` view, for an array of uninitialized values.

```
Lemma harray_uninit_himpl_hrecord : ∀ p z,
  harray (LibList.make z val_uninit) p ==>
  hrecord (LibList.map (fun k => (k, val_uninit)) (nat_seq 0 z)) p.
```

`Proof using.`

```
intros. unfolds hrecord, harray. rew_listx.
sets kvs: ((LibList.map (fun k => (k, val_uninit)) (nat_seq 0 z))).
asserts (M1&M2): (LibList.map fst kvs = nat_seq 0 z
  ∧ LibList.map snd kvs = LibList.make z val_uninit).
{ subst kvs. generalize 0%nat as o.
  induction z as [|z']; intros; simpl; rew_listx; simpl.
  { auto. }
  { forwards× (N1&N2): IHz' (S o). split; fequals×. } }
```

```

 $xsimpl z.$ 
{ applies M1. }
{ rewrites  $\leftarrow (\gg \text{hfields\_eq\_hcells } M1)$ . rew_listx. }

```

Qed.

Exploiting the above lemma, we derive the specification of record allocation.

```

Lemma triple_alloc_hrecord :  $\forall ks,$ 
 $ks = \text{nat\_seq } 0 (\text{LibListExec.length } ks) \rightarrow$ 
triple (val_alloc_hrecord ks)
 $\backslash []$ 
 $(\text{funloc } p \Rightarrow \text{hrecord} (\text{LibListExec.map} (\text{fun } k \Rightarrow (k, \text{val\_uninit})) ks) p).$ 

```

Proof using.

```

intros E. xapp triple_alloc_nat. intros p.
xsimpl p. { auto. } rewrite E. rewrite length_nat_seq.
rewrite LibListExec.map_eq. xchange harray_uninit_himpl_hrecord.

```

Qed.

The following lemma converts in the other direction, from the `hrecord` view to the `harray` view.

```

Lemma hrecord_himpl_harray :  $\forall p kvs,$ 
 $\text{hrecord } kvs p ==> \text{harray} (\text{LibList.map } \text{snd } kvs) p.$ 

```

Proof using.

```

intros. unfolds hrecord, harray. xpull. intros z M.
asserts Hz: ( $z = \text{length } kvs$ ).
{ lets E:  $\text{length\_nat\_seq } 0 \% \text{nat } z$ . rewrite  $\leftarrow M$  in E. rew_listx in *. }
xchange  $\leftarrow (\gg \text{hfields\_eq\_hcells } p M)$ . { rew_listx. }
xsimpl. rew_listx. rewrite Hz.

```

Qed.

Exploiting the above lemma, we derive the specification of record deallocation.

```

Lemma triple_dealloc_hrecord :  $\forall kvs p,$ 
triple (val_dealloc p)
 $(\text{hrecord } kvs p)$ 
 $(\text{fun } _ \Rightarrow \backslash []).$ 

```

Proof using.

```

intros. xtriple. xchange hrecord_himpl_harray.
xapp triple_dealloc. xsimpl.

```

Qed.

End REALIZATION.

Chapter 47

Library SLF.Rich

47.1 Rich: Assertions, Loops, N-ary Functions

Set Implicit Arguments.

From *SLF* Require Import LibSepReference LibSepTLCbuffer.

From *SLF* Require Basic Repr.

Implicit Types P : Prop.

Implicit Types H : hprop.

Implicit Types Q : val \rightarrow hprop.

Implicit Type $p\ q$: loc.

Implicit Type k : nat.

Implicit Type $i\ n$: int.

Implicit Type v : val.

Implicit Types b : bool.

Implicit Types L : list val.

47.2 First Pass

This chapter introduces support for additional language constructs:

- assertions,
- if-statements that are not in A-normal form (needed for loops)
- while-loops,
- n-ary functions (bonus contents).

Regarding assertions, we present a reasoning rule such that:

- assertions may be expressed in terms of mutable data, and possibly to perform local side-effects, and

- the program remains correct whether assertions are executed or not.

Regarding loops, we explain why traditional Hoare-style reasoning rules based on loop invariants are limited, in that they prevent useful applications of the frame rule. We present alternative reasoning rules compatible with the frame rule, and demonstrate their benefits on practical examples.

Regarding n-ary functions, that is, functions with several arguments, there are essentially three possible approaches:

- native n-ary functions, e.g., *function(x, y) { t }* in C syntax;
- curried functions, e.g., `fun x => fun y => t` in OCaml syntax;
- tupled functions, e.g., `fun (x,y) => t` in OCaml syntax.

In this chapter, we describe the first two approaches. The third approach (tupled functions) involves algebraic data types, which are beyond the scope of this course.

47.2.1 Reasoning Rule for Assertions

Module ASSERTIONS.

Assume an additional primitive operation, allowing to write terms of the form `val_assert t`, to dynamically check at runtime that the term *t* produces the boolean value `true`, equivalently to the OCaml expression `assert(t)`.

Parameter `val_assert : prim`.

The reasoning rule for assertions should ensure that:

- the body of the assertion always evaluates to `true`,
- the program remains correct if the assertion is not evaluated.

Let us introduce a boolean flag to control the evaluation of assertions. The value of this flag is purposely left unspecified.

Parameter `evaluate_assertions : bool`.

The program should be correct for both possible values of `evaluate_assertions`. The semantics of the evaluation of assertions is formally captured by the following two rules.

- The rule `eval_assert_enabled` applies when `evaluate_assertions` is set to `true`. The rule describes the evaluation of the body of the assertion, and the check that the output value is `true`.
- The rule `eval_assert_disabled` applies when `evaluate_assertions` is set to `false`. The rule totally ignores the body of the assertion. It treats the assertion as immediately returning the `unit` value, in the current state.

```

Parameter eval_assert_enabled : ∀ s t s',
  evaluate_assertions = true →
  eval s t s' (val_bool true) →
  eval s (val_assert t) s' val_unit.

```

```

Parameter eval_assert_disabled : ∀ s t,
  evaluate_assertions = false →
  eval s (val_assert t) s val_unit.

```

Note that it might be tempting to consider a “unifying” evaluation rule that evaluates the body of the assertion, checks that the result is `true`, and, moreover, imposes that the assertion does not modify the state.

```

Parameter eval_assert_no_effect : ∀ s t v,
  eval s t s (val_bool true) →
  eval s (val_assert t) s val_unit.

```

Yet, such a rule would be overly restrictive, for two reasons. First, it might be useful for an assertion to allocate local data for evaluating a particular property. Second, there are useful examples of assertions that do modify existing cells from the heap. For example, an assertion that appears in real programs is `assert (find x = find y)`, where the `find` operation finds the representative of a node from a Union-Find data structure, an operation that performs path compression.

At this point, our goal is to state a reasoning rule for `val_assert t` that does not depend on the value of the boolean flag `evaluate_assertions`. In other words, we are seeking for a rule that is correct both with respect to `eval_assert_enabled` and with respect to `eval_assert_disabled`.

Consider the evaluation of `val_assert t` in a state described by H . The output state should still be described by H , although the internal representation of the state might have changed (e.g., for path compression in the case of a Union-Find data structure). The body of the assertion should evaluate correctly in a state described by H , and should return the value `true`, in an output state still described by H .

As usual for primitive operations, we first establish a rule for Hoare triples, then deduce a rule for Separation Logic triples.

```

Lemma hoare_assert : ∀ t H,
  hoare t H (fun r ⇒ \[r = true] \* H) →
  hoare (val_assert t) H (fun _ ⇒ H).

```

Proof using.

```

intros M. intros s K. forwards (s'&v&R&N): M K.
rewrite hstar_hpure_l in N. destruct N as (->&K').
case_eq evaluate_assertions; intros C.
{ ∃ s' val_unit. split.
  { applys eval_assert_enabled C R. }
  { applys K'. } }
{ ∃ s val_unit. split.
  { applys eval_assert_disabled C R. }
  { applys K'. } }

```

```
{ ∃ s val_unit. split.
```

```
{ applies eval_assert_disabled C. }
{ applies K. } }
```

Qed.

```
Lemma triple_assert : ∀ t H,
  triple t H (fun r ⇒ \[r = true] \* H) →
  triple (val_assert t) H (fun _ ⇒ H).
```

Proof using.

```
intros M. intros H'. specializes M H'. applies hoare_assert.
  applies hoare_conseq M. { xsimpl. } { xsimpl. auto. }
```

Qed.

End ASSERTIONS.

47.2.2 Semantics of Conditionals not in Administrative Normal Form

To state reasoning rule for while-loops in a concise manner, it is useful to generalize the construct `if b then t1 else t2` to the form `if t0 then t1 else t2`.

To specify the behavior of a term of the form `if t0 then t1 else t2`, let us assume the evaluation rule shown below. This rule generalizes the rule `eval_if`. It first evaluates `t0` into a value `v0`, then it evaluates the term `if v0 then t1 else t2`.

```
Parameter eval_if_trm : ∀ s1 s2 s3 v0 v t0 t1 t2,
  eval s1 t0 s2 v0 →
  eval s2 (trm_if v0 t1 t2) s3 v →
  eval s1 (trm_if t0 t1 t2) s3 v.
```

With respect to this evaluation rule `eval_if_trm`, we can prove a corresponding reasoning rule. We first state it in Hoare-logic, then in Separation Logic, following the usual proof pattern.

```
Lemma hoare_if_trm : ∀ Q' t0 t1 t2 H Q,
  hoare t0 H Q' →
  (forall v, hoare (trm_if v t1 t2) (Q' v) Q) →
  hoare (trm_if t0 t1 t2) H Q.
```

Proof using.

```
intros M1 M2. intros s1 K1. lets (s2&v0&R2&K2): M1 K1.
forwards (s3&v&R3&K3): M2 K2. ∃ s3 v. splits.
{ applies eval_if_trm R2 R3. }
{ applies K3. }
```

Qed.

```
Lemma triple_if_trm : ∀ Q' t0 t1 t2 H Q,
  triple t0 H Q' →
  (forall v, triple (trm_if v t1 t2) (Q' v) Q) →
  triple (trm_if t0 t1 t2) H Q.
```

Proof using.

```

introv M1 M2. intros HF. applys hoare_if_trm (Q' \*+ HF).
{ applys hoare_conseq. applys M1 HF. { xsimpl. } { xsimpl. } }
{ intros v. applys M2. }

```

Qed.

The reasoning rule can also be reformulated in weakest-precondition form. The rule below generalizes the rule *wp_if*.

Lemma *wp_if_trm* : $\forall t0\ t1\ t2\ Q,$

$$\text{wp } t0 (\text{fun } v \Rightarrow \text{wp } (\text{trm_if } v\ t1\ t2)\ Q) ==> \text{wp } (\text{trm_if } t0\ t1\ t2)\ Q.$$

Proof using.

```

intros. unfold wp. xsimpl; intros H M H'. applys hoare_if_trm.
{ applys M. }
{ intros v. simpl. rewrite hstar_hexists. applys hoare_hexists. intros HF.
  rewrite (hstar_comm HF). rewrite hstar_assoc. applys hoare_hpure.
  intros N. applys N. }

```

Qed.

47.2.3 Semantics and Basic Evaluation Rules for While-Loops

Module WHILELOOPS.

Assume the grammar of term to be extended with a loop construct *trm_while* $t1\ t2$, corresponding to the OCaml expression *while* $t1$ do $t2$, and written *While* $t1$ *Do* $t2$ *Done* in our example programs.

Parameter *trm_while* : $\text{trm} \rightarrow \text{trm} \rightarrow \text{trm}.$

Notation "'while' $t1$ 'do' $t2$ 'done'" :=

```

  (trm_while t1 t2)
  (in custom trm at level 0,
  t1 custom trm at level 99,
  t2 custom trm at level 99,
  format "[v' 'while' t1 'do' '/ '[' t2 ']' '/ 'done' ']'")
  : trm_scope.

```

The semantics of this loop construct can be described in terms of the one-step unfolding of the loop: *while* $t1$ do $t2$ is a term that behaves exactly like the term *if* $t1$ then ($t2$; *while* $t1$ do $t2$) *else* () .

Parameter *eval_while* : $\forall s1\ s2\ t1\ t2\ v,$

```

  eval s1 (trm_if t1 (trm_seq t2 (trm_while t1 t2)) val_unit) s2 v ->
  eval s1 (trm_while t1 t2) s2 v.

```

This evaluation rule translates directly into a reasoning rule: to prove a triple for the term *while* $t1$ do $t2$, it suffices to prove a triple for the term *if* $t1$ then ($t2$; *while* $t1$ do $t2$) *else* () .

There is a catch in that reasoning principle, namely the fact that the loop *while* t_1 do t_2 appears again inside the term *if* t_1 then (t_2 ; *while* t_1 do t_2) *else* (). Nevertheless, this is not a problem if the user is carrying out a proof by induction. In that case, an induction hypothesis about the behavior of *while* t_1 do t_2 is available. We show an example proof further on.

For the moment, let us state the reasoning rules.

Lemma hoare_while : $\forall t_1 t_2 H Q,$
 $\text{hoare}(\text{trm_if } t_1 (\text{trm_seq } t_2 (\text{trm_while } t_1 t_2)) \text{ val_unit}) H Q \rightarrow$
 $\text{hoare}(\text{trm_while } t_1 t_2) H Q.$

Proof using.

intros $M K.$ *forwards* \times ($s' \& v \& R1 \& K1$): $M.$
 $\exists s' v.$ *splits* $\times.$ { *applys* \times *eval_while.* }

Qed.

Lemma triple_while : $\forall t_1 t_2 H Q,$
 $\text{triple}(\text{trm_if } t_1 (\text{trm_seq } t_2 (\text{trm_while } t_1 t_2)) \text{ val_unit}) H Q \rightarrow$
 $\text{triple}(\text{trm_while } t_1 t_2) H Q.$

Proof using.

intros $M.$ *intros* $H'.$ *apply* *hoare_while.* *applys* $\times M.$

Qed.

Lemma wp_while : $\forall t_1 t_2 Q,$
 $\text{wp}(\text{trm_if } t_1 (\text{trm_seq } t_2 (\text{trm_while } t_1 t_2)) \text{ val_unit}) Q$
 $\implies \text{wp}(\text{trm_while } t_1 t_2) Q.$

Proof using.

intros. *repeat unfold wp.* *xsimpl;* *intros* $H M H'.$
applys *hoare_while.* *applys* $M.$

Qed.

47.2.4 Separation-Logic-friendly Reasoning Rule for While-Loops

One may be tempted to introduce an invariant-based reasoning rule for while loops. Traditional invariant-based rules from Hoare logic usually admit a relatively simple statement, because they only target partial correctness, and because the termination condition is restricted to a simple expression that does not alter the state.

In our set up, targeting total correctness and a construct of the form *trm_while* t_1 t_2 , the invariant-based reasoning rule can be stated as shown below.

An invariant I describes the state at the entry and exit point of the loop. This invariant is actually of the form $I \mathbf{b} X$, where the boolean value \mathbf{b} is *false* to indicate that the loop has terminated, and where the value X belongs to a type A used to express the decreasing measure that justifies the termination of the loop.

Lemma triple_while_inv_not_framable :
 $\forall (A:\text{Type}) (I:\text{bool} \rightarrow A \rightarrow \text{hprop}) (R:A \rightarrow A \rightarrow \text{Prop}) t_1 t_2,$

$\text{wf } R \rightarrow$
 $(\forall b X, \text{triple } t1 (I b X) (\text{fun } r \Rightarrow \exists b', \forall [r = b'] \ast I b' X)) \rightarrow$
 $(\forall b X, \text{triple } t2 (I b X) (\text{fun } _- \Rightarrow \exists b' Y, \forall [R Y X] \ast I b' Y)) \rightarrow$
 $\text{triple } (\text{trm_while } t1 t2) (\exists b X, I b X) (\text{fun } r \Rightarrow \exists Y, I \text{false } Y).$

Proof using.

```

introv WR M1 M2. applys triple_hexists. intros b.
applys triple_hexists. intros X.
gen b. induction_wf IH: WR X. intros. applys triple_while.
applys triple_if_trm (fun (r:val) => \exists b', \forall [r = b'] \ast I b' X).
{ applys M1. }
{ intros v. applys triple_hexists. intros b'. applys triple_hpure. intros ->.
  applys triple_if. case_if.
  { applys triple_seq.
    { applys M2. }
    { applys triple_hexists. intros b''. applys triple_hexists. intros Y.
      applys triple_hpure. intros HR. applys IH HR. } }
  { applys triple_val. xsimpl. } }

```

Qed.

The above rule is correct yet limited, because it precludes the possibility to apply the frame rule “over the remaining iterations of the loop”.

This possibility can be exploited by carrying a proof by induction and invoking the rule `triple_while`, which unfolds the loop body. In that scheme, the frame rule can be applied to the term `while t1 do t2` that occurs in the `if t1 then (t2; (while t1 do t2)) else ()`.

We can reduce the noise associated with applying the rule `triple_while` by assigning a name, say `t`, to denote the term `while t1 do t2`. The corresponding rule, shown below, asserts that `t` admits the same behavior as the term `if t1 then (t2; t) else ()`.

Lemma `triple_while'` : $\forall t1 t2 H Q,$

```

(\forall t,
  (\forall H' Q', \text{triple } (\text{trm\_if } t1 (\text{trm\_seq } t2 t) \text{val\_unit}) H' Q' ->
   triple t H' Q') ->
  triple t H Q) ->
triple (trm_while t1 t2) H Q.

```

Proof using.

introv M. applys M. introv K. applys triple_while. applys K.

Qed.

The proof scheme that consists of setting up an induction then applying the reasoning rule `triple_while'` can be factored into the following lemma, which is stated using an invariant that appears only in the precondition. The postcondition is an abstract `Q`. With this presentation, the rule features an “invariant”, yet it remains possible to invoke the frame rule over the “remaining iterations” of the loop.

Lemma `triple_while_inv` :

```

 $\forall (A:\text{Type}) (I:\text{bool} \rightarrow A \rightarrow \text{hprop}) (R:A \rightarrow A \rightarrow \text{Prop}) t1\ t2,$ 
 $\text{let } Q := (\text{fun } r \Rightarrow \exists Y, I \text{ false } Y) \text{ in}$ 
 $\text{wf } R \rightarrow$ 
 $(\forall t b X,$ 
 $(\forall b' Y, R\ Y\ X \rightarrow \text{triple } t\ (I\ b'\ Y)\ Q) \rightarrow$ 
 $\text{triple } (\text{trm\_if } t1\ (\text{trm\_seq } t2\ t) \text{ val\_unit})\ (I\ b\ X)\ Q) \rightarrow$ 
 $\text{triple } (\text{trm\_while } t1\ t2)\ (\exists b X, I\ b\ X)\ Q.$ 

```

Proof using.

```

introv WR M. applys triple_hexists. intros b0.
applys triple_hexists. intros X0.
gen b0. induction_wf IH: WR X0. intros. applys triple_while.
applys M. intros b' Y HR'. applys IH HR'.

```

Qed.

The rule `triple_while_inv` admits a constrained precondition of the form $(\exists b X, I b X)$. To exploit this rule, one almost always needs to first invoke the consequence-frame rule.

The rule `triple_while_inv_conseq_frame`, shown below, conveniently bakes in frame and consequence rules into the statement of `triple_while_inv`.

Lemma `triple_while_inv_conseq_frame` :

```

 $\forall (A:\text{Type}) (I:\text{bool} \rightarrow A \rightarrow \text{hprop}) (R:A \rightarrow A \rightarrow \text{Prop}) H' t1\ t2\ H\ Q,$ 
 $\text{let } Q' := (\text{fun } r \Rightarrow \exists Y, I \text{ false } Y) \text{ in}$ 
 $\text{wf } R \rightarrow$ 
 $(H ==> (\exists b X, I b X) \nast H') \rightarrow$ 
 $(\forall t b X,$ 
 $(\forall b' Y, R\ Y\ X \rightarrow \text{triple } t\ (I\ b'\ Y)\ Q') \rightarrow$ 
 $\text{triple } (\text{trm\_if } t1\ (\text{trm\_seq } t2\ t) \text{ val\_unit})\ (I\ b\ X)\ Q') \rightarrow$ 
 $Q' \nast H' ==> Q \rightarrow$ 
 $\text{triple } (\text{trm\_while } t1\ t2)\ H\ Q.$ 

```

Proof using.

```

introv WR WH M WQ. applys triple_conseq_frame WH WQ.
applys triple_while_inv WR M.

```

Qed.

The above rule can be equivalently reformulated in weakest-precondition style.

Lemma `wp_while_inv_conseq` :

```

 $\forall (A:\text{Type}) (I:\text{bool} \rightarrow A \rightarrow \text{hprop}) (R:A \rightarrow A \rightarrow \text{Prop}) t1\ t2,$ 
 $\text{let } Q := (\text{fun } r \Rightarrow \exists Y, I \text{ false } Y) \text{ in}$ 
 $\text{wf } R \rightarrow$ 
 $(\exists b X, I b X)$ 
 $\nast \nabla [\forall t b X,$ 
 $(\forall b' Y, R\ Y\ X \rightarrow (I\ b'\ Y) ==> \text{wp } t\ Q) \rightarrow$ 
 $(I\ b\ X) ==> \text{wp } (\text{trm\_if } t1\ (\text{trm\_seq } t2\ t) \text{ val\_unit})\ Q]$ 
 $==> \text{wp } (\text{trm\_while } t1\ t2)\ Q.$ 

```

Proof using.

```
intros WR. sets H: ( $\exists b X$ ,  $I b X$ ). xpull. intros M.
rewrite wp_equiv. applys triple_while_inv WR.
intros N. rewrite ← wp_equiv. applys M.
intros HR. rewrite wp_equiv. applys N HR.
```

Qed.

47.3 More Details

47.3.1 Semantics and Basic Evaluation Rules for For-Loops

Module FORLOOPS.

In the previous section, we have focused on while loops and presented encodings, evaluation rules, and reasoning rules. In this section, we present a similar construction for for-loops.

A for loop takes the form $trm_for \ x \ n1 \ n2 \ t3$, matching the OCaml syntax `for x = n1 to n2 do t3 done`. We assume the bounds $n1$ and $n2$ to be already evaluated to integers—i.e., we assume A-normal form.

Parameter $trm_for : \text{var} \rightarrow \text{trm} \rightarrow \text{trm} \rightarrow \text{trm} \rightarrow \text{trm}$.

```
Notation "'for' x '=' t1 'to' t2 'do' t3 'done'" :=
  (trm_for x t1 t2 t3)
  (in custom trm at level 0,
  x at level 0,
  t1 custom trm at level 99,
  t2 custom trm at level 99,
  t3 custom trm at level 99,
  format "[v] 'for' x '=' t1 'to' t2 'do' /' [t3]' /' 'done' ']")
  : trm_scope.
```

Close Scope trm_scope .

The semantics of `for x = n1 to n2 do t3` is encoded using a conditional. If $n1 \leq n2$, then we execute the body $t3$ with x bound to $n1$ (that is, `subst x n1 t3`), and continue with the remainig iterations, which are described by `for x = n1+1 to n2 do t3 done`. Eventually, we reach a situation where $n1 > n2$. At this point, the loop terminates, and returns the unit value.

```
Parameter eval_for :  $\forall s1 s2 x n1 n2 t3 v$ ,
  eval s1 (trm_if (val_le n1 n2) (trm_seq (trm_let x n1 t3)
    (trm_for x (n1+1) n2 t3)) val_unit) s2 v →
  eval s1 (trm_for x n1 n2 t3) s2 v.
```

The direct reasoning rules for for-loops w.r.t. Hoare triples and Separation Logic triples simply unfold the encoding.

```

Lemma hoare_for : ∀ x n1 n2 t3 H Q,
  hoare (trm_if (val_le n1 n2) (trm_seq (trm_let x n1 t3)
    (trm_for x (n1+1) n2 t3)) val_unit) H Q →
  hoare (trm_for x n1 n2 t3) H Q.

```

Proof using.

```

intros M K forwards× (s'&v&R1&K1): M.
∃ s' v. splits×. { applys× eval_for. }

```

Qed.

```

Lemma triple_for : ∀ x n1 n2 t3 H Q,
  triple (trm_if (val_le n1 n2) (trm_seq (trm_let x n1 t3)
    (trm_for x (n1+1) n2 t3)) val_unit) H Q →
  triple (trm_for x n1 n2 t3) H Q.

```

Proof using. intros M. intros H'. apply hoare_for. applys× M. Qed.

Just like for while loops, we can reduce the noise associated with applying the rule triple_for, by pre-processing the encoding based on the conditional. This rule may be useful for carrying out proofs by induction.

Follow carefully through the proof, as it exhibits reasoning patterns that are useful for the follow-up exercises.

```

Lemma triple_for' : ∀ x n1 n2 t3 H Q,
  (∀ (tloop:int→trm) (i:int),
    (∀ i H' Q',
      (If i ≤ n2
        then (∃ H1, triple (subst x i t3) H' (fun v ⇒ H1)
          ∧ triple (tloop (i+1)) H1 Q')
        else (H' ==> Q' val_unit)) →
      triple (tloop i) H' Q') →
    triple (tloop i) H Q) →
  triple (trm_for x n1 n2 t3) H Q.

```

Proof using.

```

intros M. applys M (fun i ⇒ trm_for x i n2 t3). intros K.
applys triple_for. applys triple_if_trm.
{ applys triple_conseq_frame triple_le. xsimpl. xsimpl. }
{ intros v. applys triple_hpure. intros →. applys triple_if.
  do 2 case_if; try solve [ false; math ]. }
{ forwards (H1&K1&K2):K. applys triple_seq H1.
  { applys triple_let_val. applys K1. }
  { applys K2. } }
{ applys triple_val. applys K. } }

```

Qed.

We can derive a simpler rule for the case of loops with no iterations.

Exercise: 4 stars, standard, optional (triple_for_ge) Prove the specification of for loops in the case where the lower bound of the loop exceeds the upper bound of the loop. Hint: the proof shares similarities with that of triple_for'.

Lemma triple_for_ge : $\forall x n1 n2 t3 H Q,$

$n1 > n2 \rightarrow$

$H ==> Q \text{ val_unit} \rightarrow$

$\text{triple } (\text{trm_for } x n1 n2 t3) H Q.$

Proof using. Admitted.

□

We can derive an reasoning rule expressed using a loop invariant: if $I \ n1$ holds before the loop, and each iteration takes the state from $I \ i$ to $I \ (i+1)$, then $I \ (n2+1)$ holds after the last iteration. The benefits of this presentation, compared with triple_for', is that it bakes in the application of the induction principle.

Exercise: 5 stars, standard, especially useful (triple_for_inv_conseq_frame) Prove the invariant-based reasoning rule for for-loops. Hint: the proof shares similarities with that of triple_for'.

Lemma triple_for_inv_conseq_frame : $\forall (I:\text{int} \rightarrow \text{hprop}) H' x n1 n2 t3 H Q,$

$n1 \leq n2+1 \rightarrow$

$(H ==> I \ n1 \ \backslash* H') \rightarrow$

$(\forall i, n1 \leq i \leq n2+1 \rightarrow$

$\text{triple } (\text{subst } x i t3) (I \ i) (\text{fun } u \Rightarrow I \ (i+1)) \rightarrow$

$(\text{fun } v \Rightarrow I \ (n2+1) \ \backslash* H') ==> Q \rightarrow$

$\text{triple } (\text{trm_for } x n1 n2 t3) H Q.$

Proof using. Admitted.

□

End FORLOOPS.

47.3.2 Treatment of Generalized Conditionals and Loops in wpgen

Close Scope wp_scope.

The formula generator `wpfn` may be extended to take into account the generalized form `if t0 then t1 else t2`. The corresponding formula is `wpfn_let (aux t0) (fun v => mkstruct (wpfn_if v (aux t1) (aux t2)))`, where `wpfn_let` is used to compute the `wpfn` of the argument of the conditional, and where `wpfn_if` is used to compute the `wpfn` of a conditional with a argument already evaluated to a value.

This pattern is captured by the auxiliary definition `wpfn_if_trm`.

Fixpoint `wpfn` (`E:ctx`) (`t:trm`) : `hprop := mkstruct match t with ... | trm_if t0 t1 t2 => wpfn_if_trm (wpfn t0) (wpfn t1) (wpfn t2) ...`

where `wpfn_if_trm` is defined as shown below.

Definition wp_{gen}_if_trm ($F0\ F1\ F2:\text{formula}$) : formula :=
 $\wp_{\text{gen}}\text{-let } F0 \ (\text{fun } v \Rightarrow \text{mkstruct} \ (\wp_{\text{gen}}\text{-if } v\ F1\ F2)).$

The soundness of this extension of `wpgen` is captured by the following lemma.

Lemma wpgen_if_trm_sound : $\forall F0\ F1\ F2\ t0\ t1\ t2,$

```

formula_sound t0 F0 →
formula_sound t1 F1 →
formula_sound t2 F2 →
formula_sound (trm_if t0 t1 t2) (wpgen_if_trm F0 F1 F2).

```

Proof using.

```

introv S0 S1 S2. unfold wpgen_if_trm. intros Q. unfold wpgen_let.
applys himpl_trans S0. applys himpl_trans; [ | applys wp_if_trm ].
applys wp_conseq. intros v. applys mkstruct_sound.
intros Q'. applys wpgen_if_sound S1 S2.

```

Qed.

To handle while loops in `wpgen`, we introduce the auxiliary definition `wpgen_while`.

```
Fixpoint wpigen (E:ctx) (t:trm) : hprop := mkstruct match t with ... | trm_while t1 t2 => wpigen_while (wpigen t1) (wpigen t2) ...
```

The definition of `wpgen_while` quantifies over an abstract formula F , while denotes the behavior of the while loop. The weakest precondition for the loop w.r.t. postcondition Q is described as $F \ Q$, or, more precisely `mkstruct F Q`, to keep track of the fact that F denotes a formula on which one may apply any structural reasoning rule.

To establish that `mkstruct F Q` is entailed by the heap predicate that describes the current state, the user is provided with an assumption: the fact that `mkstruct F Q'` can be proved from the weakest precondition of the term `if t1 then (t2; t3) else ()`, where the weakest precondition of `t3`, which denotes the recursive call to the loop, is described by `F`.

Definition wpgen_while ($F1\ F2:\text{formula}$) : formula := fun $Q \Rightarrow$

\forall F,

```
\[ \forall Q', \text{mkstruct } (\text{wpgen\_if\_trm } F1
    (\text{mkstruct } (\text{wpgen\_seq } F2 (\text{mkstruct } F)))
    (\text{mkstruct } (\text{wpgen\_val } \text{val\_unit}))) \; Q'
==> \text{mkstruct } F \; Q' ]
```

\-* (mkstruct F Q).

Let us axiomatize the fact that `wpgen` is generalized to handle the new term construct `term_while t1 t2`.

Parameter *wpgen_while_eq* : $\forall E\ t1\ t2,$

`wpgen E (trm_while t1 t2) = mkstruct (wpgen_while (wpgen E t1) (wpgen E t2)).`

The soundness proof of `wpgen` with respect to the treatment of while-loops goes as follows.

Lemma wp_gen_while_sound : $\forall t1\ t2\ F1\ F2,$
 $\text{formula_sound } t1\ F1 \rightarrow$

```

formula_sound t2 F2 →
formula_sound (trm_while t1 t2) (wp_gen_while F1 F2).

```

Proof using.

```

intros S1 S2. intros Q. unfolds wp_gen_while.
applys himpl_hforall_l (wp (trm_while t1 t2)).
applys himpl_trans. 2:{ rewrite × ← mkstruct_wp. }
rewrite hwand_hpure_l. { auto. } intros Q'.
applys mkstruct_monotone. intros Q''.
applys himpl_trans. 2:{ applys wp_while. }
applys himpl_trans.
2:{ applys wp_gen_if_trm_sound.
{ applys S1. }
{ applys mkstruct_sound. applys wp_gen_seq_sound.
{ applys S2. }
{ applys mkstruct_sound. applys wp_sound. } }
{ applys mkstruct_sound. applys wp_gen_val_sound. } }
{ auto. }

```

Qed.

47.3.3 Notation and Tactics for Manipulating While-Loops

The associated piece of notation for displaying characteristic formulae are defined as follows.

Notation "'If_trm' F0 'Then' F1 'Else' F2" :=
 $((\text{wp_gen_if_trm } F0\ F1\ F2))$
(in custom wp at level 69) : wp_scope.

Declare Scope wp_gen_scope.

Notation "'While' F1 'Do' F2 'Done'" :=
 $((\text{wp_gen_while } F1\ F2))$
(in custom wp at level 69, F2 at level 68,
format "[v' 'While' F1 'Do' /' [F2]' /' 'Done']'")

: wp_scope.

The tactic *xapply* is useful for applying an assumption of the form $H ==> \text{mkstruct } F$ Q to a goal of the form $H' ==> \text{mkstruct } F$ Q' , with the ramified frame rule relating H , H' , Q and Q' . In essence, *xapply* applies an hypothesis “modulo consequence-frame”.

Lemma mkstruct_apply : $\forall H1\ H2\ F\ Q1\ Q2,$
 $H1 ==> \text{mkstruct } F\ Q1 \rightarrow$
 $H2 ==> H1 \setminus * (Q1 \setminus * \text{protect } Q2) \rightarrow$
 $H2 ==> \text{mkstruct } F\ Q2.$

Proof using.

```

intros M1 M2. xchange M2. xchange M1. applys mkstruct_ramified.

```

Qed.

```
Tactic Notation "xapply" constr(E) :=
  applys mkstruct_apply; [ applys E | xsimpl; unfold protect ].
```

The tactic *xwhile* is useful for reasoning about a while-loop. In essence, the tactic *while* applies the reasoning rule *wp_while*.

```
Lemma xwhile_lemma : ∀ F1 F2 H Q,
  (∀ F,
    (∀ Q', mkstruct (wpgen_if_trm F1 (mkstruct (wpgen_seq F2 (mkstruct F))))
      (mkstruct (wpgen_val val_unit))) Q'
      ==> mkstruct F Q')
    → H ==> mkstruct F Q) →
  H ==> mkstruct (wpgen_while F1 F2) Q.
```

Proof using.

```
introv M. applys himpl_trans. 2:{ applys mkstruct_erase. }
unfold wpigen_while. xsimpl. intros F N. applys M. applys N.
```

Qed.

```
Tactic Notation "xwhile" :=
  xseq_xlet_if_needed; applys xwhile_lemma.
```

Note: a similar construction could be set up for dealing with for loops.

47.3.4 Example of the Application of Frame During Loop Iterations

Section DemoLoopFrame.

Import ProgramSyntax Basic Repr.

Global Opaque MList.

Consider the following function, which computes the length of a linked list with head at location *p*, using a while loop and a reference named *a* to count the number of cells being traversed.

OCaml:

```
let mlength_loop p = let a = ref 0 in let r = ref p in while !r != null do incr a; r := !p.tail; done; let n = !a in free a; free r; n
```

Definition mlength_loop : val :=

```
<{ fun 'p =>
  let 'a = ref 0 in
  let 'r = ref 'p in
  while let 'p1 = !'r in ('p1 ≠ null) do
    incr 'a;
    let 'p1 = !'r in
    let 'q = 'p1'.tail in
    'r := 'q
  done;
  let 'n = !'a in
```

```

free 'a;
free 'r;
'n }>.

```

This function is specified and verified as follows.

```

Lemma triple_mlength_loop : ∀ L p,
  triple (mlength_loop p)
  (MList L p)
  (fun r ⇒ \[r = length L] \* MList L p).

```

Proof using.

Let's compute the weakest precondition and pretend that *xwpgen* includes support for loops. *xwp. xapp. intros a. xapp. intros r.*

```
rewrite wpgen_while_eq. xwp_simpl.
```

We call the *xwhile* tactic to handle the loop. The formula *F* then denotes “the behavior of the loop”. *xwhile. intros F HF.*

We next state the induction principle for the loop, in the form $I \text{ p n} ==> F Q$, where *I p n* denotes the loop invariant, and *Q* describes the final output of the loop. *asserts KF:* $(\forall p n,$

```

r ~~> p \* a ~~> n \* MList L p
==> mkstruct F (fun _ ⇒ r ~~> null \* a ~~> (length L + n) \* MList L p)).

```

We carry out a proof by induction on the length of the list *L*. { *induction_wf IH: list_sub L. intros*.

```

applys himpl_trans HF. clear HF. xlet.
xapp. xapp. xchange MList_if. xif; intros C; case_if.
{ xpull. intros x q L' →. xseq. xapp. xapp. xapp. xapp.

```

At this point, we reason about the recursive call. We use the tactic *xapply* to apply the induction hypothesis modulo the frame rule. Here, the head cell of the list is framed out over the scope of the recursive call, which operates only on the tail of the list. *xapply (IH L'). { auto. } intros _.*

```

xchange ← MList_cons. { xsimpl. rew_list. math. } }
{ xpull. intros →. xval. xsimpl. { congruence. }
  subst. xchange × ← (MList_nil null). } }
xapply KF. xpull. xapp. xapp. xapp. xval. xsimpl. math.

```

Qed.

End DemoLoopFrame.

47.3.5 Reasoning Rule for Loops in an Affine Logic

Module LOOPRULEAFFINE.

Recall from *Affine* the combined structural rule that includes the affine top predicate $\backslash GC$.

Parameter triple_conseq_frame_hgc : ∀ H2 H1 Q1 t H Q,

```

triple t H1 Q1 →
H ==> H1 \* H2 →
Q1 \*+ H2 ===> Q \*+ \GC →
triple t H Q.

```

In that setting, it is useful to integrate $\backslash GC$ into the rule `triple_while_inv_conseq_frame`, to allow discarding the data allocated by the loop iterations but not described in the final postcondition.

Lemma `triple_while_inv_conseq_frame_hgc` :

```

∀ (A:Type) (I:bool→A→hprop) (R:A→A→Prop) H' t1 t2 H Q,
let Q' := (fun r ⇒ ∃ Y, I false Y) in
wf R →
(H ==> (∃ b X, I b X) \* H') →
(∀ t b X,
  (∀ b' Y, R Y X → triple t (I b' Y) Q') →
  triple (trm_if t1 (trm_seq t2 t) val_unit) (I b X) Q') →
Q' \*+ H' ===> Q \*+ \GC →
triple (trm_while t1 t2) H Q.

```

Proof using.

```

intros WR WH M WQ. applys triple_conseq_frame_hgc WH WQ.
applys triple_while_inv WR M.

```

Qed.

End LOOPRULEAFFINE.

End WHILELOOPS.

47.3.6 Curried Functions of Several Arguments

Module CURRIEDFUN.

Open Scope liblist_scope.

Implicit Types $f : \text{var}$.

We next give a quick presentation of the notation, lemmas and tactics involved in the manipulation of curried functions of several arguments.

We focus here on the particular case of recursive functions with 2 arguments, to illustrate the principles at play. Set up for non-recursive and recursive functions of arity 2 and 3 can be found in the file `LibSepReference`.

One may attempt to generalize these definitions to handle arbitrary arities. Yet, to obtain an arity-generic treatment of functions, it appears simpler to work with primitive n-ary functions, whose treatment is presented in the next section.

Consider a curried recursive functions that expects two arguments: `val_fix f x1 (trm_fun x2 t)` describes such a function, where f denotes the name of the function for recursive calls, $x1$ and $x2$ denote the arguments, and t denotes the body. Observe that the inner function,

the one that expects x_2 , is not recursive, and that it is not a value but a term (because it may refer to the variable x_1 bound outside of it).

We introduce the notation $\text{Fix } f \ x_1 \ x_2 := t$ for such a recursive function with two arguments.

```
Notation "'Fix' f x1 x2 ':=' t" :=
  (val_fix f x1 (trm_fun x2 t))
  (at level 69, f, x1, x2 at level 0,
  format "'Fix' f x1 x2 ':=' t").
```

An application of a function with two arguments takes the form $f \ v_1 \ v_2$, which is actually parsed as $\text{trm_app} (\text{trm_app } f \ v_1) \ v_2$.

This expression is an application of a term to a value, and not of a value to a value. Thus, this expression cannot be evaluated using the rule *eval_app_fun*. We therefore need a distinct rule for first evaluating the arguments of a function application to values, before we can evaluate the application of a value to a value.

The rule *eval_app_args* serves that purpose. To state it, we first need to characterize whether a term is a value or not, using the predicate *trm_is_val* defined next.

```
Definition trm_is_val (t:trm) : Prop :=
  match t with trm_val v => True | _ => False end.
```

The statement of *eval_app_args* is as shown below. For an expression of the form $\text{trm_app } t_1 \ t_2$, where either t_1 or t_2 is not a value, it enables reducing both t_1 and t_2 to values, leaving a premise describing the evaluation of a term of the form $\text{trm_app } v_1 \ v_2$, for which the rule *eval_app_fun* applies.

```
Parameter eval_app_args : ∀ s1 s2 s3 s4 t1 t2 v1 v2 r,
  (¬ trm_is_val t1 ∨ ¬ trm_is_val t2) →
  eval s1 t1 s2 v1 →
  eval s2 t2 s3 v2 →
  eval s3 (trm_app v1 v2) s4 r →
  eval s1 (trm_app t1 t2) s4 r.
```

Using the above rule, we can establish an evaluation rule for the term $v_0 \ v_1 \ v_2$. There, v_0 denotes a recursive function of two arguments named x_1 and x_2 , the values v_1 and v_2 denote the two arguments, and f denotes the name of the function available for making recursive calls from the body t_1 .

The key idea is that the behavior of $v_0 \ v_1 \ v_2$ is similar to that of the term $\text{subst } x_2 \ v_2 (\text{subst } x_1 \ v_1 (\text{subst } f \ v_0 \ t_1))$. We state this property using the predicate *eval_like*, introduced in the chapter Rules.

```
Lemma eval_like_app_fix2 : ∀ v0 v1 v2 f x1 x2 t1,
  v0 = val_fix f x1 (trm_fun x2 t1) →
  (x1 ≠ x2 ∧ f ≠ x2) →
  eval_like (subst x2 v2 (subst x1 v1 (subst f v0 t1))) (v0 v1 v2).
```

Proof using.

```

introv E (N1&N2). introv R. applys× eval_app_args.
{ applys eval_app_fix E simpl. do 2 (rewrite var_eq_spec; case_if).
  applys eval_fun. }
{ applys× eval_val. }
{ applys× eval_app_fun. }

```

Qed.

We next derive the specification triple for applications of the form $v_0 \ v_1 \ v_2$.

```

Lemma triple_app_fix2 : ∀ f x1 x2 v0 v1 v2 t1 H Q,
  v0 = val_fix f x1 (trm_fun x2 t1) →
  (x1 ≠ x2 ∧ f ≠ x2) →
  triple (subst x2 v2 (subst x1 v1 (subst f v0 t1))) H Q →
  triple (v0 v1 v2) H Q.

```

Proof using.

```
introv E N M1. applys triple_eval_like M1. applys× eval_like_app_fix2.
```

Qed.

The reasoning rule above can be reformulated in weakest-precondition style.

```

Lemma wp_app_fix2 : ∀ f x1 x2 v0 v1 v2 t1 Q,
  v0 = val_fix f x1 (trm_fun x2 t1) →
  (x1 ≠ x2 ∧ f ≠ x2) →
  wp (subst x2 v2 (subst x1 v1 (subst f v0 t1))) Q ==> wp (trm_app v0 v1 v2) Q.

```

Proof using. introv EQ1 N. applys wp_eval_like. applys× eval_like_app_fix2. Qed.

Finally, it is useful to extend the tactic *xwp*, so that it exploits the rule *wp_app_fix2* in the same way as it exploits *wp_app_fix*.

To that end, we state a lemma featuring a conclusion expressed as a triple, and a premise expressed using *wpgen*. Observe the substitution context associated with *wpgen*: it is instantiated as $(f, v0)::(x1, v1)::(x2, v2)::\text{nil}$, so as to perform the relevant substitutions. Note also how the side-condition expressing the freshness conditions on the variables, using a comparison function for variables that computes in Coq.

```

Lemma xwp_lemma_fix2 : ∀ f v0 v1 v2 x1 x2 t H Q,
  v0 = val_fix f x1 (trm_fun x2 t) →
  (var_eq x1 x2 = false ∧ var_eq f x2 = false) →
  H ==> wpgen ((f, v0)::(x1, v1)::(x2, v2)::\text{nil}) t Q →
  triple (v0 v1 v2) H Q.

```

Proof using.

```
introv M1 N M2. repeat rewrite var_eq_spec in N. rew_bool_eq in *.
rewrite ← wp_equiv. xchange M2.
xchange (» wpgen_sound
  (((f, v0)::\text{nil}) ++ ((x1, v1)::\text{nil}) ++ ((x2, v2)::\text{nil})) t Q).
do 2 rewrite isubst_app. do 3 rewrite ← subst_eq_isubst_one.
applys× wp_app_fix2.
```

Qed.

The lemma gets integrated into the implementation of *xwp* as follows.

```
Tactic Notation "xwp" :=
  intros;
  first [ applys xwp_lemma_fun; [ reflexivity || ]
    | applys xwp_lemma_fix; [ reflexivity || ]
    | applys xwp_lemma_fix2; [ reflexivity | splits; reflexivity || ] ];
  xwp_simpl.
```

This tactic *xwp* also appears in the file *LibSepReference.v*. It is exploited in several examples from the chapter **Basic**.

```
End CURRIEDFUN.
```

47.3.7 Primitive n-ary Functions

```
Module PRIMITIVEARYFUN.
```

We next present an alternative treatment to functions of several arguments. The idea is to represent function arguments using lists. The verification tool CFML is implemented following this approach.

On the one hand, the manipulation of lists adds a little bit of boilerplate. On the other hand, when using this representation, all the definitions and lemmas are inherently arity-generic, that is, they work for any number of arguments.

We introduce the short names *vars*, *vals* and *trms* to denote lists of variables, lists of values, and lists of terms, respectively.

These names are not only useful to improve conciseness, they also enable the set up of useful coercions, as illustrated further on.

```
Definition vars : Type := list var.
Definition vals : Type := list val.
Definition trms : Type := list trm.
```

```
Implicit Types xs : vars.
```

```
Implicit Types vs : vals.
```

```
Implicit Types ts : trms.
```

We assume the grammar of terms and values to include primitive n-ary functions and primitive n-ary applications, featuring list of arguments.

Thereafter, for conciseness, we focus on the treatment of recursive functions, and do not describe the simpler case of non-recursive functions.

```
Parameter val_fixs : var → vars → trm → val.
Parameter trm_fixs : var → vars → trm → trm.
Parameter trm_apps : trm → trms → trm.
```

The substitution function is a bit tricky to update for dealing with list of variables. A definition along the following lines computes well, and is recognized as structurally recursive by Coq.

```

Fixpoint subst (y:var) (w:val) (t:trm) : trm := let aux t := (subst y w t) in let aux_no_captures
xs t := (If List.In y xs then t else aux t) in match t with | trm_fixs f xs t1 => trm_fixs
f xs (If f = y then t1 else aux_no_captures xs t1) | trm_apps t0 ts => trm_apps (aux t0)
(List.map aux ts) ... end.

```

The evaluation rules also need to be updated to handle list of arguments. A n-ary function from the grammar of terms evaluates to the corresponding n-ary function from the grammar of values.

```

Parameter eval_fixs : ∀ m f xs t1,
  xs ≠ nil →
  eval m (trm_fixs f xs t1) m (val_fixs f xs t1).

```

Note that, for technical reasons, we need to ensure that list of arguments is nonempty. Indeed, a function with zero arguments would beta-reduce to its body as soon as it is defined, because it is not waiting for any argument, resulting in an infinite sequence of reductions.

The application of a n-ary function to values takes the form `trm_apps (trm_val v0) ((trm_val v1):: .. ::(trm_val vn)::nil)`.

If the function v_0 is defined as `val_fixs f xs t1`, where xs denotes the list $x_1::x_2::\dots::x_n::nil$, then the beta-reduction of the function application triggers the evaluation of the term `subst xn vn (... (subst x1 v1 (subst f v0 t1)) ...)`.

To describe the evaluation rule in an arity-generic way, we need to introduce the list vs made of the values provided as arguments, that is, the list $v_1::v_2::\dots::v_n::nil$.

With this list vs , the n-ary application can then be written as the term `trm_apps (trm_val v0) (trms_vals vs)`, where the operation `trms_vals` converts a list of values into a list of terms by applying the constructor `trm_val` to every value from the list.

```

Coercion trms_vals (vs:vals) : trms :=
  LibList.map trm_val vs.

```

Note that we declare the operation `trms_vals` as a coercion, just like `trm_val` is a coercion. Doing so enables us to write a n-ary application in the form $v_0\ vs$.

To describe the iterated substitution `subst xn vn (... (subst x1 v1 (subst f v0 t1)) ...)`, we introduce the operation `substn xs vs t`, which substitutes the variables xs with the values vs inside the t . It is defined recursively, by iterating calls to the function `subst` for substituting the variables one by one.

```

Fixpoint substn (xs:list var) (vs:list val) (t:trm) : trm :=
  match xs,vs with
  | x::xs', v::vs' => substn xs' vs' (subst x v t)
  | _,_ => t
  end.

```

This substitution operation is well-behaved only if the list xs and the list vs have the same lengths. It is also needed for reasoning about the evaluation rule to guarantee that the list of variables xs contains variables that are distinct from each others and distinct from f , and to guarantee that the list xs is not empty.

To formally capture all these invariants, we introduce the predicate `var_fixs f xs n`, where `n` denotes the number of arguments the function is being applied to. Typically, `n` is equal to the length of the list of arguments `vs`).

```
Definition var_fixs (f:var) (xs:vars) (n:nat) : Prop :=
  LibList.noduplicates (f::xs)
  ∧ length xs = n
  ∧ xs ≠ nil.
```

The evaluation of a recursive function `v0` defined as `val_fixs f xs t1` on a list of arguments `vs` triggers the evaluation of the term `substn xs vs (subst f v0 t1)`, same as `substn (f::xs) (v0::vs) t1`. The evaluation rule is stated as follows, using the predicate `var_fixs` to enforce the appropriate invariants on the variable names.

```
Parameter eval_apps_fixs : ∀ v0 vs f xs t1 s1 s2 r,
  v0 = val_fixs f xs t1 →
  var_fixs f xs (LibList.length vs) →
  eval s1 (substn (f::xs) (v0::vs) t1) s2 r →
  eval s1 (trm_apps v0 (trms_vals vs)) s2 r.
```

The corresponding reasoning rule admits a somewhat similar statement.

```
Lemma triple_apps_fixs : ∀ v0 vs f xs t1 H Q,
  v0 = val_fixs f xs t1 →
  var_fixs f xs (LibList.length vs) →
  triple (substn (f::xs) (v0::vs) t1) H Q →
  triple (trm_apps v0 vs) H Q.
```

Proof using.

```
  introv E N M. applys triple_eval_like M.
  introv R. applys eval_apps_fixs.
```

Qed.

The statement of the above lemma applies only to terms that are of the form `trm_apps (trm_val v0) (trms_vals vs)`. Yet, in practice, goals are generally of the form `trm_apps (trm_val v0) ((trm_val v1):: .. :: (trm_val vn)::nil)`.

The two forms are convertible. Yet, in most cases, Coq is not able to synthesize the list `vs` during the unification process.

Fortunately, it is possible to reformulate the lemma using an auxiliary conversion function named `trms_to_vals`, whose evaluation by Coq's unification process is able to synthesize the list `vs`.

The operation `trms_to_vals ts`, if all the terms in `ts` are of the form `trm_val vi`, returns a list of values `vs` such that `ts` is equal to `trms_vals vs`. Otherwise, the operation returns `None`. The definition and specification of the operation `trms_to_vals` are as follows.

```
Fixpoint trms_to_vals (ts:trms) : option vals :=
  match ts with
  | nil ⇒ Some nil
  | (trm_val v)::ts' ⇒
```

```

match trms_to_vals ts' with
| None => None
| Some vs' => Some (v :: vs')
end
| _ => None
end.

```

The specification of the function `trms_to_vals` asserts that if `trms_to_vals` ts produces some list of values vs , then ts is equal to `trms_vals` vs .

Lemma `trms_to_vals_spec` : $\forall ts\ vs,\ trms_to_vals\ ts = \text{Some}\ vs \rightarrow ts = \text{trms_vals}\ vs.$

Proof using.

```

intros ts. induction ts as [| t ts'|]; simpl; introv E.
{ inverts E. auto. }
{ destruct t; inverts E as E. cases (trms_to_vals ts') as C; inverts E.
  rename v0 into vs'. rewrite $\times$  (IHts' vs'). }

```

Qed.

Here is a demo showing how `trms_to_vals` computes in practice.

Lemma `demo_trms_to_vals` : $\forall v1\ v2\ v3,\$

```

 $\exists vs,$ 
 $\text{trms\_to\_vals} ((\text{trm\_val}\ v1) :: (\text{trm\_val}\ v2) :: (\text{trm\_val}\ v3) :: \text{nil}) = \text{Some}\ vs$ 
 $\wedge vs = vs.$ 

```

Proof using.

```
intros. esplit. split. simpl. eauto. Abort.
```

Using `trms_to_vals`, we can reformulate the rule `triple_apps_fixs` in such a way that the rule can be smoothly applied on goals of the form `trm_apps (trm_val v0) ((trm_val v1):: .. ::(trm_val vn)::nil)`.

Lemma `triple_apps_fixs'` : $\forall v0\ ts\ vs\ f\ xs\ t1\ H\ Q,$

```

 $v0 = \text{val\_fixs}\ f\ xs\ t1 \rightarrow$ 
 $\text{trms\_to\_vals}\ ts = \text{Some}\ vs \rightarrow$ 
 $\text{var\_fixs}\ f\ xs\ (\text{LibList.length}\ vs) \rightarrow$ 
 $\text{triple}(\text{substn}(f :: xs)(v0 :: vs)\ t1)\ H\ Q \rightarrow$ 
 $\text{triple}(\text{trm\_apps}\ v0\ ts)\ H\ Q.$ 

```

Proof using.

```

introv E T N M. rewrites (@trms_to_vals_spec _ _ T).
applys $\times$  triple_apps_fixs.

```

Qed.

Finally, let us show how to integrate the rule `triple_apps_fixs'` into the tactic `xwp`. To that end, we reformulate the rule by making two small changes.

The first change consists of replacing the predicate `var_fixs` which checks the well-formedness properties of the list of variables xs by an executable version of this predicate, with a result

in `bool`. This way, the tactic `reflexivity` can prove all the desired facts, when the lemma is invoked on a concrete function. We omit the details, and simply state the type of the boolean function `var_fixs_exec`.

Parameter `var_fixs_exec` : `var → vars → nat → bool`.

The second change consists of introducing the `wpgen` function for reasoning about the body of the function. Concretely, the substitution `substn (f::xs) (v0::vs)` is described by the substitution context `List.combine (f::xs) (v0::vs)`.

The statement of the lemma for `xwp` is as follows. We omit the proof details. (They may be found in the implementation of the CFML tool.)

```
Parameter xwp_lemma_fixs :  $\forall v0\ ts\ vs\ f\ xs\ t1\ H\ Q,$ 
 $v0 = \text{val\_fixs } f\ xs\ t1 \rightarrow$ 
 $\text{trms\_to\_vals } ts = \text{Some } vs \rightarrow$ 
 $\text{var\_fixs\_exec } f\ xs\ (\text{LibList.length } vs) \rightarrow$ 
 $H ==> (\text{wpgen } (\text{combine } (f :: xs) (v0 :: vs))\ t1)\ Q \rightarrow$ 
 $\text{triple } (\text{trm\_apps } v0\ ts)\ H\ Q.$ 
```

47.4 Optional Material

47.4.1 A Coercion for Parsing Primitive N-ary Applications

One last practical detail for working with primitive n-ary functions in a smooth way consists of improving the parsing of applications.

Writing an application in the form `trm_apps f (x::y::nil)` to denote the application of a function `f` to two arguments `x` and `y` is fairly verbose, in comparison with the syntax `f x y`, which we were able to set up by declaring `trm_app` as a *Funclass* coercion—recall chapter Rules.

If we simply declare `trm_apps` as a *Funclass* coercion, then we can write `f (x::y::nil)` in place of `trm_apps f (x::y::nil)`, however we still need to write the arguments in the form `x::y::nil`.

Fortunately, there is a trick that allows the expression `f x y` to be interpreted by Coq as `trm_apps f (x::y::nil)`. This trick is arity-generic: it works for any number of arguments. It is described next.

Module NARYSYNTAX.

To explain the working of our coercion trick, let us consider a simplified grammar of terms, including only the constructor `trm_apps` for n-ary applications, and the construct `trm_val` for values.

```
Inductive trm : Type :=
| trm_val : val → trm
| trm_apps : trm → list trm → trm.
```

We introduce the data type **apps**, featuring two constructors named `apps_init` and `apps_next`, to represent the syntax tree.

Inductive apps : Type :=

```
| apps_init : trm → trm → apps
| apps_next : apps → trm → apps.
```

For example, the term `trm_apps f (x::y::z::nil)` is represented as the expression `apps_next (apps_init (apps_init f x) y) z`.

Internally, the parsing proceeds as follows.

- The application of a term to a term, that is, $t_1 t_2$, gets interpreted via a *Funclass* coercion as `apps_init t1 t2`, which is an expression of type **apps**.
- The application of an object of type **apps** to a term, that is $a_1 t_2$, gets interpreted via another *Funclass* coercion as `apps_next a1 t2`.

Coercion `apps_init : trm >-> Funclass`.

Coercion `apps_next : apps >-> Funclass`.

From a term of the form `apps_next (apps_init (apps_init f x) y) z`, the corresponding application `trm_apps f (x::y::z::nil)` can be computed by a Coq function, called `apps_to_trm`, that processes the syntax tree of the **apps** expression. This function is implemented recursively.

- In the base case, `apps_init t1 t2` describes the application of a function to one argument: `trm_apps t1 (t2::nil)`.
- In the “next” case, if an **apps** structure a_1 describes a term `trm_apps t_0 ts`, then `apps_next a1 t2` describes the term `trm_apps t_0 (ts++t2::nil)`, that is, an application to one more argument.

```
Fixpoint apps_to_trm (a:apps) : trm :=
  match a with
  | apps_init t1 t2 ⇒ trm_apps t1 (t2::nil)
  | apps_next a1 t2 ⇒
    match apps_to_trm a1 with
    | trm_apps t0 ts ⇒ trm_apps t0 (List.app ts (t2::nil))
    | t0 ⇒ trm_apps t0 (t2::nil)
    end
  end.
```

The function `apps_to_trm` is declared as a coercion from **apps** to **trm**, so that any iterated application can be interpreted as the corresponding `trm_apps` term. Coq raises an “ambiguous coercion path” warning, but this warning may be safely ignored here.

Coercion `apps_to_trm : apps >-> trm`.

The following demo shows how the term $f \ x \ y \ z$ is parsed as `apps_to_trm (apps_next (apps_init f x) y) z`, which then simplifies by `simpl` to `trm_apps f (x::y::z::nil)`.

```
Lemma apps_demo : ∀ (f x y z:trm),  
  (f x y z : trm) = trm_apps f (x::y::z::nil).
```

Proof using. intros. simpl. Abort.

End NARYSYNTAX.

End PRIMITIVENARYFUN.

47.4.2 Historical Notes

Regarding while loops, the possibility to frame over the remaining iterations, as illustrated with the example of `triple_mlength_loop`, is inherently available when a loop is encoded as a recursive functions, or when a loop is presented in CPS-style (typical with assembly-level code). The statement of a reasoning rule directly applicable to a non-encoded loop construct, and allowing to frame over the remaining iterations, has appeared independently in work by *Charguéraud* 2010 (in Bib.v) and *Tuerk* 2010 (in Bib.v).

Chapter 48

Library `SLF.Nondet`

48.1 Nondet: Triples for Nondeterministic Languages

Set Implicit Arguments.

From *SLF* Require Export LibSepReference.

Close Scope *val_scope*.

Close Scope *trm_scope*.

Implicit Types *f* : var.

Implicit Types *b* : **bool**.

Implicit Types *s* : state.

Implicit Types *t* : **trm**.

Implicit Types *v* : **val**.

Implicit Types *n* : int.

Implicit Types *p* : loc.

Implicit Types *h* : heap.

Implicit Types *P* : Prop.

Implicit Types *H* : hprop.

Implicit Types *Q* : **val** → hprop.

48.2 First Pass

In previous chapters, we have considered a quasi-deterministic language; only the addresses of allocated data could vary between two executions of a same term starting in a same state.

In this chapter, we discuss the interpretation of Separation Logic triples in the more general case of non-deterministic languages. To that end, we extend the language with a new primitive operator that produces a random number: the term `val_rand n` non-deterministically evaluates to an integer between 0 inclusive and *n* exclusive.

This chapter is organized as follows:

- Statement of a big-step semantics for non-deterministic languages

- Proof of reasoning rules for triples
- Proof of reasoning rules in weakest-precondition style
- Bonus section: definition of triples for non-deterministic languages with respect to small-step semantics.

For simplicity, we omit from the semantics the treatment of non-recursive functions, which can be encoded using recursive functions; and, likewise, omit the treatment of sequences, which can be encoded using let-bindings.

48.2.1 Non-Deterministic Big-Step Semantics: the Predicate **evaln**

Previously, we worked with the big-step judgment $\text{eval } s \ t \ s' \ v$, which relates one input state (and term) with one output state (and value). This judgment is well-suited for a deterministic semantics, because one input state leads to at most one result. For a non-deterministic semantics, however, we need to relate one input with a set of possible results.

In Coq, a set of output states can be represented as a set of pairs made of a state and a value, that is, $(\text{state} \times \text{val}) \rightarrow \text{Prop}$. This type is isomorphic to the type $\text{val} \rightarrow \text{state} \rightarrow \text{Prop}$, which corresponds exactly to the type of postconditions $\text{val} \rightarrow \text{hprop}$. Thus, we are interested in setting up a judgment that relates an “input configuration”, described by a state and a term, with a postcondition describing the set of possible “output configurations”.

The non-deterministic big-step judgment therefore takes the form $\text{evaln } s \ t \ Q$, and relates an input state s and a term t with a postcondition Q . An output configuration consists of an output state s' and an output value v that, together, satisfy the postcondition Q , in the sense that $Q \vee s'$ holds.

In the context of program verification, we are not so much interested in characterizing exactly the set of all output configurations reachable by the program, but mainly interested in showing that all output configurations satisfy a desired postcondition Q . For this reason, it is perfectly fine for the set of configurations described by Q to over-approximate the set of configurations actually reachable by the program. We’ll explain later on how to characterize the set of output configurations in a precise manner.

The judgment $\text{evaln } s \ t \ Q$ is defined inductively. It asserts that every possible execution starting from configuration (s,t) satisfies the postcondition Q . The definition of **evaln** is adapted from that of **eval**. Four cases are particularly interesting.

- Consider the case of a value. The judgment $\text{evaln } s \ t \ Q$ asserts that the value v in the state s satisfies the postcondition Q . The premise for deriving that judgment is thus that $Q \vee s$ must hold.
- Consider the case of a let-binding. $\text{let } x = t_1 \text{ in } t_2$. The first premise of the evaluation rule describes the evaluation of the subterm t_1 . It takes the form $\text{evaln } s \ t_1 \ Q$. The second premise describes the evaluation of the continuation $\text{subst } x \ v_1 \ t_2$ in a state s_2 , where the value v_1 and the state s_2 correspond to a configuration that can be produced by t_1 . In other words, v_1 and s_2 are assumed to satisfy $Q_1 \vee v_1 \ s_2$.

- Consider the case of a non-deterministic term: `val_rand n`, evaluated in a state `s`. The first premise of the rule requires $n > 0$. The second premise requires that, for any value n_1 that `val_rand n` may evaluate to (that is, such that $0 \leq n_1 < n$), the configuration made of n_1 and `s` satisfies the postcondition Q , in the sense that $Q \ n_1 \ s$ holds.
- Consider the case of an allocation: `val_ref v`, evaluated in a state `s`. The premise of the rule asserts that the postcondition Q should hold of any configuration made of a fresh location `p` and a state `s'` obtained as the update of `s` with a binding from `p` to `v`.

Inductive `evaln` : state \rightarrow trm \rightarrow (val \rightarrow state \rightarrow Prop) \rightarrow Prop :=

$$\begin{aligned}
& | \text{evaln_val} : \forall s v Q, \\
& \quad Q v s \rightarrow \\
& \quad \mathbf{evaln} s (\text{trm_val } v) Q \\
& | \text{evaln_fix} : \forall s f x t1 Q, \\
& \quad Q (\text{val_fix } f x t1) s \rightarrow \\
& \quad \mathbf{evaln} s (\text{trm_fix } f x t1) Q \\
& | \text{evaln_app_fix} : \forall s v1 v2 f x t1 Q, \\
& \quad v1 = \text{val_fix } f x t1 \rightarrow \\
& \quad \mathbf{evaln} s (\text{subst } x v2 (\text{subst } f v1 t1)) Q \rightarrow \\
& \quad \mathbf{evaln} s (\text{trm_app } v1 v2) Q \\
& | \text{evaln_let} : \forall Q1 s x t1 t2 Q, \\
& \quad \mathbf{evaln} s t1 Q1 \rightarrow \\
& \quad (\forall v1 s2, Q1 v1 s2 \rightarrow \mathbf{evaln} s2 (\text{subst } x v1 t2) Q) \rightarrow \\
& \quad \mathbf{evaln} s (\text{trm_let } x t1 t2) Q \\
& | \text{evaln_if} : \forall s b t1 t2 Q, \\
& \quad \mathbf{evaln} s (\text{if } b \text{ then } t1 \text{ else } t2) Q \rightarrow \\
& \quad \mathbf{evaln} s (\text{trm_if } (\text{val_bool } b) t1 t2) Q \\
& | \text{evaln_add} : \forall s n1 n2 Q, \\
& \quad Q (\text{val_int } (n1 + n2)) s \rightarrow \\
& \quad \mathbf{evaln} s (\text{val_add } (\text{val_int } n1) (\text{val_int } n2)) Q \\
& | \text{evaln_rand} : \forall s n Q, \\
& \quad n > 0 \rightarrow \\
& \quad (\forall n1, 0 \leq n1 < n \rightarrow Q n1 s) \rightarrow \\
& \quad \mathbf{evaln} s (\text{val_rand } (\text{val_int } n)) Q \\
& | \text{evaln_ref} : \forall s v Q, \\
& \quad (\forall p, \neg \text{Fmap.indom } s p \rightarrow \\
& \quad Q (\text{val_loc } p) (\text{Fmap.update } s p v)) \rightarrow \\
& \quad \mathbf{evaln} s (\text{val_ref } v) Q \\
& | \text{evaln_get} : \forall s p Q, \\
& \quad \text{Fmap.indom } s p \rightarrow \\
& \quad Q (\text{Fmap.read } s p) s \rightarrow \\
& \quad \mathbf{evaln} s (\text{val_get } (\text{val_loc } p)) Q \\
& | \text{evaln_set} : \forall s p v Q,
\end{aligned}$$

```

Fmap.indom s p →
Q val_unit (Fmap.update s p v) →
evaln s (val_set (val_loc p) v) Q
| evaln_free : ∀ s p Q,
  Fmap.indom s p →
  Q val_unit (Fmap.remove s p) →
  evaln s (val_free (val_loc p)) Q.

```

Observe that **evaln** $s t Q$ cannot hold if there exists one possible execution of (s,t) that runs into an error, i.e., that reaches a configuration that is stuck. This property reflects on the fact that the judgment **evaln** asserts that every possible execution terminates safely.

Observe also that **evaln** is covariant in the postcondition: if **evaln** $s t Q_1$ holds, and Q_2 is weaker than Q_1 , then **evaln** $s t Q_2$ also holds. This property reflects on the fact that it is always possible to further over-approximate the set of possible results.

```

Lemma evaln_conseq : ∀ s t Q1 Q2,
  evaln s t Q1 →
  Q1 ==> Q2 →
  evaln s t Q2.

```

Proof using.

```

introv M W.
asserts W' : (∀ v h, Q1 v h → Q2 v h). { auto. } clear W.
induction M; try solve [ constructors× ].

```

Qed.

The judgment **evaln** $s t Q$ can interpreted in at least five different ways:

- The judgment **evaln** may be viewed as the natural generalization of **eval**, generalizing from one output to a set of possible output.
- The judgment **evaln** may be viewed as an inductive definition of a weakest-precondition judgment. Interestingly, if we swap the order of the arguments to **evaln** $t Q s$, then the partial application **evaln** $t Q$ has type **hprop**, and its interpretation matches exactly that of a weakest precondition for a Hoare triple. We'll formalize this claim further on in this chapter, and we'll point out shortly afterwards the difference between the rules that define **evaln** and the tradition weakest-precondition reasoning rules.
- The judgment **evaln** may be viewed as a CPS-version of the predicate **eval**. Indeed, The output of **eval**, made of an output value and an output state, are “passed on” to the continuation Q .
- The judgment **evaln** may be viewed as a particular form of denotational semantics for the language. Each program is interpreted as (i.e., mapped to) a set of mathematical objects, which “simply” consists of pairs of states and “syntactic” values. In particular, functions are interpreted as plain pieces of syntax (not as functions over Scott domains, as usually done).

- The judgment **evaln** may be viewed as a generalized form of a typing relation. To make the analogy clear, let us adapt the definition of **evaln** in two ways, and focus on the evaluation rule for let-bindings. First, let us get rid of the state, i.e., consider a language without side-effects, for simplicity.

$\text{evaln } t1 \ Q1 \rightarrow (\text{forall } v1, Q1 \ v1 \rightarrow \text{evaln } (\text{subst } x \ v1 \ t2) \ Q) \rightarrow \text{evaln } (\text{trm_let } x \ t1 \ t2) \ Q$

Second, let us switch from a substitution-based semantics to an environment-based semantics—nothing but change of presentation.

$\text{evaln } E \ t1 \ Q1 \rightarrow (\text{forall } v1, Q1 \ v1 \rightarrow \text{evaln } ((x,v1)::E) \ t2 \ Q) \rightarrow \text{evaln } E \ (\text{trm_let } x \ t1 \ t2) \ Q$

Now, let's consider a typing property. For example, expressing that a term has type *int* amounts to asserting that this term produces a value *v* of the form *val_int n* for some *n*. This property may be captured by the postcondition $\text{fun } v \Rightarrow \exists n, v = \text{val_int } n$. Let us compare the previous rule with the traditional typing rule shown below.

$\text{typing } E \ t1 \ Q1 \rightarrow \text{typing } ((x,Q1)::E) \ t2 \ Q \rightarrow \text{typing } E \ (\text{trm_let } x \ t1 \ t2) \ Q$

The only difference is that the evaluation rule maps *x* to any value *v1* such that *v1* has type *Q1*, whereas the typing rule directly maps *x* to the type *Q1*. The two presentations are thus essentially equivalent.

In what follows, we discuss the difference between the presentation of the judgment **evaln**, and the rules that define a weakest precondition. We focus on the particular case of a let-binding.

Exercise: 1 star, standard, optional (evaln_let') The rule **evaln_let** shares similarities with the statement of the weakest-precondition style reasoning rule for let-bindings. Prove the following alternative statement, with a wp-style flavor.

Lemma evaln_let' : $\forall s1 \ x \ t1 \ t2 \ Q,$
 $\text{evaln } s1 \ t1 \ (\text{fun } v1 \ s2 \Rightarrow \text{evaln } s2 \ (\text{subst } x \ v1 \ t2) \ Q) \rightarrow$
 $\text{evaln } s1 \ (\text{trm_let } x \ t1 \ t2) \ Q.$

Proof using. *Admitted.*

□

Exercise: 2 stars, standard, optional (evaln_let_of_evaln_let') Reciprocally, prove that the statement considered in the inductive definition of **evaln** is derivable from **evaln_let'**. More precisely, prove the statement below by using **evaln_let'** and **evaln_conseq**.

Lemma evaln_let_of_evaln_let' : $\forall Q1 \ s1 \ x \ t1 \ t2 \ Q,$
 $\text{evaln } s1 \ t1 \ Q1 \rightarrow$
 $(\forall v1 \ s2, Q1 \ v1 \ s2 \rightarrow \text{evaln } s2 \ (\text{subst } x \ v1 \ t2) \ Q) \rightarrow$
 $\text{evaln } s1 \ (\text{trm_let } x \ t1 \ t2) \ Q.$

Proof using. *Admitted.*

□

One way wonder whether we could have used the wp-style formulation of the semantics of let-bindings directly in the definition of **evaln**. The answer is negative. Doing so would lead to an invalid inductive definition, involving a “non strictly positive occurrence”. To check it out, uncomment the definition below to observe Coq’s complaint.

Inductive evaln' : state -> trm -> (val->state->Prop) -> Prop := | evaln'_let : forall Q1 s1 x t1 t2 Q, evaln' s1 t1 (fun v1 s2 => evaln' s2 (subst x v1 t2) Q) -> evaln' s1 (trm_let x t1 t2) Q.

48.2.2 Interpretation of **evaln** w.r.t. **eval**

Given that **evaln** describes “all possible executions” and that **eval** describes “one possible execution”, there must be formal relationships between the two predicates. These relationships are investigated next.

Recall the big-step evaluation judgment **eval**. It is the same as before, only extended with evaluation rule for the random number generator, namely **eval_rand**.

Inductive **eval** : state → trm → state → val → Prop :=

- | eval_val : ∀ s v,
 eval s (trm_val v) s v
- | eval_fix : ∀ s f x t1,
 eval s (trm_fix f x t1) s (val_fix f x t1)
- | eval_app_fix : ∀ s1 s2 v1 v2 f x t1 v,
 v1 = val_fix f x t1 →
 eval s1 (subst x v2 (subst f v1 t1)) s2 v →
 eval s1 (trm_app v1 v2) s2 v
- | eval_let : ∀ s1 s2 s3 x t1 t2 v1 r,
 eval s1 t1 s2 v1 →
 eval s2 (subst x v1 t2) s3 r →
 eval s1 (trm_let x t1 t2) s3 r
- | eval_if : ∀ s1 s2 b v t1 t2,
 eval s1 (if b then t1 else t2) s2 v →
 eval s1 (trm_if (val_bool b) t1 t2) s2 v
- | eval_add : ∀ s n1 n2,
 eval s (val_add (val_int n1) (val_int n2)) s (val_int (n1 + n2))
- | eval_rand : ∀ s n n1,
 0 ≤ n1 < n →
 eval s (val_rand (val_int n)) s (val_int n1)
- | eval_ref : ∀ s v p,
 ¬ Fmap.indom s p →
 eval s (val_ref v) (Fmap.update s p v) (val_loc p)
- | eval_get : ∀ s p,
 Fmap.indom s p →

```

eval s (val_get (val_loc p)) s (Fmap.read s p)
| eval_set :  $\forall s p v,$ 
  Fmap.indom s p  $\rightarrow$ 
  eval s (val_set (val_loc p) v) (Fmap.update s p v) val_unit
| eval_free :  $\forall s p,$ 
  Fmap.indom s p  $\rightarrow$ 
  eval s (val_free (val_loc p)) (Fmap.remove s p) val_unit.

```

The judgment **evaln** s t Q asserts that any possible execution of the program (t,s) terminates on an output satisfying the postcondition Q. Thus, if one particular execution terminates on the output (s',v), as described by the judgment **eval** s t s' v, it must be the case that Q v s' holds. This result is formalized by the following lemma.

Exercise: 3 stars, standard, optional (evaln_inv_eval) Assume **evaln** s t Q to hold. Prove that the postcondition Q holds of any output that (s,t) may evaluate to according to the judgment **eval**.

Lemma evaln_inv_eval : $\forall s t v s' Q,$
evaln s t Q \rightarrow
eval s t s' v \rightarrow
Q v s'.

Proof using. *Admitted.*

□

The judgment **evaln** s t Q asserts that any possible execution of the program (t,s) terminates on an output satisfying the postcondition Q. This judgment implies, in particular, that there exists at least one such execution, described by a judgment of the form **eval** s t s' v, where v and s' satisfy the postcondition Q. This second result is formalized by the following lemma.

Exercise: 4 stars, standard, especially useful (evaln_inv_exists_eval) Assume **evaln** s t Q to hold. Prove that there exists an output (s',v) that (s,t) may evaluate to according to the judgment **eval**, and that this output satisfies Q.

Hint: the proof may be carried out either with or without leveraging the lemma **evaln_inv_eval**, proved above.

Hint: for the case **eval_ref**, use the following line to assert the existence of a fresh location: *forwards* \times (p&D&N): (*exists_fresh* null s)..

Lemma evaln_inv_exists_eval : $\forall s t Q,$
evaln s t Q \rightarrow
 $\exists s' v, \text{eval } s t s' v \wedge Q v s'.$

Proof using. *Admitted.*

□

The judgment **evaln** s t Q associates with a program configuration (s,t) an over-approximation of its possible output configurations, described by Q. For the purpose of defining of Hoare

triple, working with over-approximation is perfectly fine. In other contexts, however, it might be interesting to characterize exactly the set of output configurations.

The set of results to which a program (s,t) may evaluate is precisely characterized by the predicate $\text{fun } v \ s' \Rightarrow \text{eval } s \ t \ s' \ v$. Let $\text{post } s \ t$ denote exactly that predicate.

Definition $\text{post } (s:\text{state}) \ (t:\text{trm}) : \text{val} \rightarrow \text{hprop} :=$
 $\text{fun } v \ s' \Rightarrow \text{eval } s \ t \ s' \ v$.

The judgment $\text{evaln } s \ t \ (\text{post } s \ t)$ essentially captures the safety of the program (s,t) : it asserts that all possible executions terminate without error.

Let us prove that $\text{post } s \ t$ corresponds to the smallest possible post-condition for evaln . In other words, assuming that $\text{evaln } s \ t \ Q$ holds for some Q , we can prove that $\text{evaln } s \ t \ (\text{post } s \ t)$ holds, and that the entailment $\text{post } s \ t \ ==> Q$ holds. This entailment captures the fact that $\text{post } s \ t$ denotes a smaller set of results than Q . Note that $\text{evaln } s \ t \ (\text{post } s \ t)$ does not hold for a program for which one particular execution may diverge or get stuck.

Exercise: 5 stars, standard, especially useful (evaln_post_of_evaln) Prove the fact that if $\text{evaln } s \ t \ Q$ holds for some Q , then it holds for the smallest postcondition, namely $\text{post } s \ t$. Hint: in the let-binding case, you'll need to guess an appropriate intermediate postcondition for the subterm t_1 .

Lemma $\text{evaln_post_of_evaln} : \forall s \ t \ Q,$

$\text{evaln } s \ t \ Q \rightarrow$
 $\text{evaln } s \ t \ (\text{post } s \ t)$.

Proof using. *Admitted.*

□

Exercise: 2 stars, standard, especially useful (evaln_inv_post_qimpl) Prove the fact that if $\text{evaln } s \ t \ Q$ holds, then $\text{post } s \ t ==> Q$, i.e. the smallest-possible postcondition entails the postcondition Q . Hint: the proof is only one line long.

Lemma $\text{evaln_inv_post_qimpl} : \forall s \ t \ Q,$

$\text{evaln } s \ t \ Q \rightarrow$
 $\text{post } s \ t ==> Q$.

Proof using. *Admitted.*

□

48.2.3 Triples for Non-Deterministic Big-Step Semantics

A Hoare triple, written $\text{hoaren } t \ H \ Q$, asserts that, for any input state s satisfying the precondition H , all executions of t in that state s terminate and produce an output satisfying the postcondition Q .

Definition $\text{hoaren } (t:\text{trm}) \ (H:\text{hprop}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{Prop} :=$
 $\forall (s:\text{state}), \ H \ s \rightarrow \text{evaln } s \ t \ Q$.

A Separation Logic triple, written `triplen t H Q`, asserts that the term `t` satisfies a Hoare triple with precondition `H * H'` and postcondition `Q * H'`, for any heap predicate `H'` describing the “rest of the word”. Compared with the definition of `triple`, we simply replaced `hoare` with `hoaren`.

```
Definition triplen (t:trm) (H:hprop) (Q:val→hprop) : Prop :=  
  ∀ (H':hprop), hoaren t (H \* H') (Q \*+ H').
```

48.2.4 Triple-Style Reasoning Rules for a Non-Deterministic Semantics.

To derive reasoning rules for triples, we proceed in two steps, just as in the previous chapters: first, we establish rules for `hoaren`, then we derive rules for `triplen`. For the first part, the proofs are very similar to those from previous chapters. For the second part, the proofs are exactly the same as in the previous chapters.

Structural rules

```
Lemma hoaren_conseq : ∀ t H' Q' H Q,  
  hoaren t H' Q' →  
  H ==> H' →  
  Q' ===> Q →  
  hoaren t H Q.
```

Proof using.

unfolds hoaren. introv M MH MQ HF. applys× evaln_conseq.

Qed.

```
Lemma hoaren_hexists : ∀ t (A:Type) (J:A→hprop) Q,  
  (∀ x, hoaren t (J x) Q) →  
  hoaren t (hexists J) Q.
```

Proof using. *introv M. intros h (x&Hh). applys M Hh. Qed.*

```
Lemma hoaren_hpure : ∀ t (P:Prop) H Q,  
  (P → hoaren t H Q) →  
  hoaren t (\[P] \* H) Q.
```

Proof using.

*introv M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).
lets E: hempty_inv HP. subst. rewrite Fmap.union_empty_l. applys¬ M.*

Qed.

Rules for term constructs

```
Lemma hoaren_val : ∀ v H Q,  
  H ==> Q v →  
  hoaren (trm_val v) H Q.
```

Proof using. *introv M. intros h K. applys× evaln_val. Qed.*

```
Lemma hoaren_fix : ∀ f x t1 H Q,
```

$H \Rightarrow Q$ (val_fix $f x t1 \rightarrow$
 $\text{hoaren} (\text{trm_fix } f x t1) H Q.$

Proof using. introv M . intros $h K$. applys \times evaln_fix. Qed.

Lemma hoaren_app_fix : $\forall v1 v2 f x t1 H Q,$
 $v1 = \text{val_fix } f x t1 \rightarrow$
 $\text{hoaren} (\text{subst } x v2 (\text{subst } f v1 t1)) H Q \rightarrow$
 $\text{hoaren} (\text{trm_app } v1 v2) H Q.$

Proof using. introv $E M$. intros $h K$. applys \times evaln_app_fix. Qed.

Lemma hoaren_let : $\forall x t1 t2 H Q Q1,$
 $\text{hoaren} t1 H Q1 \rightarrow$
 $(\forall v1, \text{hoaren} (\text{subst } x v1 t2) (Q1 v1) Q) \rightarrow$
 $\text{hoaren} (\text{trm_let } x t1 t2) H Q.$

Proof using. introv $M1 M2$. intros $h K$. applys \times evaln_let. Qed.

Lemma hoaren_if : $\forall (b:\text{bool}) t1 t2 H Q,$
 $\text{hoaren} (\text{if } b \text{ then } t1 \text{ else } t2) H Q \rightarrow$
 $\text{hoaren} (\text{trm_if } b t1 t2) H Q.$

Proof using. introv M . intros $h K$. applys \times evaln_if. Qed.

Rules for primitive operations

Lemma hoaren_add : $\forall H n1 n2,$
 $\text{hoaren} (\text{val_add } n1 n2)$
 H
 $(\text{fun } r \Rightarrow \exists [r = \text{val_int } (n1 + n2)] \ \text{(* } H).$

Proof using.

intros. intros $h K$. applys \times evaln_add. rewrite \times hstar_hpure_l.

Qed.

Lemma hoaren_rand : $\forall n,$
 $n > 0 \rightarrow$
 $\text{hoaren} (\text{val_rand } n)$
 []
 $(\text{fun } r \Rightarrow \exists n1, \exists [0 \leq n1 < n] \ \text{(* } \exists [r = \text{val_int } n1]).$

Proof using.

introv N . intros $h K$. applys \times evaln_rand.

lets ->: hempty_inv K .

intros $n1 H1$. applys \times hexists_intro $n1$. rewrite \times hstar_hpure_l.
split \times . applys \times hpure_intro.

Qed.

Lemma hoaren_ref : $\forall H v,$
 $\text{hoaren} (\text{val_ref } v)$
 H
 $(\text{fun } r \Rightarrow (\exists p, \exists [r = \text{val_loc } p] \ \text{(* } p \rightsquigarrow v) \ \text{(* } H).$

Proof using.

```

intros. intros s1 K. applys evaln_ref. intros p D.
unfold update. applys hstar_intro K.
{ applys hexists_intro p. rewrite hstar_hpure_l.
  split×. applys hsingle_intro. }
{ applys× disjoint_single_of_not_indom. }

```

Qed.

```

Lemma hoaren_get : ∀ H v p,
  hoaren (val_get p)
    ((p ~~> v) \* H)
    (fun r ⇒ \[r = v] \* (p ~~> v) \* H).

```

Proof using.

```

intros. intros s K.
lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
lets E1: hsingl_inv P1. lets I1: indom_single p v.
applys evaln_get.
{ applys× Fmap.indom_union_l. subst. applys indom_single. }
{ rewrite hstar_hpure_l. split.
  { subst h1. rewrite× Fmap.read_union_l. rewrite× Fmap.read_single. }
  { applys× hstar_intro. } }

```

Qed.

```

Lemma hoaren_set : ∀ H w p v,
  hoaren (val_set (val_loc p) v)
    ((p ~~> w) \* H)
    (fun r ⇒ \[r = val_unit] \* (p ~~> v) \* H).

```

Proof using.

```

intros. intros s1 K.
lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
lets E1: hsingl_inv P1. lets I1: indom_single p v.
applys evaln_set.
{ applys× Fmap.indom_union_l. subst. applys indom_single. }
{ rewrite hstar_hpure_l. split×.
  { subst h1. rewrite× Fmap.update_union_l. rewrite× update_single.
    applys× hstar_intro.
    { applys× hsingl_intro. }
    { applys Fmap.disjoint_single_set D. }
    { applys indom_single. } } }

```

Qed.

```

Lemma hoaren_free : ∀ H p v,
  hoaren (val_free (val_loc p))
    ((p ~~> v) \* H)
    (fun r ⇒ \[r = val_unit] \* H).

```

Proof using.

```

intros. intros s1 K.
lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
lets E1: hsingl_inv P1. lets I1: indom_single p v.
applys evaln_free.
{ applys× Fmap.indom_union_l. subst. applys indom_single. }
{ rewrite hstar_hpure_l. split×.
  { subst h1. rewrite× Fmap.remove_union_single_l.
    { intros Dl. applys× Fmap.disjoint_inv_not_indom_both D Dl. } } }
Qed.

```

The proofs of reasoning rules for `triplens` are exactly the same as in the chapter Rules. For example, we show below the proof of the reasoning rule for let-bindings.

```

Lemma triplen_let : ∀ x t1 t2 H Q Q1,
  triplen t1 H Q1 →
  (forall v1, triplen (subst x v1 t2) (Q1 v1) Q) →
  triplen (trm_let x t1 t2) H Q.

```

Proof using.

```

introv M1 M2. intros HF. applys hoaren_let.
{ applys M1. }
{ intros v. applys hoaren_conseq M2; xsimpl. }

```

Qed.

Statements and proofs of reasoning rules for other term constructs can be directly adapted from those in the file `LibSepReference`.

48.3 More Details

48.3.1 Weakest-Precondition Style Presentation.

In chapter WPsem, we discussed several possible definitions of the weakest-precondition operator, namely `wp`, in Separation Logic. In this chapter, we present yet another possible definition, based on the judgment `evaln`.

Consider the judgment `evaln s t Q`. Assume the arguments were reordered, yielding the judgment `evaln t Q s`. Consider now the partial application `evaln t Q`. This partial application has type `state → Prop`, that is, `hprop`. This predicate `evaln t Q` denotes a predicate that characterizes the set of input states `s` in which the evaluation (or, rather, any possible evaluation) of the term `t` produces an output satisfying `Q`. It thus corresponds exactly to the notion of weakest precondition for Hoare Logic.

The judgment `hoarewpn t Q`, defined below, simply reorder the arguments of `evaln` so as to produce this weakest-precondition operator. Note that this is a Hoare Logic style predicate, which talks about the full state.

```

Definition hoarewpn (t:trm) (Q:val→hprop) : hprop :=
  fun s ⇒ evaln s t Q.

```

On top of `hoarewpn`, we can define the Separation Logic version of weakest precondition, written `wpn t Q`. At a high level, `wpn` extends `hoarewpn` with a description of “the rest of the world”. More precisely, `wpn t Q` describes a heap predicate, that, if extended with a heap predicate H describing the rest of the world, yields the weakest precondition with respect to the postcondition $Q \setminus *^+ H$, that is, Q extended with H .

Definition `wpn (t:trm) (Q:val → hprop) : hprop :=`
 $\forall H, H \setminus -* (\text{hoarewpn } t (Q \setminus *^+ H))$.

This definition is associated with the following introduction rule, which reads as follows: to prove $H ==> \text{wpn } t Q$, which asserts that t admits H as precondition and Q as postcondition, it suffices to prove that, for any H' describing the rest of the world, the entailment $H \setminus * H' ==> \text{hoarewpn } t (Q \setminus *^+ H')$ holds, asserting that the term t admits, in Hoare logic, the precondition $H \setminus * H'$ and the postcondition $Q \setminus *^+ H'$.

Lemma `wpn_of_hoarewpn : ∀ H t Q,`
 $(\forall H', H \setminus * H' ==> \text{hoarewpn } t (Q \setminus *^+ H')) \rightarrow$
 $H ==> \text{wpn } t Q$.

Proof using.

```
intros M. unfolds wpn. applys himpl_hforall_r. intros H'.
applys himpl_hwand_r. rewrite hstar_comm. applys M.
```

Qed.

Exercise: 4 stars, standard, especially useful (`wpn_equiv`) Prove the standard equivalence $(H ==> \text{wpn } t Q) \leftrightarrow (\text{triplen } t H Q)$ to relate the predicates `wpn` and `triplen`. Hint: using lemma `wpn_of_hoarewpn` can be helpful.

Lemma `wpn_equiv : ∀ H t Q,`
 $(H ==> \text{wpn } t Q) \leftrightarrow (\text{triplen } t H Q)$.

Proof using. Admitted.

□

Remark: as mentioned in the chapter `WPsem`, it is possible to define the predicate `triplen t H Q` as $H ==> \text{wpn } t Q$, that is, to define `triplen` as a notion derived from `hoarewpn` and `wpn`.

48.3.2 Hoare Logic WP-Style Rules for a Non-Deterministic Semantics.

Given that the semantics expressed by the predicate `evaln` has a weakest-precondition flavor, there are good chances that deriving weakest-precondition style reasoning rules from `evaln` could be even easier than deriving the rules for `triplen`. Thus, let us investigate whether this is indeed the case, by stating and proving reasoning rules for the judgments `hoarewpn` and `wpn`. We begin with rules for `hoarewpn`.

Structural rules

Lemma `hoarewpn_conseq : ∀ t Q1 Q2,`

$Q1 \implies Q2 \rightarrow$
 $(\text{hoarewpn } t \ Q1) \implies (\text{hoarewpn } t \ Q2).$

Proof using.

`intros W. unfold hoarewpn. intros h K. applys evaln_conseq K W.`

Qed.

Rules for term constructs

Lemma `hoarewpn_val` : $\forall v Q,$
 $Q v \implies \text{hoarewpn} (\text{trm_val } v) Q.$

Proof using.

`unfold hoarewpn. intros. intros h K. applys evaln_val.`

Qed.

Lemma `hoarewpn_fix` : $\forall f x t Q,$
 $Q (\text{val_fix } f x t) \implies \text{hoarewpn} (\text{trm_fix } f x t) Q.$

Proof using.

`unfold hoarewpn. intros. intros h K. applys evaln_fix.`

Qed.

Lemma `hoarewpn_app_fix` : $\forall f x v1 v2 t1 Q,$
 $v1 = \text{val_fix } f x t1 \rightarrow$
 $\text{hoarewpn} (\text{subst } x v2 (\text{subst } f v1 t1)) Q \implies \text{hoarewpn} (\text{trm_app } v1 v2) Q.$

Proof using.

`unfold hoarewpn. intros. intros h K. applys evaln_app_fix.`

Qed.

Lemma `hoarewpn_let` : $\forall x t1 t2 Q,$
 $\text{hoarewpn } t1 (\text{fun } v \Rightarrow \text{hoarewpn} (\text{subst } x v t2) Q)$
 $\implies \text{hoarewpn} (\text{trm_let } x t1 t2) Q.$

Proof using.

`unfold hoarewpn. intros. intros h K. applys evaln_let.`

Qed.

Lemma `hoarewpn_if` : $\forall b t1 t2 Q,$
 $\text{hoarewpn} (\text{if } b \text{ then } t1 \text{ else } t2) Q \implies \text{hoarewpn} (\text{trm_if } b t1 t2) Q.$

Proof using.

`unfold hoarewpn. intros. intros h K. applys evaln_if.`

Qed.

Rules for primitives. We state their specifications following the presentation described near the end of chapter Wand.

Lemma `hoarewpn_add` : $\forall Q n1 n2,$
 $(Q (\text{val_int } (n1 + n2))) \implies \text{hoarewpn} (\text{val_add } (\text{val_int } n1) (\text{val_int } n2)) Q.$

Proof using.

`unfolds hoarewpn. intros. intros h K. applys evaln_add.`

Qed.

Lemma hoarewpn_rand : $\forall Q n,$
 $n > 0 \rightarrow$
 $(\forall n1, [0 \leq n1 < n] \rightarrow Q (\text{val_int } n1))$
 $\implies \text{hoarewpn} (\text{val_rand} (\text{val_int } n)) Q.$

Proof using.

unfolds hoarewpn. introv N. xsimpl. intros h K.
applys evaln_rand. intros n1 H1. lets K': hforall_inv K n1.
rewrite hwand_hpure_l in K'.

Qed.

Lemma hoarewpn_ref : $\forall Q v,$
 $(\forall p, (p \rightsquigarrow v) \rightarrow Q (\text{val_loc } p)) \implies \text{hoarewpn} (\text{val_ref } v) Q.$

Proof using.

unfolds hoarewpn. intros. intros h K. *applys* evaln_ref. intros p D.
lets K': hforall_inv (rm K) p.
applys hwand_inv (single p v) K'.
{ *applys* hsingl_intro. }
{ *applys* disjoint_single_of_not_indom. }

Qed.

Lemma hoarewpn_get : $\forall v p Q,$
 $(p \rightsquigarrow v) \backslash* (p \rightsquigarrow v \rightarrow Q v) \implies \text{hoarewpn} (\text{val_get } p) Q.$

Proof using.

unfolds hoarewpn. intros. intros h K.
lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
*forwards**: hwand_inv h1 P2.
lets E1: hsingl_inv P1. *lets* I1: indom_single p v.
applys evaln_get.
{ *applys* Fmap.indom_union_l. *subst*. *applys* indom_single. }
{ *subst* h1. *rewrite* Fmap.read_union_l. *rewrite* Fmap.read_single. }

Qed.

Lemma hoarewpn_set : $\forall v w p Q,$
 $(p \rightsquigarrow v) \backslash* (p \rightsquigarrow w \rightarrow Q \text{ val_unit}) \implies \text{hoarewpn} (\text{val_set } p w) Q.$

Proof using.

unfolds hoarewpn. intros. intros h K.
lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
lets E1: hsingl_inv P1. *lets* I1: indom_single p v.
forwards: hwand_inv (single p w) P2.
{ *applys* hsingl_intro. }
{ *subst* h1. *applys* Fmap.disjoint_single_set D. }
{ *applys* evaln_set.
{ *applys* Fmap.indom_union_l. *subst*. *applys* indom_single. }
{ *subst* h1. *rewrite* Fmap.update_union_l. *rewrite* update_single. } }

Qed.

```

Lemma hoarewpn_free : ∀ v p Q,
  (p ~~> v) \* (Q val_unit) ==> hoarewpn (val_free p) Q.

Proof using.
  unfolds hoarewpn. intros. intros h K.
  lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
  lets E1: hsingl_inv P1. lets I1: indom_single p v.
  applys_eq evaln_free.
  { applys× Fmap.indom_union_l. subst. applys indom_single. }
  { subst h1. rewrite¬ Fmap.remove_union_single_l.
    intros Dl. applys× Fmap.disjoint_inv_not_indom_both D Dl. }
Qed.

```

48.3.3 Separation Logic WP-Style Rules for a Non-Deterministic Semantics.

In the previous section, we established reasoning rules for `hoarewpn`. Based on these rules, we are ready to derive reasoning rules for `wpn`.

Structural rules

```

Lemma wpn_conseq_frame : ∀ t H Q1 Q2,
  Q1 \*+ H ==> Q2 →
  (wpn t Q1) \* H ==> (wpn t Q2).

```

Proof using.

```

  introv M. unfold wpn. xsimpl.
  intros H'. xchange (hforall_specialize (H \* H')).
  applys hoarewpn_conseq. xchange M.

```

Qed.

```

Lemma wpn_ramified_trans : ∀ t H Q1 Q2,
  H ==> (wpn t Q1) \* (Q1 \-* Q2) →
  H ==> (wpn t Q2).

```

Proof using.

```

  introv M. xchange M. applys wpn_conseq_frame. applys qwand_cancel.

```

Qed.

Rules for term constructs

```

Lemma wpn_val : ∀ v Q,
  Q v ==> wpn (trm_val v) Q.

```

Proof using.

```

  intros. unfold wpn. xsimpl. intros H'.
  xchange (» hoarewpn_val (Q \*+ H')).

```

Qed.

```

Lemma wpn_fix : ∀ f x t Q,
  Q (val_fix f x t) ==> wpn (trm_fix f x t) Q.

```

Proof using.

```
intros. unfold wpn. xsimpl. intros H'.
xchange (» hoarewpn_fix (Q \*+ H')).
```

Qed.

Lemma wpn_app_fix : $\forall f x v1 v2 t1 Q,$
 $v1 = \text{val_fix } f x t1 \rightarrow$
 $\text{wpn} (\text{subst } x v2 (\text{subst } f v1 t1)) Q ==> \text{wpn} (\text{trm_app } v1 v2) Q.$

Proof using.

```
intros. unfold wpn. xsimpl. intros H'.
xchange (hforall_specialize H').
applys× hoarewpn_app_fix.
```

Qed.

Exercise: 4 stars, standard, especially useful (wpn_let) Derive the reasoning rule wpn_let from hoarewpn_let.

Lemma wpn_let : $\forall x t1 t2 Q,$
 $\text{wpn} t1 (\text{fun } v \Rightarrow \text{wpn} (\text{subst } x v t2) Q) ==> \text{wpn} (\text{trm_let } x t1 t2) Q.$

Proof using. Admitted.

□

Lemma wpn_if : $\forall b t1 t2 Q,$
 $\text{wpn} (\text{if } b \text{ then } t1 \text{ else } t2) Q ==> \text{wpn} (\text{trm_if } b t1 t2) Q.$

Proof using.

```
intros. unfold wpn. xsimpl. intros H'.
xchange (hforall_specialize H').
applys hoarewpn_if.
```

Qed.

Rules for primitives.

Lemma wpn_add : $\forall Q n1 n2,$
 $(Q (\text{val_int } (n1 + n2))) ==> \text{wpn} (\text{val_add } (\text{val_int } n1) (\text{val_int } n2)) Q.$

Proof using.

```
intros. unfold wpn. xsimpl. intros H'.
xchange (» hoarewpn_add (Q \*+ H')).
```

Qed.

Lemma wpn_rand : $\forall Q n,$
 $n > 0 \rightarrow$
 $(\forall n1, [0 \leq n1 < n] \rightarrow Q (\text{val_int } n1))$
 $==> \text{wpn} (\text{val_rand } (\text{val_int } n)) Q.$

Proof using.

```
introv N. unfold wpn. xsimpl. intros H'.
applys himpl_trans; [| applys× hoarewpn_rand ].
```

```
xsiml. intros n1. xchange (hforall_specialize n1).
```

```
intros H1. rewrite  $\times$  hwand_hpure_l.
```

Qed.

Lemma *wpn_ref* : $\forall Q v,$

```
( $\forall p, (p \rightsquigarrow v) \rightarrow Q (\text{val\_loc } p)$ ) ==> wpn (val_ref v) Q.
```

Proof using.

```
intros. unfold wpn. xsiml. intros H'.
```

```
applys himpl_trans; [| applys hoarewpn_ref ].
```

```
xsiml. intros p. xchange (hforall_specialize p).
```

Qed.

Lemma *wpn_get* : $\forall v p Q,$

```
( $p \rightsquigarrow v) \rightarrow (* (p \rightsquigarrow v \rightarrow Q v) ==> \text{wpn} (\text{val\_get } p) Q.$ 
```

Proof using.

```
intros. unfold wpn.
```

```
applys himpl_hforall_r. intros H'. applys himpl_hwand_r.
```

```
rewrite hstar_comm.
```

```
applys himpl_trans; [| applys hoarewpn_get v ]. simpl.
```

```
rewrite hstar_assoc. applys himpl_frame_r.
```

```
xsiml.
```

Qed.

Lemma *wpn_set* : $\forall v w p Q,$

```
( $p \rightsquigarrow v) \rightarrow (* (p \rightsquigarrow w \rightarrow Q \text{ val\_unit}) ==> \text{wpn} (\text{val\_set } p w) Q.$ 
```

Proof using.

```
intros. unfold wpn.
```

```
applys himpl_hforall_r. intros H'. applys himpl_hwand_r.
```

```
rewrite hstar_comm.
```

```
applys himpl_trans; [| applys hoarewpn_set v ]. simpl.
```

```
rewrite hstar_assoc. applys himpl_frame_r.
```

```
xsiml.
```

Qed.

Lemma *wpn_free* : $\forall v p Q,$

```
( $p \rightsquigarrow v) \rightarrow (* (Q \text{ val\_unit}) ==> \text{wpn} (\text{val\_free } p) Q.$ 
```

Proof using.

```
intros. unfold wpn.
```

```
applys himpl_hforall_r. intros H'. applys himpl_hwand_r.
```

```
applys himpl_trans; [| applys hoarewpn_free v ]. xsiml.
```

Qed.

Rules for primitives, alternative presentation using triples.

Lemma *triplen_add* : $\forall n1 n2,$

```
triplen (val_add n1 n2)
```

```
\[]
```

```
(fun r => \[r = val_int (n1 + n2)]).
```

Proof using.

```
intros. rewrite ← wpn_equiv.  
applys himpl_trans; [| applys wpn_add ]. xsimpl×.
```

Qed.

Lemma triplen_rand : $\forall n,$

$n > 0 \rightarrow$

triplen (val_rand n)

\[]

(fun r => \exists n1, \[0 \leq n1 < n] \(* \[r = val_int n1]).

Proof using.

```
introv N. rewrite ← wpn_equiv.  
applys himpl_trans; [| applys× wpn_rand ]. xsimpl×.
```

Qed.

Lemma triplen_ref : $\forall v,$

triplen (val_ref v)

\[]

(fun r => \exists p, \[r = val_loc p] \(* p \sim\sim> v).

Proof using.

```
intros. rewrite ← wpn_equiv.  
applys himpl_trans; [| applys wpn_ref ]. xsimpl×.
```

Qed.

Lemma triplen_get : $\forall v p,$

triplen (val_get p)

(p \sim\sim> v)

(fun r => \[r = v] \(* (p \sim\sim> v)).

Proof using.

```
intros. rewrite ← wpn_equiv.  
applys himpl_trans; [| applys wpn_get ]. xsimpl×.
```

Qed.

Lemma triplen_set : $\forall w p v,$

triplen (val_set (val_loc p) v)

(p \sim\sim> w)

(fun _ => p \sim\sim> v).

Proof using.

```
intros. rewrite ← wpn_equiv.  
applys himpl_trans; [| applys wpn_set ]. xsimpl×.
```

Qed.

Lemma triplen_free : $\forall p v,$

triplen (val_free (val_loc p))

(p \sim\sim> v)

```
(fun _ => []).
```

Proof using.

```
intros. rewrite ← wpn_equiv.
applys himpl_trans; [| applys wpn_free ]. xsimpl×.
```

Qed.

Exercise: 4 stars, standard, optional (wpn_rand_of_triplen_rand) The proof of lemma triplen_rand shows that a triple-based specification of val_rand is derivable from a wp-style specification. In this exercise, we aim to prove the reciprocal. Concretely, prove the following specification by exploiting wpn_rand. Hint: make use of wpn_equiv.

Lemma wpn_rand_of_triplen_rand : $\forall n Q,$

$$\begin{aligned} n > 0 \rightarrow \\ (\forall n_1, \exists [0 \leq n_1 < n] \rightarrow Q (\text{val_int } n_1)) \\ ==> \text{wpn} (\text{val_rand} (\text{val_int } n)) Q. \end{aligned}$$

Proof using. Admitted.

□

48.4 Optional Material

So far, this chapter has focused on handling non-determinism in the context of using a big-step semantics. In this bonus section, we investigate the treatment of non-determinism using a small-step semantics. Moreover, we establish equivalence proofs relating (non-deterministic) small-step and big-step semantics.

48.4.1 Interpretation of evaln w.r.t. eval and terminates

The predicate **terminates** $s t$ asserts that all executions of the configuration t/s terminate—none of them diverges or get stuck. Its definition is a simplified version of **evaln** where all occurrences of Q are removed. In the rule for let-bindings, namely **terminates_let**, the quantification over the configuration $v1/s2$ is done by referring to the big-step judgment **eval**.

```
Inductive terminates : state → trm → Prop :=
| terminates_val : ∀ s v,
  terminates s (trm_val v)
| terminates_fix : ∀ s f x t1,
  terminates s (trm_fix f x t1)
| terminates_app_fix : ∀ s v1 v2 f x t1,
  v1 = val_fix f x t1 →
  terminates s (subst x v2 (subst f v1 t1)) →
  terminates s (trm_app v1 v2)
| terminates_let : ∀ s x t1 t2,
```

```

terminates s t1 →
  (forall v1 s2, eval s t1 s2 v1 → terminates s2 (subst x v1 t2)) →
    terminates s (trm_let x t1 t2)
| terminates_if : forall s b t1 t2,
  terminates s (if b then t1 else t2) →
  terminates s (trm_if (val_bool b) t1 t2)
| terminates_add : forall s n1 n2,
  terminates s (val_add (val_int n1) (val_int n2))
| terminates_rand : forall s n,
  n > 0 →
  terminates s (val_rand (val_int n))
| terminates_ref : forall s v,
  terminates s (val_ref v)
| terminates_get : forall s p,
  Fmap.indom s p →
  terminates s (val_get (val_loc p))
| terminates_set : forall s p v,
  Fmap.indom s p →
  terminates s (val_set (val_loc p) v)
| terminates_free : forall s p,
  Fmap.indom s p →
  terminates s (val_free (val_loc p)).

```

Section EvalnTerminates.

Hint Constructors **eval evaln**.

Exercise: 5 stars, standard, especially useful (**evaln_iff_terminates_and_post**)
 Prove that **evaln** s t Q is equivalent to the conjunction of **terminates** s t and to a partial correctness result asserting that if an evaluation of t/s terminates on some result then this result satisfies Q.

Lemma **evaln_iff_terminates_and_post** : forall s t Q,
 evaln s t Q ↔ (**terminates** s t ∧ (forall v s', **eval** s t s' v → Q v s')).

Proof using. Admitted.

□

Let **Any** denotes the postcondition that accepts any result.

Definition **Any** : val → state → Prop :=
 fun v s ⇒ **True**.

Hint Unfold **Any**.

Exercise: 2 stars, standard, especially useful (**terminates_iff_evaln_any**) Prove that **terminates** s t is equivalent to **evaln** s t **Any**. Hint: exploit **evaln_iff_terminates_and_post**.

```
Lemma terminates_iff_evaln_any : ∀ s t,
  terminates s t ↔ evaln s t Any.
```

Proof using. Admitted.

□

End EvalnTerminates.

48.4.2 Small-Step Evaluation Relation

The judgment **step** $s t s' t'$ describes the small-step reduction relation: it asserts that the program configuration (s,t) can take one reduction step towards the program configuration (s',t') . Its definition, shown below, is standard.

Inductive **step** : state → trm → state → trm → Prop :=

$$\begin{aligned} & \mid \text{step_let_ctx} : \forall s1 s2 x t1 t1' t2, \\ & \quad \text{step } s1 t1 s2 t1' \rightarrow \\ & \quad \text{step } s1 (\text{trm_let } x t1 t2) s2 (\text{trm_let } x t1' t2) \\ \\ & \mid \text{step_fix} : \forall s f x t1, \\ & \quad \text{step } s (\text{trm_fix } f x t1) s (\text{val_fix } f x t1) \\ & \mid \text{step_app_fix} : \forall s v1 v2 f x t1, \\ & \quad v1 = \text{val_fix } f x t1 \rightarrow \\ & \quad \text{step } s (\text{trm_app } v1 v2) s (\text{subst } x v2 (\text{subst } f v1 t1)) \\ & \mid \text{step_if} : \forall s b t1 t2, \\ & \quad \text{step } s (\text{trm_if } (\text{val_bool } b) t1 t2) s (\text{if } b \text{ then } t1 \text{ else } t2) \\ & \mid \text{step_let} : \forall s x t2 v1, \\ & \quad \text{step } s (\text{trm_let } x v1 t2) s (\text{subst } x v1 t2) \\ \\ & \mid \text{step_add} : \forall s n1 n2, \\ & \quad \text{step } s (\text{val_add } (\text{val_int } n1) (\text{val_int } n2)) s (\text{val_int } (n1 + n2)) \\ & \mid \text{step_rand} : \forall s n n1, \\ & \quad 0 \leq n1 < n \rightarrow \\ & \quad \text{step } s (\text{val_rand } (\text{val_int } n)) s (\text{val_int } n1) \\ & \mid \text{step_ref} : \forall s v p, \\ & \quad \text{Fmap.indom } s p \rightarrow \\ & \quad \text{step } s (\text{val_ref } v) (\text{Fmap.update } s p v) (\text{val_loc } p) \\ & \mid \text{step_get} : \forall s p, \\ & \quad \text{Fmap.indom } s p \rightarrow \\ & \quad \text{step } s (\text{val_get } (\text{val_loc } p)) s (\text{Fmap.read } s p) \\ & \mid \text{step_set} : \forall s p v, \end{aligned}$$

```

Fmap.indom s p →
step s (val_set (val_loc p) v) (Fmap.update s p v) val_unit
| step_free : ∀ s p,
  Fmap.indom s p →
  step s (val_free (val_loc p)) (Fmap.remove s p) val_unit.

```

Consider a configuration (s, t) , where t is not a value. If this configuration cannot take any reduction step, it is said to be “stuck”.

The judgment **evals** $s t s' t'$ corresponds to the reflexive-transitive closure of **step**. Concretely, this judgment asserts that the configuration (s, t) can reduce in zero, one, or several evals to (s', t') .

```

Inductive evals : state → trm → state → trm → Prop :=
| evals_refl : ∀ s t,
  evals s t s t
| evals_step : ∀ s1 s2 s3 t1 t2 t3,
  step s1 t1 s2 t2 →
  evals s2 t2 s3 t3 →
  evals s1 t1 s3 t3.

```

48.4.3 Small-Step Characterization of **evalns**: Attempts

For a non-deterministic language, **evals** $s t s' t'$ asserts that there exists one possible evaluation from (s, t) to (s', t') . This judgment says nothing about all other possible evaluations.

On the contrary, the predicate **evaln** $s t Q$, introduced earlier in this chapter, says something about all possible evaluations. More precisely, **evaln** $s t Q$ asserts that all possible evaluations of (s, t) reach a final configuration satisfying the postcondition Q .

We are thus interested in defining a judgment of the form **evalns** $s t Q$, that is logically equivalent to **evaln** $s t Q$, but whose definition is based on the small-step semantics.

In the particular case of a deterministic semantics, we could define **evalns** $s t Q$ in terms of the transitive evaluation relation **evals** as follows.

```

Definition evalns_attempt_1 (s:state) (t:trm) (Q:val→hprop) : Prop :=
  ∃ v s', evals s t s' (trm_val v) ∧ Q v s'.

```

Let's check out several candidate definitions for the judgment **evalns**.

The definition **evalns_attempt_2** $s t Q$ asserts that any execution starting from configuration (s, t) and ending on a final configuration (s', v) is such that the final configuration satisfies the postcondition Q . Yet, this definition fails to rule out the possibility of executions that get stuck. Thus, in it is only applicable for semantics that include error-propagation rules, and for which there are no stuck terms.

```

Definition evalns_attempt_2 (s:state) (t:trm) (Q:val→hprop) : Prop :=
  ∀ v s', evals s t s' (trm_val v) → Q v s'.

```

The definition **evalns_attempt_3** $s t Q$ asserts that any execution starting from configuration (s, t) and reaching a state (s', t') is such that either t' is a value v and (s', v) satisfies the

postcondition Q , or (s',t') can take a step. Yet, this definition fails to capture the fact that all execution of t should terminate. Thus, it is only useful for capturing partial correctness properties.

```
Definition evalns_attempt_3 (s:state) (t:trm) (Q:val → hprop) : Prop :=
  ∀ s2 t2, evals s t s2 t2 →
    (exists v2, t2 = trm_val v2 ∧ Q v2 s2)
    ∨ (exists s3 t3, step s2 t2 s3 t3).
```

The definition $\text{evalns_attempt_4 } s \ t \ Q$ asserts that every execution prefix starting from (s,t) may be completed into an execution that does terminate on a configuration that satisfies the postcondition Q .

```
Definition evalns_attempt_4 (s:state) (t:trm) (Q:val → hprop) : Prop :=
  ∀ s2 t2, evals s t s2 t2 →
  ∃ v3 s3, evals s2 t2 s3 (trm_val v3) ∧ Q v3 s3.
```

Solution follows.

Consider the following program.

OCaml:

```
let rec f () = if (val_rand 2) = 0 then () else f ()
```

Consider an execution that has already performed a number of recursive calls. This execution may terminate in a finite number of evaluation evals. Indeed, the next call to `val_rand` may return zero. However, not all executions terminate. Indeed, the execution path where all calls to `val_rand` return 1, the program runs for ever.

The key challenge is to capture the property that “every possible execution terminates”. To that end, let’s consider yet another approach, based on the idea of bounding the number of execution steps.

The judgment **nbevals** $n \ s \ t \ s' \ t'$, defined below, asserts that the configuration (s,t) may reduce in exactly n steps to (s',t') . Its definition follows that of the judgment **evals**, only with an extra argument for counting the number of steps.

```
Inductive nbevals : nat → state → trm → state → trm → Prop :=
| nbevals_refl : ∀ s t,
  nbevals 0 s t s t
| nbevals_step : ∀ (n:nat) s1 s2 s3 t1 t2 t3,
  step s1 t1 s2 t2 →
  nbevals n s2 t2 s3 t3 →
  nbevals (S n) s1 t1 s3 t3.
```

Using the judgment **nbevals**, we are able to bound the length of all executions. The judgment **steps_at_most** $nmax \ s \ t$, defined below, asserts that there does not exist any execution starting from (s,t) that exceeds $nmax$ reduction steps.

```
Definition steps_at_most (nmax:nat) (s:state) (t:trm) : Prop :=
  ∀ (n:nat) s2 t2, n > nmax → ¬ (nbevals n s t s2 t2).
```

The judgment $\text{evalns_attempt_5 } s \ t \ Q$ corresponds to the conjunction of the judgment $\text{evalns_attempt_3 } s \ t \ Q$, which asserts partial correctness, and of the judgment $\exists nmax$,

`steps_at_most nmax s t`, which asserts that there exists an upper bound to the length of all possible executions.

```
Definition evalns_attempt_5 (s:state) (t:trm) (Q:val → hprop) : Prop :=
  (evalns_attempt_3 s t Q)
  ∧ (exists nmax, steps_at_most nmax s t).
```

Is the definition of `evalns_attempt_5` satisfying? Not quite. First, this definition is fairly complex, and not so easy to work with. Second, and perhaps most importantly, this definition does not apply to all programming languages. It applies only to semantics for which each configuration admits at most a finite number of possible transitions.

If we view the possible executions of a program as a tree, with each branch corresponding to a possible execution, then definition `evalns_attempt_5` only properly captures the notion of total correctness for “finitely branching trees”, in which every node has a finite number of branches.

Concretely, the definition `evalns_attempt_5` would rule out legitimate programs in a language that includes an “unbounded” sources of non-determinism. For example, consider a random number generator that applies to the unit argument and may return any integer value in \mathbb{Z} . Such an operator could be formalized as follows.

```
Parameter val_unbounded_rand : val.
Parameter evaln_unbounded_rand : ∀ s Q,
  (forall n1, Q n1 s) →
  evaln s (val_unbounded_rand val_unit) Q.
```

Solution: Consider the following program.

OCaml:

```
let rec f n = if n > 0 then f (n-1) else () in f (unbounded_rand())
```

The number of execution evals can be arbitrary. Yet, any given program execution terminates.

Arguably, a stand-alone piece of hardware does not feature such “unbounded” source of non-determinism, because each transition at the hardware level involves the manipulation of at most a finite number of bits. However, as soon as I/O is involved, unbounded non-determinism may arise. For example, a language that features an `input_string` method allowing the user to input strings of arbitrary size is a language with a source of unbounded non-determinism.

Remark: the fact that `evalns_attempt_5` only properly captures total correctness for finitely branching trees is related to a result known as König’s lemma. This result from graph theory, in the particular case of trees, asserts that “every infinite tree contains either a vertex of infinite degree or an infinite path”, or equivalently, asserts that “a finitely branching tree is infinite iff it has an infinite path”.

The contraposed statement asserts that a finitely branching tree (corresponding to executions in a language with bounded determinism) is finite (i.e., admits a bound on the depth) iff it has no infinite path (i.e., if all executions terminate).

In summary:

- `evalns_attempt_1` applies only to deterministic semantics.
- `evalns_attempt_2` applies only to complete semantics, i.e., semantics without stuck terms.
- `evalns_attempt_3` captures partial correctness only, and says nothing about termination.
- `evalns_attempt_4` also fails to properly capture termination.
- `evalns_attempt_5` captures total correctness only for semantics that feature only bounded sources of non-determinism, i.e., with a finite number of possible transitions from each configuration.

Our attempts to define a judgment `evalns s t Q` in terms of `evals` or in terms of its depth-indexed variant `nbevals` have failed. It thus appears necessary to search instead for a definition of `evalns` expressed directly in terms of the one-step reduction relation `step`.

48.4.4 Small-Step Characterization of `evaln`: A Solution

In what follows, we present an inductive definition for `evalns s t Q`. To begin with, let us consider the particular case of a deterministic language.

The predicate `evalds s t Q` asserts that the (deterministic) evaluation starting from configuration (s,t) produces an output satisfying the predicate Q . It is defined inductively as follows.

- Base case: `evalds s v Q` holds if the postcondition holds of this current state, that is, $Q \vee s$ holds.
- Step case: `evalds s t Q` holds if (s,t) one-step reduces to the configuration (s',t') and `evalds s' t' Q` holds.

```
Inductive evalds : state -> trm -> (val -> hprop) -> Prop :=
| evalds_val : ∀ s v Q,
  Q v s →
  evalds s v Q
| evalds_step : ∀ s t s' t' Q,
  step s t s' t' →
  evalds s' t' Q →
  evalds s t Q.
```

Let us now generalize the inductive definition of `evalds` to the general case of non-deterministic semantics.

- Base case: it does not change, that is, `evalns s v Q` requires $Q \vee s$.

- Step case: this case needs to be refined to account for all possible evaluations, and not just the unique possible evaluation. There are two requirements. First, **evalns** s t Q requires that the configuration (s,t) is not stuck, that is, it requires the existence of at least one possible reduction step. Second, **evalns** s t Q requires that, for any configuration (s',t') that (s,t) might reduce to, the property **evalns** s' t' Q holds.

The definition of **evalns** may thus be formalized as shown below.

```
Inductive evalns : state -> trm -> (val -> hprop) -> Prop :=
| evalns_val : ∀ s v Q,
  Q v s →
  evalns s v Q
| evalns_step : ∀ s t Q,
  (exists s' t', step s t s' t') →
  (forall s' t', step s t s' t' → evalns s' t' Q) →
  evalns s t Q.
```

Observe how this definition allows for **evalns** s t Q to hold even for the program such as the counter-example considered for definition **evalns_attempt_5**, i.e., the program that performs *unbounded_rand()* recursive calls. The judgment **evalns** holds for this program, although there exists no bound on the depth of the corresponding derivation. This possibility stems from the fact that the $\forall s' t'$ quantification in constructor **evalns_step** introduces an infinite branching factor in the derivation tree, for a language that includes the primitive operation *unbounded_rand*.

Exercise: 2 stars, standard, optional (evalns_val_inv) Prove the following inversion lemma, which asserts that **evalns** s v Q implies Q s v .

```
Lemma evalns_val_inv : ∀ s v Q,
  evalns s v Q →
  Q v s.
```

Proof using. *Admitted.*

□

In the remaining of this chapter, we present:

- a definition of Separation Logic triples based on **evalns**; the corresponding judgment is named **triplens**
- the proof of the reasoning rules associated with **triplens**;
- a formal proof of equivalence between **evalns** and **evaln**, relating the small-step-based definition to the big-step-based definition introduced in the first part of this chapter.

48.4.5 Triples for Small-Step Semantics

The judgment `hoaren t H Q` defines Hoare triples in terms of the small-step-based judgment `evalns`. It mimics the definition of `hoaren`, which was used for defining triples with respect to the big-step judgment `evaln`.

Definition `hoarens (t:trm) (H:hprop) (Q:val → hprop) : Prop :=`
 $\forall (s:\text{state}), H\ s \rightarrow \text{evalns}\ s\ t\ Q.$

The judgment `triplens` is the counterpart of `triplen`, introducing Separation Logic triples defined w.r.t. `hoarens`.

Definition `triplens (t:trm) (H:hprop) (Q:val → hprop) : Prop :=`
 $\forall (H':\text{hprop}), \text{hoarens}\ t\ (H\ \backslash*\ H')\ (Q\ \backslash*+\ H').$

There is nothing new in deriving rules for `triplens` from rules for `hoarens`. Thus, in what follows, we'll focus on the novel aspects, which consists of deriving reasoning rules for `evalns` and for `hoarens`, with respect to the small-step semantics captured by the relation `step`.

48.4.6 Reasoning Rules for `seval`

First, we establish reasoning rules for `evalns`.

The structural rules for `evalns` asserts that `evalns s t Q` is covariant in the postcondition `Q`.

Lemma `evalns_conseq : ∀ s t Q Q',`
`evalns s t Q' →`
`Q' ==> Q →`
`evalns s t Q.`

Proof using.

```
intros M WQ. induction M.
{ applys evalns_val. applys× WQ. }
{ rename H1 into IH.
  applys evalns_step.
  { auto. }
  { intro HR. applys× IH. } }
```

Qed.

The rules for term constructs are established next. Note that there is no rule stated here for the case of values, because such a rule is already provided by the constructor `evalns_val`.

Lemma `evalns_fix : ∀ s f x t1 Q,`
`Q (val_fix f x t1) s →`
`evalns s (trm_fix f x t1) Q.`

Proof using.

```
intros M. applys evalns_step.
{ do 2 esplit. constructor. }
{ intro R. inverts R. { applys evalns_val. applys M. } }
```

Qed.

```
Lemma evalns_app_fix : ∀ s f x v1 v2 t1 Q,  
  v1 = val_fix f x t1 →  
  evalns s (subst x v2 (subst f v1 t1)) Q →  
  evalns s (trm_app v1 v2) Q.
```

Proof using.

```
  introv E M. applys evalns_step.  
  { do 2 esplit. constructors×. }  
  { introv R. invert R; try solve [intros; false].  
    introv → → → → R. inverts E. applys M. }
```

Qed.

Exercise: 5 stars, standard, especially useful (evalns_let) Prove the big-step reasoning rule for let-bindings for **evalns**.

```
Lemma evalns_let : ∀ s x t1 t2 Q1 Q,  
  evalns s t1 Q1 →  
  (forall s1 v1, Q1 v1 s1 → evalns s1 (subst x v1 t2) Q) →  
  evalns s (trm_let x t1 t2) Q.
```

Proof using. Admitted.

□

```
Lemma evalns_if : ∀ s b t1 t2 Q,  
  evalns s (if b then t1 else t2) Q →  
  evalns s (trm_if b t1 t2) Q.
```

Proof using.

```
  introv M. applys evalns_step.  
  { do 2 esplit. constructors×. }  
  { introv R. inverts R; tryfalse. { applys M. } }
```

Qed.

48.4.7 Reasoning Rules for hoarens

Let's now prove reasoning rules for the Hoare triples judgment **hoarens**.

The consequence rule exploits the covariance result for **evalns**.

```
Lemma hoarens_conseq : ∀ t H' Q' H Q,  
  hoarens t H' Q' →  
  H ==> H' →  
  Q' ===> Q →  
  hoarens t H Q.
```

Proof using.

```
  introv M MH MQ HF. applys evalns_conseq M MQ. applys× MH.
```

Qed.

The other two structural rules, which operate on the precondition, admit exactly the same proofs as in the previous chapters.

Lemma hoarens_hexists : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\forall x, \text{hoarens } t (J x) Q) \rightarrow$
 $\text{hoarens } t (\text{hexists } J) Q.$

Proof using. *introv M. intros h (x&Hh). applys M Hh. Qed.*

Lemma hoarens_hpure : $\forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{hoarens } t H Q) \rightarrow$
 $\text{hoarens } t (\setminus [P] \setminus * H) Q.$

Proof using.

introv M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).
lets E: hempty_inv HP. subst. rewrite Fmap.union_empty_l. applys¬ M.

Qed.

The reasoning rules for terms follow directly from the reasoning rules established for **evalns**.

Lemma hoarens_val : $\forall v H Q,$
 $H ==> Q v \rightarrow$
 $\text{hoarens } (\text{trm_val } v) H Q.$

Proof using. *introv M. intros h K. applys× evalns_val. Qed.*

Lemma hoarens_fix : $\forall f x t1 H Q,$
 $H ==> Q (\text{val_fix } f x t1) \rightarrow$
 $\text{hoarens } (\text{trm_fix } f x t1) H Q.$

Proof using. *introv M. intros h K. applys× evalns_fix. Qed.*

Lemma hoarens_app_fix : $\forall v1 v2 f x t1 H Q,$
 $v1 = \text{val_fix } f x t1 \rightarrow$
 $\text{hoarens } (\text{subst } x v2 (\text{subst } f v1 t1)) H Q \rightarrow$
 $\text{hoarens } (\text{trm_app } v1 v2) H Q.$

Proof using. *introv E M. intros h K. applys× evalns_app_fix. Qed.*

Lemma hoarens_let : $\forall x t1 t2 H Q Q1,$
 $\text{hoarens } t1 H Q1 \rightarrow$
 $(\forall v1, \text{hoarens } (\text{subst } x v1 t2) (Q1 v1) Q) \rightarrow$
 $\text{hoarens } (\text{trm_let } x t1 t2) H Q.$

Proof using.

introv M1 M2. intros h K. applys× evalns_let.
 $\{ \text{introv } K'. \text{applys× M2.} \}$

Qed.

Lemma hoarens_if : $\forall (b:\text{bool}) t1 t2 H Q,$
 $\text{hoarens } (\text{if } b \text{ then } t1 \text{ else } t2) H Q \rightarrow$
 $\text{hoarens } (\text{trm_if } b t1 t2) H Q.$

Proof using. *introv M1. intros h K. applys× evalns_if. Qed.*

The evaluation rules for primitive operations are proved in a way that is extremely similar to the proofs used for the big-step case, i.e., to the proofs establishing the reasoning rules for hoarens.

```
Lemma hoarens_add : ∀ H n1 n2,
  hoarens (val_add n1 n2)
  H
  (fun r ⇒ \[r = val_int (n1 + n2)] \* H).
```

Proof using.

```
intros. intros s K. applys evalns_step.
{ do 2 esplit. constructors×. }
{ introv R. inverts R. applys evalns_val. rewrite¬ hstar_hpure_l. }
```

Qed.

```
Lemma hoarens_rand : ∀ n,
  n > 0 →
  hoarens (val_rand n)
  \[]
  (fun r ⇒ \exists n1, \[0 ≤ n1 < n] \* \[r = val_int n1]).
```

Proof using.

```
introv N. intros s K. lets ->: hempty_inv K. applys evalns_step.
{ do 2 esplit. applys× step_rand 0. math. }
{ introv R. inverts R; tryfalse.
  applys evalns_val. applys hexists_intro n1. rewrite¬ hstar_hpure_l.
  split¬. applys× hpure_intro. }
```

Qed.

```
Lemma hoarens_ref : ∀ H v,
  hoarens (val_ref v)
  H
  (fun r ⇒ (\exists p, \[r = val_loc p] \* p ~~> v) \* H).
```

Proof using.

```
intros. intros s K. applys evalns_step.
{ forwards¬ (p&D&N): (exists_fresh null s).
  esplit. ∃ (val_loc p). applys× step_ref. }
{ introv R. inverts R; tryfalse. applys evalns_val.
  unfold update. applys¬ hstar_intro.
  { ∃ p. rewrite¬ hstar_hpure_l. split¬. { applys¬ hsingl_intro. } }
  { applys× disjoint_single_of_not_indom. } }
```

Qed.

```
Lemma hoarens_get : ∀ H v p,
  hoarens (val_get p)
  ((p ~~> v) \* H)
  (fun r ⇒ \[r = v] \* (p ~~> v) \* H).
```

Proof using.

```

intros. intros s K. destruct K as (s1&s2&P1&P2&D&U).
lets E1: hsingle_inv P1. subst s s1. applys evalns_step.
{ do 2 esplit. applys× step_get. applys indom_union_l. applys indom_single. }
{ introv R. inverts R; tryfalse. applys evalns_val.
  rewrite¬ hstar_hpure_l. split¬.
  { rewrite read_union_l. { rewrite× read_single. } { applys indom_single. } }
  { applys× hstar_intro. } }

```

Qed.

Lemma hoarens_set : $\forall H w p v,$
 $\text{hoarens}(\text{val_set}(\text{val_loc } p) v)$
 $((p \sim\sim> w) \ast H)$
 $(\text{fun } r \Rightarrow \lambda[r = \text{val_unit}] \ast (p \sim\sim> v) \ast H).$

Proof using.

```

intros. intros s K. destruct K as (s1&s2&P1&P2&D&U).
lets E1: hsingle_inv P1. subst s s1. applys evalns_step.
{ do 2 esplit. applys× step_set. applys indom_union_l. applys indom_single. }
{ introv R. inverts R; tryfalse. applys evalns_val.
  rewrite hstar_hpure_l. split¬.
  { rewrite update_union_l; [| applys indom_single ].}
  { rewrite update_single. applys¬ hstar_intro.
    { applys¬ hsingl_intro. }
    { applys× disjoint_single_set. } } }

```

Qed.

Lemma hoarens_free : $\forall H p v,$
 $\text{hoarens}(\text{val_free}(\text{val_loc } p))$
 $((p \sim\sim> v) \ast H)$
 $(\text{fun } r \Rightarrow \lambda[r = \text{val_unit}] \ast H).$

Proof using.

```

intros. intros s K. destruct K as (s1&s2&P1&P2&D&U).
lets E1: hsingle_inv P1. subst s s1. applys evalns_step.
{ do 2 esplit. applys× step_free. applys indom_union_l. applys indom_single. }
{ introv R. inverts R; tryfalse. applys evalns_val.
  rewrite hstar_hpure_l. split¬.
  { rewrite remove_union_single_l. { auto. } }
  { intros N'. applys disjoint_inv_not_indom_both D N'.
    applys indom_single. } }

```

Qed.

From there, reasoning rules for **triplens** can be derived from the rules for **hoarens** exactly like in previous chapters, i.e., using exactly the same proofs as for deriving rules for **triple** from rules for **hoare**.

48.4.8 Equivalence Between Non-Deterministic Small-Step and Big-Step Sem.

We end this chapter with the proof of equivalence between `hoarens` and `hoaren`, establishing a formal relationship between triples defined with respect to a small-step semantics and those defined with respect to a big-step semantics.

We start by establishing the equivalence between `evalns` and `evaln`. We focus first on the direction from `evalns` to `evaln`.

We begin with a key lemma: if a configuration (s_1, t_1) takes a step to (s_2, t_2) , then this first configuration admits the same postconditions as the second configuration.

Lemma evaln_of_step_and_evaln : $\forall s_1 t_1 Q,$

$$\begin{aligned} & (\exists s_2 t_2, \text{step } s_1 t_1 s_2 t_2) \rightarrow \\ & (\forall s_2 t_2, \text{step } s_1 t_1 s_2 t_2 \rightarrow \text{evaln } s_2 t_2 Q) \rightarrow \\ & \text{evaln } s_1 t_1 Q. \end{aligned}$$

Proof using.

```

introv (s2&t2&R1) RS. gen Q. induction R1; intros.
{ applys evaln_let (fun v1 s2 => evaln s2 (subst x v1 t2) Q).
  { applys IHR1. intros s1b t1b Kb.
    forwards M: RS. { applys step_let_ctxt Kb. }
    { inverts M as M1 M2. applys evaln_conseq M1. applys M2. } }
  { intros v1 s' K. applys K. } }
{ applys evaln_fix. forwards M: RS. { applys step_fix. }
  { inverts M. } }
{ applys evaln_app_fix. forwards M: RS. { applys step_app_fix. }
  { applys M. } }
{ applys evaln_if. forwards M: RS. { applys step_if. } { applys M. } }
{ applys evaln_let (fun v' s' => v' = v1 ∧ s' = s).
  { applys evaln_val. }
  { intros ? ? (->&->). forwards M: RS. { applys step_let. }
    { applys M. } } }
{ applys evaln_add. forwards M: RS. { applys step_add. }
  { inverts M. } }
{ applys evaln_rand.
  { math. }
  { intros n2 N2. forwards M: RS. { applys step_rand n2. }
    { inverts M. } } }
{ applys evaln_ref. intros p' D.
  forwards M: RS p'. { applys step_ref. } { inverts M. } }
{ applys evaln_get. forwards M: RS. { applys step_get. }
  { inverts M. } }
{ applys evaln_set. forwards M: RS. { applys step_set. }
  { inverts M. } }
```

$\{ \text{applys} \times \text{evaln_free}. \text{forwards } M : RS. \{ \text{applys} \times \text{step_free}. \}$
 $\{ \text{inverts} \times M. \} \}$

Qed.

By exploiting the above lemma over all the evals of an execution, we obtain the fact that **evalns** implies **evaln**.

Lemma evaln_of_evalns : $\forall s t Q,$

evalns $s t Q \rightarrow$
evaln $s t Q.$

Proof using.

intros $M.$ induction $M.$
 $\{ \text{applys} \times \text{evaln_val}. \}$
 $\{ \text{applys} \times \text{evaln_of_step_and_evaln}. \}$

Qed.

Let's now turn to the second direction, from **evaln** to **evalns**. The proof is carried out by induction on the big-step relation.

Lemma evalns_of_evaln : $\forall s t Q,$

evaln $s t Q \rightarrow$
evalns $s t Q.$

Proof using.

intros $M.$ induction $M.$
 $\{ \text{applys} \times \text{evalns_val}. \}$
 $\{ \text{applys evalns_step}.$
 $\{ \text{do 2 esplit. applys step_fix.} \}$
 $\{ \text{intros } K. \text{inverts } K. \text{applys} \times \text{evalns_val}. \} \}$
 $\{ \text{rename } H \text{ into } E. \text{applys evalns_step.} \}$
 $\{ \text{do 2 esplit. applys} \times \text{step_app_fix.} \}$
 $\{ \text{intros } K. \text{invert } K; \text{try solve [intros; false].} \}$
 $\{ \text{intros } \rightarrow \rightarrow \rightarrow \rightarrow R. \text{inverts } E. \text{applys IHM.} \} \}$
 $\{ \text{rename } M \text{ into } M1, H \text{ into } M2, \text{IHM into IHM1}, H0 \text{ into IHM2.} \}$
 $\text{tests } C: (\exists v1, t1 = \text{trm_val } v1).$
 $\{ \text{destruct } C \text{ as } (v1\&\rightarrow). \text{applys evalns_step.} \}$
 $\{ \text{do 2 esplit. applys} \times \text{step_let.} \}$
 $\{ \text{intros } K. \text{inverts } K \text{ as } K1 K2.$
 $\{ \text{inverts } K1. \}$
 $\{ \text{inverts } IHM1 \text{ as } K3 K4.$
 $\{ \text{applys} \times \text{IHM2.} \}$
 $\{ \text{destruct } K3 \text{ as } (?\&?\&K5). \text{inverts } K5. \} \} \}$
 $\{ \text{applys evalns_step.} \}$
 $\{ \text{inverts } IHM1 \text{ as } K3 K4.$
 $\{ \text{false} \times C. \}$
 $\{ \text{destruct } K3 \text{ as } (?\&?\&K5). \text{do 2 esplit. applys} \times \text{step_let_ctx.} \} \}$

```

{ introv K. inverts K as K'; [|false × C].
  inverts IHM1 as K5 K6; [|false × C|].
  specializes K6 K'. applys× evalns_let. } } }

{ applys evalns_step.
  { do 2 esplit. applys× step_if. }
  { introv K. inverts K. applys IHM. } }

{ applys evalns_step.
  { do 2 esplit. applys× step_add. }
  { introv K. inverts K. applys× evalns_val. } }

{ applys evalns_step.
  { do 2 esplit. applys× step_rand 0. math. }
  { introv K. inverts K; tryfalse. applys× evalns_val. } }

{ applys evalns_step.
  { forwards¬ (p&D&N): (exists_fresh null s).
    do 2 esplit. applys× step_ref. }
  { introv K. inverts K; tryfalse. applys× evalns_val. } }

{ applys evalns_step.
  { do 2 esplit. applys× step_get. }
  { introv K. inverts K; tryfalse. applys× evalns_val. } }

{ applys evalns_step.
  { do 2 esplit. applys× step_set. }
  { introv K. inverts K; tryfalse. applys× evalns_val. } }

{ applys evalns_step.
  { do 2 esplit. applys× step_free. }
  { introv K. inverts K; tryfalse. applys× evalns_val. } }

```

Qed.

Combining the two directions, we obtain the desired equivalence between the big-step and the small-step characterization of total correctness.

Lemma evalns_eq_evalns :

evalns = evaln.

Proof using.

```

extens. intros s t Q. iff M.
{ applys× evaln_of_evalns. }
{ applys× evalns_of_evaln. }

```

Qed.

As immediate corollary, we obtain the equivalence between the big-step and the small-step characterization of Hoare triples, for the general case of a non-deterministic language.

Lemma phoare_eq_hoarens :

hoarens = hoaren.

Proof using. unfold hoarens, hoaren. rewrite× evalns_eq_evalns. Qed.

48.4.9 Historical Notes

At a very low-level, one may view a piece of hardware as being totally deterministic: at each tick of the processor’s clock, every hardware component makes a transition according to non-ambiguous rules. Yet, complex hardware architectures are not so deterministic. For example, on a multicore hardware, which of two concurrent writes reaches the memory first is, from the perspective of a program, essentially random. At a higher-level, interactions with the outside world, such as user input, can be seen as sources of non-determinism in a program semantics.

Non-deterministic semantics may be harder to reason about than deterministic semantics. To tame that complexity, a semantics may be “determinized” by parameterizing it with a trace of events, each event reflecting one “choice” that was made during the execution. Reasoning about a determinized semantics equipped with traces may, depending on the context, be easier or carry more information than reasoning about a non-deterministic semantics. Reasoning about traces is, however, beyond the scope of this course.

Non-determinism is most often described using a small-step semantics. Capturing the semantics of triples for a non-deterministic languages using a big-step judgment, as done in this chapter with the predicate **evaln**, appears to be a novel approach, as of Jan. 2021.

Chapter 49

Library `SLF.Partial`

49.1 Partial: Triples for Partial Correctness

Set Implicit Arguments.

From *SLF* Require Export Nondet.

Close Scope *val_scope*.

Close Scope *trm_scope*.

Implicit Types *f* : var.

Implicit Types *b* : **bool**.

Implicit Types *s* : state.

Implicit Types *t* : **trm**.

Implicit Types *v* : **val**.

Implicit Types *n* : int.

Implicit Types *p* : loc.

Implicit Types *h* : heap.

Implicit Types *P* : Prop.

Implicit Types *H* : hprop.

Implicit Types *Q* : **val** → hprop.

49.2 First Pass

In this chapter, we investigate definitions of Hoare triples and Separation Logic triples with respect to partial correctness, that is, without imposing termination.

Partial correctness is a weaker property than total correctness. However, it may be suited for reasoning about programs that are not meant to terminate, and may be interesting to consider as an intermediate target when reasoning about programs for which termination is much harder to establish than correctness.

This chapter is organized as follows:

- Partial correctness with respect to big-step semantics.

- Bonus contents: partial correctness with respect to small-step semantics.

This chapter, like the previous chapter `Nondet`, omits sequences and non-recursive functions, as these features can be encoded.

49.2.1 Big-Step Characterization of Partial Correctness

For total correctness, the judgment `evals s t Q` asserts that any execution of the program (s,t) terminates on an output satisfying Q . In contrast, the partial correctness judgment `evalnp s t Q` asserts that any execution of the program (s,t) either terminates on an output satisfying Q , or diverges (that is, executes for ever).

The definition of the predicate `evalnp` is obtained by taking the constructors from the inductive definition of `evals`, and considering the coinductive interpretation of these constructors. The coinductive interpretation allows for infinite derivation. It thereby introduces the possibility of diverging executions. Importantly, the predicate `evalnp` still disallows executions that get stuck.

```
CoInductive evalnp : state → trm → (val → state → Prop) → Prop :=
| evalnp_val : ∀ s v Q,
  Q v s →
  evalnp s (trm_val v) Q
| evalnp_fix : ∀ s f x t1 Q,
  Q (val_fix f x t1) s →
  evalnp s (trm_fix f x t1) Q
| evalnp_app_fix : ∀ s1 v1 v2 f x t1 Q,
  v1 = val_fix f x t1 →
  evalnp s1 (subst x v2 (subst f v1 t1)) Q →
  evalnp s1 (trm_app v1 v2) Q
| evalnp_let : ∀ Q1 s1 x t1 t2 Q,
  evalnp s1 t1 Q1 →
  (forall v1 s2, Q1 v1 s2 → evalnp s2 (subst x v1 t2) Q) →
  evalnp s1 (trm_let x t1 t2) Q
| evalnp_if : ∀ s1 b t1 t2 Q,
  evalnp s1 (if b then t1 else t2) Q →
  evalnp s1 (trm_if (val_bool b) t1 t2) Q
| evalnp_add : ∀ s n1 n2 Q,
  Q (val_int (n1 + n2)) s →
  evalnp s (val_add (val_int n1) (val_int n2)) Q
| evalnp_rand : ∀ s n Q,
  n > 0 →
  (forall n1, 0 ≤ n1 < n → Q n1 s) →
  evalnp s (val_rand (val_int n)) Q
| evalnp_ref : ∀ s v Q,
  (forall p, ¬ Fmap.indom s p →
```

$$\begin{aligned}
& Q (\text{val_loc } p) (\text{Fmap.update } s p v) \rightarrow \\
& \mathbf{evalnp} s (\text{val_ref } v) Q \\
| & \text{evalnp_get : } \forall s p Q, \\
& \text{Fmap.indom } s p \rightarrow \\
& Q (\text{Fmap.read } s p) s \rightarrow \\
& \mathbf{evalnp} s (\text{val_get } (\text{val_loc } p)) Q \\
| & \text{evalnp_set : } \forall s p v Q, \\
& \text{Fmap.indom } s p \rightarrow \\
& Q \text{ val_unit } (\text{Fmap.update } s p v) \rightarrow \\
& \mathbf{evalnp} s (\text{val_set } (\text{val_loc } p) v) Q \\
| & \text{evalnp_free : } \forall s p Q, \\
& \text{Fmap.indom } s p \rightarrow \\
& Q \text{ val_unit } (\text{Fmap.remove } s p) \rightarrow \\
& \mathbf{evalnp} s (\text{val_free } (\text{val_loc } p)) Q.
\end{aligned}$$

The judgment $\text{hoarenp } t H Q$ captures partial correctness, in terms of **evalnp**. It is defined through the usual pattern for Hoare triples.

Definition $\text{hoarenp } (t:\text{trm}) (H:\text{hprop}) (Q:\text{val} \rightarrow \text{hprop}) : \text{Prop} :=$
 $\forall s, H s \rightarrow \mathbf{evalnp} s t Q.$

One key property is the covariance of **evalnp**, which induces the rule of consequence for hoarenp . An interesting about the proof of covariance is that it is not carried out by induction, like **evalnp_conseq**, but by coinduction. Observe carefully the details of that proof.

Lemma $\text{evalnp_conseq} : \forall s t Q1 Q2,$
 $\mathbf{evalnp} s t Q1 \rightarrow$
 $Q1 \implies Q2 \rightarrow$
 $\mathbf{evalnp} s t Q2.$

Proof using.

intros $M W$. *unfolds* **qimpl**, **himpl**.

gen $s t Q1 Q2$. *cofix* IH . *intros*. *inverts* M ; *try solve* [constructors \times].

Qed.

Exercise: 3 stars, standard, optional (evalnp_inv_eval) Assume $\mathbf{evalnp} s t Q$ to hold. Prove that the postcondition Q holds of any output that (s,t) may evaluate to according to the judgment **eval**.

Lemma $\text{evalnp_inv_eval} : \forall s t v s' Q,$
 $\mathbf{evalnp} s t Q \rightarrow$
 $\mathbf{eval} s t s' v \rightarrow$
 $Q v s'.$

Proof using. *Admitted.*

□

49.2.2 From Total to Partial Correctness

Total correctness is a stronger property than partial correctness: $\text{hoaren } t \ H \ Q$ entails $\text{hoarenp } t \ H \ Q$. To formalize this result, we first establish that $\text{evaln } s \ t \ Q$, which is defined inductively, entails $\text{evalnp } s \ t \ Q$, which is defined coinductively.

The proof is a straightforward induction, reflecting on the fact that a coinductive interpretation of a set of rules is always derivable from an inductive interpretation of a same set of derivation rules.

```
Lemma evalnp_of_evaln : ∀ s t Q,
  evaln s t Q →
  evalnp s t Q.
```

Proof using. `intros M. induction M; try solve [constructors ×]. Qed.`

```
Lemma hoaren_of_cohoare : ∀ t H Q,
  hoaren t H Q →
  hoarenp t H Q.
```

Proof using.

`intros M. intros s K. specializes M K. applys evalnp_of_evaln M.`

Qed.

49.2.3 Characterization of Divergence

The judgment partial correctness judgment $\text{evalnp } s \ t \ Q$ asserts that any execution of the program (s,t) either terminates on an output satisfying Q , or diverge. By instantiating Q as the predicate that is always false, we are thus able to assert that “every possible execution of the program (s,t) diverges”.

Let us name `Empty` this “empty” postcondition, which characterizes an empty set of output configurations.

```
Definition Empty : val → state → Prop :=
  fun v s ⇒ False.
```

The judgment $\text{divn } s \ t$, defined as $\text{evalnp } s \ t \ \text{Empty}$, asserts that every possible execution of the program (s,t) diverges.

```
Definition divn (s:state) (t:trm) : Prop :=
  evalnp s t Empty.
```

49.2.4 Big-Step-Based Reasoning Rules

For the partial correctness triple (hoarenp) , we can derive all the usual reasoning rules. The proofs are the same as for the reasoning rules for deriving total correctness triples (hoaren) .

Structural rules

```
Lemma hoarenp_conseq : ∀ t H' Q' H Q,
  hoarenp t H' Q' →
```

$H ==> H' \rightarrow$
 $Q' ==> Q \rightarrow$
 $\text{hoarenp } t H Q.$

Proof using.

unfolds hoarenp. introv M MH MQ HF. applys× evalnp_conseq.

Qed.

Lemma hoarenp_hexists : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$
 $(\forall x, \text{hoarenp } t (J x) Q) \rightarrow$
 $\text{hoarenp } t (\text{hexists } J) Q.$

Proof using. introv M. intros h (x&Hh). applys M Hh. Qed.

Lemma hoarenp_hpure : $\forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{hoarenp } t H Q) \rightarrow$
 $\text{hoarenp } t (\lambda [P] \lambda^* H) Q.$

Proof using.

introv M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).

lets E: hempty_inv HP. subst. rewrite Fmap.union_empty_l. applys¬ M.

Qed.

Rules for term constructs

Lemma hoarenp_val : $\forall v H Q,$
 $H ==> Q v \rightarrow$
 $\text{hoarenp } (\text{trm_val } v) H Q.$

Proof using. introv M. intros h K. applys× evalnp_val. Qed.

Lemma hoarenp_fix : $\forall f x t1 H Q,$
 $H ==> Q (\text{val_fix } f x t1) \rightarrow$
 $\text{hoarenp } (\text{trm_fix } f x t1) H Q.$

Proof using. introv M. intros h K. applys× evalnp_fix. Qed.

Lemma hoarenp_app_fix : $\forall v1 v2 f x t1 H Q,$
 $v1 = \text{val_fix } f x t1 \rightarrow$
 $\text{hoarenp } (\text{subst } x v2 (\text{subst } f v1 t1)) H Q \rightarrow$
 $\text{hoarenp } (\text{trm_app } v1 v2) H Q.$

Proof using. introv E M. intros h K. applys× evalnp_app_fix. Qed.

Lemma hoarenp_let : $\forall x t1 t2 H Q Q1,$
 $\text{hoarenp } t1 H Q1 \rightarrow$
 $(\forall v1, \text{hoarenp } (\text{subst } x v1 t2) (Q1 v1) Q) \rightarrow$
 $\text{hoarenp } (\text{trm_let } x t1 t2) H Q.$

Proof using. introv M1 M2. intros h K. applys× evalnp_let. Qed.

Lemma hoarenp_if : $\forall (b:\text{bool}) t1 t2 H Q,$
 $\text{hoarenp } (\text{if } b \text{ then } t1 \text{ else } t2) H Q \rightarrow$
 $\text{hoarenp } (\text{trm_if } b t1 t2) H Q.$

Proof using. introv M. intros h K. applys× evalnp_if. Qed.

Rules for primitive operations

Lemma hoarenp_add : $\forall H \ n1 \ n2,$
hoarenp (val_add $n1 \ n2)$
 H
(fun $r \Rightarrow \exists [r = \text{val_int } (n1 + n2)] \ \& H).$
Proof using.
intros. intros $h \ K$. applys evalnp_add. rewrite hstar_hpure_l.
Qed.

Lemma hoarenp_rand : $\forall n,$
 $n > 0 \rightarrow$
hoarenp (val_rand $n)$
 $\exists []$
(fun $r \Rightarrow \exists n1, \exists [0 \leq n1 < n] \ \& \exists [r = \text{val_int } n1]).$

Proof using.
introv N . intros $h \ K$. applys evalnp_rand.
lets ->: hempty_inv K .
intros $n1 \ H1$. applys hexists_intro $n1$. rewrite hstar_hpure_l.
split. applys hpure_intro.
Qed.

Lemma hoarenp_ref : $\forall H \ v,$
hoarenp (val_ref $v)$
 H
(fun $r \Rightarrow (\exists p, \exists [r = \text{val_loc } p] \ \& p \rightsquigarrow v) \ \& H).$

Proof using.
intros. intros $s1 \ K$. applys evalnp_ref. intros $p \ D$.
unfolds update. applys hstar_intro K .
{ applys hexists_intro p . rewrite hstar_hpure_l.
split. applys hsingl_intro. }
{ applys disjoint_single_of_not_indom. }
Qed.

Lemma hoarenp_get : $\forall H \ v \ p,$
hoarenp (val_get $p)$
(($p \rightsquigarrow v$) $\& H)$
(fun $r \Rightarrow \exists [r = v] \ \& (p \rightsquigarrow v) \ \& H).$

Proof using.
intros. intros $s \ K$.
lets ($h1 \& h2 \& P1 \& P2 \& D \& -\rightarrow$): hstar_inv (rm K).
lets $E1$: hsingl_inv $P1$. lets $I1$: indom_single $p \ v$.
applys evalnp_get.
{ applys Fmap.indom_union_l. subst. applys indom_single. }
{ rewrite hstar_hpure_l. split. }

```
{ subst h1. rewrite× Fmap.read_union_l. rewrite× Fmap.read_single. }
{ applys× hstar_intro. } }
```

Qed.

```
Lemma hoarenp_set : ∀ H w p v,
  hoarenp (val_set (val_loc p) v)
  ((p ~~> w) \* H)
  (fun r ⇒ \[r = val_unit] \* (p ~~> v) \* H).
```

Proof using.

```
intros. intros s1 K.
lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
lets E1: hsingl_inv P1. lets I1: indom_single p v.
applys evalnp_set.
{ applys× Fmap.indom_union_l. subst. applys indom_single. }
{ rewrite hstar_hpure_l. split×.
  { subst h1. rewrite Fmap.update_union_l.
    2:{ applys indom_single. }
    rewrite update_single. applys hstar_intro.
    { applys hsingl_intro. }
    { applys P2. }
    { applys Fmap.disjoint_single_set D. } } }
```

Qed.

```
Lemma hoarenp_free : ∀ H p v,
  hoarenp (val_free (val_loc p))
  ((p ~~> v) \* H)
  (fun r ⇒ \[r = val_unit] \* H).
```

Proof using.

```
intros. intros s1 K.
lets (h1&h2&P1&P2&D&->): hstar_inv (rm K).
lets E1: hsingl_inv P1. lets I1: indom_single p v.
applys evalnp_free.
{ applys× Fmap.indom_union_l. subst. applys indom_single. }
{ rewrite hstar_hpure_l. split×.
  { subst h1. rewrite× Fmap.remove_union_single_l.
    { intros Dl. applys× Fmap.disjoint_inv_not_indom_both D Dl. } } }
```

Qed.

49.2.5 Big-Step-Based Reasoning Rules for Divergence

In a partial correctness setting, it is also interesting to derive reasoning rules for establishing that a program diverges. These rules are stated and proved next.

```
Lemma divn_app_fix : ∀ s1 v1 v2 f x t1,
  v1 = val_fix f x t1 →
```

$\text{divn } s1 (\text{subst } x v2 (\text{subst } f v1 t1)) \rightarrow$
 $\text{divn } s1 (\text{trm_app } v1 v2).$

Proof using. *introv E M. unfolds divn. applys evalnp_app_fix.* Qed.

Exercise: 2 stars, standard, especially useful (divn_let_ctx) Prove the divergence rule for a let-binding covering the case where the first subterm diverges.

Lemma divn_let_ctx : $\forall s1 x t1 t2,$
 $\text{divn } s1 t1 \rightarrow$
 $\text{divn } s1 (\text{trm_let } x t1 t2).$

Proof using. *Admitted.*

□

Lemma divn_let : $\forall s1 x t1 t2 Q1,$
 $\text{evalnp } s1 t1 Q1 \rightarrow$
 $(\forall s2 v1, \text{divn } s2 (\text{subst } x v1 t2)) \rightarrow$
 $\text{divn } s1 (\text{trm_let } x t1 t2).$

Proof using. *introv M1 M2. unfolds divn. applys evalnp_let M1.* Qed.

Lemma divn_if : $\forall s1 b t1 t2,$
 $\text{divn } s1 (\text{if } b \text{ then } t1 \text{ else } t2) \rightarrow$
 $\text{divn } s1 (\text{trm_if } (\text{val_bool } b) t1 t2).$

Proof using. *introv M. unfolds divn. applys evalnp_if.* Qed.

49.3 Optional Material

49.3.1 Interpretation of evalnp w.r.t. eval and safe

The predicate **safe** $s t$ asserts that an execution of the configuration t/s can never get stuck. Its definition is a simplified version of **evalnp** where all occurrences of Q are removed. In the rule for let- bindings, namely **safe_let**, the quantification over the configuration $v1/s2$ is done by referring to the big-step judgment **eval**.

```
CoInductive safe : state → trm → Prop :=
| safe_val : ∀ s v,
  safe s (trm_val v)
| safe_fix : ∀ s f x t1,
  safe s (trm_fix f x t1)
| safe_app_fix : ∀ s v1 v2 f x t1,
  v1 = val_fix f x t1 →
  safe s (subst x v2 (subst f v1 t1)) →
  safe s (trm_app v1 v2)
| safe_let : ∀ s x t1 t2,
  safe s t1 →
```

```


$$\begin{aligned}
& (\forall v1 s2, \mathbf{eval} s t1 s2 v1 \rightarrow \mathbf{safe} s2 (\mathbf{subst} x v1 t2)) \rightarrow \\
& \quad \mathbf{safe} s (\mathbf{trm\_let} x t1 t2) \\
| \quad & \mathbf{safe\_if} : \forall s b t1 t2, \\
& \quad \mathbf{safe} s (\mathbf{if} b \mathbf{then} t1 \mathbf{else} t2) \rightarrow \\
& \quad \mathbf{safe} s (\mathbf{trm\_if} (\mathbf{val\_bool} b) t1 t2) \\
| \quad & \mathbf{safes\_add} : \forall s n1 n2, \\
& \quad \mathbf{safe} s (\mathbf{val\_add} (\mathbf{val\_int} n1) (\mathbf{val\_int} n2))) \\
| \quad & \mathbf{safe\_rand} : \forall s n, \\
& \quad n > 0 \rightarrow \\
& \quad \mathbf{safe} s (\mathbf{val\_rand} (\mathbf{val\_int} n)) \\
| \quad & \mathbf{safe\_ref} : \forall s v, \\
& \quad \mathbf{safe} s (\mathbf{val\_ref} v) \\
| \quad & \mathbf{safe\_get} : \forall s p, \\
& \quad \mathbf{Fmap.indom} s p \rightarrow \\
& \quad \mathbf{safe} s (\mathbf{val\_get} (\mathbf{val\_loc} p)) \\
| \quad & \mathbf{safe\_set} : \forall s p v, \\
& \quad \mathbf{Fmap.indom} s p \rightarrow \\
& \quad \mathbf{safe} s (\mathbf{val\_set} (\mathbf{val\_loc} p) v) \\
| \quad & \mathbf{safe\_free} : \forall s p, \\
& \quad \mathbf{Fmap.indom} s p \rightarrow \\
& \quad \mathbf{safe} s (\mathbf{val\_free} (\mathbf{val\_loc} p)).
\end{aligned}$$


```

Section EvalnpSafe.

Hint Constructors **eval evalnp**.

Exercise: 5 stars, standard, especially useful (evalnp_iff_safe_and_post) Prove that **evalnp** $s t Q$ is equivalent to the conjunction of **safe** $s t$ and to a partial correctness result asserting that if an evaluation of t/s terminates on a particular result, then this result satisfies Q .

Lemma evalnp_iff_safe_and_post : $\forall s t Q,$

$$\mathbf{evalnp} s t Q \leftrightarrow (\mathbf{safe} s t \wedge (\forall v s', \mathbf{eval} s t s' v \rightarrow Q v s')).$$

Proof using. Admitted.

□

Let **Any** denotes the postcondition that accepts any result.

Definition Any : **val** → state → Prop :=

$$\text{fun } v s \Rightarrow \mathbf{True}.$$

Hint Unfold Any.

Exercise: 2 stars, standard, especially useful (safe_iff_evalnp_any) Prove that **safe** $s t$ is equivalent to **evalnp** $s t$ **Any**. Hint: a direct proof fails—why? The solution is to exploit evalnp_iff_safe_and_post.

Lemma safe_iff_evalnp_any : $\forall s t,$

safe $s t \leftrightarrow \text{evalnp } s t \text{ Any.}$

Proof using. Admitted.

□

End EvalnpSafe.

49.3.2 Small-Step Characterization of Partial Correctness

So far, this chapter has focused on handling partial correctness using a big-step semantics. In this bonus section, we investigate the treatment of partial correctness using a small-step semantics.

Before we get started, let us state and prove a basic lemma asserting that values cannot take a step. It will be useful throughout the section.

Lemma exists_step_val_inv : $\forall s v,$
 $(\exists s' t', \text{step } s v s' t') \rightarrow$
False.

Proof using. introv $(s' \& t' \& S)$. inverts S . Qed.

In the chapter Nondet, we presented a definition named *evals_attempt_3* that defined a small-step characterization of partial correctness. Let us rename this definition to evalnps.

The judgment evalnps $s t Q$ asserts that, for any configuration (s',t') that can be reached from (s,t) , either t' is a value that, together with s' , satisfy the postcondition Q ; or the configuration (s',t') can take a step, that is, it is not a stuck configuration.

Definition evalnps ($s:\text{state}$) ($t:\text{trm}$) ($Q:\text{val} \rightarrow \text{state} \rightarrow \text{Prop}$) : Prop :=
 $\forall s' t', \text{evals } s t s' t' \rightarrow$
 $(\exists v, t' = \text{trm_val } v \wedge Q v s')$
 $\vee (\exists s'' t'', \text{step } s' t' s'' t'').$

On top of evalnps, we define a partial correctness Hoare triple judgment hoarenps in the usual way.

Definition hoarenps ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : Prop :=
 $\forall s, H s \rightarrow \text{evalnps } s t Q.$

The property evalnps $s t Q$ may be viewed as a form of typing: the configuration (s,t) admits the “type” Q .

- Base case: if t is a value, we assert that a configuration (s,v) has type Q iff $Q v s$ holds.
- Preservation property: if the configuration (s,t) has type Q and reduces to (s',t') , then the configuration (s',t') also admits the same type Q .
- Progress property: if the configuration (s,t) has type Q , then either t is of the form $\text{trm_val } v$ and (s,v) admits type Q , or the configuration (s,t) can take a reduction step to some configuration (s',t') .

The progress property is, in essence, a partial correctness property. Indeed, it ensures that programs never get stuck, yet without imposing termination.

In what follows, we formally state and prove the three properties associated with the judgment `evalnps s t Q`: the characterization of the base case, the preservation property, and the progress property.

First, let's prove an introduction lemma and an inversion lemma for the base case, that is, the characteriation of values.

Lemma evalnps_val : $\forall s v Q,$

$$\begin{aligned} & Q v s \rightarrow \\ & \text{evalnps } s v Q. \end{aligned}$$

Proof using.

intros M R. inverts R as.

$$\begin{aligned} & \{ \text{left} \times. \} \\ & \{ \text{intros } s1 t1 R1 R2. \text{false. inverts } R1. \} \end{aligned}$$

Qed.

Lemma evalnps_val_inv : $\forall s v Q,$

$$\begin{aligned} & \text{evalnps } s v Q \rightarrow \\ & Q v s. \end{aligned}$$

Proof using.

intros M. forwards N: M s v. { applys evals_refl. }

destruct N as [(v1&E1&P1)|(s2&t2&R)].

$$\begin{aligned} & \{ \text{inverts} \times E1. \} \\ & \{ \text{inverts } R. \} \end{aligned}$$

Qed.

Second, let's prove the preservation property.

Exercise: 3 stars, standard, especially useful (evalnps_inv_step) Prove that the property `evalnps` for a given postcondition Q is preserved when a program takes a reduction step.

Lemma evalnps_inv_step : $\forall s1 t1 s2 t2 Q,$

$$\begin{aligned} & \text{evalnps } s1 t1 Q \rightarrow \\ & \text{step } s1 t1 s2 t2 \rightarrow \\ & \text{evalnps } s2 t2 Q. \end{aligned}$$

Proof using. *Admitted.*

□

Third, let's prove the progress property.

Exercise: 4 stars, standard, especially useful (evalnps_not_val_inv) Prove that if the property `evalnps` holds for a given postcondition Q , and for a configuration that has not already reached a value, and this configuration can take a step. Moreover, the configuration reached also satisfies the property `evalnps` for that same Q .

```

Lemma evalnps_not_val_inv : ∀ s1 t1 Q,
  evalnps s1 t1 Q →
  (forall v, t1 ≠ trm_val v) →
  ∃ s2 t2, step s1 t1 s2 t2 ∧ evalnps s2 t2 Q.

```

Proof using. Admitted.

□

49.3.3 Reasoning Rules w.r.t. Small-Step Characterization

We present two proofs of reasoning rules established with respect to the judgment `hoarenps`. The first proof targets the rule of consequence. This result leverages the covariance property for `evalnps`.

```

Lemma evalnps_conseq : ∀ s t Q Q',
  evalnps s t Q' →
  Q' ==> Q →
  evalnps s t Q.

```

Proof using.

```

intros M WQ. introv R. lets [(v&E&P)|N]; M R.
{ left. ∃ v. split¬. applys× WQ. }
{ right×. }

```

Qed.

```

Lemma hoarenps_conseq : ∀ t H' Q' H Q,
  hoarenps t H' Q' →
  H ==> H' →
  Q' ==> Q →
  hoarenps t H Q.

```

Proof using.

```

intros M WH WQ. intros h K. applys evalnps_conseq WQ.
applys M. applys WH K.

```

Qed.

The second proof targets the reasoning rule for let-bindings, stated for the judgment `hoarenps`. This result leverages as key intermediate lemma an introduction rule for let-bindings with respect to `evalnps`.

Exercise: 5 stars, standard, especially useful (evalnps_let) Prove the reasoning rule for let-bindings for `evalnps`.

```

Lemma evalnps_let : ∀ s x t1 t2 Q1 Q,
  evalnps s t1 Q1 →
  (forall s1 v1, Q1 v1 s1 → evalnps s1 (subst x v1 t2) Q) →
  evalnps s (trm_let x t1 t2) Q.

```

Proof using. Admitted.

□

Lemma hoarenp_s_let : $\forall x t1 t2 H Q Q1,$
hoarenp_s $t1 H Q1 \rightarrow$
 $(\forall v1, \text{hoarenp}_s (\text{subst } x v1 t2) (Q1 v1) Q) \rightarrow$
hoarenp_s (trm_let $x t1 t2) H Q.$

Proof using.

introv $M1 M2.$ intros $h K.$ applys \times evalnps_let.
{ introv $K'.$ applys \times $M2.$ }

Qed.

49.3.4 Small-Step Characterization of Divergence

Recall the definition of evalnps.

Definition evalnps ($s:\text{state}$) ($t:\text{trm}$) ($Q:\text{val-} \rightarrow \text{state-} \rightarrow \text{Prop}$) : Prop := forall $s' t', \text{evals } s t' \rightarrow$ (exists $v, t' = \text{trm_val } v \wedge Q v s') \vee$ (exists $s'' t'', \text{step } s' t' s'' t''$).

The judgment evalnps $s t Q$ can be specialized with the empty post- condition $Q := \text{Empty}$ to express divergence, just like we did earlier on for setting up the definition of divn.

Recall that $\text{Empty} \vee s$ is equivalent to **False**. Thus, let's take the definition of evalnps $s t Q$ and replace $Q \vee s'$ with **False**. The first clause of the disjunction, $\exists v, t' = \text{trm_val } v \wedge Q v s'$ is never satisfiable and thus may be removed. What remains is a simpler, direct characterization of the property that “all executions diverge”.

The corresponding predicate, named divns $s t$ below, asserts that any execution prefix can be extended by at least one step.

Definition divns ($s:\text{state}$) ($t:\text{trm}$) : Prop :=
 $\forall s' t', \text{evals } s t s' t' \rightarrow$
 $\exists s'' t'', \text{step } s' t' s'' t''.$

Exercise: 3 stars, standard, optional (divns_iff_evalnps_Empty) Prove the equivalence between divns and the specialization of evalnps to Empty.

Lemma divns_iff_evalnps_Empty : $\forall s t,$
divns $s t \leftrightarrow \text{evalnps } s t \text{ Empty}.$

Proof using. Admitted.

□

Exercise: 2 stars, standard, optional (divns_inv_step) Prove that if a diverging configuration takes a step, then it reaches a configuration that also diverges.

Lemma divns_inv_step : $\forall s1 t1 s2 t2,$
divns $s1 t1 \rightarrow$
 $\text{step } s1 t1 s2 t2 \rightarrow$
divns $s2 t2.$

Proof using. Admitted.

□

Exercise: 2 stars, standard, optional (divns_val_inv) Prove that the execution of a value does not diverge.

Lemma `divns_val_inv` : $\forall s v,$
 $\neg \text{divns } s v.$

Proof using. *Admitted.*

□

Exercise: 5 stars, standard, optional (divns_let_ctx) Prove the divergence rule for a let-binding covering the case where the first subterm diverges, with respect to the definition `divns`.

Lemma `divns_let_ctx` : $\forall s1 x t1 t2,$
 $\text{divns } s1 t1 \rightarrow$
 $\text{divns } s1 (\text{trm_let } x t1 t2).$

Proof using. *Admitted.*

□

49.3.5 Coinductive, Small-Step Characterization of Partial Correctness

In the previous section, we characterized partial correctness using the judgment `evalnps`, which is defined in terms of the transitive closure of the small-step relation `evals`. In this section, we present an alternative definition, based on a coinductive definition.

Concretely, we start from the inductively defined judgment `evals`, which characterizes total correctness, and we consider the coinductive interpretation of the same rules. The resulting judgment is written `evalnpz` $s t Q$. This coinductive definition, because it allows for infinite derivations, covers the case of diverging computations that take involve infinitely many reduction steps.

```
CoInductive evalnpz : state → trm → (val → hprop) → Prop :=  
| evalnpz_val : ∀ s v Q,  
  Q v s →  
  evalnpz s v Q  
| evalnpz_step : ∀ s t Q,  
  (exists s' t', step s t s' t') →  
  (forall s' t', step s t s' t' → evalnpz s' t' Q) →  
  evalnpz s t Q.
```

We name `hoarenpz` the associated Hoare triple definition.

```
Definition hoarenpz (t:trm) (H:hprop) (Q:val → hprop) : Prop :=  
  ∀ s, H s → evalnpz s t Q.
```

Here again, we focus on the proofs of two particular reasoning rules, established with respect to **evalnpz**.

Firstly, let's prove the rule of consequence. The key intermediate result is to establish the covariance of **evalnpz**. Observe that this proof is to be conducted by coinduction.

```
Lemma evalnpz_conseq : ∀ s t Q Q',
  evalnpz s t Q' →
  Q' ==> Q →
  evalnpz s t Q.
```

Proof using.

```
introv M WQ. gen s t Q'. cofix IH; intros. destruct M.
{ applys evalnpz_val. applys× WQ. }
{ applys evalnpz_step.
  { eauto. }
  { introv R'. lets M': H0 R'. applys IH M' WQ. } }
```

Qed.

```
Lemma hoarenpz_conseq : ∀ t H' Q' H Q,
  hoarenpz t H' Q' →
  H ==> H' →
  Q' ==> Q →
  hoarenpz t H Q.
```

Proof using.

```
introv M WH WQ. intros h K. applys evalnpz_conseq WQ.
applys M. applys WH K.
```

Qed.

Secondly, we let's prove the reasoning rule for let-bindings. Here again, the key intermediate result is to establish a rule for **evalnpz**.

Exercise: 5 stars, standard, especially useful (hoarenpz_let) Prove the reasoning rule for let-bindings stated for **evalnpz**. Hint: to set up the coinductive proof, follow the template that appears at the beginning of the proof of evalnpz_conseq.

```
Lemma evalnpz_let : ∀ s x t1 t2 Q1 Q,
  evalnpz s t1 Q1 →
  (forall s1 v1, Q1 v1 s1 → evalnpz s1 (subst x v1 t2) Q) →
  evalnpz s (trm_let x t1 t2) Q.
```

Proof using. Admitted.

□

```
Lemma hoarenpz_let : ∀ x t1 t2 H Q Q1,
  hoarenpz t1 H Q1 →
  (forall v1, hoarenpz (subst x v1 t2) (Q1 v1) Q) →
  hoarenpz (trm_let x t1 t2) H Q.
```

Proof using.

```

intros M1 M2. intros h K. applys× evalnpz_let.
{ introv K'. applys× M2. }
Qed.

```

To express diverge, the judgment **evalnp** $s t Q$ may be specialized with the postcondition $Q := \text{Empty}$. Starting from the definition of **evalnpz**, replacing $Q \vee s$ with **False** is equivalent to removing the constructor **evalnpz_val** altogether. Thus, divergence of a configuration (s,t) may be characterized directly by the coinductive judgment **divnz** $s t$ defined below, featuring a single constructor.

```

CoInductive divnz : state → trm → Prop :=
| divnz_step : ∀ s1 t1,
  (exists s2 t2, step s1 t1 s2 t2) →
  (forall s2 t2, step s1 t1 s2 t2 → divnz s2 t2) →
  divnz s1 t1.

```

Exercise: 4 stars, standard, optional (divnz_iff_evalnpz_Empty) Prove the equivalence between **divnz** and the specialization of the judgment **evalnp** to **Empty**.

Lemma divnz_iff_evalnpz_Empty : ∀ s t,
divnz s t \leftrightarrow **evalnpz** s t **Empty**.

Proof using. Admitted.

□

49.3.6 Equivalence Between the Two Small-Step Charact. of Partial Correctness

This section establishes that **hoarenpz** is equivalent to **hoarenp**s. The key result is to prove **evalnpz** equivalent to **evalnp**s.

Lemma evalnpz_eq_evalnp :
evalnpz = **evalnp**s.

Proof using.

```

extens. intros s t Q. iff M.
{ introv R. gen M. induction R; intros.
  { inverts M as M1 M2.
    { left×. }
    { right. applys M1. } }
  { rename H into R1, R into R2. inverts M as M1 M2.
    { false. inverts R1. }
    { applys IHR. applys M2 R1. } } }
{ gen s t Q. cofix IH. intros.
  tests C: (exists v, t = trm_val v).
  { destruct C as (v&t>). applys evalnpz_val. applys evalnp_val_inv M. }
  { lets (s2&t2&R2&RS): evalnp_not_val_inv M C. applys evalnpz_step. }
}

```

```

{  $\exists s2 t2. \{$ 
{ intros s' t' R'. applys IH. applys evalnps_inv_step M R'. } } }

```

Qed.

Lemma hoarenpz_eq_hoarenp :

hoarenpz = hoarenp.

Proof using. unfold hoarenp, hoarenpz. rewrite \times evalnps_eq_evalnps. Qed.

As a corollary, we can establish the equivalence between the two small-step characterization of divergence. Indeed, both characterizations are obtained by specializing the postcondition of their underlying evaluation judgment to the empty postcondition, namely `Empty`.

Lemma divnz_eq_divns :

divnz = divns.

Proof using.

```

extens. intros s t.
rewrite divns_iff_evalnps_Empty.
rewrite divnz_iff_evalnps_Empty.
rewrite $\times$  evalnps_eq_evalnps.

```

Qed.

49.3.7 Equivalence Between Small-Step and Big-Step Partial Correctness

Semantics

We end this chapter with the proof of equivalence between `hoarenp` and `hoarenpz`, establishing a formal relationship between partial correctness triples defined with respect to a small-step semantics and those defined with respect to a big-step semantics.

The first step of the proof is to formally relate `evalnps` and `evalnp`, which are both defined coinductively. Then, we can derive the equality between `hoarenpz` and `hoarenp`, and conclude using the equality `hoarenpz_eq_hoarenp`, which was established earlier on.

The first direction is from `evalnps` to `evalnp`. To establish it, we need 3 inversion lemmas for let-bindings.

```

Lemma evalnps_let_inv_ctx :  $\forall x s1 t1 t2 Q Q1,$ 
evalnps s1 (trm_let x t1 t2) Q  $\rightarrow$ 
(fun v s  $\Rightarrow$  evals s1 t1 s v) ===> Q1  $\rightarrow$ 
evalnps s1 t1 Q1.

```

Proof using.

```

cofix IH. introv M WQ. inverts M as M0 M'.
tests C: ( $\exists v1, t1 = \text{trm\_val } v1$ ).
{ destruct C as (v1&->). applys evalnps_val.
  applys WQ. applys evals_refl. }
{ applys evalnps_step.
  { destruct M0 as (s'&t'&S). inverts S as. 2:{ false $\times$  C. } }

```

```

intros S.  $\exists$ . applys S. }
{ intros s2 t1' S. applys IH.
  { applys M'. applys step_let_ctxt S. }
  { intros v s R. applys WQ. applys evals_step S R. } } }

```

Qed.

Lemma evalnpz_let_inv_cont : $\forall s1 s2 v1 x t1 t2 Q,$
evalnpz $s1 (\text{trm_let } x t1 t2) Q \rightarrow$
evals $s1 t1 s2 v1 \rightarrow$
evalnpz $s2 (\text{subst } x v1 t2) Q.$

Proof using.

```

introv M R. gen_eq t1': (trm_val v1). induction R; intros; subst.
{ inverts M as _ M'. applys M'. applys step_let. }
{ rename H into S, R into R', t0 into t1'.
  inverts M as _ M'. applys× IHR. applys M'.
  applys step_let_ctxt S. }

```

Qed.

Lemma evalnpz_let_inv : $\forall s1 x t1 t2 Q,$
evalnpz $s1 (\text{trm_let } x t1 t2) Q \rightarrow$
 $\exists Q1, \text{evalnpz } s1 t1 Q1$
 $\wedge (\forall v1 s2, Q1 v1 s2 \rightarrow \text{evalnpz } s2 (\text{subst } x v1 t2) Q).$

Proof using.

```

introv M.  $\exists (\text{fun } v s \Rightarrow \text{evals } s1 t1 s (\text{trm\_val } v)).$  split.
{ applys× evalnpz_let_inv_ctxt M. }
{ introv K. applys× evalnpz_let_inv_cont M K. }

```

Qed.

Using these inversion lemmas, we can prove by induction the first direction, from **evalnpz** to **evalnp**.

Lemma evalnp_of_evalnpz : $\forall s t Q,$
evalnpz $s t Q \rightarrow$
evalnp $s t Q.$

Proof using.

```

cofix IH. introv M. destruct t; try solve [ inverts M; false_invert ].
{ inverts M as.
  { intros R. applys evalnp_val R. }
  { intros (s'&t'&S) _. inverts S. } }
{ inverts M as.
  { intros (s'&t'&S) .. inverts S. } }
{ inverts M as (s'&t'&S) M'. inverts S. }
{ rename v into f, v0 into x, t into t1.
  inverts M as (s'&t'&S) M1. inverts S.
  applys evalnp_fix. }

```

```

forwards  $M1'$ :  $M1$ . { applys  $\text{step\_fix}$ . }
inverts  $M1'$  as. { auto. } { intros.  $\text{false} \times \text{exists\_step\_val\_inv}$ . } }

{ inverts M as ( $s' \& t' \& S$ )  $M1$ . inverts S as.
  { applys evalnp_app_fix. { reflexivity. } applys IH.
    applys M1. applys step_app_fix. }
  { applys evalnp_add.
    forwards  $M1'$ :  $M1$ . { applys step_add. }
    inverts  $M1'$  as. { auto. } { intros.  $\text{false} \times \text{exists\_step\_val\_inv}$ . } }
  { introv N applys evalnp_rand. { math. }
    intros n1' N1.
    forwards  $M1'$ :  $M1$ . { applys step_rand n1'. }
    inverts  $M1'$  as. { auto. } { intros.  $\text{false} \times \text{exists\_step\_val\_inv}$ . } }
  { introv D applys evalnp_ref intros p' D'.
    forwards  $M1'$ :  $M1$ . { applys step_ref p'. }
    inverts  $M1'$  as. { auto. } { intros.  $\text{false} \times \text{exists\_step\_val\_inv}$ . } }
  { introv D applys evalnp_get.
    forwards  $M1'$ :  $M1$ . { applys step_get. }
    inverts  $M1'$  as. { auto. } { intros.  $\text{false} \times \text{exists\_step\_val\_inv}$ . } }
  { introv D applys evalnp_set.
    forwards  $M1'$ :  $M1$ . { applys step_set. }
    inverts  $M1'$  as. { auto. } { intros.  $\text{false} \times \text{exists\_step\_val\_inv}$ . } }
  { introv D applys evalnp_free.
    forwards  $M1'$ :  $M1$ . { applys step_free. }
    inverts  $M1'$  as. { auto. } { intros.  $\text{false} \times \text{exists\_step\_val\_inv}$ . } }
}

{ inverts M as ( $s' \& t' \& S$ )  $M'$ . inverts S. }
{ lets ( $Q1 \& M1 \& M2$ ):  $\text{evalnpz\_let\_inv } M$ . applys evalnp_let.
  { applys IH applys M1. }
  { introv R applys IH applys M2 R. } }
{ inverts M as ( $s' \& t' \& S$ )  $M1$ . inverts S applys evalnp_if.
  { applys IH applys M1 applys step_if. } }

```

Qed.

For the reciprocal direction, we also need one key inversion lemma. It is stated below. Its hypothesis is **evalnp** $s t Q$, and its conclusion corresponds to the disjunction of the constructors of the inductive definition of **evalnpz** $s t Q$.

Lemma evalnp_inv_step : $\forall s t Q$,

$$\begin{aligned} \mathbf{evalnp} \ s \ t \ Q &\rightarrow \\ &(\exists v, t = \text{trm_val } v \wedge Q v s) \\ &\vee (\exists s' t', \mathbf{step} \ s \ t \ s' t' \wedge \mathbf{evalnp} \ s' t' Q) \\ &\quad \wedge (\forall s' t', \mathbf{step} \ s \ t \ s' t' \rightarrow \mathbf{evalnp} \ s' t' Q)). \end{aligned}$$

Proof using.

```

introv M gen s Q induction t; intros; inverts M as.
{ introv R left eauto. }

```

```

{ introv R. right. split.
  { ∃. split. { applys step_fix. } { applys× evalnp_val. } }
  { intros s' t' S. inverts S. applys× evalnp_val. } }

{ introv M1. right. split.
  { ∃. split. { applys× step_app_fix. } { auto. } }
  { intros s' t' S. inverts S as E. inverts E. auto. } }

{ introv R. right. split.
  { ∃. split. { applys step_add. } { applys× evalnp_val. } }
  { intros s' t' S. inverts S. applys× evalnp_val. } }

{ introv N R. right. split.
  { ∃. split. { applys× step_rand 0. math. }
    { applys× evalnp_val. applys R. math. } }
  { intros s' t' S. inverts S; tryfalse. applys× evalnp_val. } }

{ introv R. right. split.
  { forwards¬ (p&D&N): (exists_fresh null s).
    ∃. split. { applys× step_ref. } { applys× evalnp_val. } }
  { intros s' t' S. inverts S; tryfalse. applys× evalnp_val. } }

{ introv D R. right. split.
  { ∃. split. { applys× step_get. } { applys× evalnp_val. } }
  { intros s' t' S. inverts S; tryfalse. applys× evalnp_val. } }

{ introv D R. right. split.
  { ∃. split. { applys× step_set. } { applys× evalnp_val. } }
  { intros s' t' S. inverts S; tryfalse. applys× evalnp_val. } }

{ introv D R. right. split.
  { ∃. split. { applys× step_free. } { applys× evalnp_val. } }
  { intros s' t' S. inverts S; tryfalse. applys× evalnp_val. } }

{ introv M1 M2. rename v into x. right. split.
  { forwards [(v1&->&HQ1)|( (s'&t'&S'&M')&_)]: IHt1 M1.
    { ∃. split. { applys step_let. } { applys× M2. } }
    { ∃. split.
      { applys step_let_ctxt S'. }
      { applys evalnp_let M'. applys M2. } } }

  { intros s' t' S. inverts S as S.
    { forwards [(v1&->&HQ1)|(_&M3)]: IHt1 M1.
      { inverts S. }
      { specializes M3 S. applys evalnp_let M3 M2. } }
    { inverts M1 as R. applys× M2. } } }

  { introv M1. right. split.
    { ∃. split. { applys step_if. } { auto. } }
    { intros s' t' S. inverts S. auto. } } }

```

Qed.

Using this inversion lemma, it is straightforward to derive the implication from **evalnp**

to **evalnpz**.

Lemma evalnpz_of_coeval : $\forall s t Q,$

evalnp $s t Q \rightarrow$

evalnpz $s t Q.$

Proof.

cofix *IH*. introv M . lets C : evalnp_inv_step M .

destruct C as $[(v\&-\>&HQ)|((s'\&t'\&S\&M1)\&M2)].$

{ applys evalnpz_val $HQ.$ }

{ applys evalnpz_step.

{ $\exists. applys S.$ }

{ intros $s1 t1 S'. applys IH. applys M2 S'.$ } }

Qed.

Combining the two directions yields the equality between **evalnpz** and **evalnp**, and that between **evalnps** and **evalnp**.

Lemma evalnpz_eq_evalnp :

evalnpz = **evalnp**.

Proof using.

extens. intros $s t Q. iff M.$

{ applys× evalnp_of_evalnpz. }

{ applys× evalnpz_of_coeval. }

Qed.

Lemma hoarencso_eq_hoarenp :

hoarencso = **hoarenp**.

Proof using. unfold hoarenp, hoarencso. rewrite× evalnpz_eq_evalnp. **Qed.**

Lemma hoarenps_eq_hoarenp :

hoarenps = **hoarenp**.

Proof using.

rewrite ← hoarencso_eq_hoarenps. rewrite× hoarencso_eq_hoarenp.

Qed.

As another corollary, we can establish the equivalence between the big-step and the small-step characterization of divergence. Indeed, both are obtained by specializing the postcondition to Empty.

Lemma divns_eq_divn :

divns = **divn**.

Proof using.

extens. intros $s t.$ unfold divn.

rewrite divns_if evalnps_Empty.

rewrite ← evalnpz_eq_evalnp.

rewrite× evalnpz_eq_evalnps.

Qed.

49.3.8 Historical Notes

Many program verification tools target partial correctness only, or provide separate tooling for justifying termination.

The soundness of program logics delivering partial correctness is nearly always justified with respect to a small-step semantics of the programming language. Partial correctness is generally chosen because it appears better suited for reasoning about programs that involve some amount of concurrency, and for programs that are not meant to terminate (e.g., system kernels).

Capturing partial correctness using a coinductive big-step judgment, as done in this chapter with the predicate `evalnp`, appears to be novel as of Jan. 2021.

Chapter 50

Library `SLF.Postscript`

50.1 Postscript: Conclusion and Perspectives

50.2 Beyond the Scope of This Course

This course has focused on the foundations of Separation Logic for sequential programs, and discussed the set up of tooling for computing weakest preconditions.

There are numerous aspects of Separation Logic that were not covered in this course, including:

- Concurrent Separation Logic, for reasoning about concurrent behaviors, or even, behaviors of weak memory models.
- Automated reasoning using Separation Logic, typically for a restricted fragment of Separation Logic.
- Separation Logic for low-level programs, at the byte level, and with assembly-style code written in continuation-passing style.
- Separation Logic for object-oriented programs.
- Reasoning about I/O, integer overflows, floating-point arithmetics, etc.

50.3 Historical Notes

The most relevant historical notes were included near the end of each chapter. Additional information may be found in the ICFP'20 paper *Separation Logic for Sequential Programs*, by Arthur Charguéraud. http://www.chargueraud.org/research/2020/seq_seplogic/seq_seplogic.pdf

In addition, the reader may be interested in checking out Peter O'Hearn's account of the early days of Separation Logic: http://www0.cs.ucl.ac.uk/staff/p.ohearn/SeparationLogic/Separation_Logic.pdf

50.4 Tools Leveraging Separation Logic

Ideas of Separation Logic have had significant influence on the field of programming languages from various perspectives. For a broad survey of Separation Logic, we refer to Peter O’Hearn’s CACM paper 2019. <https://dl.acm.org/doi/10.1145/3211968> (make sure to also download the “supplementary material” link at the bottom of the page).

Here is a non-exhaustive list of active projects leveraging Separation Logic.

- The tool “Infer” leverages for static analysis, that is, fully automated program analysis to produce a list of potential bugs. <https://fbinfer.com/> <https://fbinfer.com/docs/separation-logic-and-bi-abduction/>
- The programming language “Rust” features a type system with “borrows”, a notion directly inspired by Separation Logic.
- The tool “VeriFast” targets Java programs. <https://people.cs.kuleuven.be/~bart.jacobs/verifast> and <https://github.com/verifast/verifast>
- The tool “Verifiable C” tool, from the VST project, targets the verification of C code, using interactive Coq proofs based on Separation Logic. <https://vst.cs.princeton.edu/> See also volume 5 of Software Foundations.
- The tool “Viper” is based on a variant of Separation Logic, its front-end supports several languages, and its backend leverages the SMT solver Z3; it also supports user annotations in the code for guiding proofs. <https://www.inf.ethz.ch/research/viper.html>
- The “Isabelle Refinement Framework” leverages Separation Logic is the process of refining monadic high-order logic definitions into efficient imperative code. It can be used to produce verified ML code, as well as LLVM IR code. <https://www21.in.tum.de/~lammich/pub/itp15.pdf>
- The tool “CFML”, which targets the verification of OCaml programs. The original version of CFML leverages a version of characteristic formulae slightly more complicated than the one presented in this volume. The new version, CFML 2.0, directly leverages all the ideas from this course. It is still under active development.
- The verified compiler “CakeML” implements technology similar to that of CFML, and leverages it to verify components from its standard library and from its runtime.
- The Iris project develops advanced logics for reasoning about concurrent programs using Separation Logic.

50.5 Related Courses

The following courses focus on reasoning about programs, and should be of interest to the reader:

- Verifiable C, volume 5 of Sofware Foundations, by Andrew Appel. <https://softwarefoundations.cis.upenn.edu/current/index.html>
- Formal Reasoning About Programs, by Adam Chlipala. <http://adam.chlipala.net/frap/>
- The Iris tutorial, by Birkedal and Bizjak, presents the core ideas of Iris' concurrent Separation Logic. <https://iris-project.org/tutorial-material.html>

Teaser: Arthur Charguéraud has for project to write a follow-up course on practical program verification using CFML, focusing on the specification and verification of classic data structures and algorithms.

50.6 Acknowledgments

The lead author of this volume, Arthur Charguéraud, wishes to thank:

- Benjamin Pierce, for demonstrating the benefits of Coq-based teaching, for coordinating the Software Foundations series, and for encouraging me to write this volume.
- The Software Foundations (SF) community, for helping to polish the course material and the tooling, and contributing to the success of teaching using this material.
- François Pottier, with whom I developed several extensions of Separation Logic, and thereby obtained a deeper understanding of this logic.
- Jacques-Henri Jourdan, who introduced me to a number of techniques used in the Iris project, and contributed to the definition of `mkstruct`.
- Xavier Leroy, with whom I had numerous discussions on mechanized semantics.
- Jean-Christophe Filliâtre, for numerous discussions on program verification.
- Andrew Appel, Lars Birkedal, Adam Chlipala, Magnus Myreen, Gerwin Klein, Peter Lammich, and Zhong Shao, who kindly answered questions on related work aspects.
- Jonathan Leivent for reporting typos on the beta-version of this course.

Chapter 51

Library `SLF.Bib`

51.1 Bib: Bibliography

51.2 Resources cited in this volume

An extended related work section on the history of Separation Logic may be found in the paper *Charguéraud 2020*, available from: http://www.chargueraud.org/research/2020/seq_seplogic/seq_seplog.pdf

The references most relevant to the contents of the course appear below.

Birkedal, Torp-Smith and Yang 2006 Lars Birkedal, Noah Torp-smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules for algol-like languages. In *Logical Methods in Computer Science*.

Burstall 1972 R. M. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. In *Machine Intelligence 7*, Edinburgh University Press.

Charguéraud 2010 Arthur Charguéraud. Characteristic Formulae for Mechanized Program Verification. PhD thesis, Université Paris Diderot, December 2010.

Charguéraud 2011 Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming (ICFP)*. <https://doi.org/10.1145/2034773.2034828>

Charguéraud 2020 Arthur Charguéraud. Separation Logic for Sequential Programs. In *International Conference on Function Programming (ICFP)*. <https://dl.acm.org/doi/abs/10.1145/3408998>

Chlipala et al 2009 Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-Order Imperative Programs. In *International Conference on Functional Programming (ICFP)*. <https://doi.org/10.1145/1596550>. See also the other papers: <https://ynot.cs.harvard.edu/>

Guéneau, Jourdan, Charguéraud, and Pottier 2019 Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *Interactive Theorem Proving (ITP)*. <https://doi.org/10.4230/LIPIcs.ITP.2019.31>

Guéneau, Myreen, Kumar and Norrish 2017 Armaël Guéneau, Magnus O. Myreen, Ramaña Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-54434-1_22

- Dijkstra* 1975 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM. <https://doi.org/10.1145/360933.360975>
- Gordon* 1989 Michael J. C. Gordon. Mechanizing Programming Logics in Higher Order Logic. Springer-Verlag, Berlin, Heidelberg, https://doi.org/10.1007/978-1-4612-3658-0_10
- Krishnaswami, Birkedal, and Aldrich* 2010 Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying Event-Driven Programs Using Ramified Frame Properties. In Workshop on Types in Language Design and Implementation (TLI). <https://doi.org/10.1145/1708016.170>
- Hennessy and Milner* 1985 Matthew Hennessy and Robin Milner. Algebraic laws for non-determinism and concurrency. Journal of the ACM, 32. <https://dl.acm.org/doi/10.1145/2455.2460>
- Honda, Berger, and Yoshida* 2006 Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. International Colloquium on Automata, Languages and Programming (ICALP). https://doi.org/10.1007/11787006_31
- Hobor and Villard* 2013 Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In Symposium on Principles of Programming Languages (POPL). <https://doi.org/10.1145/2429069.2429131>
- Ishtiaq and O'Hearn* 2001 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. SIGPLAN Notice 36. <https://doi.org/10.1145/373243.375719>
- Jung et al* 2018 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ale{\v{s}} Bizjak, Lars Birkedal, Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming. <https://doi.org/10.1017/S0956796818000>
- O'Hearn and Pym* 1999 Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. The Bulletin of Symbolic Logic 5. <https://doi.org/10.2307/421090>
- O'Hearn, Reynolds, and Yang* 2001 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. Workshop on Computer Science Logic (CSL). https://doi.org/10.1007/3-540-44802-0_1
- Reynolds* 2002 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. Annual IEEE Symposium on Logic in Computer Science (LICS). 10.1109/LICS.2002.1029817
- Reynolds* 2000 John C. Reynolds. Intuitionistic Reasoning about Shared Mutable Data Structure <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.5999&rep=rep1&type=pdf> July 2001
- Tuerk* 2010 Thomas Tuerk. Local Reasoning about While-Loops. In International Conference on Verified Software: Theories, Tools and Experiments (VSTTE).
- Weber* 2004 Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In Computer Science Logic (CSL) https://doi.org/10.1007/978-3-540-30124-0_21
- Yu, Hamid, and Shao* 2003 Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. European Symposium on Programming (ESOP). https://doi.org/10.1007/3-540-36575-3_25
- Date*