

Software Foundations: PLF

August 4, 2026

Contents

1	Library <code>PLF.Maps</code>	2
1.1	Maps: Total and Partial Maps	2
1.2	The Standard Library	2
1.3	Identifiers	3
1.4	Total Maps	3
1.5	Partial maps	6
2	Library <code>PLF.Imp</code>	10
2.1	Imp: Simple Imperative Programs	10
2.2	Arithmetic and Boolean Expressions	10
2.2.1	Syntax	11
2.2.2	Evaluation	12
2.2.3	Optimization	12
2.3	Rocq Automation	14
2.3.1	Tacticals	14
2.3.2	Defining New Tactics	18
2.3.3	The <i>lia</i> Tactic	19
2.3.4	A Few More Handy Tactics	20
2.4	Evaluation as a Relation	20
2.4.1	Inference Rule Notation	22
2.4.2	Equivalence of the Definitions	23
2.4.3	Computational vs. Relational Definitions	25
2.5	Expressions With Variables	27
2.5.1	States	27
2.5.2	Syntax	27
2.5.3	Notations	28
2.5.4	Evaluation	29
2.6	Commands	30
2.6.1	Syntax	30
2.6.2	Desugaring Notations	31
2.6.3	<code>Locate</code> Again	32
2.6.4	More Examples	32
2.7	Evaluating Commands	33

2.7.1	Evaluation as a Function (Failed Attempt)	33
2.7.2	Evaluation as a Relation	34
2.7.3	Determinism of Evaluation	37
2.8	Reasoning About Imp Programs	38
2.9	Additional Exercises	39
3	Library PLF.Preface	46
3.1	Preface	46
3.2	Welcome	46
3.3	Overview	46
3.3.1	Program Verification	47
3.3.2	Type Systems	48
3.3.3	Further Reading	48
3.4	Practicalities	48
3.4.1	Recommended Citation Format	48
3.5	Note for Instructors	48
3.6	Thanks	49
4	Library PLF.Equiv	50
4.1	Equiv: Program Equivalence	50
4.2	Behavioral Equivalence	51
4.2.1	Definitions	52
4.2.2	Simple Examples	53
4.3	Properties of Behavioral Equivalence	60
4.3.1	Behavioral Equivalence is an Equivalence	60
4.3.2	Behavioral Equivalence Is a Congruence	61
4.4	Program Transformations	64
4.4.1	The Constant-Folding Transformation	65
4.4.2	Soundness of Constant Folding	68
4.4.3	Soundness of $(0 + n)$ Elimination, Redux	72
4.5	Proving Inequivalence	73
4.6	Extended Exercise: Nondeterministic Imp	76
4.7	Additional Exercises	81
5	Library PLF.Hoare	83
5.1	Hoare: Hoare Logic, Part I	83
5.2	Assertions	84
5.2.1	Notations for Assertions	85
5.2.2	Example Assertions	87
5.2.3	Assertion Implication	88
5.3	Hoare Triples, Informally	88
5.4	Hoare Triples, Formally	90
5.5	Proof Rules	91

5.5.1	Skip	91
5.5.2	Sequencing	91
5.5.3	Assignment	92
5.5.4	Consequence	97
5.5.5	Automation	100
5.5.6	Sequencing + Assignment	104
5.5.7	Conditionals	106
5.5.8	While Loops	114
5.6	Summary	119
5.7	Additional Exercises	120
5.7.1	Havoc	120
5.7.2	Assert and Assume	122
6	Library <code>PLF.Hoare2</code>	129
6.1	Hoare2: Hoare Logic, Part II	129
6.2	Decorated Programs	129
6.2.1	Example: Swapping	130
6.2.2	Example: Simple Conditionals	131
6.2.3	Example: Reduce to Zero	132
6.2.4	Example: Division	132
6.2.5	From Decorated Programs to Formal Proofs	133
6.3	Formal Decorated Programs	136
6.3.1	Syntax	136
6.3.2	Extracting Verification Conditions	142
6.3.3	More Automation	146
6.4	Finding Loop Invariants	148
6.4.1	Example: Slow Subtraction	149
6.4.2	Exercise: Slow Assignment	152
6.4.3	Example: Parity	152
6.4.4	Example: Finding Square Roots	154
6.4.5	Example: Squaring	156
6.4.6	Exercise: Factorial	157
6.4.7	Exercise: Minimum	157
6.4.8	Exercise: Two Loops	158
6.4.9	Exercise: Power Series	159
6.5	Weakest Preconditions (Optional)	162
7	Library <code>PLF.HoareAsLogic</code>	166
7.1	HoareAsLogic: Hoare Logic as a Logic	166
7.2	Hoare Logic and Model Theory	166
7.3	Hoare Logic and Proof Theory	167
7.4	Derivability	169
7.5	Soundness and Completeness	170

7.6	Postscript: Decidability	172
8	Library <code>PLF.Smallstep</code>	174
8.1	Smallstep: Small-step Operational Semantics	174
8.2	A Toy Language	175
8.3	Relations	178
8.3.1	Values	180
8.3.2	Strong Progress and Normal Forms	182
8.4	Multi-Step Reduction	189
8.4.1	Examples	191
8.4.2	Normal Forms Again	192
8.4.3	Equivalence of Big-Step and Small-Step	195
8.4.4	Additional Exercises	196
8.5	Small-Step Imp	198
8.6	Concurrent Imp	201
8.7	A Small-Step Stack Machine	207
8.8	Aside: A <i>normalize</i> Tactic	208
9	Library <code>PLF.Types</code>	210
9.1	Types: Type Systems	210
9.2	Typed Arithmetic Expressions	210
9.2.1	Syntax	210
9.2.2	Operational Semantics	212
9.2.3	Normal Forms and Values	213
9.2.4	Typing	214
9.2.5	Progress	217
9.2.6	Type Preservation	218
9.2.7	Type Soundness	220
9.2.8	Additional Exercises	220
10	Library <code>PLF.Stlc</code>	223
10.1	Stlc: The Simply Typed Lambda-Calculus	223
10.2	Overview	224
10.3	Syntax	225
10.3.1	Types	226
10.3.2	Terms	226
10.4	Operational Semantics	228
10.4.1	Values	228
10.4.2	Substitution	229
10.4.3	Reduction	231
10.4.4	Examples	233
10.5	Typing	235
10.5.1	Contexts	235

10.5.2	Typing Relation	236
10.5.3	Examples	237
11	Library <code>PLF.StlcProp</code>	240
11.1	StlcProp: Properties of STLC	240
11.2	Canonical Forms	240
11.3	Progress	241
11.4	Preservation	243
11.4.1	The Weakening Lemma	243
11.4.2	The Substitution Lemma	244
11.4.3	Main Theorem	246
11.5	Type Soundness	248
11.6	Uniqueness of Types	248
11.7	Context Invariance (Optional)	248
11.8	Additional Exercises	252
11.8.1	Exercise: STLC with Arithmetic	255
11.8.2	The Technical Theorems	257
12	Library <code>PLF.MoreStlc</code>	259
12.1	MoreStlc: More on the Simply Typed Lambda-Calculus	259
12.2	Simple Extensions to STLC	259
12.2.1	Numbers	259
12.2.2	Let Bindings	259
12.2.3	Pairs	260
12.2.4	Unit	262
12.2.5	Sums	262
12.2.6	Lists	264
12.2.7	General Recursion	265
12.2.8	Records	268
12.3	Exercise: Formalizing the Extensions	270
12.3.1	Examples	281
12.3.2	Properties of Typing	287
12.3.3	Weakening	291
13	Library <code>PLF.Sub</code>	295
13.1	Sub: Subtyping	295
13.2	Concepts	295
13.2.1	A Motivating Example	295
13.2.2	Subtyping and Object-Oriented Languages	296
13.2.3	The Subsumption Rule	297
13.2.4	The Subtype Relation	297
13.2.5	Exercises	302
13.3	Formal Definitions	305

13.3.1	Core Definitions	306
13.3.2	Subtyping	309
13.3.3	Typing	311
13.4	Properties	312
13.4.1	Inversion Lemmas for Subtyping	313
13.4.2	Canonical Forms	314
13.4.3	Progress	315
13.4.4	Inversion Lemmas for Typing	317
13.4.5	Weakening	319
13.4.6	Substitution	319
13.4.7	Preservation	320
13.4.8	Records, via Products and Top	321
13.4.9	Exercises	322
13.5	Exercise: Adding Products	323
13.5.1	Formalized “Thought Exercises”	324
14	Library <code>PLF.Typechecking</code>	329
14.1	Typechecking: A Typechecker for STLC	329
14.2	Comparing Types	329
14.3	The Typechecker	330
14.4	Digression: Improving the Notation	331
14.5	Properties	332
14.6	Exercises	334
15	Library <code>PLF.Records</code>	344
15.1	Records: Adding Records to STLC	344
15.2	Adding Records	344
15.3	Formalizing Records	345
15.3.1	Examples	351
15.3.2	Properties of Typing	352
15.3.3	Weakening	355
16	Library <code>PLF.References</code>	358
16.1	References: Typing Mutable References	358
16.2	Definitions	359
16.3	Syntax	359
16.4	Pragmatics	364
16.4.1	Side Effects and Sequencing	364
16.4.2	References and Aliasing	364
16.4.3	Shared State	365
16.4.4	Objects	365
16.4.5	References to Compound Types	366
16.4.6	Null References	367

16.4.7	Garbage Collection	367
16.5	Operational Semantics	367
16.5.1	Locations	367
16.5.2	Stores	368
16.5.3	Reduction	370
16.6	Typing	374
16.6.1	Store typings	374
16.6.2	The Typing Relation	377
16.7	Properties	379
16.7.1	Well-Typed Stores	379
16.7.2	Extending Store Typings	380
16.7.3	Preservation, Finally	382
16.7.4	Substitution Lemma	382
16.7.5	Assignment Preserves Store Typing	383
16.7.6	Weakening for Stores	384
16.7.7	Preservation!	385
16.7.8	Progress	388
16.8	References and Nontermination	390
16.9	Additional Exercises	393
17	Library <code>PLF.RecordSub</code>	394
17.1	RecordSub: Subtyping with Records	394
17.2	Core Definitions	394
17.3	Subtyping	398
17.3.1	Definition	398
17.3.2	Examples	399
17.3.3	Properties of Subtyping	401
17.4	Typing	403
17.4.1	Typing Examples	404
17.4.2	Properties of Typing	404
17.4.3	Weakening	408
18	Library <code>PLF.Norm</code>	412
18.1	Norm: Normalization of STLC	412
18.2	Language	413
18.2.1	Weakening	418
18.3	Normalization	422
18.3.1	Membership in R_T Is Invariant Under Reduction	424
18.3.2	Closed Instances of Terms of Type t Belong to R_T	426

19 Library PLF.PE	435
19.1 PE: Partial Evaluation	435
19.2 Generalizing Constant Folding	436
19.2.1 Partial States	436
19.2.2 Arithmetic Expressions	437
19.2.3 Boolean Expressions	441
19.3 Partial Evaluation of Commands, Without Loops	443
19.3.1 Assignment	443
19.3.2 Conditional	445
19.3.3 The Partial Evaluation Relation	449
19.3.4 Examples	450
19.3.5 Correctness of Partial Evaluation	451
19.4 Partial Evaluation of Loops	453
19.4.1 Examples	456
19.4.2 Correctness	457
19.5 Partial Evaluation of Flowchart Programs	463
19.5.1 Basic blocks	463
19.5.2 Flowchart programs	465
19.5.3 Partial Evaluation of Basic Blocks and Flowchart Programs	465
20 Library PLF.Postscript	469
20.1 Postscript	469
20.2 Looking Back	469
20.3 Looking Around	470
20.4 Looking Forward	473
21 Library PLF.Bib	475
21.1 Bib: Bibliography	475
21.2 Resources cited in this volume	475
22 Library PLF.LibTactics	476
22.1 LibTactics: A Collection of Handy General-Purpose Tactics	476
22.2 Fixing Stdlib	477
22.3 Tools for Programming with Ltac	477
22.3.1 Identity Continuation	477
22.3.2 Untyped Arguments for Tactics	477
22.3.3 Optional Arguments for Tactics	478
22.3.4 Wildcard Arguments for Tactics	478
22.3.5 Position Markers	478
22.3.6 List of Arguments for Tactics	479
22.3.7 Databases of Lemmas	481
22.3.8 On-the-Fly Removal of Hypotheses	482
22.3.9 Numbers as Arguments	483

22.3.10	Testing Tactics	484
22.3.11	Testing evars and non-evars	485
22.3.12	Check No Evar in Goal	485
22.3.13	Helper Function for Introducing Evars	486
22.3.14	Tagging of Hypotheses	486
22.3.15	More Tagging of Hypotheses	486
22.3.16	Deconstructing Terms	486
22.3.17	Action at Occurrence and Action Not at Occurrence	487
22.3.18	An Alias for <code>eq</code>	488
22.4	Check as a tactic instead of a top-level command, with possibility to turn off	488
22.5	Common Tactics for Simplifying Goals Like <code>intuition</code>	488
22.6	Backward and Forward Chaining	489
22.6.1	Application	489
22.6.2	Assertions	491
22.6.3	Instantiation and Forward-Chaining	493
22.6.4	Experimental Tactics for Application	503
22.6.5	Adding Assumptions	504
22.6.6	Application of Tautologies	504
22.6.7	Application Modulo Equalities	505
22.6.8	Absurd Goals	508
22.7	Introduction and Generalization	509
22.7.1	Introduction	509
22.7.2	Introduction using $\Rightarrow$ and $=\gg$	511
22.7.3	Generalization	514
22.7.4	Naming	515
22.8	Rewriting	517
22.8.1	Replace	520
22.8.2	Change	520
22.8.3	Renaming	521
22.8.4	Unfolding	521
22.8.5	Simplification	523
22.8.6	Reduction	524
22.8.7	Substitution	524
22.8.8	Tactics to Work with Proof Irrelevance	525
22.8.9	Proving Equalities	526
22.9	Inversion	528
22.9.1	Basic Inversion	528
22.9.2	Inversion with Substitution	529
22.9.3	Injection with Substitution	532
22.9.4	Inversion and Injection with Substitution –rough implementation	533
22.9.5	Case Analysis	534
22.10	Induction	537

22.11	Coinduction	539
22.12	Decidable Equality	539
22.13	Equivalence	540
22.14	N-ary Conjunctions and Disjunctions	540
22.15	Tactics to Prove Typeclass Instances	546
22.16	Tactics to Invoke Automation	546
22.16.1	Definitions for Parsing Compatibility	546
22.16.2	<i>hint</i> to Add Hints Local to a Lemma	547
22.16.3	<i>jauto</i> , a New Automation Tactic	547
22.16.4	Definitions of Automation Tactics	547
22.16.5	Parsing for Light Automation	550
22.16.6	Parsing for Strong Automation	558
22.17	Tactics to Sort Out the Proof Context	567
22.17.1	Hiding Hypotheses	567
22.17.2	Sorting Hypotheses	570
22.17.3	Clearing Hypotheses	570
22.18	Tactics for Development Purposes	572
22.18.1	Skipping Subgoals	572
22.19	Compatibility with standard library	574
23	Library <code>PLF.UseTactics</code>	575
23.1	UseTactics: Tactic Library for Rocq: A Gentle Introduction	575
23.2	Tactics for Naming and Performing Inversion	576
23.2.1	The Tactic <i>introv</i>	576
23.2.2	The Tactic <i>inverts</i>	577
23.3	Tactics for N-ary Connectives	580
23.3.1	The Tactic <i>splits</i>	580
23.3.2	The Tactic <i>branch</i>	581
23.4	Tactics for Working with Equality	581
23.4.1	The Tactic <i>asserts_rewrite</i>	581
23.4.2	The Tactic <i>fequals</i>	582
23.4.3	The Tactic <i>applies_eq</i>	582
23.5	Some Convenient Shorthands	583
23.5.1	The Tactic <i>unfolds</i>	584
23.5.2	The Tactics <i>false</i> and <i>tryfalse</i>	584
23.5.3	The Tactic <i>gen</i>	585
23.5.4	The Tactics <i>admits</i> , <i>admit_rewrite</i> and <i>admit_goal</i>	585
23.5.5	The Tactic <i>sort</i>	587
23.6	Tactics for Advanced Lemma Instantiation	587
23.6.1	Working of <i>lets</i>	588
23.6.2	Working of <i>applies</i> , <i>forwards</i> and <i>specializes</i>	590
23.7	Summary	590

24 Library <code>PLF.UseAuto</code>	592
24.1 UseAuto: Theory and Practice of Automation in Rocq Proofs	592
24.2 Basic Features of Proof Search	593
24.2.1 Strength of Proof Search	593
24.2.2 Basics	593
24.2.3 Conjunctions	594
24.2.4 Disjunctions	596
24.2.5 Existentials	596
24.2.6 Negation	597
24.2.7 Equalities	597
24.3 How Proof Search Works	598
24.3.1 Search Depth	598
24.3.2 Backtracking	600
24.3.3 Adding Hints	602
24.3.4 Integration of Automation in Tactics	603
24.4 Example Proofs using Automation	604
24.4.1 Determinism	604
24.4.2 Preservation for STLC	607
24.4.3 Progress for STLC	608
24.4.4 BigStep and SmallStep	609
24.4.5 Preservation for STLCRef	610
24.4.6 Progress for STLCRef	613
24.4.7 Subtyping	614
24.5 Advanced Topics in Proof Search	615
24.5.1 Stating Lemmas in the Right Way	615
24.5.2 Unfolding of Definitions During Proof-Search	616
24.5.3 Automation for Proving Absurd Goals	617
24.5.4 Automation for Transitivity Lemmas	619
24.6 Decision Procedures	621
24.6.1 Lia	622
24.6.2 Ring	623
24.6.3 Congruence	623
24.7 Summary	624
25 Library <code>PLF.MapsTest</code>	626
26 Library <code>PLF.ImpTest</code>	629
27 Library <code>PLF.PrefaceTest</code>	636
28 Library <code>PLF.EquivTest</code>	638
29 Library <code>PLF.HoareTest</code>	645

30 Library	PLF.Hoare2Test	655
31 Library	PLF.HoareAsLogicTest	659
32 Library	PLF.SmallstepTest	663
33 Library	PLF.TypesTest	669
34 Library	PLF.StlcTest	674
35 Library	PLF.StlcPropTest	677
36 Library	PLF.MoreStlcTest	683
37 Library	PLF.SubTest	687
38 Library	PLF.TypecheckingTest	693
39 Library	PLF.RecordsTest	696
40 Library	PLF.ReferencesTest	704
41 Library	PLF.RecordSubTest	708
42 Library	PLF.NormTest	715
43 Library	PLF.PETest	717
44 Library	PLF.PostscriptTest	719
45 Library	PLF.BibTest	721
46 Library	PLF.LibTacticsTest	723
47 Library	PLF.UseTacticsTest	725
48 Library	PLF.UseAutoTest	727

Chapter 1

Library `PLF.Maps`

1.1 Maps: Total and Partial Maps

Maps (or *dictionaries*) are ubiquitous data structures both in ordinary programming and in the theory of programming languages; we’re going to need them in many places in the coming chapters.

They also make a nice case study using ideas we’ve seen in previous chapters, including building data structures out of higher-order functions (from *Basics* and *Poly*) and the use of reflection to streamline proofs (from *IndProp*).

We’ll define two flavors of maps: *total* maps, which include a “default” element to be returned when a key being looked up doesn’t exist, and *partial* maps, which instead return an `option` to indicate success or failure. Partial maps are defined in terms of total maps, using `None` as the default element.

1.2 The Standard Library

One small digression before we begin...

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (or, transitively, all the earlier chapters). Instead, in this chapter and from now on, we’re going to import the definitions and theorems we need directly from Rocq’s standard library. You should not notice much difference, though, because we’ve been careful to name our own definitions and theorems the same as their counterparts in the standard library, wherever they overlap.

```
From Stdlib Require Import Arith.  
From Stdlib Require Import Bool.  
From Stdlib Require Export Strings.String.  
From Stdlib Require Import FunctionalExtensionality.  
From Stdlib Require Import List.  
Import LISTNOTATIONS.
```

Documentation for the standard library can be found at <https://rocq-prover.org/doc/V9.0.0/stdlib/index.html>.

The `Search` command is a good way to look for theorems involving objects of specific types. See *Lists* for a reminder of how to use it.

If you want to find out how or where a notation is defined, the `Locate` command is useful. For example, where is the natural addition operation defined in the standard library?

```
Locate "+".
```

(There are several uses of the `+` notation, but only one for naturals.)

```
Print Init.Nat.add.
```

We'll see some more uses of `Locate` in the *Imp* chapter.

1.3 Identifiers

To define maps, we first need a type for the keys that we will use to index into our maps. In *Lists.v* we introduced a fresh type `id` for a similar purpose; here and for the rest of *Software Foundations* we will use the `string` type from Rocq's standard library.

To compare strings, we use the function `eqb_refl` from the `String` module in the standard library.

```
Check String.eqb_refl :
```

```
  ∀ x : string, (x =? x)%string = true.
```

We will often use a few basic properties of string equality... `Check String.eqb_eq :`

```
  ∀ n m : string, (n =? m)%string = true ↔ n = m.
```

```
Check String.eqb_neq :
```

```
  ∀ n m : string, (n =? m)%string = false ↔ n ≠ m.
```

```
Check String.eqb_spec :
```

```
  ∀ x y : string, reflect (x = y) (String.eqb x y).
```

1.4 Total Maps

Our main job in this chapter will be to build a definition of partial maps that is similar in behavior to the one we saw in the *Lists* chapter, plus accompanying lemmas about its behavior.

This time around, though, we're going to use *functions*, rather than lists of key-value pairs, to build maps. The advantage of this representation is that it offers a more “extensional” view of maps: two maps that respond to queries in the same way will be represented as exactly the same function, rather than just as “equivalent” list structures. This simplifies proofs that use maps.

We build up to partial maps in two steps. First, we define a type of *total maps* that return a default value when we look up a key that is not present in the map.

```
Definition total_map (A : Type) := string → A.
```

Intuitively, a total map over an element type A is just a function that can be used to look up **strings**, yielding A s.

The function `t_empty` yields an empty total map, given a default element; this map always returns the default element when applied to any string.

```
Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ => v).
```

More interesting is the map-updating function, which (as always) takes a map m , a key x , and a value v and returns a new map that takes x to v and takes every other key to whatever m does. The novelty here is that we achieve this effect by wrapping a new function around the old one.

```
Definition t_update {A : Type} (m : total_map A)
  (x : string) (v : A) :=
  fun x' => if String.eqb x x' then v else m x'.
```

This definition is a nice example of higher-order programming: `t_update` takes a *function* m and yields a new function `fun x' => ...` that behaves like the desired map.

For example, we can build a map taking **strings** to **bools**, where "foo" and "bar" are mapped to **true** and every other key is mapped to **false**, like this:

```
Definition examplemap :=
  t_update (t_update (t_empty false) "foo" true)
    "bar" true.
```

Next, let's introduce some notations to facilitate working with maps.

First, we use the following notation to represent an empty total map with a default value.

```
Notation "'_-' '!'>' v" := (t_empty v)
  (at level 100, right associativity).
```

```
Example example_empty := ( false).
```

We next introduce a symbolic notation for extending an existing map with a new binding.

```
Notation "x '!'>' v ';' m" := (t_update m x v)
  (at level 100, v constr at level 100, right associativity).
```

The `examplemap` above can now be defined as follows:

```
Definition examplemap' :=
  ( "bar" !-> true;
    "foo" !-> true;
    __ !-> false
  ).
```

This completes the definition of total maps. Note that we don't need to define a *find* operation on this representation of maps because it is just function application!

```
Example update_example1 : examplemap' "baz" = false.
```

Proof. `reflexivity`. Qed.

```
Example update_example2 : examplemap' "foo" = true.
```


Proof. reflexivity. Qed.

Example update_example3 : examplemap' "quux" = false.

Proof. reflexivity. Qed.

Example update_example4 : examplemap' "bar" = true.

Proof. reflexivity. Qed.

When we use maps in later chapters, we'll need several fundamental facts about how they behave.

Even if you don't work the following exercises, make sure you thoroughly understand the statements of the lemmas!

(Some of the proofs require the functional extensionality axiom, which was discussed in the [Logic](#) chapter.)

Exercise: 1 star, standard, optional (t_apply_empty) First, the empty map returns its default element for all keys:

Lemma t_apply_empty : $\forall (A : \text{Type}) (x : \text{string}) (v : A),$
 $(\_\_ \text{!-} \rightarrow v) \ x = v.$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (t_update_eq) Next, if we update a map m at a key x with a new value v and then look up x in the map resulting from the `update`, we get back v :

Lemma t_update_eq : $\forall (A : \text{Type}) (m : \text{total\_map } A) \ x \ v,$
 $(x \text{!-} \rightarrow v ; m) \ x = v.$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (t_update_neq) On the other hand, if we update a map m at a key $x1$ and then look up a *different* key $x2$ in the resulting map, we get the same result that m would have given:

Theorem t_update_neq : $\forall (A : \text{Type}) (m : \text{total\_map } A) \ x1 \ x2 \ v,$
 $x1 \neq x2 \rightarrow$
 $(x1 \text{!-} \rightarrow v ; m) \ x2 = m \ x2.$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (t_update_shadow) If we update a map `m` at a key `x` with a value `v1` and then update again with the same key `x` and another value `v2`, the resulting map behaves the same (gives the same result when applied to any key) as the simpler map obtained by performing just the second `update` on `m`:

Lemma `t_update_shadow` : $\forall (A : \text{Type}) (m : \text{total\_map } A) x v1 v2,$
 $(x \mapsto v2 ; x \mapsto v1 ; m) = (x \mapsto v2 ; m).$

Proof.

Admitted.

□

Exercise: 2 stars, standard (t_update_same) Given strings `x1` and `x2`, we can use the tactic `destruct (eqb_spec x1 x2)` to simultaneously perform case analysis on the result of `String.eqb x1 x2` and generate hypotheses about the equality (in the sense of `=`) of `x1` and `x2`. With the example in chapter *IndProp* as a template, use `String.eqb_spec` to prove the following theorem, which states that if we update a map to assign key `x` the same value as it already has in `m`, then the result is equal to `m`:

Theorem `t_update_same` : $\forall (A : \text{Type}) (m : \text{total\_map } A) x,$
 $(x \mapsto m x ; m) = m.$

Proof.

Admitted.

□

Exercise: 3 stars, standard, especially useful (t_update_permute) Similarly, use `String.eqb_spec` to prove one final property of the `update` function: If we update a map `m` at two distinct keys, it doesn't matter in which order we do the updates.

Theorem `t_update_permute` : $\forall (A : \text{Type}) (m : \text{total\_map } A)$
 $v1 v2 x1 x2,$

$x2 \neq x1 \rightarrow$
 $(x1 \mapsto v1 ; x2 \mapsto v2 ; m)$
 $=$
 $(x2 \mapsto v2 ; x1 \mapsto v1 ; m).$

Proof.

Admitted.

□

1.5 Partial maps

Lastly, we define *partial maps* on top of total maps. A partial map with elements of type `A` is simply a total map with elements of type `option A` and default element `None`.

Definition `partial_map (A : Type) := total_map (option A).`

Definition empty $\{A : \text{Type}\} : \text{partial\_map } A :=$
 $\text{t\_empty None}.$

Definition update $\{A : \text{Type}\} (m : \text{partial\_map } A)$
 $(x : \text{string}) (v : A) :=$

$(x \mapsto \text{Some } v ; m).$

We introduce a similar notation for partial maps: **Notation** "x'|->' v ',' m" := (update
 $m \ x \ v)$

(at level 0, x constr, v at level 200, right associativity).

We can also hide the last case when it is empty. **Notation** "x'|->' v" := (update empty
 $x \ v)$

(at level 0, x constr, v at level 200).

Definition examplepmap :=

("Church" $\mapsto$ true ; "Turing" $\mapsto$ false).

We now straightforwardly lift all of the basic lemmas about total maps to partial maps.

Lemma apply_empty : $\forall (A : \text{Type}) (x : \text{string}),$
 $\text{@empty } A \ x = \text{None}.$

Proof.

intros. unfold empty. rewrite t_apply_empty.

reflexivity.

Qed.

Lemma update_eq : $\forall (A : \text{Type}) (m : \text{partial\_map } A) \ x \ v,$
 $(x \mapsto v ; m) \ x = \text{Some } v.$

Proof.

intros. unfold update. rewrite t_update_eq.

reflexivity.

Qed.

The `update_eq` lemma is used very often in proofs. Adding it to Rocq's global "hint database" allows proof-automation tactics such as `auto` to find it. **#[global] Hint Resolve**
`update_eq : core.`

Theorem update_neq : $\forall (A : \text{Type}) (m : \text{partial\_map } A) \ x1 \ x2 \ v,$
 $x2 \neq x1 \rightarrow$
 $(x2 \mapsto v ; m) \ x1 = m \ x1.$

Proof.

intros $A \ m \ x1 \ x2 \ v \ H.$

unfold update. rewrite t_update_neq.

- reflexivity.

- apply $H.$

Qed.

Lemma update_shadow : $\forall (A : \text{Type}) (m : \text{partial\_map } A) \ x \ v1 \ v2,$
 $(x \mapsto v2 ; x \mapsto v1 ; m) = (x \mapsto v2 ; m).$

Proof.

```
intros A m x v1 v2. unfold update. rewrite t_update_shadow.
reflexivity.
```

Qed.

Theorem update_same : $\forall (A : \text{Type}) (m : \text{partial\_map } A) x v,$
 $m x = \text{Some } v \rightarrow$
 $(x \mapsto v ; m) = m.$

Proof.

```
intros A m x v H. unfold update. rewrite ← H.
apply t_update_same.
```

Qed.

Theorem update_permute : $\forall (A : \text{Type}) (m : \text{partial\_map } A)$
 $x1 x2 v1 v2,$

$x2 \neq x1 \rightarrow$
 $(x1 \mapsto v1 ; x2 \mapsto v2 ; m) = (x2 \mapsto v2 ; x1 \mapsto v1 ; m).$

Proof.

```
intros A m x1 x2 v1 v2. unfold update.
apply t_update_permute.
```

Qed.

One last thing: For partial maps, it's convenient to introduce a notion of map inclusion, stating that all the entries in one map are also present in another:

Definition includedin $\{A : \text{Type}\} (m m' : \text{partial\_map } A) :=$
 $\forall x v, m x = \text{Some } v \rightarrow m' x = \text{Some } v.$

We can then show that map update preserves map inclusion, that is:

Lemma includedin_update : $\forall (A : \text{Type}) (m m' : \text{partial\_map } A)$
 $(x : \text{string}) (vx : A),$

$\text{includedin } m m' \rightarrow$
 $\text{includedin } (x \mapsto vx ; m) (x \mapsto vx ; m').$

Proof.

```
unfold includedin.
intros A m m' x vx H.
intros y vy.
destruct (eqb_spec x y) as [Hxy | Hxy].
- rewrite Hxy.
  rewrite update_eq. rewrite update_eq. intro H1. apply H1.
- rewrite update_neq.
  + rewrite update_neq.
    × apply H.
    × apply Hxy.
  + apply Hxy.
```

Qed.

This property is quite useful for reasoning about languages with variable binding – e.g., the Simply Typed Lambda Calculus, which we will see in *Programming Language Foundations*, where maps are used to keep track of which program variables are defined in a given scope.

Chapter 2

Library **PLF.Imp**

2.1 Imp: Simple Imperative Programs

In this chapter, we take a more serious look at how to use Rocq as a tool to study other things. Our case study is a *simple imperative programming language* called Imp, embodying a tiny core fragment of conventional mainstream languages such as C and Java.

Here is a familiar mathematical function written in Imp.

$Z := X; Y := 1; \text{while } Z \neq 0 \text{ do } Y := Y * Z; Z := Z - 1 \text{ end}$

We concentrate here on defining the *syntax* and *semantics* of Imp; later, in *Programming Language Foundations (Software Foundations, volume 2)*, we develop a theory of *program equivalence* and introduce *Hoare Logic*, a popular logic for reasoning about imperative programs.

```
Set Warnings "-notation-overridden".
From Stdlib Require Import Bool.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat. Import NAT.
From Stdlib Require Import Lia.
From Stdlib Require Import List. Import LISTNOTATIONS.
From Stdlib Require Import Strings.String.
From PLF Require Import Maps.
```

2.2 Arithmetic and Boolean Expressions

We'll present Imp in three parts: first a core language of *arithmetic and boolean expressions*, then an extension of these with *variables*, and finally a language of *commands* including assignment, conditionals, sequencing, and loops.

2.2.1 Syntax

Module AEXP.

These two definitions specify the *abstract syntax* of arithmetic and boolean expressions.

Inductive aexp : Type :=

| ANum (n : nat)
| APlus (a1 a2 : aexp)
| AMinus (a1 a2 : aexp)
| AMult (a1 a2 : aexp).

Inductive bexp : Type :=

| BTrue
| BFalse
| BEq (a1 a2 : aexp)
| BNeq (a1 a2 : aexp)
| BLe (a1 a2 : aexp)
| BGt (a1 a2 : aexp)
| BNot (b : bexp)
| BAnd (b1 b2 : bexp).

In this chapter, we'll mostly elide the translation from the concrete syntax that a programmer would actually write to these abstract syntax trees – the process that, for example, would translate the string "1 + 2 × 3" to the AST

APlus (ANum 1) (AMult (ANum 2) (ANum 3)).

The optional chapter *ImpParser* develops a simple lexical analyzer and parser that can perform this translation. You do not need to understand that chapter to understand this one, but if you haven't already taken a course where these techniques are covered (e.g., a course on compilers) you may want to skim it.

For comparison, here's a conventional BNF (Backus-Naur Form) grammar defining the same abstract syntax:

a := nat | a + a | a - a | a * a
b := true | false | a = a | a <> a | a <= a | a > a | ~ b | b && b
Compared to the Rocq version above...

- The BNF is more informal – for example, it gives some suggestions about the surface syntax of expressions (like the fact that the addition operation is written with an infix +) while leaving other aspects of lexical analysis and parsing (like the relative precedence of +, -, and ×, the use of parens to group subexpressions, etc.) unspecified. Some additional information – and human intelligence – would be required to turn this description into a formal definition, e.g., for implementing a compiler.

The Rocq version consistently omits all this information and concentrates on the abstract syntax only.

- Conversely, the BNF version is lighter and easier to read. Its informality makes it flexible, a big advantage in situations like discussions at the blackboard, where conveying general ideas is more important than nailing down every detail precisely.

Indeed, there are dozens of BNF-like notations and people switch freely among them – usually without bothering to say which kind of BNF they’re using, because there is no need to: a rough-and-ready informal understanding is all that’s important.

It’s good to be comfortable with both sorts of notations: informal ones for communicating between humans and formal ones for carrying out implementations and proofs.

2.2.2 Evaluation

Evaluating an arithmetic expression produces a number.

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) × (aeval a2)
  end.
```

Example test_aeval:

```
aeval (APlus (ANum 2) (ANum 2)) = 4.
```

Proof. reflexivity. Qed.

Similarly, evaluating a boolean expression yields a boolean.

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ (aeval a1) =? (aeval a2)
  | BNeq a1 a2 ⇒ negb ((aeval a1) =? (aeval a2))
  | BLe a1 a2 ⇒ (aeval a1) <=? (aeval a2)
  | BGt a1 a2 ⇒ negb ((aeval a1) <=? (aeval a2))
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

2.2.3 Optimization

We haven’t defined very much yet, but we can already get some mileage out of the definitions. Suppose we define a function that takes an arithmetic expression and slightly simplifies it, changing every occurrence of $0 + e$ (i.e., `(APlus (ANum 0) e)`) into just `e`.


```

Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | APlus (ANum 0) e2 ⇒ optimize_0plus e2
  | APlus e1 e2 ⇒ APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 ⇒ AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 ⇒ AMult (optimize_0plus e1) (optimize_0plus e2)
  end.

```

To gain confidence that our optimization is doing the right thing we can test it on some examples and see if the output looks OK.

Example test_optimize_0plus:

```

optimize_0plus (APlus (ANum 2)
                     (APlus (ANum 0)
                          (APlus (ANum 0) (ANum 1))))
= APlus (ANum 2) (ANum 1).

```

Proof. reflexivity. Qed.

But if we want to be certain the optimization is correct – that evaluating an optimized expression *always* gives the same result as the original – we should prove it!

Theorem optimize_0plus_sound: $\forall a$,
 $\text{aeval} (\text{optimize\_0plus } a) = \text{aeval } a$.

Proof.

```

intros a. induction a.
- reflexivity.
- destruct a1 eqn:Ea1.
  + destruct n eqn:En.
    × simpl. apply IHa2.
    × simpl. rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity. Qed.

```

2.3 Rocq Automation

The amount of repetition in this last proof is a little annoying. And if either the language of arithmetic expressions or the optimization being proved sound were significantly more complex, it would start to be a real problem.

So far, we’ve been doing all our proofs using just a small handful of Rocq’s tactics and completely ignoring its powerful facilities for constructing parts of proofs automatically. This section introduces some of these facilities, and we will see more over the next several chapters. Getting used to them will take some energy – Rocq’s automation is a power tool – but it will allow us to scale up our efforts to more complex definitions and more interesting properties without becoming overwhelmed by boring, repetitive, low-level details.

2.3.1 Tacticals

Tacticals is Rocq’s term for tactics that take other tactics as arguments – “higher-order tactics,” if you will.

The `try` Tactical

If `T` is a tactic, then `try T` is a tactic that is just like `T` except that, if `T` fails, `try T` *successfully* does nothing at all (rather than failing). `Theorem silly1 :  $\forall (P : \text{Prop}), P \rightarrow P$ .`

Proof.

```
intros P HP.  
try reflexivity.   apply HP. Qed.
```

`Theorem silly2 :  $\forall ae, \text{aeval } ae = \text{aeval } ae$ .`

Proof.

```
try reflexivity. Qed.
```

There is not much reason to use `try` in completely manual proofs like these, but it is very useful for doing automated proofs in conjunction with the `; tactical`, which we show next.

The `; Tactical` (Simple Form)

In its most common form, the `; tactical` takes two tactics as arguments. The compound tactic `T;T'` first performs `T` and then performs `T'` on *each subgoal* generated by `T`.

For example, consider the following trivial lemma:

`Lemma foo :  $\forall n, 0 \leq n \rightarrow n = \text{true}$ .`

Proof.

```
intros.  
destruct n.  
- simpl. reflexivity.  
- simpl. reflexivity.
```

Qed.

We can simplify this proof using the `; tactical`:

Lemma `foo'` : $\forall n, 0 \leq n \rightarrow n = \text{true}$.

Proof.

```
intros.
destruct n;

simpl;

reflexivity.
Qed.
```

Using `try` and `; together`, we can get rid of the repetition in the proof that was bothering us a little while ago.

Theorem `optimize_0plus_sound'`: $\forall a,$
 $\text{aeval} (\text{optimize\_0plus } a) = \text{aeval } a$.

Proof.

```
intros a.
induction a;

  try (simpl; rewrite IHa1; rewrite IHa2; reflexivity).
- reflexivity.
-
  destruct a1 eqn:Ea1;

  try (simpl; simpl in IHa1; rewrite IHa1;
    rewrite IHa2; reflexivity).
+ destruct n eqn:En;
  simpl; rewrite IHa2; reflexivity. Qed.
```

Rocq experts often use this “...; `try`...” idiom after a tactic like `induction` to take care of many similar cases all at once. Indeed, this practice has an analog in informal proofs. For example, here is an informal proof of the optimization theorem that matches the structure of the formal one:

Theorem: For all arithmetic expressions `a`,
 $\text{aeval} (\text{optimize\_0plus } a) = \text{aeval } a$.

Proof: By induction on `a`. Most cases follow directly from the IH. The remaining cases are as follows:

- Suppose `a = ANum n` for some `n`. We must show
 $\text{aeval} (\text{optimize\_0plus } (\text{ANum } n)) = \text{aeval } (\text{ANum } n)$.
This is immediate from the definition of `optimize_0plus`.
- Suppose `a = APlus a1 a2` for some `a1` and `a2`. We must show

$\text{aeval } (\text{optimize\_0plus } (\text{APlus } a1 \ a2)) = \text{aeval } (\text{APlus } a1 \ a2).$

Consider the possible forms of $a1$. For most of them, `optimize_0plus` simply calls itself recursively for the subexpressions and rebuilds a new expression of the same form as $a1$; in these cases, the result follows directly from the IH.

The interesting case is when $a1 = \text{ANum } n$ for some n . If $n = 0$, then

$\text{optimize\_0plus } (\text{APlus } a1 \ a2) = \text{optimize\_0plus } a2$

and the IH for $a2$ is exactly what we need. On the other hand, if $n = \text{S } n'$ for some n' , then again `optimize_0plus` simply calls itself recursively, and the result follows from the IH. $\square$

However, this proof can still be improved: the first case (for $a = \text{ANum } n$) is very trivial – even more trivial than the cases that we said simply followed from the IH – yet we have chosen to write it out in full. It would be better and clearer to drop it and just say, at the top, “Most cases are either immediate or direct from the IH. The only interesting case is the one for `APlus...`” We can make the same improvement in our formal proof too. Here’s how it looks:

Theorem `optimize_0plus_sound`’: $\forall a,$

$\text{aeval } (\text{optimize\_0plus } a) = \text{aeval } a.$

Proof.

`intros a.`

`induction a;`

`try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);`

`try reflexivity.`

`-`

`destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;
rewrite IHa2; reflexivity).`

`+ destruct n;`

`simpl; rewrite IHa2; reflexivity. Qed.`

The ; Tactical (General Form)

The `;` tactical also has a more general form than the simple $T;T'$ we’ve seen above. If T , $T1$, ..., Tn are tactics, then

$T; T1 \mid T2 \mid \dots \mid Tn$

is a tactic that first performs T and then performs $T1$ on the first subgoal generated by T , performs $T2$ on the second subgoal, etc.

So $T;T'$ is just special notation for the case when all of the Ti ’s are the same tactic; i.e., $T;T'$ is shorthand for:

$T; T' \mid T' \mid \dots \mid T'$

The `repeat` Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails or until it succeeds but doesn't make any progress.

Here is an example proving that 10 is in a long list using `repeat`.

`Theorem In10 : In 10 [1;2;3;4;5;6;7;8;9;10].`

`Proof.`

`repeat (try (left; reflexivity); right).`

`Qed.`

The tactic `repeat T` never fails: if the tactic `T` doesn't apply to the original goal, then `repeat succeeds` without changing the goal at all (i.e., it repeats zero times).

`Theorem In10' : In 10 [1;2;3;4;5;6;7;8;9;10].`

`Proof.`

`repeat simpl.`

`repeat (left; reflexivity).`

`repeat (right; try (left; reflexivity)).`

`Qed.`

The tactic `repeat T` does not have any upper bound on the number of times it applies `T`. If `T` is a tactic that *always* succeeds (and makes progress), then `repeat T` will loop forever.

`Theorem repeat_loop : ∀ (m n : nat),`

`m + n = n + m.`

`Proof.`

`intros m n.`

Admitted.

Wait – did we just write an infinite loop in Rocq?!?

Sort of.

While evaluation in Rocq's term language, Gallina, is guaranteed to terminate, *tactic* evaluation is not. This does not affect Rocq's logical consistency, however, since the job of `repeat` and other tactics is to guide Rocq in constructing proofs; if the construction process diverges (i.e., it does not terminate), this simply means that we have failed to construct a proof at all, not that we have constructed a bad proof.

Exercise: 3 stars, standard (optimize_0plus_b_sound) Since the `optimize_0plus` transformation doesn't change the value of `aexps`, we should be able to apply it to all the `aexps` that appear in a `bexp` without changing the `bexp`'s value. Write a function that performs this transformation on `bexps` and prove it is sound. Use the tacticals we've just seen to make the proof as short and elegant as possible.

`Fixpoint optimize_0plus_b (b : bexp) : bexp`

`. Admitted.`

`Example optimize_0plus_b_test1:`

`optimize_0plus_b` (BNot (BGt (APlus (ANum 0) (ANum 4)) (ANum 8))) =
 (BNot (BGt (ANum 4) (ANum 8))).

Proof. *Admitted.*

Example `optimize_0plus_b_test2`:

`optimize_0plus_b` (BAnd (BLe (APlus (ANum 0) (ANum 4)) (ANum 5)) BTrue) =
 (BAnd (BLe (ANum 4) (ANum 5)) BTrue).

Proof. *Admitted.*

Theorem `optimize_0plus_b_sound` : $\forall b$,
`beval` (`optimize_0plus_b b`) = `beval b`.

Proof.

Admitted.

□

Exercise: 4 stars, standard, optional (optimize) *Design exercise:* The optimization implemented by our `optimize_0plus` function is only one of many possible optimizations on arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct. (You will probably find it easiest to start small – add just a single, simple optimization and its correctness proof – and build up incrementally to something more interesting.)

2.3.2 Defining New Tactics

Rocq also provides facilities for “programming” in tactic scripts.

The `Ltac` idiom illustrated below gives a handy way to define “shorthand tactics” that bundle several tactics into a single command.

`Ltac` also includes syntactic pattern-matching on the goal and context, as well as general programming facilities.

It is useful for proof automation and there are several idioms for programming with `Ltac`. Because it is a language style you might not have seen before, a good reference is the textbook “Certified Programming with dependent types” *CPDT*, which is more advanced than what we will need in this course, but is considered by many the best reference for `Ltac` programming.

Just for future reference: Rocq provides two other ways of defining new tactics. There is a `Tactic Notation` command that allows defining new tactics with custom control over their concrete syntax. And there is also a low-level API that can be used to build tactics that directly manipulate Rocq’s internal structures. We will not need either of these for present purposes.

Here’s an example `Ltac` script called *invert*.

`Ltac invert H` :=
`inversion H; subst; clear H.`

This defines a new tactic called *invert* that takes a hypothesis *H* as an argument and performs the sequence of commands *inversion H*; *subst*; *clear H*. This gives us quick way to do inversion on evidence and constructors, rewrite with the generated equations, and remove the redundant hypothesis at the end.

```
Lemma invert_example1:  $\forall \{a \ b \ c : \text{nat}\}, [a \ ; b] = [a ; c] \rightarrow b = c.$ 
  intros.
  invert H.
  reflexivity.
Qed.
```

2.3.3 The *lia* Tactic

The *lia* tactic implements a decision procedure for integer linear arithmetic, a subset of propositional logic and arithmetic.

If the goal is a universally quantified formula made out of

- numeric constants, addition (+ and *S*), subtraction (- and *pred*), and multiplication by constants (this is what makes it Presburger arithmetic),
- equality (= and $\neq$) and ordering ($\leq$ and $>$), and
- the logical connectives $\wedge$, $\vee$, $\neg$, and $\rightarrow$,

then invoking *lia* will either solve the goal or fail, meaning that the goal is actually false. (If the goal is *not* of this form, *lia* will fail.)

```
Example silly_presburger_example :  $\forall \ m \ n \ o \ p,$ 
   $m + n \leq n + o \wedge o + 3 = p + 3 \rightarrow$ 
   $m \leq p.$ 
```

Proof.

```
  intros. lia.
```

Qed.

```
Example add_comm_lia :  $\forall \ m \ n,$ 
   $m + n = n + m.$ 
```

Proof.

```
  intros. lia.
```

Qed.

```
Example add_assoc_lia :  $\forall \ m \ n \ p,$ 
   $m + (n + p) = m + n + p.$ 
```

Proof.

```
  intros. lia.
```

Qed.

(Note the *From Stdlib Require Import Lia.* at the top of this file, which makes *lia* available.)

2.3.4 A Few More Handy Tactics

Finally, here are some miscellaneous tactics that you may find convenient.

- `clear  $H$` : Delete hypothesis H from the context.
- `subst  $x$` : Given a variable x , find an assumption $x = e$ or $e = x$ in the context, replace x with e throughout the context and current goal, and clear the assumption.
- `subst`: Substitute away *all* assumptions of the form $x = e$ or $e = x$ (where x is a variable).
- `rename... into...`: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named x , then `rename  $x$  into  $y$` will change all occurrences of x to y .
- `assumption`: Try to find a hypothesis H in the context that exactly matches the goal; if one is found, solve the goal.
- `contradiction`: Try to find a hypothesis H in the context that is logically equivalent to `False`. If one is found, solve the goal.
- `constructor`: Try to find a constructor c (from some `Inductive` definition in the current environment) that can be applied to solve the current goal. If one is found, behave like `apply  $c$` .

We'll see examples of all of these as we go along.

2.4 Evaluation as a Relation

We have presented `aeval` and `beval` as functions defined by `Fixpoints`. Another way to think about evaluation – one that is often more flexible – is as a *relation* between expressions and their values. This perspective leads to `Inductive` definitions like the following...

`Module AEVALR_FIRST_TRY.`

```
Inductive aevalR : aexp → nat → Prop :=
| E_ANum ( $n$  : nat) :
  aevalR (ANum  $n$ )  $n$ 
| E_APlus ( $e1$   $e2$  : aexp) ( $n1$   $n2$  : nat) :
  aevalR  $e1$   $n1$  →
  aevalR  $e2$   $n2$  →
  aevalR (APlus  $e1$   $e2$ ) ( $n1 + n2$ )
| E_AMinus ( $e1$   $e2$  : aexp) ( $n1$   $n2$  : nat) :
  aevalR  $e1$   $n1$  →
  aevalR  $e2$   $n2$  →
```



```

    aevalR (AMinus e1 e2) (n1 - n2)
| E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
    aevalR e1 n1 →
    aevalR e2 n2 →
    aevalR (AMult e1 e2) (n1 × n2).

```

Module HYPOTHESISNAMES.

A small notational aside. We could also write the definition of **aevalR** as follow, with explicit names for the hypotheses in each case:

```

Inductive aevalR : aexp → nat → Prop :=
| E_ANum (n : nat) :
    aevalR (ANum n) n
| E_APlus (e1 e2 : aexp) (n1 n2 : nat)
    (H1 : aevalR e1 n1)
    (H2 : aevalR e2 n2) :
    aevalR (APlus e1 e2) (n1 + n2)
| E_AMinus (e1 e2 : aexp) (n1 n2 : nat)
    (H1 : aevalR e1 n1)
    (H2 : aevalR e2 n2) :
    aevalR (AMinus e1 e2) (n1 - n2)
| E_AMult (e1 e2 : aexp) (n1 n2 : nat)
    (H1 : aevalR e1 n1)
    (H2 : aevalR e2 n2) :
    aevalR (AMult e1 e2) (n1 × n2).

```

This style gives us more control over the names that Rocq chooses during proofs involving **aevalR**, at the cost of making the definition a little more verbose.

End HYPOTHESISNAMES.

It will be convenient to have an infix notation for **aevalR**. We'll write $e ==> n$ to mean that arithmetic expression e evaluates to value n .

```

Notation "e '==>' n"
:= (aevalR e n)
   (at level 90, left associativity)
: type_scope.

```

End AEVALR_FIRST_TRY.

As we saw in our case study of regular expressions in chapter *IndProp*, Rocq provides a way to use this notation in the definition of **aevalR** itself.

Reserved Notation "e '==>' n" (at level 90, left associativity).

```

Inductive aevalR : aexp → nat → Prop :=
| E_ANum (n : nat) :
    (ANum n) ==> n

```

```

| E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
  (e1 ==> n1) →
  (e2 ==> n2) →
  (APlus e1 e2) ==> (n1 + n2)
| E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
  (e1 ==> n1) →
  (e2 ==> n2) →
  (AMinus e1 e2) ==> (n1 - n2)
| E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
  (e1 ==> n1) →
  (e2 ==> n2) →
  (AMult e1 e2) ==> (n1 × n2)

where "e '==>' n" := (aevalR e n) : type_scope.

```

2.4.1 Inference Rule Notation

In informal discussions, it is convenient to write the rules for **aevalR** and similar relations in the more readable graphical form of *inference rules*, where the premises above the line justify the conclusion below the line.

For example, the constructor **E_APlus**...

| E_APlus : forall (e1 e2 : aexp) (n1 n2 : nat), aevalR e1 n1 -> aevalR e2 n2 -> aevalR (APlus e1 e2) (n1 + n2)
 ...can be written like this as an inference rule:
 e1 ==> n1 e2 ==> n2

(E_APlus) APlus e1 e2 ==> n1+n2

Formally, there is nothing deep about inference rules: they are just implications.

You can read the rule name on the right as the name of the constructor and read each of the linebreaks between the premises above the line (as well as the line itself) as $\rightarrow$.

All the variables mentioned in the rule (*e1*, *n1*, etc.) are implicitly bound by universal quantifiers at the beginning. (Such variables are often called *metavariables* to distinguish them from the variables of the language we are defining. At the moment, our arithmetic expressions don't include variables, but we'll soon be adding them.)

The whole collection of rules is understood as being wrapped in an **Inductive** declaration. In informal prose, this is sometimes indicated by saying something like “Let **aevalR** be the smallest relation closed under the following rules...”.

For example, we could define $\Rightarrow$ as the smallest relation closed under these rules:

(E_ANum) ANum n ==> n
 e1 ==> n1 e2 ==> n2

(E_APlus) APlus e1 e2 ==> n1+n2

$$e1 ==> n1 \quad e2 ==> n2$$

(E_AMinus) AMinus $e1 \ e2 ==> n1 - n2$
 $e1 ==> n1 \quad e2 ==> n2$

(E_AMult) AMult $e1 \ e2 ==> n1 * n2$

Exercise: 1 star, standard, optional (beval_rules) Here, again, is the Rocq definition of the `beval` function:

Fixpoint `beval` ($e : \text{bexp}$) : bool := match e with | BTrue => true | BFalse => false | BEq $a1 \ a2$ => (aeval $a1$) ==? (aeval $a2$) | BNeq $a1 \ a2$ => negb ((aeval $a1$) ==? (aeval $a2$)) | BLe $a1 \ a2$ => (aeval $a1$) <=? (aeval $a2$) | BGt $a1 \ a2$ => ~((aeval $a1$) <=? (aeval $a2$)) | BNot b => negb (beval b) | BAnd $b1 \ b2$ => andb (beval $b1$) (beval $b2$) end.

Write out a corresponding definition of boolean evaluation as a relation (in inference rule notation).

Definition `manual_grade_for_beval_rules` : option (nat × string) := None.

□

2.4.2 Equivalence of the Definitions

It is straightforward to prove that the relational and functional definitions of evaluation agree:

Theorem `aevalR_iff_aeval` : $\forall \ a \ n,$
 $(a ==> n) \leftrightarrow \text{aeval } a = n.$

Proof.

```
split.
-
  intros H.
  induction H; simpl.
+
  reflexivity.
+
  rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
+
  rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
+
  rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
-
  generalize dependent n.
  induction a;
    simpl; intros; subst.
+
```

```

    apply E_ANum.
+
    apply E_APlus.
    × apply IHa1. reflexivity.
    × apply IHa2. reflexivity.
+
    apply E_AMinus.
    × apply IHa1. reflexivity.
    × apply IHa2. reflexivity.
+
    apply E_AMult.
    × apply IHa1. reflexivity.
    × apply IHa2. reflexivity.
Qed.

```

Again, we can make the proof quite a bit shorter using some tacticals.

```

Theorem aevalR_iff_aeval' : ∀ a n,
  (a ==> n) ↔ aeval a = n.
Proof.
  split.
-
  intros H; induction H; subst; reflexivity.
-
  generalize dependent n.
  induction a; simpl; intros; subst; constructor;
    try apply IHa1; try apply IHa2; reflexivity.
Qed.

```

Exercise: 3 stars, standard (bevalR) Write a relation **bevalR** in the same style as **aevalR**, and prove that it is equivalent to **beval**.

Reserved Notation "e '==>b' b" (at level 90, left associativity).

Inductive **bevalR**: **bexp** → **bool** → Prop :=

```

where "e '==>b' b" := (bevalR e b) : type_scope
.

```

```

Lemma bevalR_iff_beval : ∀ b bv,
  b ==>b bv ↔ beval b = bv.

```

Proof.

Admitted.

□

End AEXP.

2.4.3 Computational vs. Relational Definitions

For the definitions of evaluation for arithmetic and boolean expressions, the choice of whether to use functional or relational definitions is mainly a matter of taste: either way works fine.

However, there are many situations where relational definitions of evaluation work much better than functional ones.

Module AEVALR_DIVISION.

For example, suppose that we wanted to extend the arithmetic operations with division:

```
Inductive aexp : Type :=
| ANum (n : nat)
| APlus (a1 a2 : aexp)
| AMinus (a1 a2 : aexp)
| AMult (a1 a2 : aexp)
| ADiv (a1 a2 : aexp).
```

Extending the definition of `aeval` to handle this new operation would not be straightforward (what should we return as the result of `ADiv (ANum 5) (ANum 0)`?). But extending `aevalR` is very easy.

```
Reserved Notation "e '==>' n"
(at level 90, left associativity).
```

```
Inductive aevalR : aexp → nat → Prop :=
| E_ANum (n : nat) :
  (ANum n) ==> n
| E_APlus (a1 a2 : aexp) (n1 n2 : nat) :
  (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)
| E_AMinus (a1 a2 : aexp) (n1 n2 : nat) :
  (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)
| E_AMult (a1 a2 : aexp) (n1 n2 : nat) :
  (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)
| E_ADiv (a1 a2 : aexp) (n1 n2 n3 : nat) :
  (a1 ==> n1) → (a2 ==> n2) → (n2 > 0) →
  (mult n2 n3 = n1) → (ADiv a1 a2) ==> n3
```

```
where "a '==>' n" := (aevalR a n) : type_scope.
```

Notice that this evaluation relation corresponds to a *partial* function: There are some inputs for which it does not specify an output.

End AEVALR_DIVISION.

Module AEVALR_EXTENDED.

Or suppose that we want to extend the arithmetic operations by a nondeterministic number generator *any* that, when evaluated, may yield any number.

(Note that this is not the same as making a *probabilistic* choice among all possible numbers – we’re not specifying any particular probability distribution for the results, just saying what results are *possible*.)

Reserved Notation "e '==>' n" (at level 90, left associativity).

Inductive **aexp** : Type :=

```
| AAny
| ANum (n : nat)
| APlus (a1 a2 : aexp)
| AMinus (a1 a2 : aexp)
| AMult (a1 a2 : aexp).
```

Again, extending **aeval** would be tricky, since now evaluation is *not* a deterministic function from expressions to numbers; but extending **aevalR** is no problem...

Inductive **aevalR** : **aexp** → **nat** → Prop :=

```
| E_Any (n : nat) :
  AAny ==> n
| E_ANum (n : nat) :
  (ANum n) ==> n
| E_APlus (a1 a2 : aexp) (n1 n2 : nat) :
  (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)
| E_AMinus (a1 a2 : aexp) (n1 n2 : nat) :
  (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)
| E_AMult (a1 a2 : aexp) (n1 n2 : nat) :
  (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)
```

where "a '==>' n" := (**aevalR** a n) : *type_scope*.

End **AEVALR_EXTENDED**.

At this point you maybe wondering: Which of these styles should I use by default?

In the examples we’ve just seen, relational definitions turned out to be more useful than functional ones. For situations like these, where the thing being defined is not easy to express as a function, or indeed where it is *not* a function, there is no real choice. But what about when both styles are workable?

One point in favor of relational definitions is that they can be more elegant and easier to understand.

Another is that Rocq automatically generates nice inversion and induction principles from **Inductive** definitions.

On the other hand, functional definitions can often be more convenient:

- Functions are automatically deterministic and total; for a relational definition, we have to *prove* these properties explicitly if we need them.
- With functions we can also take advantage of Rocq’s computation mechanism to simplify expressions during proofs.

Furthermore, functions can be directly “extracted” from Gallina to executable code in OCaml or Haskell.

Ultimately, the choice often comes down to either the specifics of a particular situation or simply a question of taste. Indeed, in large Rocq developments it is common to see a definition given in *both* functional and relational styles, plus a lemma stating that the two coincide, allowing further proofs to switch from one point of view to the other at will.

2.5 Expressions With Variables

Let’s return to defining `Imp`, where the next thing we need to do is to enrich our arithmetic and boolean expressions with variables.

To keep things simple, we’ll assume that all variables are global and that they only hold numbers.

2.5.1 States

Since we’ll want to look variables up to find out their current values, we’ll use total maps from the `Maps` chapter.

A *machine state* (or just *state*) represents the current values of all variables at some point in the execution of a program.

For simplicity, we assume that the state is defined for *all* variables, even though any given program is only able to mention a finite number of them. Because each variable stores a natural number, we can represent the state as a total map from strings (variable names) to `nat`, and will use 0 as default value in the store.

Definition `state := total_map nat`.

2.5.2 Syntax

We can add variables to the arithmetic expressions we had before simply by including one more constructor:

```
Inductive aexp : Type :=
| ANum (n : nat)
| Ald (x : string)
| APlus (a1 a2 : aexp)
| AMinus (a1 a2 : aexp)
| AMult (a1 a2 : aexp).
```

Defining a few variable names as notational shorthands will make examples easier to read:

Definition `W : string := "W"`.

Definition `X : string := "X"`.

Definition `Y : string := "Y"`.

Definition **Z** : **string** := "Z".

(This convention for naming program variables (**X**, **Y**, **Z**) clashes a bit with our earlier use of uppercase letters for types. Since we're not using polymorphism heavily in the chapters developed to Imp, this overloading should not cause confusion.)

The definition of **bexps** is unchanged (except that it now refers to the new **aexps**):

Inductive **bexp** : **Type** :=

```
| BTrue
| BFalse
| BEq (a1 a2 : aexp)
| BNeq (a1 a2 : aexp)
| BLe (a1 a2 : aexp)
| BGt (a1 a2 : aexp)
| BNot (b : bexp)
| BAnd (b1 b2 : bexp).
```

2.5.3 Notations

To make Imp programs easier to read and write, we introduce some notations and implicit coercions.

You do not need to understand exactly what these declarations do.

Briefly, though:

- The **Coercion** declaration stipulates that a function (or constructor) can be implicitly used by the type system to coerce a value of the input type to a value of the output type. For instance, the coercion declaration for **Ald** allows us to use plain strings when an **aexp** is expected; the string will implicitly be wrapped with **Ald**.
- *Declare Custom Entry* **com** tells Rocq to create a new “custom grammar” for parsing Imp expressions and programs. The first notation declaration after this tells Rocq that anything between $\langle \{$ and $\} \rangle$ should be parsed using the Imp grammar. Again, it is not necessary to understand the details, but it is important to recognize that we are defining *new* interpretations for some familiar operators like $+$, $-$, $\times$, $=$, $\leq$, etc., when they occur between $\langle \{$ and $\} \rangle$.

Coercion **Ald** : **string** $\rightarrow$ **aexp**.

Coercion **ANum** : **nat** $\rightarrow$ **aexp**.

Declare Custom Entry **com**.

Declare Scope *com_scope*.

Notation " $\langle \{ e \} \rangle$ " := *e*

(*e custom com, format "[hv' <{ ' / ' '[v' e ']' ' / ' }> ']"*) : *com_scope*.

Notation "(*x*)" := *x* (*in custom com, x at level 99*).

Notation "*x*" := *x* (*in custom com at level 0, x constr at level 0*).

Notation "f x .. y" := ($\lambda x. (f x) .. y$)
 (in *custom com* at level 0, *only parsing*,
 f constr at level 0, x constr at level 1,
 y constr at level 1).
 Notation "x + y" := (APlus x y) (in *custom com* at level 50, left associativity).
 Notation "x - y" := (AMinus x y) (in *custom com* at level 50, left associativity).
 Notation "x * y" := (AMult x y) (in *custom com* at level 40, left associativity).
 Notation "'true'" := true (at level 1).
 Notation "'true'" := BTrue (in *custom com* at level 0).
 Notation "'false'" := false (at level 1).
 Notation "'false'" := BFalse (in *custom com* at level 0).
 Notation "x <= y" := (BLe x y) (in *custom com* at level 70, no associativity).
 Notation "x > y" := (BGt x y) (in *custom com* at level 70, no associativity).
 Notation "x = y" := (BEq x y) (in *custom com* at level 70, no associativity).
 Notation "x <> y" := (BNeq x y) (in *custom com* at level 70, no associativity).
 Notation "x && y" := (BAnd x y) (in *custom com* at level 80, left associativity).
 Notation "'~' b" := (BNot b) (in *custom com* at level 75, right associativity).

Open Scope *com_scope*.

We can now write $3 + (X \times 2)$ instead of APlus 3 (AMult X 2), and $\text{true} \ \&\& \ \sim(X \leq 4)$ instead of BAnd true (BNot (BLe X 4)).

Definition example_aexp : aexp := <{ 3 + (X $\times$ 2) }>.

Definition example_bexp : bexp := <{ true && $\neg(X \leq 4)$ }>.

2.5.4 Evaluation

The arith and boolean evaluators must now be extended to handle variables in the obvious way, taking a state *st* as an extra argument:

```

Fixpoint aeval (st : state)
  (a : aexp) : nat :=
  match a with
  | ANum n  $\Rightarrow$  n
  | Ald x  $\Rightarrow$  st x
  | <{a1 + a2}>  $\Rightarrow$  (aeval st a1) + (aeval st a2)
  | <{a1 - a2}>  $\Rightarrow$  (aeval st a1) - (aeval st a2)
  | <{a1  $\times$  a2}>  $\Rightarrow$  (aeval st a1)  $\times$  (aeval st a2)
  end.

```

```

Fixpoint beval (st : state)
  (b : bexp) : bool :=
  match b with
  | <{true}>  $\Rightarrow$  true
  | <{false}>  $\Rightarrow$  false

```

```

| <{a1 = a2}> ⇒ (aeval st a1) =? (aeval st a2)
| <{a1 ≠ a2}> ⇒ negb ((aeval st a1) =? (aeval st a2))
| <{a1 ≤ a2}> ⇒ (aeval st a1) <=? (aeval st a2)
| <{a1 > a2}> ⇒ negb ((aeval st a1) <=? (aeval st a2))
| <{¬ b1}> ⇒ negb (beval st b1)
| <{b1 && b2}> ⇒ andb (beval st b1) (beval st b2)
end.

```

We can use our notation for total maps in the specific case of states – i.e., we write the empty state as `(-- !-> 0)`.

Definition `empty_st` := `(-- !-> 0)`.

Also, we can add a notation for a “singleton state” with just one variable bound to a value.

Notation `"x '!->' v"` := `(x !-> v ; empty_st)` (at level 100, right associativity).

Example `aexpl` :

```

aeval (X !-> 5) <{ 3 + (X × 2) }>
= 13.

```

Proof. reflexivity. Qed.

Example `aexp2` :

```

aeval (X !-> 5 ; Y !-> 4) <{ Z + (X × Y) }>
= 20.

```

Proof. reflexivity. Qed.

Example `bexpl` :

```

beval (X !-> 5) <{ true && ¬(X ≤ 4) }>
= true.

```

Proof. reflexivity. Qed.

2.6 Commands

Now we are ready to define the syntax and behavior of Imp *commands* (or *statements*).

2.6.1 Syntax

Informally, commands `c` are described by the following BNF grammar.

`c` := skip | `x := a` | `c ; c` | if `b` then `c` else `c` end | while `b` do `c` end

Here is the formal definition of the abstract syntax of commands:

Inductive `com` : Type :=

```

| CSkip
| CAsgn (x : string) (a : aexp)
| CSeq (c1 c2 : com)
| Clf (b : bexp) (c1 c2 : com)
| CWhile (b : bexp) (c : com).

```

As we did for expressions, we can use a few **Notation** declarations to make reading and writing Imp programs more convenient.

```

Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
   format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
   right associativity,
   format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
   format "'[v' 'if' x 'then' '/' y '/' 'else' '/' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
   format "'[v' 'while' x 'do' '/' y '/' 'end' ']'") : com_scope.

```

For example, here is the factorial function again, written as a formal Rocq definition. When this command terminates, the variable **Y** will contain the factorial of the initial value of **X**.

```

Definition fact_in_coq : com :=
  <{ Z := X;
    Y := 1;
    while Z ≠ 0 do
      Y := Y × Z;
      Z := Z - 1
    end }>.
Print fact_in_coq.

```

2.6.2 Desugaring Notations

Rocq offers a rich set of features to manage the increasing complexity of the objects we work with, such as coercions and notations. However, their heavy usage can make it hard to understand what the expressions we enter actually mean. In such situations it is often instructive to “turn off” those features to get a more elementary picture of things, using the following commands:

- **Unset Printing Notations** (undo with **Set Printing Notations**)
- **Set Printing Coercions** (undo with **Unset Printing Coercions**)
- **Set Printing All** (undo with **Unset Printing All**)

These commands can also be used in the middle of a proof, to elaborate the current goal and context.

```
Unset Printing Notations.
Print fact_in_coq.
Set Printing Notations.
Print example_bexp.
Set Printing Coercions.
Print example_bexp.
Print fact_in_coq.
Unset Printing Coercions.
```

2.6.3 Locate Again

Finding identifiers

When used with an identifier, the `Locate` prints the full path to every value in scope with the same name. This is useful to troubleshoot problems due to variable shadowing. `Locate aexp`.

Finding notations

When faced with an unknown notation, you can use `Locate` with a string containing one of its symbols to see its possible interpretations. `Locate "&&"`.

```
Locate ";".
```

```
Locate "while".
```

2.6.4 More Examples

Assignment:

```
Definition plus2 : com :=
  <{ X := X + 2 }>.
```

```
Definition XtimesYinZ : com :=
  <{ Z := X × Y }>.
```

Loops

```
Definition subtract_slowly_body : com :=
  <{ Z := Z - 1 ;
    X := X - 1 }>.
```

```
Definition subtract_slowly : com :=
```

```

<{ while X ≠ 0 do
  subtract_slowly_body
end }>.

```

Definition subtract_3_from_5_slowly : **com** :=
 <{ X := 3 ;
 Z := 5 ;
 subtract_slowly }>.

An infinite loop:

Definition loop : **com** :=
 <{ while true do
 skip
 end }>.

2.7 Evaluating Commands

Next we need to define what it means to evaluate an Imp command. The fact that *while* loops don't necessarily terminate makes defining an evaluation function tricky...

2.7.1 Evaluation as a Function (Failed Attempt)

Here's an attempt at defining an evaluation function for commands (with a bogus *while* case).

```

Fixpoint ceval_fun_no_while (st : state) (c : com) : state :=
  match c with
  | <{ skip }> =>
    st
  | <{ x := a }> =>
    (x !-> aeval st a ; st)
  | <{ c1 ; c2 }> =>
    let st' := ceval_fun_no_while st c1 in
    ceval_fun_no_while st' c2
  | <{ if b then c1 else c2 end }> =>
    if (beval st b)
    then ceval_fun_no_while st c1
    else ceval_fun_no_while st c2
  | <{ while b do c end }> =>
    st
  end.

```

In a more conventional functional programming language like OCaml or Haskell we could add the *while* case as follows:

Fixpoint *ceval_fun* (st : state) (c : com) : state := match c with ... | <{ while b do c end }> => if (beval st b) then *ceval_fun* st <{ c ; while b do c end }> else st end.

Rocq doesn't accept such a definition ("Error: Cannot guess decreasing argument of fix") because the function we want to define is not guaranteed to terminate. Indeed, it *doesn't* always terminate: for example, the full version of the *ceval_fun* function applied to the *loop* program above would never terminate. Since Rocq aims to be not just a functional programming language but also a consistent logic, any potentially non-terminating function needs to be rejected.

Here is an example showing what would go wrong if Rocq allowed non-terminating recursive functions:

Fixpoint *loop_false* (n : nat) : False := *loop_false* n.

That is, propositions like **False** would become provable (*loop_false* 0 would be a proof of **False**), which would be a disaster for Rocq's logical consistency.

Thus, because it doesn't terminate on all inputs, *ceval_fun* cannot be written in Rocq – at least not without additional tricks and workarounds (see chapter *ImpCEvalFun* if you're curious about those).

2.7.2 Evaluation as a Relation

Here's a better way: define **ceval** as a *relation* rather than a *function* – i.e., make its result a *Prop* rather than a *state*, similar to what we did for **aevalR** above.

This is an important change. Besides freeing us from awkward workarounds, it gives us a ton more flexibility in the definition. For example, if we add nondeterministic features like *any* to the language, we want the definition of evaluation to be nondeterministic – i.e., not only will it not be total, it will not even be a function!

We'll use the notation $st = [c] => st'$ for the **ceval** relation: $st = [c] => st'$ means that executing program *c* in a starting state *st* results in an ending state *st'*. This can be pronounced "*c* takes state *st* to *st'*".

Operational Semantics

Here is an informal definition of evaluation, presented as inference rules for readability:

(E_Skip) $st = \textit{skip} => st$
 $\textit{aeval } st \ a = n$

(E_Asgn) $st = x := a => (x \text{ !-> } n ; st)$
 $st = c1 => st' \ st' = c2 => st''$

(E_Seq) $st = c1 ; c2 => st''$
 $\textit{beval } st \ b = \text{true} \ st = c1 => st'$

(E_IfTrue) $st = \text{if } b \text{ then } c1 \text{ else } c2 \text{ end} \Rightarrow st'$
 $\text{beval } st \ b = \text{false} \ st = c2 \Rightarrow st'$

(E_IfFalse) $st = \text{if } b \text{ then } c1 \text{ else } c2 \text{ end} \Rightarrow st'$
 $\text{beval } st \ b = \text{false}$

(E_WhileFalse) $st = \text{while } b \text{ do } c \text{ end} \Rightarrow st$
 $\text{beval } st \ b = \text{true} \ st = c \Rightarrow st' \ st' = \text{while } b \text{ do } c \text{ end} \Rightarrow st''$

(E_WhileTrue) $st = \text{while } b \text{ do } c \text{ end} \Rightarrow st''$

Here is the formal definition. Make sure you understand how it corresponds to the inference rules.

Reserved Notation

"st0 '[c]=>' st1"
 (at level 40, *c* custom com at level 99,
st0 constr, *st1* constr at next level,
 format "[hv' st0 = ['/' ' '[c]' '/']=> st1 ']'").

Inductive **ceval** : **com** $\rightarrow$ state $\rightarrow$ state $\rightarrow$ Prop :=

| E_Skip : $\forall st,$
 $st = [\text{skip}] \Rightarrow st$
| E_Asgn : $\forall st \ a \ n \ x,$
 $\text{aeval } st \ a = n \rightarrow$
 $st = [x := a] \Rightarrow (x \ !\rightarrow n ; st)$
| E_Seq : $\forall c1 \ c2 \ st \ st' \ st'',$
 $st = [c1] \Rightarrow st' \rightarrow$
 $st' = [c2] \Rightarrow st'' \rightarrow$
 $st = [c1 ; c2] \Rightarrow st''$
| E_IfTrue : $\forall st \ st' \ b \ c1 \ c2,$
 $\text{beval } st \ b = \text{true} \rightarrow$
 $st = [c1] \Rightarrow st' \rightarrow$
 $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$
| E_IfFalse : $\forall st \ st' \ b \ c1 \ c2,$
 $\text{beval } st \ b = \text{false} \rightarrow$
 $st = [c2] \Rightarrow st' \rightarrow$
 $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$
| E_WhileFalse : $\forall b \ st \ c,$
 $\text{beval } st \ b = \text{false} \rightarrow$
 $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st$
| E_WhileTrue : $\forall st \ st' \ st'' \ b \ c,$
 $\text{beval } st \ b = \text{true} \rightarrow$
 $st = [c] \Rightarrow st' \rightarrow$

```

st' = [ while b do c end ] => st'' →
st = [ while b do c end ] => st''

```

```

where "st0 = [ c ] => st1" := (ceval c st0 st1).

```

The cost of defining evaluation as a relation instead of a function is that we now need to construct a *proof* that some program evaluates to some result state, rather than just letting Rocq's computation mechanism do it for us.

Example `ceval_example1`:

```

empty_st = [
  X := 2;
  if (X ≤ 1)
    then Y := 3
    else Z := 4
  end
] => (Z !-> 4 ; X !-> 2).

```

Proof.

```

apply E_Seq with (X !-> 2).
-
  apply E_Asgn. reflexivity.
-
  apply E_IfFalse.
  + reflexivity.
  + apply E_Asgn. reflexivity.

```

Qed.

Exercise: 2 stars, standard (`ceval_example2`) Example `ceval_example2`:

```

empty_st = [
  X := 0;
  Y := 1;
  Z := 2
] => (Z !-> 2 ; Y !-> 1 ; X !-> 0).

```

Proof.

Admitted.

□

Set Printing Implicit.

Check @`ceval_example2`.

Exercise: 3 stars, standard, optional (`pup_to_n`) Write an Imp program that sums the numbers from 1 to `X` (inclusive: $1 + 2 + \dots + X$) in the variable `Y`. Your program should update the state as shown in theorem `pup_to_2_ceval`, which you can reverse-engineer to discover the program you should write. The proof of that theorem will be somewhat lengthy.

Definition pup_to_n : com

. *Admitted*.

Theorem pup_to_2_ceval :

(X !-> 2) = [
 pup_to_n
] => (X !-> 0 ; Y !-> 3 ; X !-> 1 ; Y !-> 2 ; Y !-> 0 ; X !-> 2).

Proof.

Admitted.

□

2.7.3 Determinism of Evaluation

Changing from a computational to a relational definition of evaluation is a good move because it frees us from the artificial requirement that evaluation should be a total function. But it also raises a question: Is the second definition of evaluation really a partial *function*? Or is it possible that, beginning from the same state *st*, we could evaluate some command *c* in different ways to reach two different output states *st'* and *st''*?

In fact, this cannot happen: **ceval** is a partial function:

Theorem ceval_deterministic: $\forall c\ st\ st1\ st2,$

$st = [c] \Rightarrow st1 \rightarrow$

$st = [c] \Rightarrow st2 \rightarrow$

$st1 = st2.$

Proof.

intros c st st1 st2 E1 E2.

generalize dependent st2.

induction E1; intros st2 E2; inversion E2; subst.

- reflexivity.

- reflexivity.

-

rewrite (IHE1_1 st'0 H1) in *.

apply IHE1_2. assumption.

-

apply IHE1. assumption.

-

rewrite H in H5. discriminate.

-

rewrite H in H5. discriminate.

-

apply IHE1. assumption.

-

reflexivity.

-

```

    rewrite  $H$  in  $H2$ . discriminate.
-
    rewrite  $H$  in  $H4$ . discriminate.
-
    rewrite ( $IHE1\_1$   $st'0$   $H3$ ) in *.
    apply  $IHE1\_2$ . assumption. Qed.

```

2.8 Reasoning About Imp Programs

We'll get into more systematic and powerful techniques for reasoning about Imp programs in *Programming Language Foundations*, but we can already do a few things (albeit in a somewhat low-level way) just by working with the bare definitions. This section explores some examples.

Theorem `plus2_spec` : $\forall st\ n\ st'$,
 $st\ X = n \rightarrow$
 $st = [plus2] \Rightarrow st' \rightarrow$
 $st'\ X = n + 2$.

Proof.

```

  intros  $st\ n\ st'\ HX\ Heval$ .

```

Inverting *Heval* essentially forces Rocq to expand one step of the `ceval` computation – in this case revealing that st' must be st extended with the new value of X , since `plus2` is an assignment.

```

  inversion  $Heval$ . subst. clear  $Heval$ . simpl.
  apply  $t\_update\_eq$ . Qed.

```

Exercise: 3 stars, standard, optional (`XtimesYinZ_spec`) State and prove a specification of `XtimesYinZ`.

Definition `manual_grade_for_XtimesYinZ_spec` : `option (nat × string)` := `None`.

□

Exercise: 3 stars, standard, especially useful (`loop_never_stops`) **Theorem** `loop_never_stops` : $\forall st\ st'$,
 $\sim(st = [loop] \Rightarrow st')$.

Proof.

```

  intros  $st\ st'\ contra$ . unfold loop in  $contra$ .
  remember <{ while true do skip end }> as  $loopdef$ 
    eqn: $Hegloopdef$ .

```

Proceed by induction on the assumed derivation showing that *loopdef* terminates. Most of the cases are immediately contradictory and so can be solved in one step with `discriminate`.

Admitted.

□

Exercise: 3 stars, standard (no_whiles_eqv) Consider the following function:

```
Fixpoint no_whiles (c : com) : bool :=
  match c with
  | <{ skip }> =>
    true
  | <{ _ := _ }> =>
    true
  | <{ c1 ; c2 }> =>
    andb (no_whiles c1) (no_whiles c2)
  | <{ if _ then ct else cf end }> =>
    andb (no_whiles ct) (no_whiles cf)
  | <{ while _ do _ end }> =>
    false
  end.
```

This predicate yields `true` just on programs that have no while loops. Using `Inductive`, write a property `no_whilesR` such that `no_whilesR c` is provable exactly when `c` is a program with no while loops. Then prove its equivalence with `no_whiles`.

```
Inductive no_whilesR: com → Prop :=
```

.

```
Theorem no_whiles_eqv:
```

```
  ∀ c, no_whiles c = true ↔ no_whilesR c.
```

Proof.

Admitted.

□

Exercise: 4 stars, standard (no_whiles_terminating) Imp programs that don't involve while loops always terminate. State and prove a theorem `no_whiles_terminating` that says this.

Use either `no_whiles` or `no_whilesR`, as you prefer.

```
Definition manual_grade_for_no_whiles_terminating : option (nat × string) := None.
```

□

2.9 Additional Exercises

Exercise: 3 stars, standard (stack_compiler) Old HP Calculators, programming languages like Forth and Postscript, and abstract machines like the Java Virtual Machine all evaluate arithmetic expressions using a *stack*. For instance, the expression

$(2*3)+(3*(4-2))$

would be written as

2 3 * 3 4 2 - * +

and evaluated like this (where we show the program being evaluated on the right and the contents of the stack on the left):

| 2 3 * 3 4 2 - * + 2 | 3 * 3 4 2 - * + 3, 2 | * 3 4 2 - * + 6 | 3 4 2 - * + 3, 6 | 4 2 - * + 4, 3, 6 | 2 - * + 2, 4, 3, 6 | - * + 2, 3, 6 | * + 6, 6 | + 12 |

The goal of this exercise is to write a small compiler that translates **aexps** into stack machine instructions.

The instruction set for our stack language will consist of the following instructions:

- **SPush** *n*: Push the number *n* on the stack.
- **SLoad** *x*: Load the identifier *x* from the store and push it on the stack
- **SPlus**: Pop the two top numbers from the stack, add them, and push the result onto the stack.
- **SMinus**: Similar, but subtract the first number from the second.
- **SMult**: Similar, but multiply.

Inductive **sinstr** : **Type** :=

| **SPush** (*n* : **nat**)
| **SLoad** (*x* : **string**)
| **SPlus**
| **SMinus**
| **SMult**.

Write a function to evaluate programs in the stack language. It should take as input a state, a stack represented as a list of numbers (top stack item is the head of the list), and a program represented as a list of instructions, and it should return the stack after executing the program. Test your function on the examples below.

Note that it is unspecified what to do when encountering an **SPlus**, **SMinus**, or **SMult** instruction if the stack contains fewer than two elements. In a sense, it is immaterial what we do, since a correct compiler will never emit such a malformed program. But for sake of later exercises, it would be best to skip the offending instruction and continue with the next one.

Fixpoint **s_execute** (*st* : **state**) (*stack* : **list nat**)
 (*prog* : **list sinstr**)
 : **list nat**

. *Admitted*.

Check **s_execute**.

Example **s_executel** :

```

    s_execute empty_st []
      [SPush 5; SPush 3; SPush 1; SMinus]
  = [2; 5].
  Admitted.

```

Example `s_execute2` :

```

    s_execute (X !-> 3) [3;4]
      [SPush 4; SLoad X; SMult; SPlus]
  = [15; 4].
  Admitted.

```

Next, write a function that compiles an **aexp** into a stack machine program. The effect of running the program should be the same as pushing the value of the expression on the stack.

```

Fixpoint s_compile (e : aexp) : list sinstr
. Admitted.

```

After you've defined `s_compile`, prove the following to test that it works.

```

Example s_compile1 :
  s_compile <{ X - (2 × Y) }>
    = [SLoad X; SPush 2; SLoad Y; SMult; SMinus].
  Admitted.
□

```

Exercise: 3 stars, standard (execute_app) Execution can be decomposed in the following sense: executing stack program `p1 ++ p2` is the same as executing `p1`, taking the resulting stack, and executing `p2` from that stack. Prove that fact.

```

Theorem execute_app : ∀ st p1 p2 stack,
  s_execute st stack (p1 ++ p2) = s_execute st (s_execute st stack p1) p2.
Proof.
  Admitted.
□

```

Exercise: 3 stars, standard (stack_compiler_correct) Now we'll prove the correctness of the compiler implemented in the previous exercise. Begin by proving the following lemma. If it becomes difficult, consider whether your implementation of `s_execute` or `s_compile` could be simplified.

```

Lemma s_compile_correct_aux : ∀ st e stack,
  s_execute st stack (s_compile e) = aeval st e :: stack.
Proof.
  Admitted.

```

The main theorem should be a very easy corollary of that lemma.

Theorem `s_compile_correct` : $\forall (st : \text{state}) (e : \text{aexp}),$
 $s\_execute\ st\ []\ (s\_compile\ e) = [aeval\ st\ e].$

Proof.

Admitted.

□

Exercise: 3 stars, standard, optional (short_circuit) Most modern programming languages use a “short-circuit” evaluation rule for boolean **and**: to evaluate **BAnd** *b1 b2*, first evaluate *b1*. If it evaluates to **false**, then the entire **BAnd** expression evaluates to **false** immediately, without evaluating *b2*. Otherwise, *b2* is evaluated to determine the result of the **BAnd** expression.

Write an alternate version of **beval** that performs short-circuit evaluation of **BAnd** in this manner, and prove that it is equivalent to **beval**. (N.b. This is only true because expression evaluation in **Imp** is rather simple. In a bigger language where evaluating an expression might diverge, the short-circuiting **BAnd** would *not* be equivalent to the original, since it would make more programs terminate.)

Module **BREAKIMP**.

Exercise: 4 stars, standard, optional (break_imp) Imperative languages like C and Java often include a **break** or similar statement for interrupting the execution of loops. In this exercise we consider how to add **break** to **Imp**. First, we need to enrich the language of commands with an additional case.

Inductive **com** : Type :=

| CSkip
 | CBreak
 | CAsgn (*x* : string) (*a* : aexp)
 | CSeq (*c1 c2* : com)
 | Clf (*b* : bexp) (*c1 c2* : com)
 | CWhile (*b* : bexp) (*c* : com).

Notation "'break'" := CBreak (in *custom com* at level 0) : *com_scope*.

Notation "'skip'" := CSkip

(in *custom com* at level 0) : *com_scope*.

Notation "*x* := *y*" := (CAsgn *x y*)

(in *custom com* at level 0, *x* constr at level 0, *y* at level 85, no associativity,
 format "*x* := *y*") : *com_scope*.

Notation "*x* ; *y*" := (CSeq *x y*)

(in *custom com* at level 90,
 right associativity,
 format "[*v* '*x* ; '*y* ']" : *com_scope*.

Notation "'if' *x* 'then' *y* 'else' *z* 'end'" := (Clf *x y z*)

(in *custom com* at level 89, *x* at level 99, *y* at level 99, *z* at level 99,

```

format "[v' 'if' x 'then' '/' y '/' 'else' '/' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
(in custom com at level 89, x at level 99, y at level 99,
format "[v' 'while' x 'do' '/' y '/' 'end' ']'") : com_scope.

```

Next, we need to define the behavior of *break*. Informally, whenever *break* is executed in a sequence of commands, it stops the execution of that sequence and signals that the innermost enclosing loop should terminate. (If there aren't any enclosing loops, then the whole program simply terminates.) The final state should be the same as the one in which the *break* statement was executed.

One important point is what to do when there are multiple loops enclosing a given *break*. In those cases, *break* should only terminate the *innermost* loop. Thus, after executing the following...

```

X := 0; Y := 1; while 0 <> Y do while true do break end; X := 1; Y := Y - 1 end
... the value of X should be 1, and not 0.

```

One way of expressing this behavior is to add another parameter to the evaluation relation that specifies whether evaluation of a command executes a *break* statement:

```

Inductive result : Type :=
| SContinue
| SBreak.

```

Reserved Notation

```

"st0 '[ c ]=>' st1 '/' s"
(at level 40, c custom com at level 99,
st0 constr, st1 constr at next level,
format "[hv' st0 =[ '/' '[ c ]' '/' ]=> st1 / s ']'").

```

Intuitively, $st = [c] \Rightarrow st' / s$ means that, if c is started in state st , then it terminates in state st' and either signals that the innermost surrounding loop (or the whole program) should exit immediately ($s = \text{SBreak}$) or that execution should continue normally ($s = \text{SContinue}$).

The definition of the " $st = [c] \Rightarrow st' / s$ " relation is very similar to the one we gave above for the regular evaluation relation ($st = [c] \Rightarrow st'$) – we just need to handle the termination signals appropriately:

- If the command is *skip*, then the state doesn't change and execution of any enclosing loop can continue normally.
- If the command is *break*, the state stays unchanged but we signal a *SBreak*.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution can continue normally.
- If the command is of the form *if b then c1 else c2 end*, then the state is updated as in the original semantics of Imp, except that we also propagate the signal from the execution of whichever branch was taken.

- If the command is a sequence $c1 ; c2$, we first execute $c1$. If this yields a **SBreak**, we skip the execution of $c2$ and propagate the **SBreak** signal to the surrounding context; the resulting state is the same as the one obtained by executing $c1$ alone. Otherwise, we execute $c2$ on the state obtained after executing $c1$, and propagate the signal generated there.
- Finally, for a loop of the form $\text{while } b \text{ do } c \text{ end}$, the semantics is almost the same as before. The only difference is that, when b evaluates to true, we execute c and check the signal that it raises. If that signal is **SContinue**, then the execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop, and the resulting state is the same as the one resulting from the execution of the current iteration. In either case, since break only terminates the innermost loop, while signals **SContinue**.

Based on the above description, complete the definition of the **ceval** relation.

Inductive **ceval** : **com** $\rightarrow$ state $\rightarrow$ **result** $\rightarrow$ state $\rightarrow$ Prop :=
 | E_Skip : $\forall st,$
 $st = [\text{CSkip}] \Rightarrow st / \text{SContinue}$

where $"st' = [c] \Rightarrow st' / s" := (\text{ceval } c \text{ } st \text{ } s \text{ } st')$.

Now prove the following properties of your definition of **ceval**:

Theorem **break_ignore** : $\forall c \text{ } st \text{ } st' \text{ } s,$
 $st = [\text{break}; c] \Rightarrow st' / s \rightarrow$
 $st = st'.$

Proof.

Admitted.

Theorem **while_continue** : $\forall b \text{ } c \text{ } st \text{ } st' \text{ } s,$
 $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st' / s \rightarrow$
 $s = \text{SContinue}.$

Proof.

Admitted.

Theorem **while_stops_on_break** : $\forall b \text{ } c \text{ } st \text{ } st',$
 $\text{beval } st \text{ } b = \text{true} \rightarrow$
 $st = [c] \Rightarrow st' / \text{SBreak} \rightarrow$
 $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st' / \text{SContinue}.$

Proof.

Admitted.

Theorem **seq_continue** : $\forall c1 \text{ } c2 \text{ } st \text{ } st' \text{ } st'',$
 $st = [c1] \Rightarrow st' / \text{SContinue} \rightarrow$
 $st' = [c2] \Rightarrow st'' / \text{SContinue} \rightarrow$
 $st = [c1 ; c2] \Rightarrow st'' / \text{SContinue}.$

Proof.

Admitted.

Theorem seq_stops_on_break : $\forall c1\ c2\ st\ st',$
 $st = [c1] \Rightarrow st' / \text{SBreak} \rightarrow$
 $st = [c1 ; c2] \Rightarrow st' / \text{SBreak}.$

Proof.

Admitted.

□

Exercise: 3 stars, advanced, optional (while_break_true) Theorem while_break_true

: $\forall b\ c\ st\ st',$
 $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st' / \text{SContinue} \rightarrow$
 $\text{beval } st' \ b = \text{true} \rightarrow$
 $\exists st'', st'' = [c] \Rightarrow st' / \text{SBreak}.$

Proof.

Admitted.

□

Exercise: 4 stars, advanced, optional (ceval_deterministic) Theorem ceval_deterministic:

$\forall (c:\text{com})\ st\ st1\ st2\ s1\ s2,$
 $st = [c] \Rightarrow st1 / s1 \rightarrow$
 $st = [c] \Rightarrow st2 / s2 \rightarrow$
 $st1 = st2 \wedge s1 = s2.$

Proof.

Admitted.

□ End BREAKIMP.

Exercise: 4 stars, standard, optional (add_for_loop) Add C-style **for** loops to the language of commands, update the **ceval** definition to define the semantics of **for** loops, and add cases for **for** loops as needed so that all the proofs in this file are accepted by Rocq.

A **for** loop should be parameterized by (a) a statement executed initially, (b) a test that is run on each iteration of the loop to determine whether the loop should continue, (c) a statement executed at the end of each loop iteration, and (d) a statement that makes up the body of the loop. (You don't need to worry about making up a concrete Notation for **for** loops, but feel free to play with this too if you like.)

Chapter 3

Library **PLF.Preface**

3.1 Preface

3.2 Welcome

This electronic book is a survey of basic concepts in the mathematical study of programs and programming languages. Topics include advanced use of the Rocq proof assistant, operational semantics, Hoare logic, and static type systems. The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity will be helpful.

As with all of the books in the *Software Foundations* series, this one is one hundred percent formalized and machine-checked: the entire text is literally a script for Rocq. It is intended to be read alongside (or inside) an interactive session with Rocq. All the details in the text are fully formalized in Rocq, and most of the exercises are designed to be worked using Rocq.

The files are organized into a sequence of core chapters, covering about one half semester's worth of material and organized into a coherent linear narrative, plus a number of “offshoot” chapters covering additional topics. All the core chapters are suitable for both upper-level undergraduate and graduate students.

The book builds on the material from *Logical Foundations* (*Software Foundations*, volume 1). It can be used together with that book for a one-semester course on the theory of programming languages. Or, for classes where students who are already familiar with some or all of the material in *Logical Foundations*, there is plenty of additional material to fill most of a semester from this book alone.

3.3 Overview

The book develops two main conceptual threads:

(1) formal techniques for *reasoning about the properties of specific programs* (e.g., the fact that a sorting function or a compiler obeys some formal specification); and

(2) the use of *type systems* for establishing well-behavedness guarantees for *all* programs in a given programming language (e.g., the fact that well-typed Java programs cannot be subverted at runtime).

Each of these is easily rich enough to fill a whole course in its own right, and tackling both together naturally means that much will be left unsaid. Nevertheless, we hope readers will find that these themes illuminate and amplify each other and that bringing them together creates a good foundation for digging into any of them more deeply. Some suggestions for further reading can be found in the [Postscript](#) chapter. Bibliographic information for all cited works can be found in the file [Bib](#).

3.3.1 Program Verification

In the first part of the book, we introduce two broad topics of critical importance in building reliable software (and hardware): techniques for proving specific properties of particular *programs* and for proving general properties of whole programming *languages*.

For both of these, the first thing we need is a way of representing programs as mathematical objects, so we can talk about them precisely, plus ways of describing their behavior in terms of mathematical functions or relations. Our main tools for these tasks are *abstract syntax* and *operational semantics*, a method of specifying programming languages by writing abstract interpreters. At the beginning, we work with operational semantics in the so-called “big-step” style, which leads to simple and readable definitions when it is applicable. Later on, we switch to a lower-level “small-step” style, which helps make some useful distinctions (e.g., between different sorts of nonterminating program behaviors) and which is applicable to a broader range of language features, including concurrency.

The first programming language we consider in detail is *Imp*, a tiny toy language capturing the core features of conventional imperative programming: variables, assignment, conditionals, and loops.

We study two different ways of reasoning about the properties of *Imp* programs. First, we consider what it means to say that two *Imp* programs are *equivalent* in the intuitive sense that they exhibit the same behavior when started in any initial memory state. This notion of equivalence then becomes a criterion for judging the correctness of *metaprograms* – programs that manipulate other programs, such as compilers and optimizers. We build a simple optimizer for *Imp* and prove that it is correct.

Second, we develop a methodology for proving that a given *Imp* program satisfies some formal specifications of its behavior. We introduce the notion of *Hoare triples* – *Imp* programs annotated with pre- and post-conditions describing what they expect to be true about the memory in which they are started and what they promise to make true about the memory in which they terminate – and the reasoning principles of *Hoare Logic*, a domain-specific logic specialized for convenient compositional reasoning about imperative programs, with concepts like “loop invariant” built in.

This part of the course is intended to give readers a taste of the key ideas and mathematical tools used in a wide variety of real-world software and hardware verification tasks.

3.3.2 Type Systems

Our other major topic, covering approximately the second half of the book, is *type systems* – powerful tools for establishing properties of *all* programs in a given language.

Type systems are the best established and most popular example of a highly successful class of formal verification techniques known as *lightweight formal methods*. These are reasoning techniques of modest power – modest enough that automatic checkers can be built into compilers, linkers, or program analyzers and thus be applied even by programmers unfamiliar with the underlying theories. Other examples of lightweight formal methods include hardware and software model checkers, contract checkers, and run-time monitoring techniques.

This also completes a full circle with the beginning of the book: the language whose properties we study in this part, the *simply typed lambda-calculus*, is essentially a simplified model of the core of Rocq itself!

3.3.3 Further Reading

This text is intended to be self contained, but readers looking for a deeper treatment of particular topics will find some suggestions for further reading in the [Postscript](#) chapter.

3.4 Practicalities

3.4.1 Recommended Citation Format

If you want to refer to this volume in your own writing, please do so as follows:

```
@book {Pierce:SF2, author = {Benjamin C. Pierce and Arthur Azevedo de Amorim
and Chris Casinghino and Marco Gaboardi and Michael Greenberg and Cătălin Hrițcu
and Vilhelm Sjöberg and Andrew Tolmach and Brent Yorgey}, editor = {Benjamin C.
Pierce}, title = "Programming Language Foundations", series = "Software Foundations",
volume = "2", year = "2026", publisher = "Electronic textbook", note = {Version 7.0,
\URL{http://softwarefoundations.cis.upenn.edu} }, }
```

3.5 Note for Instructors

If you plan to use these materials in your own course, you will undoubtedly find things you'd like to change, improve, or add. Your contributions are welcome! Please see the [Preface](#) to *Logical Foundations* for instructions.

3.6 Thanks

Development of 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 4

Library **PLF.Equiv**

4.1 Equiv: Program Equivalence

```
Set Warnings "-notation-overridden".
From PLF Require Import Maps.
From Stdlib Require Import Bool.
From Stdlib Require Import Arith.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import PeanoNat. Import NAT.
From Stdlib Require Import EqNat.
From Stdlib Require Import Lia.
From Stdlib Require Import List. Import LISTNOTATIONS.
From Stdlib Require Import FunctionalExtensionality.
From PLF Require Export Imp.
```

Before You Get Started:

- Create a fresh directory for this volume. Do not try to mix the files from this volume with files from *Logical Foundations* in the same directory: the result will not make you happy.

You can either start with an empty directory and populate it with the files listed below (and others as you go along), or else download the whole PLF zip file and unzip it.

- The new directory should contain at least the following files:
 - *Imp.v* (make sure it is the one from the PLF distribution, not the one from LF: they are slightly different);
 - *Maps.v* (ditto)
 - *Equiv.v* (this file)
 - *_CoqProject*, containing the following line:

* Q . PLF

- If you see errors like this...
Compiled library PLF.Maps (in file .../plf/Maps.vo) makes inconsistent assumptions over library Rocq.Init.Logic
... it may mean something went wrong with the above steps. Doing “*make clean*” (or manually removing everything except *.v* and *_CoqProject* files) may help.
- If you are using VSCode with the VSCoq extension, you’ll then want to open a new window in VSCode, click *Open Folder* > *plf*, and run *make*. If you get an error like “Cannot find a physical path...” error, it may be because you didn’t open plf directly (you instead opened a folder containing both *lf* and *plf*, for example).

Advice for Working on Exercises:

- Most of the Rocq proofs we ask you to do in this chapter are similar to proofs that we’ve provided. Before starting to work on exercises, take the time to work through our proofs (both informally and in Rocq) and make sure you understand them in detail. This will save you a lot of effort.
- The Rocq proofs we’re doing now are sufficiently complicated that it is more or less impossible to complete them by random experimentation or following your nose. You need to start with an idea about why the property is true and how the proof is going to go. The best way to do this is to write out at least a sketch of an informal proof on paper – one that intuitively convinces you of the truth of the theorem – before starting to work on the formal one. Alternately, grab a friend and try to convince them that the theorem is true; then try to formalize your explanation.
- Use automation to save work! The proofs in this chapter can get pretty long if you try to write out all the cases explicitly.

4.2 Behavioral Equivalence

In an earlier chapter, we investigated the correctness of a very simple program transformation: the *optimize_0plus* function. The programming language we were considering was the first version of the language of arithmetic expressions – with no variables – so in that setting it was very easy to define what it means for a program transformation to be correct: it should always yield a program that evaluates to the same number as the original.

To talk about the correctness of program transformations for the full Imp language – in particular, assignment – we need to consider the role of mutable state and develop a more sophisticated notion of correctness, which we’ll call *behavioral equivalence*.

For example:

- $X + 2$ is behaviorally equivalent to $1 + X + 1$
- $X - X$ is behaviorally equivalent to 0
- $(X - 1) + 1$ is *not* behaviorally equivalent to X

4.2.1 Definitions

For **aexprs** and **bexprs** with variables, the definition we want is clear: Two **aexprs** or **bexprs** are “behaviorally equivalent” if they evaluate to the same result in every state.

Definition `aequiv (a1 a2 : aexpr) : Prop :=`
 `∀ (st : state),`
 `aeval st a1 = aeval st a2.`

Definition `bequiv (b1 b2 : bexpr) : Prop :=`
 `∀ (st : state),`
 `beval st b1 = beval st b2.`

Here are some simple examples of equivalences of arithmetic and boolean expressions.

Theorem `aequiv_example:`

```
aequiv
  <{ X - X }>
  <{ 0 }>.
```

Proof.

```
intros st. simpl. lia.
```

Qed.

Theorem `bequiv_example:`

```
bequiv
  <{ X - X = 0 }>
  <{ true }>.
```

Proof.

```
intros st. unfold beval.
rewrite aequiv_example. reflexivity.
```

Qed.

For commands, the situation is a little more subtle. We can’t simply say “two commands are behaviorally equivalent if they evaluate to the same ending state whenever they are started in the same initial state,” because some commands, when run in some starting states, don’t terminate in any final state at all!

What we need instead is this: two commands are behaviorally equivalent if, for any given starting state, they either (1) both diverge or else (2) both terminate in the same final state. A compact way to express this is “if the first one terminates in a particular state then so does the second, and vice versa.”


```

Definition cequiv (c1 c2 : com) : Prop :=
  ∀ (st st' : state),
    (st =[ c1 ] => st') ↔ (st =[ c2 ] => st').

```

4.2.2 Simple Examples

For examples of command equivalence, let's start by looking at a trivial equivalence involving *skip*:

```

Theorem skip_left : ∀ c,
  cequiv
    <{ skip; c }>
    c.

```

Proof.

```

  intros c st st'.
  split; intros H.
  -
    inversion H. subst.
    inversion H2. subst.
    assumption.
  -
    apply E_Seq with st.
    + apply E_Skip.
    + assumption.

```

Qed.

Exercise: 2 stars, standard (skip_right) Prove that adding a *skip* after a command also results in an equivalent program

```

Theorem skip_right : ∀ c,
  cequiv
    <{ c; skip }>
    c.

```

Proof.

Admitted.

□

Similarly, here is a simple equivalence that optimizes *if* commands:

```

Theorem if_true_simple : ∀ c1 c2,
  cequiv
    <{ if true then c1 else c2 end }>
    c1.

```

Proof.

```

  intros c1 c2.
  split; intros H.

```

```

-
  inversion H; subst.
+ assumption.
+ discriminate.
-
  apply E_IfTrue.
+ reflexivity.
+ assumption. Qed.

```

Of course, no programmer would write a conditional whose condition is literally `true`. (At least, no human programmer – compilers and macro preprocessors do this sort of thing internally all the time!) But they might write one whose condition is *equivalent* to `true`:

Theorem: If *b* is equivalent to `true`, then `if b then c1 else c2 end` is equivalent to *c1*.

Proof:

- ($\rightarrow$) We must show, for all *st* and *st'*, that if $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$ then $st = [c1] \Rightarrow st'$.

Proceed by cases on the rules that could possibly have been used to show $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$, namely `E_IfTrue` and `E_IfFalse`.

- Suppose the final rule in the derivation of $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$ was `E_IfTrue`. We then have, by the premises of `E_IfTrue`, that $st = [c1] \Rightarrow st'$. This is exactly what we set out to prove.
- On the other hand, suppose the final rule in the derivation of $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$ was `E_IfFalse`. We then know that $\text{beval } st \ b = \text{false}$ and $st = [c2] \Rightarrow st'$.

Recall that *b* is equivalent to `true`, i.e., for all *st*, $\text{beval } st \ b = \text{beval } st \ <\{\text{true}\}> .$ In particular, this means that $\text{beval } st \ b = \text{true}$, since $\text{beval } st \ <\{\text{true}\}> = \text{true}$. But this is a contradiction, since `E_IfFalse` requires that $\text{beval } st \ b = \text{false}$. Thus, the final rule could not have been `E_IfFalse`.

- ($\leftarrow$) We must show, for all *st* and *st'*, that if $st = [c1] \Rightarrow st'$ then $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$.

Since *b* is equivalent to `true`, we know that $\text{beval } st \ b = \text{beval } st \ <\{\text{true}\}> = \text{true}$. Together with the assumption that $st = [c1] \Rightarrow st'$, we can apply `E_IfTrue` to derive $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$. $\square$

Here is the formal version of this proof:

```

Theorem if_true:  $\forall \ b \ c1 \ c2,$ 
  bequiv b <\{true\}>  $\rightarrow$ 
  cequiv
    <\{ if b then c1 else c2 end \}>
    c1.

```

Proof.

```
intros b c1 c2 Hb.
split; intros H.
-
  inversion H; subst.
  +
    assumption.
  +
    unfold bequiv in Hb. simpl in Hb.
    rewrite Hb in H5.
    discriminate.
-
  apply E_lfTrue; try assumption.
  unfold bequiv in Hb. simpl in Hb.
  apply Hb. Qed.
```

Exercise: 2 stars, standard, especially useful (if_false) Theorem `if_false` : $\forall b\ c1\ c2,$

```
bequiv b <{false}> →
cequiv
  <{ if b then c1 else c2 end }>
  c2.
```

Proof.

Admitted.

□

Exercise: 3 stars, standard (swap_if_branches) Show that we can swap the branches of an `if` if we also negate its condition.

Theorem `swap_if_branches` : $\forall b\ c1\ c2,$

```
cequiv
  <{ if b then c1 else c2 end }>
  <{ if ¬ b then c2 else c1 end }>.
```

Proof.

Admitted.

□

For *while* loops, we can give a similar pair of theorems. A loop whose guard is equivalent to `false` is equivalent to *skip*, while a loop whose guard is equivalent to `true` is equivalent to *while true do skip end* (or any other non-terminating program).

The first of these facts is easy.

Theorem `while_false` : $\forall b\ c,$

```
bequiv b <{false}> →
cequiv
```

```
<{ while b do c end }>
<{ skip }>.
```

Proof.

```
intros b c Hb. split; intros H.
-
  inversion H; subst.
+
  apply E_Skip.
+
  rewrite Hb in H2. discriminate.
-
  inversion H. subst.
  apply E_WhileFalse.
  apply Hb. Qed.
```

Exercise: 2 stars, advanced, optional (while_false_informal) Write an informal proof of `while_false`.

□

To prove the second fact, we need an auxiliary lemma stating that `while` loops whose guards are equivalent to `true` never terminate.

Lemma: If `b` is equivalent to `true`, then it cannot be the case that $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'$.

Proof: Suppose that $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'$. We show, by induction on a derivation of $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'$, that this assumption leads to a contradiction. The only two cases to consider are `E_WhileFalse` and `E_WhileTrue`; the others are contradictory.

- Suppose $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'$ is proved using rule `E_WhileFalse`. Then by assumption `beval st b = false`. But this contradicts the assumption that `b` is equivalent to `true`.
- Suppose $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'$ is proved using rule `E_WhileTrue`. We must have:
 1. `beval st b = true`, and 2. there is some `st0` such that $st = [c] \Rightarrow st0$ and $st0 = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'$. 3. Also, we are given an induction hypothesis saying that $st0 = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'$ leads to a contradiction,

We obtain a contradiction by 2 and 3. □

```
Lemma while_true_nonterm : ∀ b c st st',
  bequiv b <{true}> →
  ~( st = [ while b do c end ] => st' ).
```

Proof.

```
intros b c st st' Hb.
```

```

intros H.
remember <{ while b do c end }> as cw eqn:Hegcw.
induction H;

inversion Hegcw; subst; clear Hegcw.
-
  unfold bequiv in Hb.
  rewrite Hb in H. discriminate.
-
  apply IHceval2. reflexivity. Qed.

```

Exercise: 2 stars, standard, optional (while_true_nonterm_informal) Explain what the lemma `while_true_nonterm` means in English.

□

Exercise: 2 stars, standard, especially useful (while_true) Prove the following theorem. *Hint*: You'll want to use `while_true_nonterm` here.

Theorem `while_true` : $\forall b\ c,$
 bequiv $b\ <\{\text{true}\}> \rightarrow$
 cequiv
 $<\{\text{while } b \text{ do } c \text{ end}\}>$
 $<\{\text{while true do skip end}\}>.$

Proof.

Admitted.

□

A more interesting fact about *while* commands is that any number of copies of the body can be “unrolled” without changing meaning.

Loop unrolling is an important transformation in any real compiler, so its correctness is of more than just academic interest!

Theorem `loop_unrolling` : $\forall b\ c,$
 cequiv
 $<\{\text{while } b \text{ do } c \text{ end}\}>$
 $<\{\text{if } b \text{ then } c ; \text{while } b \text{ do } c \text{ end else skip end}\}>.$

Proof.

```

intros b c st st'.
split; intros Hce.
-
  inversion Hce; subst.
+
  apply E_IfFalse.
  × assumption.
  × apply E_Skip.

```

```

+
  apply E_IfTrue.
  × assumption.
  × apply E_Seq with (st' := st'0).
    - assumption.
    - assumption.
-
  inversion Hce; subst.
+
  inversion H5; subst.
  apply E_WhileTrue with (st' := st'0).
  × assumption.
  × assumption.
  × assumption.
+
  inversion H5; subst. apply E_WhileFalse. assumption. Qed.

```

Exercise: 2 stars, standard, optional (seq_assoc) Theorem `seq_assoc` : $\forall c1\ c2\ c3,$
`cequiv <{(c1; c2); c3}> <{c1; (c2; c3)}>`.

Proof.

Admitted.

□

Proving program properties involving assignments is one place where the fact that we are treating equality on program states extensionally (e.g., $x \mapsto m\ x ; m$ and m are equal maps) comes in handy.

Theorem `identity_assignment` : $\forall x,$

```

cequiv
  <{ x := x }>
  <{ skip }>.

```

Proof.

```

intros.
split; intro H; inversion H; subst; clear H.

```

```

-
  rewrite t_update_same.
  apply E_Skip.
-
  assert (Hx : st' = [ x := x ] => (x !-> st' x ; st')).
  { apply E_Asgn. reflexivity. }
  rewrite t_update_same in Hx.
  apply Hx.

```

Qed.

Exercise: 2 stars, standard, especially useful (assign_aequiv) *Theorem assign_aequiv*

$\vdash \forall (X : \text{string}) (a : \text{aexp}),$
 $\text{aequiv} \langle \{ X \} \rangle a \rightarrow$
 $\text{cequiv} \langle \{ \text{skip} \} \rangle \langle \{ X := a \} \rangle.$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (equiv_classes) Given the following programs, group together those that are equivalent in Imp. Your answer should be given as a list of lists, where each sub-list represents a group of equivalent programs. For example, if you think programs (a) through (h) are all equivalent to each other, but not to (i), your answer should look like this:

[prog-a;prog-b;prog-c;prog-d;prog-e;prog-f;prog-g;prog-h] ; [prog-i]

Write down your answer below in the definition of `equiv_classes`.

Definition prog_a : **com** :=

```
<{ while X > 0 do
  X := X + 1
end }>.
```

Definition prog_b : **com** :=

```
<{ if (X = 0) then
  X := X + 1;
  Y := 1
else
  Y := 0
end;
X := X - Y;
Y := 0 }>.
```

Definition prog_c : **com** :=

```
<{ skip }> .
```

Definition prog_d : **com** :=

```
<{ while X ≠ 0 do
  X := (X × Y) + 1
end }>.
```

Definition prog_e : **com** :=

```
<{ Y := 0 }>.
```

Definition prog_f : **com** :=

```
<{ Y := X + 1;
  while X ≠ Y do
    Y := X + 1
  end }>.
```

```

Definition prog_g : com :=
  <{ while true do
    skip
  end }>.

Definition prog_h : com :=
  <{ while X ≠ X do
    X := X + 1
  end }>.

Definition prog_i : com :=
  <{ while X ≠ Y do
    X := Y + 1
  end }>.

Definition equiv_classes : list (list com)
. Admitted.

Definition manual_grade_for_equiv_classes : option (nat × string) := None.
□

```

4.3 Properties of Behavioral Equivalence

We next consider some fundamental properties of program equivalence.

4.3.1 Behavioral Equivalence is an Equivalence

First, let's verify that the equivalences on *aexprs*, *bexprs*, and *coms* really are *equivalences* – i.e., that they are reflexive, symmetric, and transitive. The proofs are all easy.

```

Lemma refl_aequiv : ∀ (a : aexp),
  aequiv a a.

```

Proof.

```

  intros a st. reflexivity. Qed.

```

```

Lemma sym_aequiv : ∀ (a1 a2 : aexp),
  aequiv a1 a2 → aequiv a2 a1.

```

Proof.

```

  intros a1 a2 H. intros st. symmetry. apply H. Qed.

```

```

Lemma trans_aequiv : ∀ (a1 a2 a3 : aexp),
  aequiv a1 a2 → aequiv a2 a3 → aequiv a1 a3.

```

Proof.

```

  unfold aequiv. intros a1 a2 a3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.

```

```

Lemma refl_bequiv : ∀ (b : bexp),
  bequiv b b.

```


Proof.
 unfold bequiv. intros *b st*. reflexivity. Qed.

Lemma sym_bequiv : $\forall (b1\ b2 : \mathbf{bexp})$,
 bequiv *b1 b2* $\rightarrow$ bequiv *b2 b1*.
 Proof.
 unfold bequiv. intros *b1 b2 H*. intros *st*. symmetry. apply *H*. Qed.

Lemma trans_bequiv : $\forall (b1\ b2\ b3 : \mathbf{bexp})$,
 bequiv *b1 b2* $\rightarrow$ bequiv *b2 b3* $\rightarrow$ bequiv *b1 b3*.
 Proof.
 unfold bequiv. intros *b1 b2 b3 H12 H23 st*.
 rewrite (*H12 st*). rewrite (*H23 st*). reflexivity. Qed.

Lemma refl_cequiv : $\forall (c : \mathbf{com})$,
 cequiv *c c*.
 Proof.
 unfold cequiv. intros *c st st'*. reflexivity. Qed.

Lemma sym_cequiv : $\forall (c1\ c2 : \mathbf{com})$,
 cequiv *c1 c2* $\rightarrow$ cequiv *c2 c1*.
 Proof.
 unfold cequiv. intros *c1 c2 H st st'*.
 rewrite *H*. reflexivity.
 Qed.

Lemma trans_cequiv : $\forall (c1\ c2\ c3 : \mathbf{com})$,
 cequiv *c1 c2* $\rightarrow$ cequiv *c2 c3* $\rightarrow$ cequiv *c1 c3*.
 Proof.
 unfold cequiv. intros *c1 c2 c3 H12 H23 st st'*.
 rewrite *H12*. apply *H23*.
 Qed.

4.3.2 Behavioral Equivalence Is a Congruence

Less obviously, behavioral equivalence is also a *congruence*. That is, the equivalence of two subprograms implies the equivalence of the larger programs in which they are embedded:

aequiv *a a'*

cequiv (*x := a*) (*x := a'*)
 cequiv *c1 c1'* cequiv *c2 c2'*

cequiv (*c1;c2*) (*c1';c2'*)
 ... and so on for the other forms of commands.

(Note that we are using the inference rule notation here not as part of an inductive definition, but simply to write down some valid implications in a readable format. We prove

these implications below.)

We will see a concrete example of why these congruence properties are important in the following section (in the proof of `fold_constants_com_sound`), but the main idea is that they allow us to replace a small part of a large program with an equivalent small part and know that the whole large programs are equivalent *without* doing an explicit proof about the parts that didn't change – i.e., the “proof burden” of a small change to a large program is proportional to the size of the change, not the program!

Theorem `CAsgn_congruence` : $\forall x\ a\ a'$,
`aequiv a a' →`
`cequiv <{x := a}> <{x := a'}>`.

Proof.

```
intros x a a' Heqv st st'.
split; intros Hceval.
-
  inversion Hceval. subst. apply E_Asgn.
  rewrite Heqv. reflexivity.
-
  inversion Hceval. subst. apply E_Asgn.
  rewrite Heqv. reflexivity. Qed.
```

The congruence property for loops is a little more interesting, since it requires induction.

Theorem: Equivalence is a congruence for `while` – that is, if `b` is equivalent to `b'` and `c` is equivalent to `c'`, then `while b do c end` is equivalent to `while b' do c' end`.

Proof: Suppose `b` is equivalent to `b'` and `c` is equivalent to `c'`. We must show, for every `st` and `st'`, that `st = [ while b do c end ] => st'` iff `st = [ while b' do c' end ] => st'`. We consider the two directions separately.

- ($\rightarrow$) We show that `st = [ while b do c end ] => st'` implies `st = [ while b' do c' end ] => st'`, by induction on a derivation of `st = [ while b do c end ] => st'`. The only nontrivial cases are when the final rule in the derivation is `E_WhileFalse` or `E_WhileTrue`.
 - `E_WhileFalse`: In this case, the form of the rule gives us `beval st b = false` and `st = st'`. But then, since `b` and `b'` are equivalent, we have `beval st b' = false`, and `E_WhileFalse` applies, giving us `st = [ while b' do c' end ] => st'`, as required.
 - `E_WhileTrue`: The form of the rule now gives us `beval st b = true`, with `st = [ c ] => st'0` and `st'0 = [ while b do c end ] => st'` for some state `st'0`, with the induction hypothesis `st'0 = [ while b' do c' end ] => st'`.
 Since `c` and `c'` are equivalent, we know that `st = [ c' ] => st'0`. And since `b` and `b'` are equivalent, we have `beval st b' = true`. Now `E_WhileTrue` applies, giving us `st = [ while b' do c' end ] => st'`, as required.
- ($\leftarrow$) Similar. $\square$

Theorem CWhile_congruence : $\forall b\ b'\ c\ c',$
 $\text{bequiv } b\ b' \rightarrow \text{cequiv } c\ c' \rightarrow$
 $\text{cequiv } \langle \{ \text{while } b \text{ do } c \text{ end} \} \rangle \langle \{ \text{while } b' \text{ do } c' \text{ end} \} \rangle.$

Proof.

```

assert (A:  $\forall (b\ b' : \text{bexp}) (c\ c' : \text{com}) (st\ st' : \text{state}),$ 
        bequiv  $b\ b' \rightarrow \text{cequiv } c\ c' \rightarrow$ 
         $st = [ \text{while } b \text{ do } c \text{ end} ] \Rightarrow st' \rightarrow$ 
         $st = [ \text{while } b' \text{ do } c' \text{ end} ] \Rightarrow st').$ 
{ unfold bequiv,cequiv.
  intros b b' c c' st st' Hbe Hc1e Hce.
  remember  $\langle \{ \text{while } b \text{ do } c \text{ end} \} \rangle$  as cwhile
    eqn:Hegcwhile.
  induction Hce; inversion Hegcwhile; subst.
+
  apply E_WhileFalse. rewrite  $\leftarrow Hbe$ . apply H.
+
  apply E_WhileTrue with ( $st' := st'$ ).
   $\times$  rewrite  $\leftarrow Hbe$ . apply H.
   $\times$ 
    apply (Hc1e st st'). apply Hce1.
   $\times$ 
    apply IHHce2. reflexivity. }

intros. split.
- apply A; assumption.
- apply A.
  + apply sym_bequiv. assumption.
  + apply sym_cequiv. assumption.

```

Qed.

Exercise: 3 stars, standard, optional (CSeq_congruence) Theorem CSeq_congruence

: $\forall c1\ c1'\ c2\ c2',$
 $\text{cequiv } c1\ c1' \rightarrow \text{cequiv } c2\ c2' \rightarrow$
 $\text{cequiv } \langle \{ c1 ; c2 \} \rangle \langle \{ c1' ; c2' \} \rangle.$

Proof.

Admitted.

□

Exercise: 3 stars, standard (CIf_congruence) Theorem CIf_congruence : $\forall b\ b'\ c1$

$c1'\ c2\ c2',$
 $\text{bequiv } b\ b' \rightarrow \text{cequiv } c1\ c1' \rightarrow \text{cequiv } c2\ c2' \rightarrow$
 $\text{cequiv } \langle \{ \text{if } b \text{ then } c1 \text{ else } c2 \text{ end} \} \rangle$

<{ if b' then $c1'$ else $c2'$ end }>.

Proof.

Admitted.

□

For example, here are two programs and a proof of their equivalence using these congruence theorems...

Example congruence_example:
cequiv

```
<{ X := 0;
  if X = 0 then Y := 0
  else Y := 42 end }>
```

```
<{ X := 0;
  if X = 0 then Y := X - X
  else Y := 42 end }>.
```

Proof.

```
apply CSeq_congruence.
- apply refl_cequiv.
- apply CIf_congruence.
  + apply refl_bequiv.
  + apply CAsgn_congruence. unfold aequiv. simpl.
    symmetry. apply sub_diag.
  + apply refl_cequiv.
```

Qed.

Exercise: 3 stars, advanced (not_congr) We've shown that the `cequiv` relation is both an equivalence and a congruence on commands. Can you think of a relation on commands that is an equivalence but *not* a congruence? Write down the relation (formally), together with an informal sketch of a proof that it is an equivalence and a counterexample showing it is not a congruence.

Definition manual_grade_for_not_congr : option (nat × string) := None.

□

4.4 Program Transformations

A *program transformation* is a function that takes a program as input and produces a modified program as output. Compiler optimizations such as constant folding are canonical examples, but there are many others.

A program transformation is *sound* if it preserves the behavior of the original program.

Definition atrans_sound (atrans : aexp → aexp) : Prop :=

```

  ∀ (a : aexp),
    aequiv a (atrans a).
Definition btrans_sound (btrans : bexp → bexp) : Prop :=
  ∀ (b : bexp),
    bequiv b (btrans b).
Definition ctrans_sound (ctrans : com → com) : Prop :=
  ∀ (c : com),
    cequiv c (ctrans c).

```

4.4.1 The Constant-Folding Transformation

An expression is *constant* if it contains no variable references.

Constant folding is an optimization that finds constant expressions and replaces them by their values.

```

Fixpoint fold_constants_aexp (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | Ald x ⇒ Ald x
  | <{ a1 + a2 }> ⇒
    match (fold_constants_aexp a1 ,
           fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
    | (a1', a2') ⇒ <{ a1' + a2' }>
    end
  | <{ a1 - a2 }> ⇒
    match (fold_constants_aexp a1 ,
           fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
    | (a1', a2') ⇒ <{ a1' - a2' }>
    end
  | <{ a1 × a2 }> ⇒
    match (fold_constants_aexp a1 ,
           fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 × n2)
    | (a1', a2') ⇒ <{ a1' × a2' }>
    end
  end.

```

```

Example fold_aexp_ex1 :
  fold_constants_aexp <{ (1 + 2) × X }>

```

= <{ 3 × X }>.

Proof. reflexivity. Qed.

Note that this version of constant folding doesn't do other "obvious" things like eliminating trivial additions (e.g., rewriting $0 + X$ to just X): we are focusing on a single optimization for the sake of simplicity.

It is not hard to incorporate other ways of simplifying expressions – the definitions and proofs just get longer. We'll consider some in the exercises.

Example fold_aexp_ex2 :

fold_constants_aexp <{ X - ((0 × 6) + Y) }> = <{ X - (0 + Y) }>.

Proof. reflexivity. Qed.

Not only can we lift fold_constants_aexp to bexps (in the BEq, BNeq, and BLe cases); we can also look for constant *boolean* expressions and evaluate them in-place as well.

Fixpoint fold_constants_bexp (b : bexp) : bexp :=

```
match b with
| <{true}> => <{true}>
| <{false}> => <{false}>
| <{ a1 = a2 }> =>
  match (fold_constants_aexp a1,
        fold_constants_aexp a2) with
  | (ANum n1, ANum n2) =>
    if n1 =? n2 then <{true}> else <{false}>
  | (a1', a2') =>
    <{ a1' = a2' }>
  end
| <{ a1 ≠ a2 }> =>
  match (fold_constants_aexp a1,
        fold_constants_aexp a2) with
  | (ANum n1, ANum n2) =>
    if negb (n1 =? n2) then <{true}> else <{false}>
  | (a1', a2') =>
    <{ a1' ≠ a2' }>
  end
| <{ a1 ≤ a2 }> =>
  match (fold_constants_aexp a1,
        fold_constants_aexp a2) with
  | (ANum n1, ANum n2) =>
    if n1 <=? n2 then <{true}> else <{false}>
  | (a1', a2') =>
    <{ a1' ≤ a2' }>
  end
| <{ a1 > a2 }> =>
```

```

match (fold_constants_aexp a1,
      fold_constants_aexp a2) with
| (ANum n1, ANum n2) ⇒
  if n1 <=? n2 then <{false}> else <{true}>
| (a1', a2') ⇒
  <{ a1' > a2' }>
end
| <{ ¬ b1 }> ⇒
  match (fold_constants_bexp b1) with
  | <{true}> ⇒ <{false}>
  | <{false}> ⇒ <{true}>
  | b1' ⇒ <{ ¬ b1' }>
  end
| <{ b1 && b2 }> ⇒
  match (fold_constants_bexp b1,
        fold_constants_bexp b2) with
  | (<{true}>, <{true}>) ⇒ <{true}>
  | (<{true}>, <{false}>) ⇒ <{false}>
  | (<{false}>, <{true}>) ⇒ <{false}>
  | (<{false}>, <{false}>) ⇒ <{false}>
  | (b1', b2') ⇒ <{ b1' && b2' }>
  end
end.

```

Example fold_bexp_ex1 :

```

fold_constants_bexp <{ true && ¬(false && true) }>
= <{ true }>.

```

Proof. reflexivity. Qed.

Example fold_bexp_ex2 :

```

fold_constants_bexp <{ (X = Y) && (0 = (2 - (1 + 1))) }>
= <{ (X = Y) && true }>.

```

Proof. reflexivity. Qed.

To fold constants in a command, we simply apply the appropriate folding functions on all embedded expressions.

```

Fixpoint fold_constants_com (c : com) : com :=
  match c with
  | <{ skip }> ⇒
    <{ skip }>
  | <{ x := a }> ⇒
    <{ x := (fold_constants_aexp a) }>
  | <{ c1 ; c2 }> ⇒
    <{ fold_constants_com c1 ; fold_constants_com c2 }>
  end

```

```

| <{ if b then c1 else c2 end }> ⇒
  match fold_constants_bexp b with
  | <{true}> ⇒ fold_constants_com c1
  | <{false}> ⇒ fold_constants_com c2
  | b' ⇒ <{ if b' then fold_constants_com c1
              else fold_constants_com c2 end }>
  end
| <{ while b do c1 end }> ⇒
  match fold_constants_bexp b with
  | <{true}> ⇒ <{ while true do skip end }>
  | <{false}> ⇒ <{ skip }>
  | b' ⇒ <{ while b' do (fold_constants_com c1) end }>
  end
end.

```

Example fold_com_ex1 :
 fold_constants_com

```

<{ X := 4 + 5;
  Y := X - 3;
  if (X - Y) = (2 + 4) then skip
  else Y := 0 end;
  if 0 ≤ (4 - (2 + 1)) then Y := 0
  else skip end;
  while Y = 0 do
    X := X + 1
  end }>

=

<{ X := 9;
  Y := X - 3;
  if (X - Y) = 6 then skip
  else Y := 0 end;
  Y := 0;
  while Y = 0 do
    X := X + 1
  end }>.

```

Proof. reflexivity. Qed.

4.4.2 Soundness of Constant Folding

Now we need to show that what we've done is correct.

Here's the proof for arithmetic expressions:

Theorem fold_constants_aexp_sound :


```

    atrans_sound fold_constants_aexp.
Proof.
  unfold atrans_sound. intros a. unfold aequiv. intros st.
  induction a; simpl;

  try reflexivity;

  try (destruct (fold_constants_aexp a1);
        destruct (fold_constants_aexp a2);
        rewrite IHa1; rewrite IHa2; reflexivity). Qed.

```

Exercise: 3 stars, standard, optional (fold_bexp_Eq_informal) Here is an informal proof of the **BEq** case of the soundness argument for boolean expression constant folding. Read it carefully and compare it to the formal proof that follows. Then fill in the **BLE** case of the formal proof (without looking at the **BEq** case, if possible).

Theorem: The constant folding function for booleans, `fold_constants_bexp`, is sound.

Proof: We must show that b is equivalent to `fold_constants_bexp b`, for all boolean expressions b . Proceed by induction on b . We show just the case where b has the form $a1 = a2$.

In this case, we must show

$\text{beval st } \langle \{ a1 = a2 \} \rangle = \text{beval st } (\text{fold\_constants\_bexp } \langle \{ a1 = a2 \} \rangle).$

There are two cases to consider:

- First, suppose `fold_constants_aexp a1` = `ANum n1` and `fold_constants_aexp a2` = `ANum n2` for some $n1$ and $n2$.

In this case, we have

$\text{fold\_constants\_bexp } \langle \{ a1 = a2 \} \rangle = \text{if } n1 =? n2 \text{ then } \langle \{ \text{true} \} \rangle \text{ else } \langle \{ \text{false} \} \rangle$

and

$\text{beval st } \langle \{ a1 = a2 \} \rangle = (\text{aeval st } a1) =? (\text{aeval st } a2).$

By the soundness of constant folding for arithmetic expressions (Lemma `fold_constants_aexp_sound`), we know

$\text{aeval st } a1 = \text{aeval st } (\text{fold\_constants\_aexp } a1) = \text{aeval st } (\text{ANum } n1) = n1$

and

$\text{aeval st } a2 = \text{aeval st } (\text{fold\_constants\_aexp } a2) = \text{aeval st } (\text{ANum } n2) = n2,$

so

$\text{beval st } \langle \{ a1 = a2 \} \rangle = (\text{aeval } a1) =? (\text{aeval } a2) = n1 =? n2.$

Also, it is easy to see (by considering the cases $n1 = n2$ and $n1 \neq n2$ separately) that

$\text{beval st } (\text{if } n1 =? n2 \text{ then } \langle \{ \text{true} \} \rangle \text{ else } \langle \{ \text{false} \} \rangle) = \text{if } n1 =? n2 \text{ then beval st } \langle \{ \text{true} \} \rangle \text{ else beval st } \langle \{ \text{false} \} \rangle = \text{if } n1 =? n2 \text{ then true else false} = n1 =? n2.$

So

$\text{beval st } (<\{ a1 = a2 \}>) = n1 =? n2. = \text{beval st } (\text{if } n1 =? n2 \text{ then } <\{\text{true}\}> \text{ else } <\{\text{false}\}>),$

as required.

- Otherwise, one of `fold_constants_aexp a1` and `fold_constants_aexp a2` is not a constant. In this case, we must show

$\text{beval st } <\{a1 = a2\}> = \text{beval st } (<\{ (\text{fold\_constants\_aexp } a1) = (\text{fold\_constants\_aexp } a2) \}>),$

which, by the definition of `beval`, is the same as showing

$(\text{aeval st } a1) =? (\text{aeval st } a2) = (\text{aeval st } (\text{fold\_constants\_aexp } a1)) =? (\text{aeval st } (\text{fold\_constants\_aexp } a2)).$

But the soundness of constant folding for arithmetic expressions (`fold_constants_aexp_sound`) gives us

$\text{aeval st } a1 = \text{aeval st } (\text{fold\_constants\_aexp } a1) \text{ aeval st } a2 = \text{aeval st } (\text{fold\_constants\_aexp } a2),$

completing the case. $\square$

Theorem `fold_constants_bexp_sound`:

`btrans_sound fold_constants_bexp`.

Proof.

```
unfold btrans_sound. intros b. unfold bequiv. intros st.
induction b;
```

```
  try reflexivity.
```

```
-
```

```
  simpl.
```

```
  remember (fold_constants_aexp a1) as a1' eqn:Hega1'.
```

```
  remember (fold_constants_aexp a2) as a2' eqn:Hega2'.
```

```
  replace (aeval st a1) with (aeval st a1') by
```

```
    (subst a1'; rewrite ← fold_constants_aexp_sound; reflexivity).
```

```
  replace (aeval st a2) with (aeval st a2') by
```

```
    (subst a2'; rewrite ← fold_constants_aexp_sound; reflexivity).
```

```
  destruct a1'; destruct a2'; try reflexivity.
```

```
    simpl. destruct (n =? n0); reflexivity.
```

```
-
```

```
  simpl.
```

```
  remember (fold_constants_aexp a1) as a1' eqn:Hega1'.
```

```
  remember (fold_constants_aexp a2) as a2' eqn:Hega2'.
```

```
  replace (aeval st a1) with (aeval st a1') by
```

```

      (subst a1'; rewrite ← fold_constants_aexp_sound; reflexivity).
    replace (aeval st a2) with (aeval st a2') by
      (subst a2'; rewrite ← fold_constants_aexp_sound; reflexivity).
    destruct a1'; destruct a2'; try reflexivity.
    simpl. destruct (n =? n0); reflexivity.
  -
    admit.
  -
    admit.
  -
    simpl. remember (fold_constants_bexp b) as b' eqn:Heqb'.
    rewrite IHb.
    destruct b'; reflexivity.
  -
    simpl.
    remember (fold_constants_bexp b1) as b1' eqn:Heqb1'.
    remember (fold_constants_bexp b2) as b2' eqn:Heqb2'.
    rewrite IHb1. rewrite IHb2.
    destruct b1'; destruct b2'; reflexivity.
Admitted.
□

```

Exercise: 3 stars, standard (fold_constants_com_sound) Complete the *while* case of the following proof.

Theorem fold_constants_com_sound :
 ctrans_sound fold_constants_com.

Proof.

```

  unfold ctrans_sound. intros c.
  induction c; simpl.
  - apply refl_cequiv.
  - apply CAsgn_congruence.
    apply fold_constants_aexp_sound.
  - apply CSeq_congruence; assumption.
  -
    assert (bequiv b (fold_constants_bexp b)). {
      apply fold_constants_bexp_sound. }
    destruct (fold_constants_bexp b) eqn:Heqb;
    try (apply CIf_congruence; assumption).
  +
    apply trans_cequiv with c1; try assumption.
    apply if_true; assumption.
  +

```

apply trans_cequiv with *c2*; try assumption.
 apply if_false; assumption.

Admitted.

□

4.4.3 Soundness of $(0 + n)$ Elimination, Redux

Exercise: 4 stars, standard, optional (optimize_0plus_var) Recall the definition `optimize_0plus` from the `Imp` chapter of *Logical Foundations*:

Fixpoint `optimize_0plus` (a:aexp) : aexp := match a with | ANum n => ANum n | <{ 0 + a2 }> => optimize_0plus a2 | <{ a1 + a2 }> => <{ (optimize_0plus a1) + (optimize_0plus a2) }> | <{ a1 - a2 }> => <{ (optimize_0plus a1) - (optimize_0plus a2) }> | <{ a1 * a2 }> => <{ (optimize_0plus a1) * (optimize_0plus a2) }> end.

Note that this function is defined over the old version of `aexps`, without states.

Write a new version of this function that deals with variables (by leaving them alone), plus analogous ones for `bexps` and commands:

`optimize_0plus_aexp` `optimize_0plus_bexp` `optimize_0plus_com`

Fixpoint `optimize_0plus_aexp` (a : aexp) : aexp

. *Admitted.*

Fixpoint `optimize_0plus_bexp` (b : bexp) : bexp

. *Admitted.*

Fixpoint `optimize_0plus_com` (c : com) : com

. *Admitted.*

Example `test_optimize_0plus`:

`optimize_0plus_com`

<{ while X ≠ 0 do X := 0 + X - 1 end }>

= <{ while X ≠ 0 do X := X - 1 end }>.

Proof.

Admitted.

Prove that these three functions are sound, as we did for `fold_constants_×`. Make sure you use the congruence lemmas in the proof of `optimize_0plus_com` – otherwise it will be *long*!

Theorem `optimize_0plus_aexp_sound`:

`atrans_sound` `optimize_0plus_aexp`.

Proof.

Admitted.

Theorem `optimize_0plus_bexp_sound` :

`btrans_sound` `optimize_0plus_bexp`.

Proof.

Admitted.

Theorem `optimize_0plus_com_sound` :
`ctrans_sound optimize_0plus_com`.

Proof.

Admitted.

Finally, let's define a compound optimizer on commands that first folds constants (using `fold_constants_com`) and then eliminates `0 + n` terms (using `optimize_0plus_com`).

Definition `optimizer (c : com) := optimize_0plus_com (fold_constants_com c)`.

Prove that this optimizer is sound.

Theorem `optimizer_sound` :
`ctrans_sound optimizer`.

Proof.

Admitted.

□

4.5 Proving Inequivalence

Next, let's look at some programs that are *not* equivalent.

Suppose that *c1* is a command of the form

`X := a1; Y := a2`

and *c2* is the command

`X := a1; Y := a2'`

where *a2'* is formed by substituting *a1* for all occurrences of *X* in *a2*.

For example, *c1* and *c2* might be:

`c1 = (X := 42 + 53; Y := Y + X)` `c2 = (X := 42 + 53; Y := Y + (42 + 53))`

Clearly, this *particular* *c1* and *c2* are equivalent. Is this true in general?

We will see in a moment that it is not, but it is worthwhile to pause, now, and see if you can find a counter-example on your own.

More formally, here is the function that substitutes an arithmetic expression *u* for each occurrence of a given variable *x* in another expression *a*:

```
Fixpoint subst_aexp (x : string) (u : aexp) (a : aexp) : aexp :=
  match a with
  | ANum n =>
    ANum n
  | Ald x' =>
    if String.eqb x x' then u else Ald x'
  | <{ a1 + a2 }> =>
    <{ (subst_aexp x u a1) + (subst_aexp x u a2) }>
  | <{ a1 - a2 }> =>
    <{ (subst_aexp x u a1) - (subst_aexp x u a2) }>
  | <{ a1 × a2 }> =>
```

```

    <{ (subst_aexp x u a1) × (subst_aexp x u a2) }>
end.

```

```

Example subst_aexp_ex :
  subst_aexp X <{42 + 53}> <{Y + X}>
= <{ Y + (42 + 53) }>.

```

Proof. simpl. reflexivity. Qed.

And here is the property we are interested in, expressing the claim that commands *c1* and *c2* as described above are always equivalent.

```

Definition subst_equiv_property : Prop := ∀ x1 x2 a1 a2,
  cequiv <{ x1 := a1; x2 := a2 }>
    <{ x1 := a1; x2 := subst_aexp x1 a1 a2 }>.

```

Sadly, the property does *not* always hold.

Here is a counterexample:

X := X + 1; Y := X

If we perform the substitution, we get

X := X + 1; Y := X + 1

which clearly isn't equivalent.

```

Theorem subst_inequiv :
  ¬ subst_equiv_property.

```

Proof.

```

  unfold subst_equiv_property.
  intros Contra.

```

```

  remember <{ X := X + 1;
              Y := X }>

```

```

    as c1.

```

```

  remember <{ X := X + 1;
              Y := X + 1 }>

```

```

    as c2.

```

```

  assert (cequiv c1 c2) by (subst; apply Contra).

```

```

  clear Contra.

```

```

  remember (Y !-> 1 ; X !-> 1) as st1.

```

```

  remember (Y !-> 2 ; X !-> 1) as st2.

```

```

  assert (H1 : empty_st = [ c1 ] => st1);

```

```

  assert (H2 : empty_st = [ c2 ] => st2);

```

```

  try (subst;

```

```

    apply E_Seq with (st' := (X !-> 1));

```

```

    apply E_Asgn; reflexivity).

```

```

  clear Heqc1 Heqc2.

```

```

  apply H in H1.

```

```

  clear H.

```

```

assert (Hcontra : st1 = st2)
  by (apply (ceval_deterministic c2 empty_st); assumption).
clear H1 H2.

assert (Hcontra' : st1 Y = st2 Y)
  by (rewrite Hcontra; reflexivity).
subst. discriminate. Qed.

```

Exercise: 4 stars, standard, optional (better_subst_equiv) The equivalence we had in mind above was not complete nonsense – in fact, it was actually almost right. To make it correct, we just need to exclude the case where the variable **X** occurs in the right-hand side of the first assignment statement.

```

Inductive var_not_used_in_aexp (x : string) : aexp → Prop :=
| VNUNum : ∀ n, var_not_used_in_aexp x (ANum n)
| VNUIId : ∀ y, x ≠ y → var_not_used_in_aexp x (Ald y)
| VNUPlus : ∀ a1 a2,
  var_not_used_in_aexp x a1 →
  var_not_used_in_aexp x a2 →
  var_not_used_in_aexp x (<{ a1 + a2 }>)
| VNUMinus : ∀ a1 a2,
  var_not_used_in_aexp x a1 →
  var_not_used_in_aexp x a2 →
  var_not_used_in_aexp x (<{ a1 - a2 }>)
| VNUMult : ∀ a1 a2,
  var_not_used_in_aexp x a1 →
  var_not_used_in_aexp x a2 →
  var_not_used_in_aexp x (<{ a1 × a2 }>).

```

```

Lemma aeval_weakening : ∀ x st a ni,
  var_not_used_in_aexp x a →
  aeval (x !-> ni ; st) a = aeval st a.

```

Proof.

Admitted.

Using **var_not_used_in_aexp**, formalize and prove a correct version of **subst_equiv_property**.

Exercise: 3 stars, standard (inequiv_exercise) Prove that an infinite loop is not equivalent to *skip*

Theorem inequiv_exercise:

```

¬ cequiv <{ while true do skip end }> <{ skip }>.

```

Proof.

Admitted.

□

4.6 Extended Exercise: Nondeterministic Imp

As we have seen (in theorem `ceval_deterministic` in the `Imp` chapter), `Imp`'s evaluation relation is deterministic. However, *non*-determinism is an important part of the definition of many real programming languages. For example, in many imperative languages (such as C and its relatives), the order in which function arguments are evaluated is unspecified: the program fragment

```
x = 0; f(++x, x)
```

might call `f` with arguments $(1, 0)$ or $(1, 1)$, depending how the compiler chooses to order things. This can be a little confusing for programmers, but it gives compiler writers useful freedom.

In this exercise, we will extend `Imp` with a simple nondeterministic command and study how this change affects program equivalence. The new command has the syntax *HAVOC* `X`, where `X` is an identifier. The effect of executing *HAVOC* `X` is to assign an *arbitrary* number to the variable `X`, nondeterministically. For example, after executing the program:

```
HAVOC Y; Z := Y * 2
```

the value of `Y` can be any number, while the value of `Z` is twice that of `Y` (so `Z` is always even). Note that we are not saying anything about the *probabilities* of the outcomes – just that there are (infinitely) many different outcomes that can possibly happen after executing this nondeterministic code.

In a sense, a variable on which we do *HAVOC* roughly corresponds to an uninitialized variable in a low-level language like C. After the *HAVOC*, the variable holds a fixed but arbitrary number. Most sources of nondeterminism in language definitions are there precisely because programmers don't care which choice is made (and so it is good to leave it open to the compiler to choose whichever will run faster).

We call this new language *Himp* (“`Imp` extended with *HAVOC*”).

Module `HIMP`.

To formalize *Himp*, we first add a clause to the definition of commands.

```
Inductive com : Type :=
| CSkip : com
| CAsgn : string → aexp → com
| CSeq : com → com → com
| Clf : bexp → com → com → com
| CWhile : bexp → com → com
| CHavoc : string → com.
```

```
Notation "'havoc' l" := (CHavoc l)
(in custom com at level 60, l constr at level 0).
```

```
Notation "'skip'" := CSkip
(in custom com at level 0) : com_scope.
```

```
Notation "x := y" := (CAsgn x y)
(in custom com at level 0, x constr at level 0, y at level 85, no associativity,
format "x := y") : com_scope.
```


Notation "x ; y" := (CSeq x y)
 (in custom com at level 90,
 right associativity,
 format "[v' x ; '/' y ']'") : com_scope.

Notation "if x 'then' y 'else' z 'end'" := (CIf x y z)
 (in custom com at level 89, x at level 99, y at level 99, z at level 99,
 format "[v' 'if' x 'then' '/' y '/' 'else' '/' z '/' 'end' ']'") : com_scope.

Notation "while x 'do' y 'end'" := (CWhile x y)
 (in custom com at level 89, x at level 99, y at level 99,
 format "[v' 'while' x 'do' '/' y '/' 'end' ']'") : com_scope.

Exercise: 2 stars, standard (himp_ceval) Now, we must extend the operational semantics. We have provided a template for the **ceval** relation below, specifying the big-step semantics. What rule(s) must be added to the definition of **ceval** to formalize the behavior of the **HAVOC** command?

Reserved Notation

"st0 '[c]=> st1"
 (at level 40, c custom com at level 99,
 st0 constr, st1 constr at next level,
 format "[hv' st0 =['/' '[c]' '/']=> st1 ']'").

Inductive **ceval** : com → state → state → Prop :=

| E_Skip : ∀ st,
 st = [skip] => st

| E_Asgn : ∀ st a n x,
 aeval st a = n →
 st = [x := a] => (x !-> n ; st)

| E_Seq : ∀ c1 c2 st st' st'',
 st = [c1] => st' →
 st' = [c2] => st'' →
 st = [c1 ; c2] => st''

| E_IfTrue : ∀ st st' b c1 c2,
 beval st b = true →
 st = [c1] => st' →
 st = [if b then c1 else c2 end] => st'

| E_IfFalse : ∀ st st' b c1 c2,
 beval st b = false →
 st = [c2] => st' →
 st = [if b then c1 else c2 end] => st'

| E_WhileFalse : ∀ b st c,
 beval st b = false →
 st = [while b do c end] => st

| E_WhileTrue : ∀ st st' st'' b c,

```

beval st b = true →
st = [ c ] => st' →
st' = [ while b do c end ] => st'' →
st = [ while b do c end ] => st''

```

where "st = [c] => st'" := (ceval c st st').

As a sanity check, the following claims should be provable for your definition:

Example havoc_example1 : empty_st = [havoc X] => (X !-> 0).

Proof.

Admitted.

Example havoc_example2 :

empty_st = [skip; havoc Z] => (Z !-> 42).

Proof.

Admitted.

Definition manual_grade_for_Check_rule_for_HAVOC : option (nat × string) := None.

□

Finally, we repeat the definition of command equivalence from above:

Definition cequiv (c1 c2 : com) : Prop := ∀ st st' : state,
st = [c1] => st' ↔ st = [c2] => st'.

Let's apply this definition to prove some nondeterministic programs equivalent / inequivalent.

Exercise: 3 stars, standard (havoc_swap) Are the following two programs equivalent?

Definition pXY :=
<{ havoc X ; havoc Y }>.

Definition pYX :=
<{ havoc Y ; havoc X }>.

If you think they are equivalent, prove it. If you think they are not, prove that.

Theorem pXY_cequiv_pYX :
cequiv pXY pYX ∨ ¬cequiv pXY pYX.

Proof.

Admitted.

□

Exercise: 4 stars, standard, optional (havoc_copy) Are the following two programs equivalent?

Definition ptwice :=

<{ havoc X; havoc Y }>.

Definition pcopy :=
 <{ havoc X; Y := X }>.

If you think they are equivalent, then prove it. If you think they are not, then prove that. (Hint: You may find the `assert` tactic useful.)

Theorem ptwice_cequiv_pcopy :
 cequiv ptwice pcopy $\vee$ $\neg$ cequiv ptwice pcopy.

Proof. *Admitted*.

□

The definition of program equivalence we are using here has some subtle consequences on programs that may loop forever. What `cequiv` says is that the set of possible *terminating* outcomes of two equivalent programs is the same. However, in a language with nondeterminism, like Himp, some programs always terminate, some programs always diverge, and some programs can nondeterministically terminate in some runs and diverge in others. The final part of the following exercise illustrates this phenomenon.

Exercise: 4 stars, advanced (p1_p2_term) Consider the following commands:

Definition p1 : com :=
 <{ while $\neg$ (X = 0) do
 havoc Y;
 X := X + 1
 end }>.

Definition p2 : com :=
 <{ while $\neg$ (X = 0) do
 skip
 end }>.

Intuitively, `p1` and `p2` have the same termination behavior: either they loop forever, or they terminate in the same state they started in. We can capture the termination behavior of `p1` and `p2` individually with these lemmas:

Lemma p1_may_diverge : $\forall$ st st', st X $\neq$ 0 $\rightarrow$
 $\neg$ st = [p1] => st'.

Proof. *Admitted*.

Lemma p2_may_diverge : $\forall$ st st', st X $\neq$ 0 $\rightarrow$
 $\neg$ st = [p2] => st'.

Proof.

Admitted.

□

Exercise: 4 stars, advanced (p1_p2_equiv) Use these two lemmas to prove that `p1` and `p2` are actually equivalent.

Theorem `p1_p2_equiv` : `cequiv p1 p2`.

Proof. *Admitted*.

□

Exercise: 4 stars, advanced (`p3_p4_inequiv`) Prove that the following programs are *not* equivalent. (Hint: What should the value of `Z` be when `p3` terminates? What about `p4`?)

Definition `p3` : `com` :=

```
<{ Z := 1;
  while X ≠ 0 do
    havoc X;
    havoc Z
  end }>.
```

Definition `p4` : `com` :=

```
<{ X := 0;
  Z := 1 }>.
```

Theorem `p3_p4_inequiv` : $\neg$ `cequiv p3 p4`.

Proof. *Admitted*.

□

Exercise: 5 stars, advanced, optional (`p5_p6_equiv`) Prove that the following commands are equivalent. (Hint: As mentioned above, our definition of `cequiv` for Himp only takes into account the sets of possible terminating configurations: two programs are equivalent if and only if the set of possible terminating states is the same for both programs when given a same starting state *st*. If `p5` terminates, what should the final state be? Conversely, is it always possible to make `p5` terminate?)

Definition `p5` : `com` :=

```
<{ while X ≠ 1 do
  havoc X
end }>.
```

Definition `p6` : `com` :=

```
<{ X := 1 }>.
```

Theorem `p5_p6_equiv` : `cequiv p5 p6`.

Proof. *Admitted*.

□

End HIMP.

4.7 Additional Exercises

Exercise: 3 stars, standard, optional (swap_noninterfering_assignments) (Hint: You may or may not – depending how you approach it – need to use `functional_extensionality` explicitly for this one.)

Theorem `swap_noninterfering_assignments`: $\forall l1\ l2\ a1\ a2,$

$l1 \neq l2 \rightarrow$
`var_not_used_in_aexp` $l1\ a2 \rightarrow$
`var_not_used_in_aexp` $l2\ a1 \rightarrow$
`cequiv`
 $\langle \{ l1 := a1; l2 := a2 \} \rangle$
 $\langle \{ l2 := a2; l1 := a1 \} \rangle.$

Proof.

Admitted.

□

Exercise: 4 stars, standard, optional (for_while_equiv) This exercise extends the optional `add_for_loop` exercise from the `Imp` chapter, where you were asked to extend the language of commands with C-style `for` loops. Prove that the command:

`for (c1; b; c2) { c3 }`
 is equivalent to:
`c1; while b do c3; c2 end`

Exercise: 4 stars, advanced, optional (capprox) In this exercise we define an asymmetric variant of program equivalence we call *program approximation*. We say that a program $c1$ *approximates* a program $c2$ when, for each of the initial states for which $c1$ terminates, $c2$ also terminates and produces the same final state. Formally, program approximation is defined as follows:

Definition `capprox` ($c1\ c2 : \text{com}$) : `Prop` := $\forall (st\ st' : \text{state}),$
 $st = [c1] \Rightarrow st' \rightarrow st = [c2] \Rightarrow st'.$

For example, the program

`c1 = while X <> 1 do X := X - 1 end`

approximates $c2 = X := 1$, but $c2$ does not approximate $c1$ since $c1$ does not terminate when $X = 0$ but $c2$ does. If two programs approximate each other in both directions, then they are equivalent.

Find two programs $c3$ and $c4$ such that neither approximates the other.

Definition `c3` : `com`

. Admitted.

Definition `c4` : `com`

. Admitted.

Theorem `c3_c4_different` : $\neg \text{capprox } c3\ c4 \wedge \neg \text{capprox } c4\ c3.$

Proof. *Admitted.*

Find a program `cmin` that approximates every other program.

Definition `cmin` : `com`
. *Admitted.*

Theorem `cmin_minimal` : $\forall c, \text{capprox } cmin\ c.$

Proof. *Admitted.*

Finally, find a non-trivial property which is preserved by program approximation (when going from left to right).

Definition `zprop` (`c` : `com`) : `Prop`
. *Admitted.*

Theorem `zprop_preserving` : $\forall c\ c',$
 $zprop\ c \rightarrow \text{capprox } c\ c' \rightarrow zprop\ c'.$

Proof. *Admitted.*

□

Chapter 5

Library **PLF.Hoare**

5.1 Hoare: Hoare Logic, Part I

```
Set Warnings "-notation-overridden".
From PLF Require Import Maps.
From Stdlib Require Import Bool.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat.
From Stdlib Require Import PeanoNat. Import NAT.
From Stdlib Require Import Lia.
From PLF Require Export Imp.
```

In the final chapter of *Logical Foundations* (*Software Foundations*, volume 1), we began applying the mathematical tools developed in the first part of the course to studying the theory of a small programming language, Imp.

- We defined a type of *abstract syntax trees* for Imp, together with an *evaluation relation* (a partial function on states) that specifies the *operational semantics* of programs.

The language we defined, though small, captures some of the key features of full-blown languages like C, C++, and Java, including the fundamental notion of mutable state and some common control structures.

- We proved a number of *metatheoretic properties* – “meta” in the sense that they are properties of the language as a whole, rather than of particular programs in the language. These included:
 - determinism of evaluation
 - equivalence of some different ways of writing down the definitions (e.g., functional and relational definitions of arithmetic expression evaluation)
 - guaranteed termination of certain classes of programs

- correctness (in the sense of preserving meaning) of a number of useful program transformations
- behavioral equivalence of programs (in the [Equiv](#) chapter).

If we stopped here, we would already have something useful: a set of tools for defining and discussing programming languages and language features that are mathematically precise, flexible, and easy to work with, applied to a set of key properties. All of these properties are things that language designers, compiler writers, and users might care about knowing. Indeed, many of them are so fundamental to our understanding of the programming languages we deal with that we might not consciously recognize them as “theorems.” But properties that seem intuitively obvious can sometimes be quite subtle (sometimes also subtly wrong!).

We’ll return to the theme of metatheoretic properties of whole languages later in this volume when we discuss *types* and *type soundness*. In this chapter, though, we turn to a different set of issues. Our goal in this chapter is to develop the tools to work through some simple examples of *program verification* – i.e., to use the precise definition of Imp to prove formally that particular programs satisfy particular specifications of their behavior.

We’ll develop a reasoning system called *Floyd-Hoare Logic* – often shortened to just *Hoare Logic* – in which each of the syntactic constructs of Imp is equipped with a generic “proof rule” that can be used to reason compositionally about the correctness of programs involving this construct.

Hoare Logic originated in the 1960s, and it continues to be the subject of intensive research right up to the present day. It lies at the core of a multitude of tools that are being used in academia and industry to specify and verify real software systems.

Hoare Logic combines two beautiful ideas: a natural way of writing down *specifications* of programs, and a *structured proof technique* for proving that programs are correct with respect to such specifications – where by “structured” we mean that the structure of proofs directly mirrors the structure of the programs that they are about.

5.2 Assertions

An *assertion* is a logical claim about the state of a program’s memory – formally, a property of [states](#).

Definition *Assertion* $:= \text{state} \rightarrow \text{Prop}$.

For example,

- $\text{fun } st \Rightarrow st.X = 3$ holds for states st in which value of X is 3,
- $\text{fun } st \Rightarrow \text{True}$ hold for all states, and
- $\text{fun } st \Rightarrow \text{False}$ holds for no states.

Exercise: 1 star, standard, optional (assertions) Paraphrase the following assertions in English (or your favorite natural language).

Module EXASSERTIONS.

Definition assertion1 : Assertion := fun *st* => *st* X ≤ *st* Y.

Definition assertion2 : Assertion :=

fun *st* => *st* X = 3 ∨ *st* X ≤ *st* Y.

Definition assertion3 : Assertion :=

fun *st* => *st* Z × *st* Z ≤ *st* X ∧
 ¬ ((S (*st* Z)) × (S (*st* Z))) ≤ *st* X).

Definition assertion4 : Assertion :=

fun *st* => *st* Z = max (*st* X) (*st* Y).

End EXASSERTIONS.

□

5.2.1 Notations for Assertions

This way of writing assertions can be a little bit heavy, for two reasons: (1) every single assertion that we ever write is going to begin with `fun st =>`; and (2) this state *st* is the only one that we ever use to look up variables in assertions (we will almost never need to talk about two different memory states at the same time). For discussing examples informally, we'll adopt some simplifying conventions: we'll drop the initial `fun st =>`, and we'll write just *X* to mean *st* X. Thus, instead of writing

fun *st* => *st* X = m

we'll write just

¹.

Here the “doubly curly” braces `{{` and `}}` delimit the scope of an assertion. We'll see more examples below.

This example also illustrates a convention that we'll use throughout the Hoare Logic chapters: in informal assertions, capital letters like *X*, *Y*, and *Z* are Imp variables, while lowercase letters like *x*, *y*, *m*, and *n* are ordinary Rocq variables (of type `nat`). This is why, when translating from informal to formal, we replace *X* with *st* X but leave *m* alone.

The convention described above can be implemented in Rocq with a little syntax magic, using coercions and annotation scopes, much as we did with the `<{ com }>` notation in Imp. This new notation automatically lifts `aexps`, numbers, and `Props` into `Assertions` when they appear in the `{{ _ }}` scope, or when Rocq knows that the type of an expression is `Assertion`.

There is no need to understand the details of how these notation hacks work, so we hide them in the HTML version of the notes. (We barely understand some of it ourselves!) For the gory details, see the Rocq development.

Definition Aexp : Type := state → nat.

¹*X*=*m*

```

Definition assert_of_Prop ( $P$  : Prop) : Assertion := fun _  $\Rightarrow$   $P$ .
Definition Aexp_of_nat ( $n$  : nat) : Aexp := fun _  $\Rightarrow$   $n$ .
Definition Aexp_of_aexp ( $a$  : aexp) : Aexp := fun  $st$   $\Rightarrow$  aeval  $st$   $a$ .

Coercion assert_of_Prop : Sortclass >-> Assertion.
Coercion Aexp_of_nat : nat >-> Aexp.
Coercion Aexp_of_aexp : aexp >-> Aexp.

Arguments assert_of_Prop /.
Arguments Aexp_of_nat /.
Arguments Aexp_of_aexp /.

Declare Custom Entry assn. Declare Scope assertion_scope.
Bind Scope assertion_scope with Assertion.
Bind Scope assertion_scope with Aexp.
Delimit Scope assertion_scope with assertion.

```

One small limitation of this approach is that we don't have an automatic way to coerce a function application that appears within an assertion to make appropriate use of the state when its arguments should be interpreted as Imp arithmetic expressions. Instead, we introduce a notation `#f  $e_1$  ..  $e_n$` that stands for `(fun  $st$   $\Rightarrow$  f ( $e_1$   $st$ ) .. ( $e_n$   $st$ ))`, letting us manually mark such function calls when they're needed as part of an assertion.

```

Notation "# f x .. y" := (fun  $st$   $\Rightarrow$  (.. (f ((x:Aexp)  $st$ )) .. ((y:Aexp)  $st$ )))
      (in custom assn at level 2,
      f constr at level 0, x custom assn at level 1,
      y custom assn at level 1) : assertion_scope.

Notation "P -> Q" := (fun  $st$   $\Rightarrow$  (P:Assertion)  $st$   $\rightarrow$  (Q:Assertion)  $st$ ) (in custom assn
at level 99, right associativity) : assertion_scope.
Notation "P <-> Q" := (fun  $st$   $\Rightarrow$  (P:Assertion)  $st$   $\leftrightarrow$  (Q:Assertion)  $st$ ) (in custom assn
at level 95) : assertion_scope.

Notation "P \\/ Q" := (fun  $st$   $\Rightarrow$  (P:Assertion)  $st$   $\vee$  (Q:Assertion)  $st$ ) (in custom assn
at level 85, right associativity) : assertion_scope.
Notation "P /\ Q" := (fun  $st$   $\Rightarrow$  (P:Assertion)  $st$   $\wedge$  (Q:Assertion)  $st$ ) (in custom assn
at level 80, right associativity) : assertion_scope.
Notation "~ P" := (fun  $st$   $\Rightarrow$   $\neg$  ((P:Assertion)  $st$ )) (in custom assn at level 75, right
associativity) : assertion_scope.
Notation "a = b" := (fun  $st$   $\Rightarrow$  (a:Aexp)  $st$  = (b:Aexp)  $st$ ) (in custom assn at level
70) : assertion_scope.
Notation "a <> b" := (fun  $st$   $\Rightarrow$  (a:Aexp)  $st$   $\neq$  (b:Aexp)  $st$ ) (in custom assn at level
70) : assertion_scope.
Notation "a <= b" := (fun  $st$   $\Rightarrow$  (a:Aexp)  $st$   $\leq$  (b:Aexp)  $st$ ) (in custom assn at level
70) : assertion_scope.
Notation "a < b" := (fun  $st$   $\Rightarrow$  (a:Aexp)  $st$  < (b:Aexp)  $st$ ) (in custom assn at level
70) : assertion_scope.

```

Notation "a >= b" := (fun st $\Rightarrow$ (a:Aexp) st $\geq$ (b:Aexp) st) (in custom assn at level 70) : assertion_scope.
 Notation "a > b" := (fun st $\Rightarrow$ (a:Aexp) st > (b:Aexp) st) (in custom assn at level 70) : assertion_scope.
 Notation "'True'" := **True**.
 Notation "'True'" := (fun st $\Rightarrow$ True) (in custom assn at level 0) : assertion_scope.
 Notation "'False'" := **False**.
 Notation "'False'" := (fun st $\Rightarrow$ False) (in custom assn at level 0) : assertion_scope.
 Notation "a + b" := (fun st $\Rightarrow$ (a:Aexp) st + (b:Aexp) st) (in custom assn at level 50, left associativity) : assertion_scope.
 Notation "a - b" := (fun st $\Rightarrow$ (a:Aexp) st - (b:Aexp) st) (in custom assn at level 50, left associativity) : assertion_scope.
 Notation "a * b" := (fun st $\Rightarrow$ (a:Aexp) st $\times$ (b:Aexp) st) (in custom assn at level 40, left associativity) : assertion_scope.
 Notation "(x)" := x (in custom assn at level 0, x at level 99) : assertion_scope.

Occasionally we need to “escape” a raw “Rocq-defined” function to express a particularly complicated assertion. We can do that using a \$ prefix, as in $\{\{ \$(\text{raw\_rocq}) \}\}$.

For example, $\{\{ \$(\text{fun st} \Rightarrow \forall X, \text{st } X = 0) \}\}$ indicates an assertion that every variable of X maps to 0 in the given state.

Notation "\$ f" := f (in custom assn at level 0, f constr at level 0) : assertion_scope.
 Notation "x" := (x%assertion) (in custom assn at level 0, x constr at level 0) : assertion_scope.
 Declare Scope hoare_spec_scope.
 Open Scope hoare_spec_scope.
 Notation "{ { e } }" := e (at level 2, e custom assn at level 99) : assertion_scope.
 Open Scope assertion_scope.

5.2.2 Example Assertions

Here are some example assertions that take advantage of this new notation.

Module EXAMPLEPRETTYASSERTIONS.
 Definition assertion1 : Assertion := { { X = 3 } }.
 Definition assertion2 : Assertion := { { True } }.
 Definition assertion3 : Assertion := { { False } }.
 Definition assertion4 : Assertion := { { True $\vee$ False } }.
 Definition assertion5 : Assertion := { { X $\leq$ Y } }.
 Definition assertion6 : Assertion := { { X = 3 $\vee$ X $\leq$ Y } }.
 Definition assertion7 : Assertion := { { Z = (#max X Y) } }.
 Definition assertion8 : Assertion := { { Z $\times$ Z $\leq$ X
 $\wedge \neg ((\#S Z) \times (\#S Z)) \leq X$ } }.
 Definition assertion9 : Assertion := { { #add X Y > #max Y X } }.

End EXAMPLEPRETTYASSERTIONS.

5.2.3 Assertion Implication

Given two assertions P and Q , we say that P *implies* Q , written $P \multimap Q$, if, whenever P holds in some state st , Q also holds.

Definition `assert_implies` ($P\ Q : \text{Assertion}$) : `Prop` :=
 $\forall\ st, P\ st \rightarrow Q\ st.$

Note that the notation for *assertion implication* is analogous to the “usual” Rocq implication $\rightarrow$.

Notation " $P \multimap Q$ " := (`assert_implies P Q`)
 (at level 80) : `hoare_spec_scope`.

We’ll also want the “iff” variant of implication between assertions:

Notation " $P \leftrightarrow Q$ " := ($P \multimap Q \wedge Q \multimap P$)
 (at level 80) : `hoare_spec_scope`.

(The *hoare_spec_scope* annotation here tells Rocq that this notation is not global but is intended to be used in particular contexts.)

5.3 Hoare Triples, Informally

A *Hoare triple* is a claim about the state before and after executing a command. The standard notation is

$\{P\}\ c\ \{Q\}$
 meaning:

- If command c begins execution in a state satisfying assertion P ,
- and if c eventually terminates in some final state,
- then that final state will satisfy the assertion Q .

Assertion P is called the *precondition* of the triple, and Q is the *postcondition*.

Because single braces are already used for other things in Rocq, we’ll write Hoare triples with double braces:

² $\{P\}\ c\ \{Q\}$ For example,

- The Hoare triple

⁴ $\{X=0\}\ X := X + 1\ \{X=1\}$ ⁵

² P

³ Q

⁴ $X=0$

⁵ $X=1$

states that command $X := X + 1$ will transform a state in which $X = 0$ to a state in which $X = 1$.

- On the other hand,

forall m , ⁶ $X := X + 1$ ⁷

is a *proposition* stating that the Hoare triple $\{\{X = m\}\} X := X + 1 \{\{X = m + 1\}\}$ is valid for any choice of m . Note that m in the two assertions is a reference to the *Rocq* variable m , which is bound outside the Hoare triple.

Exercise: 1 star, standard, optional (triples) Paraphrase the following in English.

- 1) ⁸ c ⁹
- 2) forall m , ¹⁰ c ¹¹
- 3) ¹² c ¹³
- 4) ¹⁴ c ¹⁵
- 5) forall m , ¹⁶ c ¹⁷
- 6) forall m , ¹⁸ c ¹⁹

Exercise: 1 star, standard, optional (valid_triples) Which of the following Hoare triples are *valid* – i.e., the claimed relation between P , c , and Q is true?

- 1) ²⁰ $X := 5$ ²¹
- 2) ²² $X := X + 1$ ²³
- 3) ²⁴ $X := 5; Y := 0$ ²⁵
- 4) ²⁶ $X := 5$ ²⁷

⁶ $X=m$
⁷ $X=m+1$
⁸ True
⁹ $X=5$
¹⁰ $X=m$
¹¹ $X=m+5$
¹² $X<=Y$
¹³ $Y<=X$
¹⁴ True
¹⁵ False
¹⁶ $X=m$
¹⁷ $Y=\text{real\_factm}$
¹⁸ $X=m$
¹⁹ $(Z*Z)<=m/\wedge\sim(((SZ)*(SZ))<=m)$
²⁰ True
²¹ $X=5$
²² $X=2$
²³ $X=3$
²⁴ True
²⁵ $X=5$
²⁶ $X=2/\wedge X=3$
²⁷ $X=0$

- 5) ²⁸ skip ²⁹
- 6) ³⁰ skip ³¹
- 7) ³² while true do skip end ³³
- 8) ³⁴ while X = 0 do X := X + 1 end ³⁵
- 9) ³⁶ while X <> 0 do X := X + 1 end ³⁷

5.4 Hoare Triples, Formally

We formalize valid Hoare triples in Rocq as follows:

Definition `valid_hoare_triple`

```
(P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st st',
    st =[ c ]=> st' →
    P st →
    Q st'.
```

Notation for Hoare triples

```
Notation "{{ P }}" c "{{ Q }}" :=
  (valid_hoare_triple P c Q)
  (at level 2, P custom assn at level 99, c custom com at level 99,
   Q custom assn at level 99)
  : hoare_spec_scope.
```

Exercise: 1 star, standard (`hoare_post_true`) Prove that if Q holds in every state, then any triple with Q as its postcondition is valid.

```
Theorem hoare_post_true : ∀ (P Q : Assertion) c,
  (∀ st, Q st) →
  {{P}} c {{Q}}.
```

Proof.

Admitted.

□

```
28True
29False
30False
31True
32True
33False
34X=0
35X=1
36X=1
37X=100
```

Exercise: 1 star, standard, optional (hoare_pre_false) Prove that if P holds in no state, then any triple with P as its precondition is valid.

Theorem `hoare_pre_false` : $\forall (P\ Q : \text{Assertion})\ c,$
 $(\forall st, \neg (P\ st)) \rightarrow$
 $\{\{P\}\}\ c\ \{\{Q\}\}.$

Proof.

Admitted.

□

5.5 Proof Rules

The goal of Hoare logic is to provide a *compositional* method for proving the validity of specific Hoare triples. That is, we want the structure of a program’s correctness proof to mirror the structure of the program itself. To this end, in the sections below, we’ll introduce a rule for reasoning about each of the different syntactic forms of commands in Imp – one for assignment, one for sequencing, one for conditionals, etc. – plus a couple of “structural” rules for gluing things together. We will then be able to prove programs correct using these proof rules, without ever unfolding the definition of `valid_hoare_triple`.

5.5.1 Skip

Since `skip` doesn’t change the state, it preserves any assertion P :

(`hoare_skip`) ³⁸ `skip` ³⁹

Theorem `hoare_skip` : $\forall P,$
 $\{\{P\}\}\ \text{skip}\ \{\{P\}\}.$

Proof.

`intros P st st' H HP. inversion H; subst. assumption.`

`Qed.`

5.5.2 Sequencing

If command $c1$ takes any state where P holds to a state where Q holds, and if $c2$ takes any state where Q holds to one where R holds, then doing $c1$ followed by $c2$ will take any state where P holds to one where R holds:

⁴⁰ $c1$ ⁴¹ $c2$ ⁴² $c2$ ⁴³

³⁸ P

³⁹ P

⁴⁰ P

⁴¹ Q

⁴² Q

⁴³ R

(hoare_seq) ⁴⁴ $c1; c2$ ⁴⁵

Theorem hoare_seq : $\forall P Q R c1 c2,$
 $\{\{Q\}\} c2 \{\{R\}\} \rightarrow$
 $\{\{P\}\} c1 \{\{Q\}\} \rightarrow$
 $\{\{P\}\} c1; c2 \{\{R\}\}.$

Proof.

```
intros P Q R c1 c2 H1 H2 st st' H12 Pre.  
inversion H12; subst.  
eauto.
```

Qed.

Note that, in the formal rule `hoare_seq`, the premises are given in backwards order ($c2$ before $c1$). This matches the natural flow of information in many of the situations where we'll use the rule, since the natural way to construct a Hoare-logic proof is to begin at the end of the program (with the final postcondition) and push postconditions backwards through commands until we reach the beginning.

5.5.3 Assignment

The rule for assignment is the most fundamental of the Hoare logic proof rules. Here's how it works.

Consider this incomplete Hoare triple:

⁴⁶ $X := Y$ ⁴⁷

We want to assign Y to X and finish in a state where X is 1. What could the precondition be?

One possibility is $Y = 1$, because if Y is already 1 then assigning it to X causes X to be 1. That leads to a valid Hoare triple:

⁴⁸ $X := Y$ ⁴⁹

It may seem as though coming up with that precondition must have taken some clever thought. But there is a mechanical way we could have done it: if we take the postcondition $X = 1$ and in it replace X with Y —that is, replace the left-hand side of the assignment statement with the right-hand side—we get the precondition, $Y = 1$.

That same idea works in more complicated cases. For example:

⁵⁰ $X := X + Y$ ⁵¹

⁴⁴P
⁴⁵R
⁴⁶???
⁴⁷X=1
⁴⁸Y=1
⁴⁹X=1
⁵⁰???
⁵¹X=1

If we replace the X in $X = 1$ with $X + Y$, we get $X + Y = 1$. That again leads to a valid Hoare triple:

⁵² $X := X + Y$ ⁵³

Why does this technique work? The postcondition identifies some property P that we want to hold of the variable X being assigned. In this case, P is “equals 1”. To complete the triple and make it valid, we need to identify a precondition that guarantees that property will hold of X . Such a precondition must ensure that the same property holds of *whatever is being assigned to* X . So, in the example, we need “equals 1” to hold of $X + Y$. That’s exactly what the technique guarantees.

In general, the postcondition could be some arbitrary assertion Q , and the right-hand side of the assignment could be some arbitrary arithmetic expression a :

⁵⁴ $X := a$ ⁵⁵

The precondition would then be Q , but with any occurrences of X in it replaced by a .

Let’s introduce a notation for this idea of replacing occurrences: Define $Q[X \rightarrow a]$ to mean “ Q where a is substituted in place of X ”.

This yields the Hoare logic rule for assignment:

⁵⁶ $X := a$ ⁵⁷

One way of reading this rule is: If you want statement $X := a$ to terminate in a state that satisfies assertion Q , then it suffices to start in a state that also satisfies Q , except where a is substituted for every occurrence of X .

To many people, this rule seems “backwards” at first, because it proceeds from the postcondition to the precondition. Actually it makes good sense to go in this direction: the postcondition is often what is more important, because it characterizes what will be true after running the code.

Nonetheless, it’s also possible to formulate a “forward” assignment rule. We’ll do that later in some exercises.

Here are some valid instances of the assignment rule:

⁵⁸ (that is, $X + 1 \leq 5$) $X := X + 1$ ⁵⁹

⁶⁰ (that is, $3 = 3$) $X := 3$ ⁶¹

⁶². (that is, $0 \leq 3 \wedge 3 \leq 5$) $X := 3$ ⁶³

To formalize the rule, we must first formalize the idea of “substituting an expression for an Imp variable in an assertion”, which we refer to as assertion substitution, or `assertion_sub`.

⁵² $X+Y=1$
⁵³ $X=1$
⁵⁴???
⁵⁵ Q
⁵⁶ $Q[X \rightarrow a]$
⁵⁷ Q
⁵⁸ $(X \leq 5) [X \rightarrow X+1]$
⁵⁹ $X \leq 5$
⁶⁰ $(X=3) [X \rightarrow 3]$
⁶¹ $X=3$
⁶² $(0 \leq X \wedge X \leq 5) [X \rightarrow 3]$
⁶³ $0 \leq X \wedge X \leq 5$

Intuitively, given a proposition P , a variable X , and an arithmetic expression a , we want to derive another proposition P' that is just the same as P except that P' should mention a wherever P mentions X .

This operation is related to the idea of substituting Imp expressions for Imp variables that we saw in `Equiv (subst_aexp and friends)`. The difference is that, here, P is an arbitrary Rocq assertion, so we can't directly "edit" its text.

However, we can achieve the same effect by evaluating P in an updated state, defined as follows:

Definition `assertion_sub X (a:aexp) (P:Assertion) : Assertion :=`

```
  fun (st : state) =>
    (P%_assertion) (X !-> ((a:Aexp) st); st).
```

Notation `"P [ X |-> a ]" := (assertion_sub X a P)`

```
      (in custom assn at level 10, left associativity,
       P custom assn, X global, a custom com)
      : assertion_scope.
```

This notation allows us to write this operation as:

`P X |-> a`

That is, $P [X \mapsto a]$ stands for an assertion – let's call it P' – that behaves just like P except that, wherever P looks up the variable X in the current state, P' instead uses the value of the expression a .

To see how this works in more detail, let's calculate what happens with a couple of examples. First, suppose P' is $(X \leq 5) [X \mapsto 3]$ – that is, more formally, P' is the Rocq expression

```
fun st => (fun st' => st' X <= 5) (X !-> aeval st 3 ; st),
```

which simplifies to

```
fun st => (fun st' => st' X <= 5) (X !-> 3 ; st)
```

and further simplifies to

```
fun st => ((X !-> 3 ; st) X) <= 5
```

and finally to

```
fun st => 3 <= 5.
```

That is, P' is the assertion that 3 is less than or equal to 5 (as expected).

For a more interesting example, suppose P' is $(X \leq 5) [X \mapsto X + 1]$. Formally, P' is the Rocq expression

```
fun st => (fun st' => st' X <= 5) (X !-> aeval st (X + 1); st),
```

which simplifies to

```
fun st => (X !-> aeval st (X + 1) ; st) X <= 5
```

and further simplifies to

```
fun st => (aeval st (X + 1)) <= 5.
```

That is, P' is the assertion that $X + 1$ is at most 5.

We can demonstrate formally that we have captured intuitive meaning of "assertion substitution" by proving some example logical equivalences:

```

Module EXAMPLEASSERTIONSUB.
Example equivalent_assertion1 :
  {{ (X ≤ 5) [X |-> 3] }} «-» {{ 3 ≤ 5 }}.
Proof.
  split; unfold assert_implies, assertion_sub; intros st H;
  simpl in *; apply H.
Qed.
Example equivalent_assertion2 :
  {{ (X ≤ 5) [X |-> X + 1] }} «-» {{ (X + 1) ≤ 5 }}.
Proof.
  split; unfold assert_implies, assertion_sub; intros st H;
  simpl in *; apply H.
Qed.
End EXAMPLEASSERTIONSUB.

```

Now, using the substitution operation we’ve just defined, we can give the precise proof rule for assignment:

(hoare_asgn) ⁶⁴ $X := a$ ⁶⁵

We can prove formally that this rule is indeed valid.

Theorem hoare_asgn : $\forall Q X (a : \text{aexp}),$
 $\{\{Q [X \mapsto a]\}\} X := a \{\{Q\}\}.$

Proof.
 intros Q X a st st' HE HQ.
 inversion HE. subst.
 unfold assertion_sub in HQ. simpl in HQ. assumption. Qed.

Here’s a first formal proof of a Hoare triple using this rule.

Example assertion_sub_example :
 $\{\{(X < 5) [X \mapsto X + 1]\}\}$
 $X := X + 1$
 $\{\{X < 5\}\}.$

Proof.
 apply hoare_asgn. Qed.

Of course, we’d probably prefer to work with this simpler triple:

⁶⁶ $X := X + 1$ ⁶⁷

We will see how to do so in the next section.

Complete these Hoare triples by providing an appropriate precondition using $\exists$, then prove then with `apply hoare_asgn`. If you find that tactic doesn’t suffice, double check that

⁶⁴ $Q[X \mapsto a]$

⁶⁵ Q

⁶⁶ $X < 4$

⁶⁷ $X < 5$

you have completed the triple properly.

Exercise: 2 stars, standard, optional (hoare_asgn_examples1) [Example hoare_asgn_examples1](#)

```

:
  ∃ P,
    {{ P }}
    X := 2 × X
    {{ X ≤ 10 }}.

```

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (hoare_asgn_examples2) [Example hoare_asgn_examples2](#)

```

:
  ∃ P,
    {{ P }}
    X := 3
    {{ 0 ≤ X ∧ X ≤ 5 }}.

```

Proof. Admitted.

□

Exercise: 2 stars, standard, especially useful (hoare_asgn_wrong) The assignment rule looks backward to almost everyone the first time they see it. If it still seems puzzling to you, it may help to think a little about alternative “forward” rules. Here is a seemingly natural one:

(hoare_asgn_wrong) ⁶⁸ X := a ⁶⁹

Give a counterexample showing that this rule is incorrect and use it to complete the proof below, showing that it is really a counterexample. (Hint: The rule universally quantifies over the arithmetic expression *a*, so your counterexample needs to exhibit an *a* for which the rule doesn’t work.)

Theorem hoare_asgn_wrong : ∃ a : aexp,

¬ {{ True }} X := a {{ X = a }}.

Proof.

Admitted.

Exercise: 3 stars, advanced, optional (hoare_asgn_fwd) By using a *parameter m* (a Rocq number) to remember the original value of *X* we can define a Hoare rule for assignment that does, intuitively, “work forwards” rather than backwards.

⁶⁸True

⁶⁹X=a

(hoare_asgn_fwd) ⁷⁰ $X := a$ ⁷¹

Note that we need to write out the postcondition in “desugared” form, because it needs to talk about two different states: we use the original value of X to reconstruct the state st' before the assignment took place. (Also note that this rule is more complicated than `hoare_asgn!`)

Prove that this rule is correct.

Theorem `hoare_asgn_fwd` :

$\forall (m:\text{nat}) (a:\text{aexp}) (P : \text{Assertion}),$
 $\{\{ P \wedge X = m \}\}$
 $X := a$
 $\{\{ \$(\text{fun } st \Rightarrow (P (X \mapsto m ; st)$
 $\quad \wedge st X = \text{aeval } (X \mapsto m ; st) a)) \}\}$.

Proof.

Admitted.

□

Exercise: 2 stars, advanced, optional (`hoare_asgn_fwd_exists`) Another way to define a forward rule for assignment is to existentially quantify over the previous value of the assigned variable. Prove that it is correct.

(hoare_asgn_fwd_exists) ⁷² $X := a$ ⁷³

Theorem `hoare_asgn_fwd_exists` :

$\forall a (P : \text{Assertion}),$
 $\{\{ P \}\}$
 $X := a$
 $\{\{ \$(\text{fun } st \Rightarrow \exists m, P (X \mapsto m ; st) \wedge$
 $\quad st X = \text{aeval } (X \mapsto m ; st) a) \}\}$.

Proof.

Admitted.

□

5.5.4 Consequence

Sometimes the preconditions and postconditions we get from the Hoare rules won't quite be the ones we want in the particular situation at hand – they may be logically equivalent but have a different syntactic form that fails to unify with the goal we are trying to prove,

⁷⁰`fun st => Pst /\ st X = m`

⁷¹`fun st => P(X!->m;st) /\ st X = aeval(X!->m;st) a`

⁷²`fun st => Pst`

⁷³`fun st => exists m, P(X!->m;st) /\ st X = aeval(X!->m;st) a`

or they actually may be logically weaker (for preconditions) or stronger (for postconditions) than what we need.

For instance,

⁷⁴ $X := 3$ ⁷⁵,

follows directly from the assignment rule, but

⁷⁶ $X := 3$ ⁷⁷

does not. This triple is valid, but it is not an instance of `hoare_asgn` because `True` and $(X = 3) [X \mapsto 3]$ are not syntactically equal assertions.

However, they are logically *equivalent*, so if one triple is valid, then the other must certainly be as well. We can capture this observation with the following rule:

⁷⁸ c ⁷⁹ $P \vdash P'$

⁸⁰ c ⁸¹

Taking this line of thought a bit further, we can see that strengthening the precondition or weakening the postcondition of a valid triple always produces another valid triple. This observation is captured by two *Rules of Consequence*.

⁸² c ⁸³ $P \multimap P'$

(hoare_consequence_pre) ⁸⁴ c ⁸⁵

⁸⁶ c ⁸⁷ $Q' \multimap Q$

(hoare_consequence_post) ⁸⁸ c ⁸⁹

Here are the formal versions:

Theorem `hoare_consequence_pre` : $\forall (P P' Q : \text{Assertion}) c,$

$\{\{P'\}\} c \{\{Q\}\} \rightarrow$

$P \multimap P' \rightarrow$

$\{\{P\}\} c \{\{Q\}\}.$

Proof.

⁷⁴ $(X=3) [X \mapsto 3]$

⁷⁵ $X=3$

⁷⁶ `True`

⁷⁷ $X=3$

⁷⁸ $P,$

⁷⁹ Q

⁸⁰ P

⁸¹ Q

⁸² $P,$

⁸³ Q

⁸⁴ P

⁸⁵ Q

⁸⁶ P

⁸⁷ $Q,$

⁸⁸ P

⁸⁹ Q

```

  unfold valid_hoare_triple, "-»".
  intros P P' Q c Hhoare Himp st st' Heval Hpre.
  apply Hhoare with (st := st).
  - assumption.
  - apply Himp. assumption.
Qed.

Theorem hoare_consequence_post :  $\forall (P\ Q\ Q' : \text{Assertion})\ c,$ 
   $\{\{P\}\}\ c\ \{\{Q'\}\} \rightarrow$ 
   $Q' -\!\!\rangle\ Q \rightarrow$ 
   $\{\{P\}\}\ c\ \{\{Q\}\}.$ 

```

Proof.

```

  unfold valid_hoare_triple, "-»".
  intros P Q Q' c Hhoare Himp st st' Heval Hpre.
  apply Himp.
  apply Hhoare with (st := st).
  - assumption.
  - assumption.
Qed.

```

For example, we can use the first consequence rule like this:

⁹⁰ $-\!\!\rangle$ ⁹¹ $X := 1$ ⁹²

Or, formally...

```

Example hoare_asgn_example1 :
   $\{\{\text{True}\}\}\ X := 1\ \{\{X = 1\}\}.$ 

```

Proof.

```

  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - unfold "-»", assertion_sub, t_update; simpl.
    intros st _. reflexivity.

```

Qed.

We can also use it to prove the example mentioned earlier.

⁹³ $-\!\!\rangle$ ⁹⁴ $X := X + 1$ ⁹⁵

Or, formally ...

```

Example assertion_sub_example2 :
   $\{\{X < 4\}\}$ 
   $X := X + 1$ 

```

⁹⁰True

⁹¹(X=1) [X| ->1]

⁹²X=1

⁹³X<4

⁹⁴(X<5) [X| ->X+1]

⁹⁵X<5

```

    {{X < 5}}.
Proof.
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - unfold "->", assertion_sub, t_update.
    intros st H. simpl in *. lia.
Qed.

```

Finally, here is a combined rule of consequence that allows us to vary both the precondition and the postcondition.

⁹⁶ c ⁹⁷ $P \multimap P' \quad Q' \multimap Q$

(hoare_consequence) ⁹⁸ c ⁹⁹

Theorem hoare_consequence : $\forall (P \ P' \ Q \ Q' : \text{Assertion}) \ c,$
 $\{\{P'\}\} \ c \ \{\{Q'\}\} \rightarrow$
 $P \multimap P' \rightarrow$
 $Q' \multimap Q \rightarrow$
 $\{\{P\}\} \ c \ \{\{Q\}\}.$

```

Proof.
  intros P P' Q Q' c Htriple Hpre Hpost.
  apply hoare_consequence_pre with (P' := P').
  - apply hoare_consequence_post with (Q' := Q'); assumption.
  - assumption.
Qed.

```

5.5.5 Automation

Many of the proofs we have done so far with Hoare triples can be streamlined using the automation techniques that we introduced in the *Auto* chapter of *Logical Foundations*.

Recall that the `auto` tactic can be told to `unfold` definitions as part of its proof search. Let's give it that hint for the definitions and coercions we're using:

```

Hint Unfold assert_implies assertion_sub t_update : core.
Hint Unfold valid_hoare_triple : core.
Hint Unfold assert_of_Prop Aexp_of_nat Aexp_of_aexp : core.

```

Also recall that `auto` will search for a proof involving `intros` and `apply`. By default, the theorems that it will apply include any of the local hypotheses, as well as theorems in the “core” hint database.

⁹⁶ P'
⁹⁷ Q'
⁹⁸ P
⁹⁹ Q

The proof of `hoare_consequence_pre`, repeated below, looks like an opportune place for such automation, because all it does is `unfold`, `intros`, and `apply`. (It uses `assumption`, too, but that's just application of a hypothesis.)

```
Theorem hoare_consequence_pre' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -> P' →
  {{P}} c {{Q}}.
```

Proof.

```
unfold valid_hoare_triple, "->".
intros P P' Q c Hhoare Himp st st' Heval Hpre.
apply Hhoare with (st := st).
- assumption.
- apply Himp. assumption.
```

Qed.

Merely using `auto`, though, doesn't complete the proof.

```
Theorem hoare_consequence_pre'' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -> P' →
  {{P}} c {{Q}}.
```

Proof.

```
auto. Abort.
```

The problem is the `apply Hhoare with...` part of the proof. Rocq isn't able to figure out how to instantiate `st` without some help from us. Recall, though, that there are versions of many tactics that will use *existential variables* to make progress even when the regular versions of those tactics would get stuck.

Here, the `eapply` tactic will introduce an existential variable `?st` as a placeholder for `st`, and `eassumption` will instantiate `?st` with `st` when it discovers `st` in assumption `Heval`. By using `eapply` we are essentially telling Rocq, "Be patient: The missing part is going to be filled in later in the proof."

```
Theorem hoare_consequence_pre''' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -> P' →
  {{P}} c {{Q}}.
```

Proof.

```
unfold valid_hoare_triple, "->".
intros P P' Q c Hhoare Himp st st' Heval Hpre.
eapply Hhoare.
- eassumption.
- apply Himp. assumption.
```

Qed.

The `eauto` tactic will use `eapply` as part of its proof search. So, the entire proof can

actually be done in just one line.

```
Theorem hoare_consequence_pre''' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -> P' →
  {{P}} c {{Q}}.
```

Proof.

```
eauto.
```

Qed.

Of course, it's hard to predict that `eauto` suffices here without having gone through the original proof of `hoare_consequence_pre` to see the tactics it used. But now that we know `eauto` worked there, it's a good bet that it will also work for `hoare_consequence_post`.

```
Theorem hoare_consequence_post' : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' -> Q →
  {{P}} c {{Q}}.
```

Proof.

```
eauto.
```

Qed.

We can also use `eapply` to streamline a proof (`hoare_asgn_example1`), that we did earlier as an example of using the consequence rule:

```
Example hoare_asgn_example1' :
  {{True}} X := 1 {{X = 1}}.
```

Proof.

```
eapply hoare_consequence_pre. - apply hoare_asgn.
- unfold "->", assertion_sub, t_update.
  intros st _. simpl. reflexivity.
```

Qed.

The final bullet of that proof also looks like a candidate for automation.

```
Example hoare_asgn_example1'' :
  {{True}} X := 1 {{X = 1}}.
```

Proof.

```
eapply hoare_consequence_pre.
- apply hoare_asgn.
- auto.
```

Qed.

Now we have quite a nice proof script: it simply identifies the Hoare rules that need to be used and leaves the remaining low-level details up to Rocq to figure out.

By now it might be apparent that the *entire* proof could be automated if we added `hoare_consequence_pre` and `hoare_asgn` to the hint database. We won't do that in this chapter, so that we can get a better understanding of when and how the Hoare rules are

used. In the next chapter, [Hoare2](#), we'll dive deeper into automating entire proofs of Hoare triples.

The other example of using consequence that we did earlier, [hoare_asgn_example2](#), requires a little more work to automate. We can streamline the first line with `eapply`, but we can't just use `auto` for the final bullet, since it needs [lia](#).

```
Example assertion_sub_example2' :
```

```
  {{X < 4}}
  X := X + 1
  {{X < 5}}.
```

```
Proof.
```

```
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - auto.      unfold "->", assertion_sub, t_update.
    intros st H. simpl in *. lia.
```

```
Qed.
```

Let's introduce our own tactic to handle both that bullet and the bullet from example 1:

```
Ltac assertion_auto :=
```

```
  try auto;
  try (unfold "->", assertion_sub, t_update;
    intros; simpl in *; lia).
```

```
Example assertion_sub_example2'' :
```

```
  {{X < 4}}
  X := X + 1
  {{X < 5}}.
```

```
Proof.
```

```
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - assertion_auto.
```

```
Qed.
```

```
Example hoare_asgn_example1''' :
```

```
  {{True}} X := 1 {{X = 1}}.
```

```
Proof.
```

```
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - assertion_auto.
```

```
Qed.
```

Again, we have quite a nice proof script. All the low-level details of proofs about assertions have been taken care of automatically. Of course, [assertion_auto](#) isn't able to prove everything we could possibly want to know about assertions – there's no magic here! But it's pretty good.

Exercise: 2 stars, standard (hoare_asgn_examples_2) Prove these triples. Try to make your proof scripts nicely automated by following the examples above.

Example assertion_sub_ex1' :

```
{ { X ≤ 5 } }
  X := 2 × X
{ { X ≤ 10 } }.
```

Proof.

Admitted.

Example assertion_sub_ex2' :

```
{ { 0 ≤ 3 ∧ 3 ≤ 5 } }
  X := 3
{ { 0 ≤ X ∧ X ≤ 5 } }.
```

Proof.

Admitted.

□

5.5.6 Sequencing + Assignment

Here's an example of a program involving both sequencing and assignment. Note the use of `hoare_seq` in conjunction with `hoare_consequence_pre` and the `eapply` tactic.

Example hoare_asgn_example3 : $\forall (a:\text{aexp}) (n:\text{nat}),$

```
{ { a = n } }
  X := a;
  skip
{ { X = n } }.
```

Proof.

```
intros a n. eapply hoare_seq.
-
  apply hoare_skip.
-
  eapply hoare_consequence_pre.
  + apply hoare_asgn.
  + assertion_auto.
```

Qed.

Informally, a nice way of displaying a proof using the sequencing rule is as a “decorated program” where the intermediate assertion Q is written between $c1$ and $c2$:

¹⁰⁰ $X := a$ ¹⁰¹; $\leftarrow$ decoration for Q skip ¹⁰² We'll come back to the idea of decorated programs in much more detail in the next chapter.

¹⁰⁰ $a=n$

¹⁰¹ $X=n$

¹⁰² $X=n$

Exercise: 2 stars, standard, especially useful (hoare_asgn_example4) Translate this “decorated program” into a formal proof:

```
103 -» 104 X := 1 105 -» 106; Y := 2 107
```

Note the use of “-»” decorations, each marking a use of `hoare_consequence_pre`.

We’ve started you off by providing a use of `hoare_seq` that explicitly identifies $X = 1$ as the intermediate assertion.

Example `hoare_asgn_example4` :

```
{ { True } }
  X := 1;
  Y := 2
{ { X = 1 ∧ Y = 2 } }.
```

Proof.

```
eapply hoare_seq with (Q := { { X = 1 } }).
```

```
Admitted.
```

□

Exercise: 3 stars, standard (swap_exercise) Write an Imp program `c` that swaps the values of `X` and `Y` and show that it satisfies the following specification:

```
108 c 109
```

Your proof should not need to use `unfold valid_hoare_triple`.

Hints:

- Remember that Imp commands need to be enclosed in `<{...}>` brackets.
- Remember that the assignment rule works best when it’s applied “back to front,” from the postcondition to the precondition. So your proof will want to start at the end and work back to the beginning of your program.
- Remember that `eapply` is your friend.)

Definition `swap_program` : `com`

```
. Admitted.
```

Theorem `swap_exercise` :

```
{ { X ≤ Y } }
  swap_program
{ { Y ≤ X } }.
```

```
103 True
104 1=1
105 X=1
106 X=1 /\ 2=2
107 X=1 /\ Y=2
108 X<=Y
109 Y<=X
```

Proof.

Admitted.

□

Exercise: 4 stars, advanced (invalid_triple) Show that

¹¹⁰ $X := 3; Y := a$ ¹¹¹

is not a valid Hoare triple for some choices of a and n .

Conceptual hint: Invent a particular a and n for which the triple is invalid, then use those to complete the proof.

Technical hint: Hypothesis H below begins $\forall a\ n, \dots$. You'll want to instantiate that with the particular a and n you've invented. You can do that with `assert` and `apply`, but you may remember (from *Tactics.v* in Logical Foundations) that Rocq offers an even easier tactic: `specialize`. If you write

specialize H with (a := your_a) (n := your_n)

the hypothesis will be instantiated on *your_a* and *your_n*.

Having chosen your a and n , proceed as follows:

- Use the (assumed) validity of the given hoare triple to derive a state st' in which Y has some value $y1$
- Use the evaluation rules (`E_Seq` and `E_Asgn`) to show that Y has a *different* value $y2$ in the same final state st'
- Since $y1$ and $y2$ are both equal to $st' Y$, they are equal to each other. But we chose them to be different, so this is a contradiction, which finishes the proof.

Theorem `invalid_triple` : $\neg \forall (a : \text{aexp}) (n : \text{nat}),$
 $\{\{ a = n \}\}$
 $X := 3; Y := a$
 $\{\{ Y = n \}\}.$

Proof.

unfold valid_hoare_triple.

intros H.

Admitted.

□

5.5.7 Conditionals

What sort of rule do we want for reasoning about conditional commands?

Certainly, if the same assertion Q holds after executing either of the branches, then it holds after the whole conditional. So we might be tempted to write:

¹¹⁰ $a=n$

¹¹¹ $Y=n$

112 c1 113 114 c2 115

116 if b then c1 else c2 117

However, this is rather weak. For example, using this rule, we cannot show

118 if X = 0 then Y := 2 else Y := X + 1 end 119

since the rule doesn't tell us enough about the state in which the assignments take place in the "then" and "else" branches.

Fortunately, we can say something more precise. In the "then" branch, we know that the boolean expression *b* evaluates to *true*, and in the "else" branch, we know it evaluates to *false*. Making this information available in the premises of the rule gives us more information to work with when reasoning about the behavior of *c1* and *c2* (i.e., the reasons why they establish the postcondition *Q*).

120 c1 121 122 c2 123

(hoare_if) 124 if b then c1 else c2 end 125

To interpret this rule formally, we need to do a little work. Strictly speaking, the assertion we've written, $P \wedge b$, is the conjunction of an assertion and a boolean expression – i.e., it doesn't typecheck. To fix this, we need a way of formally "lifting" any bexp *b* to an assertion. We'll write *bassertion b* for the assertion "the boolean expression *b* evaluates to *true* (in the given state)."

Definition *bassertion b* : Assertion :=

fun *st* => (beval *st b* = true).

Coercion *bassertion* : bexp >-> Assertion.

Arguments *bassertion* /.

A useful fact about *bassertion*:

Lemma *bexp_eval_false* : $\forall b\ st,$

beval *st b* = false $\rightarrow \neg ((bassertion\ b)\ st)$.

Proof. congruence. Qed.

Hint Resolve *bexp_eval_false* : core.

112 P
113 Q
114 P
115 Q
116 P
117 Q
118 True
119 X<=Y
120 P/\b
121 Q
122 P/\~b
123 Q
124 P
125 Q

We mentioned the `congruence` tactic in passing in *Auto* when building the *find_rwd* tactic. Like *find_rwd*, `congruence` is able to automatically find that both `beval st b = false` and `beval st b = true` are being assumed, notice the contradiction, and `discriminate` to complete the proof.

Now we can formalize the Hoare proof rule for conditionals and prove it correct.

```
Theorem hoare_if : ∀ P Q (b:bexp) c1 c2 ,
  {{ P ∧ b }} c1 {{ Q }} →
  {{ P ∧ ¬ b }} c2 {{ Q }} →
  {{ P }} if b then c1 else c2 end {{ Q }}.
```

That is (unwrapping the notations):

```
Theorem hoare_if : forall P Q b c1 c2, 126 c1 127 -> 128 c2 129 -> 130 if b then c1 else c2
end 131.
```

Proof.

```
intros P Q b c1 c2 HTrue HFalse st st' HE HP.
inversion HE; subst; eauto.
```

Qed.

Example

Here is a formal proof that the program we used to motivate the rule satisfies the specification we wanted.

Example if_example :

```
{{ True }}
if (X = 0)
  then Y := 2
  else Y := X + 1
end
{{ X ≤ Y }}.
```

Proof.

```
apply hoare_if.
-
  eapply hoare_consequence_pre.
  + apply hoare_asgn.
  + assertion_auto.      unfold "->", assertion_sub, t_update, bassertion.
    simpl. intros st [_ H]. apply eqb_eq in H.
    rewrite H. lia.
-

```

```
126 funst=>Pst/\bassertionbst
127 Q
128 funst=>Pst/\~(bassertionbst)
129 Q
130 P
131 Q
```



```

    eapply hoare_consequence_pre.
  + apply hoare_asgn.
  + assertion_auto.

```

Qed.

As we did earlier, it would be nice to eliminate all the low-level proof script that isn't about the Hoare rules. Unfortunately, the *assertion_auto* tactic we wrote wasn't quite up to the job. Looking at the proof of *if_example*, we can see why. We had to unfold a definition (*bassertion*) and use a theorem (*eqb_eq*) that we didn't need in earlier proofs. So, let's add those into our tactic, and clean it up a little in the process.

```

Ltac assertion_auto' :=
  unfold "-»", assertion_sub, t_update, bassertion;
  intros; simpl in *;
  try rewrite → eqb_eq in *;
  auto; try lia.

```

Now the proof is quite streamlined.

Example *if_example*'' :

```

  {{True}}
  if X = 0
  then Y := 2
  else Y := X + 1
  end
  {{X ≤ Y}}.

```

Proof.

```

  apply hoare_if.
- eapply hoare_consequence_pre.
  + apply hoare_asgn.
  + assertion_auto'.
- eapply hoare_consequence_pre.
  + apply hoare_asgn.
  + assertion_auto'.

```

Qed.

We can even shorten it a little bit more.

Example *if_example*''' :

```

  {{True}}
  if X = 0
  then Y := 2
  else Y := X + 1
  end
  {{X ≤ Y}}.

```

Proof.

```

  apply hoare_if; eapply hoare_consequence_pre;

```

```

    try apply hoare_asgn; try assertion_auto'.
Qed.

```

For later proofs, it will help to extend *assertion_auto*' to handle inequalities, too.

```

Ltac assertion_auto'' :=
  unfold "->", assertion_sub, t_update, bassertion;
  intros; simpl in *;
  try rewrite → eqb_eq in *;
  try rewrite → leb_le in *;
  auto; try lia.

```

Exercise: 2 stars, standard (if_minus_plus) Prove the theorem below using *hoare_if*. Do not use *unfold valid_hoare_triple*. The *assertion_auto''* tactic we just defined may be useful.

```

Theorem if_minus_plus :
  {{True}}
  if (X ≤ Y)
  then Z := Y - X
  else Y := X + Z
  end
  {{Y = X + Z}}.

```

Proof.

```

  Admitted.

```

□

Exercise: One-sided conditionals

In this exercise we consider extending Imp with “one-sided conditionals” of the form *if1 b then c end*. Here *b* is a boolean expression, and *c* is a command. If *b* evaluates to *true*, then command *c* is evaluated. If *b* evaluates to *false*, then *if1 b then c end* does nothing.

We recommend that you complete this exercise before attempting the ones that follow, as it should help solidify your understanding of the material.

The first step is to extend the syntax of commands and introduce the usual notations. (We’ve done this for you, in a separate module to prevent polluting the global name space.)

Module IF1.

```

Inductive com : Type :=
| CSkip : com
| CAsgn : string → aexp → com
| CSeq : com → com → com
| Clf : bexp → com → com → com
| CWhile : bexp → com → com
| Clf1 : bexp → com → com.

```

```

Notation "'if1' x 'then' y 'end'" :=
  (Clf1 x y)
  (in custom com at level 0, x custom com at level 99).
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
   format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
   right associativity,
   format "'[v' x ; '/' y']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (Clf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
   format "'[v' 'if' x 'then' '/' 'y' '/' 'else' '/' 'z' '/' 'end'']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
   format "'[v' 'while' x 'do' '/' 'y' '/' 'end'']'") : com_scope.

```

Exercise: 2 stars, standard, especially useful (if1_ceval) Add two new evaluation rules to relation **ceval**, below, for *if1*. Let the rules for **if** guide you.

```

Reserved Notation
  "st0 '=[ ' c ' ]=>' st1 '/' s"
  (at level 40, c custom com at level 99,
   st0 constr, st1 constr at next level,
   format "'[hv' st0 =[ '/' ' ' c ']' '/' ]=> st1 / s']").

```

```

Inductive ceval : com → state → state → Prop :=
| E_Skip : ∀ st,
  st =[ skip ]=> st
| E_Asgn : ∀ st a1 n x,
  aeval st a1 = n →
  st =[ x := a1 ]=> (x !-> n ; st)
| E_Seq : ∀ c1 c2 st st' st'',
  st =[ c1 ]=> st' →
  st' =[ c2 ]=> st'' →
  st =[ c1 ; c2 ]=> st''
| E_IfTrue : ∀ st st' b c1 c2,
  beval st b = true →
  st =[ c1 ]=> st' →
  st =[ if b then c1 else c2 end ]=> st'
| E_IfFalse : ∀ st st' b c1 c2,
  beval st b = false →

```

```

    st = [ c2 ] => st' →
    st = [ if b then c1 else c2 end ] => st'
| E_WhileFalse : ∀ b st c,
  beval st b = false →
  st = [ while b do c end ] => st
| E_WhileTrue : ∀ st st' st'' b c,
  beval st b = true →
  st = [ c ] => st' →
  st' = [ while b do c end ] => st'' →
  st = [ while b do c end ] => st''

```

where "st' = [c] => st'" := (ceval c st st').

Hint Constructors **ceval** : core.

The following unit tests should be provable simply by **eauto** if you have defined the rules for *if1* correctly.

Example if1true_test :

```
empty_st = [ if1 X = 0 then X := 1 end ] => (X !-> 1).
```

Proof. Admitted.

Example if1false_test :

```
(X !-> 2) = [ if1 X = 0 then X := 1 end ] => (X !-> 2).
```

Proof. Admitted.

□

Now we have to repeat the definition and notation of Hoare triples, so that they will use the updated **com** type.

Definition valid_hoare_triple

```

(P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st st',
    st = [ c ] => st' →
    P st →
    Q st'.

```

Hint Unfold valid_hoare_triple : core.

Notation "{ { P } } c { { Q } }" :=

```
(valid_hoare_triple P c Q)
```

(at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn at level 99)

```
: hoare_spec_scope.
```

Exercise: 2 stars, standard (hoare_if1) Invent a Hoare logic proof rule for *if1*. State and prove a theorem named *hoare_if1* that shows the validity of your rule. Use **hoare_if** as a

guide. Try to invent a rule that is *complete*, meaning it can be used to prove the correctness of as many one-sided conditionals as possible. Also try to keep your rule *compositional*, meaning that any `Imp` command that appears in a premise should syntactically be a part of the command in the conclusion.

Hint: if you encounter difficulty getting `Rocq` to parse part of your rule as an assertion, try manually indicating that it should be in the assertion scope. For example, if you want e to be parsed as an assertion, write it as $(e)\%*assertion*$.

For example (`hoare_if1_good`) your rule should be strong enough to show the following Hoare triple is valid:

¹³² `if1 Y <> 0 then X := X + Y end` ¹³³

Definition `manual_grade_for_hoare_if1` : `option (nat × string) := None`.

□

Before the next exercise, we need to restate the Hoare rules of consequence (for preconditions) and assignment for the new `com` type.

Theorem `hoare_consequence_pre` : $\forall (P \ P' \ Q : \text{Assertion}) \ c,$

$\{\{P'\}\} \ c \ \{\{Q\}\} \rightarrow$

$P \multimap P' \rightarrow$

$\{\{P\}\} \ c \ \{\{Q\}\}.$

Proof.

`eauto.`

Qed.

Theorem `hoare_asgn` : $\forall \ Q \ X \ a,$

$\{\{Q \ [X \mapsto a]\}\} \ (X := a) \ \{\{Q\}\}.$

Proof.

`intros Q X a st st' Heval HQ.`

`inversion Heval; subst.`

`auto.`

Qed.

Exercise: 2 stars, standard (`hoare_if1_good`) Use your *if1* rule to prove the following (valid) Hoare triple.

Hint: `assertion_auto''` will once again get you most but not all the way to a completely automated proof. You can finish manually, or tweak the tactic further.

Hint: If you see a message like *Unable to unify "Imp.ceval Imp.CSkip st st"* with..., it probably means you are using a definition or theorem *e.g.*, `hoare_skip` from above this exercise without re-proving it for the new version of `Imp` with `if1`.

Lemma `hoare_if1_good` :

$\{\{X + Y = Z\}\}$

`if1 Y ≠ 0 then`

¹³² $X+Y=Z$

¹³³ $X=Z$

```

    X := X + Y
  end
  {{ X = Z }}.

```

Proof. Admitted.

□

End IF1.

5.5.8 While Loops

The Hoare rule for *while* loops is based on the idea of a *command invariant* (or just *invariant*): an assertion whose truth is guaranteed after executing a command, assuming it is true before.

That is, an assertion P is a command invariant of c if

134 c 135

holds. Note that the command invariant might temporarily become false in the middle of executing c , but by the end of c it must be restored.

As a first attempt at a *while* rule, we could try:

136 c 137

while b do c end P 138}

This rule is valid: if P is a command invariant of c , as the premise requires, then, no matter how many times the loop body executes, P is going to be true when the loop finally finishes.

But the rule also omits two crucial pieces of information. First, the loop terminates when b becomes false. So we can strengthen the postcondition in the conclusion:

139 c 140

while b do c end $P \wedge b$ 141}

Second, the loop body will be executed only if b is true. So we can also strengthen the precondition in the premise:

142 c 143

(hoare_while) while b do c end $P \wedge b$ 144}

134 P
 135 P
 136 P
 137 P
 138 P
 139 P
 140 P
 141 P
 142 $P \wedge b$
 143 P
 144 P

That is the Hoare *while* rule. Note how it combines aspects of *skip* and conditionals:

- If the loop body executes zero times, the rule is like *skip* in that the precondition survives to become (part of) the postcondition.
- Like a conditional, we can assume guard *b* holds on entry to the subcommand.

Theorem `hoare_while` : $\forall P (b:\text{bexp}) c,$
 $\{\{P \wedge b\}\} c \{\{P\}\} \rightarrow$
 $\{\{P\}\} \text{while } b \text{ do } c \text{ end } \{\{P \wedge \neg b\}\}.$

Proof.

```
intros P b c Hhoare st st' Heval HP.
remember <{while b do c end}> as original_command eqn:Horig.
induction Heval;
  try (inversion Horig; subst; clear Horig);
  eauto.
```

Qed.

We call *P* a *loop invariant* of *while b do c end* if
¹⁴⁵ *c* ¹⁴⁶

is a valid Hoare triple.

This means that *P* will be true at the end of the loop body whenever the loop body executes. If *P* contradicts *b*, this holds trivially since the precondition is false.

For instance, $X = 0$ is a loop invariant of

`while X = 2 do X := 1 end`

since the program will never enter the loop.

The program

`while Y > 10 do Y := Y - 1; Z := Z + 1 end`

admits an interesting loop invariant:

$X = Y + Z$

Note that this doesn't contradict the loop guard but neither is it a command invariant of

`Y := Y - 1; Z := Z + 1`

since, if $X = 5$, $Y = 0$ and $Z = 5$, running the command will set $Y + Z$ to 6. The loop guard $Y > 10$ guarantees that this will not be the case. We will see many such loop invariants in the following chapter.

Example `while_example` :

```
{X ≤ 3}
while (X ≤ 2) do
  X := X + 1
end
```

¹⁴⁵ $P \wedge b$

¹⁴⁶ P

```

  {{X = 3}}.
Proof.
  eapply hoare_consequence_post.
  - apply hoare_while.
    eapply hoare_consequence_pre.
    + apply hoare_asgn.
    + assertion_auto''.
  - assertion_auto''.
Qed.

```

If the loop never terminates, any postcondition will work.

```

Theorem always_loop_hoare : ∀ Q,
  {{True}} while true do skip end {{Q}}.
Proof.
  intros Q.
  eapply hoare_consequence_post.
  - apply hoare_while. apply hoare_post_true. auto.
  - simpl. intros st [Hinv Hguard]. congruence.
Qed.

```

Of course, this result is not surprising if we remember that the definition of `valid_hoare_triple` asserts that the postcondition must hold *only* when the command terminates. If the command doesn't terminate, we can prove anything we like about the post-condition.

Hoare rules that specify what happens *if* commands terminate, without proving that they do, are said to describe a logic of *partial* correctness. It is also possible to give Hoare rules for *total* correctness, which additionally specifies that commands must terminate. Total correctness is out of the scope of this textbook.

Exercise: *REPEAT*

Exercise: 4 stars, advanced, optional (hoare_repeat) In this exercise, we'll add a new command to our language of commands: *REPEAT* `c` `until` `b` `end`. You will write the evaluation rule for *REPEAT* and add a new Hoare rule to the language for programs involving it. (You may recall that the evaluation rule is given in an example in the *Auto* chapter. Try to figure it out yourself here rather than peeking.)

Module REPEATEXERCISE.

```

Inductive com : Type :=
| CSkip : com
| CAsgn : string → aexp → com
| CSeq : com → com → com
| Clf : bexp → com → com → com
| CWhile : bexp → com → com
| CRepeat : com → bexp → com.

```


REPEAT behaves like *while*, except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*. Because of this, the body will always execute at least once.

```

Notation "'repeat' e1 'until' b2 'end'" :=
  (CRepeat e1 b2)
  (in custom com at level 0,
   e1 custom com at level 99, b2 custom com at level 99).
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
   format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
   right associativity,
   format "[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
   format "[v' 'if' x 'then' '/' y '/' 'else' '/' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
   format "[v' 'while' x 'do' '/' y '/' 'end' ']'") : com_scope.

```

Add new rules for *REPEAT* to **ceval** below. You can use the rules for *while* as a guide, but remember that the body of a *REPEAT* should always execute at least once, and that the loop ends when the guard becomes true.

```

Inductive ceval : state → com → state → Prop :=
| E_Skip : ∀ st,
  st = [ skip ] => st
| E_Asgn : ∀ st a1 n x,
  aeval st a1 = n →
  st = [ x := a1 ] => (x !-> n ; st)
| E_Seq : ∀ c1 c2 st st' st'',
  st = [ c1 ] => st' →
  st' = [ c2 ] => st'' →
  st = [ c1 ; c2 ] => st''
| E_IfTrue : ∀ st st' b c1 c2,
  beval st b = true →
  st = [ c1 ] => st' →
  st = [ if b then c1 else c2 end ] => st'
| E_IfFalse : ∀ st st' b c1 c2,
  beval st b = false →
  st = [ c2 ] => st' →

```

```

    st = [ if b then c1 else c2 end ] => st'
| E_WhileFalse : ∀ b st c,
  beval st b = false →
  st = [ while b do c end ] => st
| E_WhileTrue : ∀ st st' st'' b c,
  beval st b = true →
  st = [ c ] => st' →
  st' = [ while b do c end ] => st'' →
  st = [ while b do c end ] => st''

```

where "st' = [c] => st'" := (ceval st c st').

A couple of definitions from above, copied here so they use the new **ceval**.

```

Definition valid_hoare_triple (P : Assertion) (c : com) (Q : Assertion)
  : Prop :=
  ∀ st st', st = [ c ] => st' → P st → Q st'.

```

```

Notation "{ { P } } c { { Q } }" :=
  (valid_hoare_triple P c Q)

```

```

  (at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn
  at level 99)
  : hoare_spec_scope.

```

To make sure you've got the evaluation rules for **repeat** right, prove that **ex1_repeat** evaluates correctly.

```

Definition ex1_repeat :=
  <{ repeat
    X := 1;
    Y := Y + 1
  until (X = 1) end }>.

```

```

Theorem ex1_repeat_works :
  empty_st = [ ex1_repeat ] => (Y !-> 1 ; X !-> 1).

```

Proof.

Admitted.

Now state and prove a theorem, *hoare_repeat*, that expresses an appropriate proof rule for **repeat** commands. Use **hoare_while** as a model, and try to make your rule as precise as possible.

For full credit, make sure (informally) that your rule can be used to prove the following valid Hoare triple:

¹⁴⁷ repeat Y := X; X := X - 1 until X = 0 end ¹⁴⁸

¹⁴⁷X>0

¹⁴⁸X=0 /\ Y>0

End REPEAT EXERCISE.

Definition manual_grade_for_hoare_repeat : option (nat × string) := None.

□

5.6 Summary

So far, we've introduced Hoare Logic as a tool for reasoning about Imp programs.

The rules of Hoare Logic are:

(hoare_asgn) ¹⁴⁹ X:=a ¹⁵⁰

(hoare_skip) ¹⁵¹ skip ¹⁵²
¹⁵³ c1 ¹⁵⁴ ¹⁵⁵ c2 ¹⁵⁶

(hoare_seq) ¹⁵⁷ c1;c2 ¹⁵⁸
¹⁵⁹ c1 ¹⁶⁰ ¹⁶¹ c2 ¹⁶²

(hoare_if) ¹⁶³ if b then c1 else c2 end ¹⁶⁴
¹⁶⁵ c ¹⁶⁶

(hoare_while) ¹⁶⁷ while b do c end ¹⁶⁸
¹⁶⁹ c ¹⁷⁰ P -> P' Q' -> Q

¹⁴⁹ Q [X | -> a]
¹⁵⁰ Q
¹⁵¹ P
¹⁵² P
¹⁵³ P
¹⁵⁴ Q
¹⁵⁵ Q
¹⁵⁶ R
¹⁵⁷ P
¹⁵⁸ R
¹⁵⁹ P /\ b
¹⁶⁰ Q
¹⁶¹ P /\ ~b
¹⁶² Q
¹⁶³ P
¹⁶⁴ Q
¹⁶⁵ P /\ b
¹⁶⁶ P
¹⁶⁷ P
¹⁶⁸ P /\ ~b
¹⁶⁹ P',
¹⁷⁰ Q'

(hoare_consequence) ¹⁷¹ c ¹⁷²

Our main task in this chapter has been to *define* the rules of Hoare logic, and prove that the definitions are sound. Having done so, we can go on and work *within* Hoare logic to prove that particular programs satisfy particular Hoare triples. In the next chapter, we'll see how Hoare logic is can be used to prove that more interesting programs satisfy interesting specifications of their behavior.

Crucially, we will do so without ever again *unfolding* the definition of Hoare triples – i.e., we will take the rules of Hoare logic as a closed world for reasoning about programs.

5.7 Additional Exercises

5.7.1 Havoc

In this exercise, we will derive proof rules for a *HAVOC* command, which is similar to the nondeterministic *any* expression from the the *Imp* chapter.

First, we enclose this work in a separate module, and recall the syntax and big-step semantics of *Himp* commands.

Module *HIMP*.

Inductive *com* : Type :=

| CSkip : *com*
| CAsgn : *string* → *aexp* → *com*
| CSeq : *com* → *com* → *com*
| Clf : *bexp* → *com* → *com* → *com*
| CWhile : *bexp* → *com* → *com*
| CHavoc : *string* → *com*.

Notation "'havoc' l" := (CHavoc l)

(in custom com at level 60, l constr at level 0).

Notation "'skip'" := CSkip

(in custom com at level 0) : *com_scope*.

Notation "x := y" := (CAsgn x y)

(in custom com at level 0, x constr at level 0, y at level 85, no associativity,
format "x := y") : *com_scope*.

Notation "x ; y" := (CSeq x y)

(in custom com at level 90,
right associativity,
format "[v' x ; '/' y ']'") : *com_scope*.

Notation "'if' x 'then' y 'else' z 'end'" := (Clf x y z)

(in custom com at level 89, x at level 99, y at level 99, z at level 99,
format "[v' 'if' x 'then' '/' 'y '/' 'else' '/' 'z '/' 'end' ']'") : *com_scope*.

¹⁷¹P

¹⁷²Q

Notation "'while' x 'do' y 'end'" := (CWhile x y)
 (in custom com at level 89, x at level 99, y at level 99,
 format "[v' 'while' x 'do' '/' 'y '/' 'end' ']'") : com_scope.

Inductive **ceval** : **com** → state → state → Prop :=

| E_Skip : ∀ st,
 st = [skip] => st
 | E_Asgn : ∀ st a1 n x,
 aeval st a1 = n →
 st = [x := a1] => (x !-> n ; st)
 | E_Seq : ∀ c1 c2 st st' st'',
 st = [c1] => st' →
 st' = [c2] => st'' →
 st = [c1 ; c2] => st''
 | E_IfTrue : ∀ st st' b c1 c2,
 beval st b = true →
 st = [c1] => st' →
 st = [if b then c1 else c2 end] => st'
 | E_IfFalse : ∀ st st' b c1 c2,
 beval st b = false →
 st = [c2] => st' →
 st = [if b then c1 else c2 end] => st'
 | E_WhileFalse : ∀ b st c,
 beval st b = false →
 st = [while b do c end] => st
 | E_WhileTrue : ∀ st st' st'' b c,
 beval st b = true →
 st = [c] => st' →
 st' = [while b do c end] => st'' →
 st = [while b do c end] => st''
 | E_Havoc : ∀ st x n,
 st = [havoc x] => (x !-> n ; st)

where "st '=[c ']=>' st'" := (**ceval** c st st').

Hint Constructors **ceval** : core.

The definition of Hoare triples is exactly as before.

Definition valid_hoare_triple (P:Assertion) (c:**com**) (Q:Assertion) : Prop :=
 ∀ st st', st = [c] => st' → P st → Q st'.

Hint Unfold valid_hoare_triple : core.

Notation "{ { P } } c { { Q } }" :=

(valid_hoare_triple P c Q)

(at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn

```
at level 99)
  : hoare_spec_scope.
```

And the precondition consequence rule is exactly as before.

```
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -> P' →
  {{P}} c {{Q}}.
```

Proof. eauto. Qed.

Exercise: 3 stars, advanced (hoare_havoc) Complete the Hoare rule for *HAVOC* commands below by defining `havoc_pre`, and prove that the resulting rule is correct.

```
Definition havoc_pre (X : string) (Q : Assertion) (st : total_map nat) : Prop
  . Admitted.
```

```
Theorem hoare_havoc : ∀ (Q : Assertion) (X : string),
  {{ $(havoc_pre X Q) }} havoc X {{ Q }}.
```

Proof.

```
Admitted.
```

□

Exercise: 3 stars, advanced (havoc_post) Complete the following proof without changing any of the provided commands. If you find that it can't be completed, your definition of `havoc_pre` is probably too strong. Find a way to relax it so that `havoc_post` can be proved.

Hint: the *assertion_auto* tactics we've built won't help you here. You need to proceed manually.

```
Theorem havoc_post : ∀ (P : Assertion) (X : string),
  {{ P }} havoc X {{ $(fun st => ∃ (n:nat), ({P [X |-> n] }) st) }}.
```

Proof.

```
intros P X. eapply hoare_consequence_pre.
```

```
- apply havoc_pre.
```

```
- Admitted.
```

□

End HIMP.

5.7.2 Assert and Assume

Exercise: 4 stars, standard (assert_vs_assume) In this exercise, we will extend IMP with two commands, `assert` and `assume`. Both commands are ways to indicate that a certain assertion should hold any time this part of the program is reached. However they differ as follows:

- If an `assert` statement fails, it causes the program to go into an error state and exit.
- If an `assume` statement fails, the program fails to evaluate at all. In other words, the program gets stuck and has no final state.

The new set of commands is:

Module HOAREASSERTASSUME.

Inductive **com** : Type :=

```
CSkip : com
CAsgn : string → aexp → com
CSeq  : com → com → com
CIf   : bexp → com → com → com
CWhile : bexp → com → com
CAssert : bexp → com
CAssume : bexp → com.
```

```
Notation "'assert' l" := (CAssert l)
      (in custom com at level 8, l custom com at level 0).
```

```
Notation "'assume' l" := (CAssume l)
      (in custom com at level 8, l custom com at level 0).
```

```
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
```

```

Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
   format "x := y") : com_scope.

```

```

Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
   right associativity,
   format "'[v' x ; ']' y ']'") : com_scope.

```

```

Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
   format "[v' 'if' x 'then' '/' y '/' 'else' '/' z '/' 'end' ']'") : com_scope.

```

```

Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
   format "'[v' 'while' x 'do' '/' y '/' 'end' ]'") : com_scope.

```

To define the behavior of `assert` and `assume`, we need to add notation for an error, which indicates that an assertion has failed. We modify the `ceval` relation, therefore, so that it relates a start state to either an end state or to `error`. The `result` type indicates the end value of a program, either a state or an error:

Inductive **result** : Type :=

```
| RNormal : state → result
| RError : result.
```

Now we are ready to give you the ceval relation for the new language.

Inductive **ceval** : **com** → state → **result** → Prop :=

```

| E_Skip : ∀ st,
  st = [ skip ] => RNormal st
| E_Asgn : ∀ st a1 n x,
  aeval st a1 = n →
  st = [ x := a1 ] => RNormal (x !-> n ; st)
| E_SeqNormal : ∀ c1 c2 st st' r,
  st = [ c1 ] => RNormal st' →
  st' = [ c2 ] => r →
  st = [ c1 ; c2 ] => r
| E_SeqError : ∀ c1 c2 st,
  st = [ c1 ] => RError →
  st = [ c1 ; c2 ] => RError
| E_IfTrue : ∀ st r b c1 c2,
  beval st b = true →
  st = [ c1 ] => r →
  st = [ if b then c1 else c2 end ] => r
| E_IfFalse : ∀ st r b c1 c2,
  beval st b = false →
  st = [ c2 ] => r →
  st = [ if b then c1 else c2 end ] => r
| E_WhileFalse : ∀ b st c,
  beval st b = false →
  st = [ while b do c end ] => RNormal st
| E_WhileTrueNormal : ∀ st st' r b c,
  beval st b = true →
  st = [ c ] => RNormal st' →
  st' = [ while b do c end ] => r →
  st = [ while b do c end ] => r
| E_WhileTrueError : ∀ st b c,
  beval st b = true →
  st = [ c ] => RError →
  st = [ while b do c end ] => RError

| E_AssertTrue : ∀ st b,
  beval st b = true →
  st = [ assert b ] => RNormal st
| E_AssertFalse : ∀ st b,
  beval st b = false →
  st = [ assert b ] => RError
| E_Assume : ∀ st b,

```



```

beval st b = true →
st =[ assume b ] => RNormal st

```

where "st '=[c ']=> r" := (ceval c st r).

We redefine hoare triples: Now, $\{\{P\}\} c \{\{Q\}\}$ means that, whenever c is started in a state satisfying P , and terminates with result r , then r is not an error and the state of r satisfies Q .

Definition valid_hoare_triple

```

(P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st r,
    st =[ c ] => r → P st →
    (∃ st', r = RNormal st' ∧ Q st').

```

Notation " $\{\{P\}\} c \{\{Q\}\}$ " :=

```

(valid_hoare_triple P c Q)

```

(at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn at level 99)

```

: hoare_spec_scope.

```

To test your understanding of this modification, give an example precondition and postcondition that are satisfied by the *assume* statement but not by the *assert* statement.

```

Theorem assert_assume_differ : ∃ (P:Assertion) b (Q:Assertion),
  ({P} assume b {Q})
  ∧ ¬ ({P} assert b {Q}).
Admitted.

```

Then prove that any triple for an *assert* also works when *assert* is replaced by *assume*.

```

Theorem assert_implies_assume : ∀ P b Q,
  ({P} assert b {Q})
  → ({P} assume b {Q}).

```

Proof.

```

Admitted.

```

Next, here are proofs for the old hoare rules adapted to the new semantics. You don't need to do anything with these.

```

Theorem hoare_asgn : ∀ Q X a,
  {Q [X |-> a]} X := a {Q}.

```

Proof.

```

unfold valid_hoare_triple.
intros Q X a st st' HE HQ.
inversion HE. subst.
∃ (X !-> aeval st a ; st). split; try reflexivity.
assumption. Qed.

```

```

Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,

```

$\{\{P'\}\} c \{\{Q\}\} \rightarrow$
 $P \multimap P' \rightarrow$
 $\{\{P\}\} c \{\{Q\}\}.$

Proof.

intros $P P' Q c$ *Hhoare Himp*.
 intros $st st' Hc HP$. apply (*Hhoare st st'*).
 - assumption.
 - apply *Himp*. assumption. Qed.

Theorem hoare_consequence_post : $\forall (P Q Q' : \text{Assertion}) c,$

$\{\{P\}\} c \{\{Q'\}\} \rightarrow$
 $Q' \multimap Q \rightarrow$
 $\{\{P\}\} c \{\{Q\}\}.$

Proof.

intros $P Q Q' c$ *Hhoare Himp*.
 intros $st r Hc HP$.
 unfold valid_hoare_triple in *Hhoare*.
 assert ($\exists st', r = \text{RNormal } st' \wedge Q' st'$).
 { apply (*Hhoare st*); assumption. }
 destruct *H* as [$st' [Hr HQ']$].
 $\exists st'$. split; try assumption.
 apply *Himp*. assumption.

Qed.

Theorem hoare_seq : $\forall P Q R c1 c2,$

$\{\{Q\}\} c2 \{\{R\}\} \rightarrow$
 $\{\{P\}\} c1 \{\{Q\}\} \rightarrow$
 $\{\{P\}\} c1 ; c2 \{\{R\}\}.$

Proof.

intros $P Q R c1 c2 H1 H2 st r H12 Pre$.
 inversion *H12*; subst.
 - eapply *H1*.
 + apply *H6*.
 + apply *H2* in *H3*. apply *H3* in *Pre*.
 destruct *Pre* as [$st'0 [Heq HQ]$].
 inversion *Heq*; subst. assumption.
 -
 apply *H2* in *H5*. apply *H5* in *Pre*.
 destruct *Pre* as [$st' [C -]$].
 inversion *C*.

Qed.

Here are the other proof rules (sanity check) Theorem hoare_skip : $\forall P,$
 $\{\{P\}\} \text{skip} \{\{P\}\}.$

Proof.

```

intros  $P$   $st$   $st'$   $H$   $HP$ . inversion  $H$ . subst.
eexists. split.
- reflexivity.
- assumption.
Qed.

Theorem hoare_if :  $\forall P Q (b:bexp) c1 c2$ ,
   $\{\{P \wedge b\}\} c1 \{\{Q\}\} \rightarrow$ 
   $\{\{P \wedge \neg b\}\} c2 \{\{Q\}\} \rightarrow$ 
   $\{\{P\}\} \text{ if } b \text{ then } c1 \text{ else } c2 \text{ end } \{\{Q\}\}$ .
Proof.
  intros  $P Q b c1 c2 HTrue HFalse st st' HE HP$ .
  inversion  $HE$ ; subst.
  -
    apply ( $HTrue st st'$ ).
    + assumption.
    + split; assumption.
  -
    apply ( $HFalse st st'$ ).
    + assumption.
    + split.
      × assumption.
      × apply bexp_eval_false. assumption.
Qed.

Theorem hoare_while :  $\forall P (b:bexp) c$ ,
   $\{\{P \wedge b\}\} c \{\{P\}\} \rightarrow$ 
   $\{\{P\}\} \text{ while } b \text{ do } c \text{ end } \{\{P \wedge \neg b\}\}$ .
Proof.
  intros  $P b c Hhoare st st' He HP$ .
  remember <{while  $b$  do  $c$  end}> as  $wcom$  eqn: $Heqwcom$ .
  induction  $He$ ;
    try (inversion  $Heqwcom$ ); subst; clear  $Heqwcom$ .
  -
    eexists. split.
    + reflexivity.
    + split; try assumption.
      apply bexp_eval_false. assumption.
  -
    clear  $IHHe1$ .
    apply  $IHHe2$ .
    + reflexivity.
    + clear  $IHHe2 He2 r$ .
      unfold valid_hoare_triple in  $Hhoare$ .

```

```

    apply Hhoare in He1.
    × destruct He1 as [st1 [Heq Hst1] ].
      inversion Heq; subst.
      assumption.
    × split; assumption.
  -
    exfalso. clear IHHe.
    unfold valid_hoare_triple in Hhoare.
    apply Hhoare in He.
    + destruct He as [st' [C -] ]. inversion C.
    + split; assumption.
Qed.

```

Finally, state Hoare rules for `assert` and `assume` and use them to prove a simple program correct. Name your rules *hoare_assert* and *hoare_assume*.

Use your rules to prove the following triple.

Example `assert_assume_example`:

```

{{True}}
  assume (X = 1);
  X := X + 1;
  assert (X = 2)
{{True}}.

```

Proof.

Admitted.

End HOAREASSERTASSUME.

□

Chapter 6

Library **PLF.Hoare2**

6.1 Hoare2: Hoare Logic, Part II

```
Set Warnings "-notation-overridden".
Ltac intuition_solver := auto.
From Stdlib Require Import Strings.String.
From PLF Require Import Maps.
From Stdlib Require Import Bool.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat.
From Stdlib Require Import PeanoNat. Import NAT.
From Stdlib Require Import Lia.
From PLF Require Export Imp.
From PLF Require Import Hoare.
Definition FILL_IN_HERE := <{True}>.
```

6.2 Decorated Programs

The beauty of Hoare Logic is that it is *syntax directed*: the structure of proofs exactly follows the structure of programs.

We can record the essential ideas of a Hoare-logic proof – omitting low-level calculational details – by “decorating” a program with appropriate assertions on each of its commands.

Such a *decorated program* carries within itself an argument for its own correctness.

For example, consider the program:

$X := m; Z := p; \text{while } X \neq 0 \text{ do } Z := Z - 1; X := X - 1 \text{ end}$ Here is one possible specification for this program, in the form of a Hoare triple:

¹ $X := m; Z := p; \text{while } X \neq 0 \text{ do } Z := Z - 1; X := X - 1 \text{ end}$ ² (Note the *parameters*

¹True

² $Z=p-m$

m and p , which stand for fixed-but-arbitrary numbers. Formally, they are simply Rocq variables of type `nat`.)

Here is a decorated version of this program, embodying a proof of this specification:

```

3 -> 4 X := m 5 -> 6; Z := p; 7 -> 8 while X <> 0 do 9 -> 10 Z := Z - 1 11; X := X - 1 12
end 13 -> 14

```

Concretely, a decorated program consists of the program’s text interleaved with assertions (sometimes multiple assertions separated by `->`).

A decorated program can be viewed as a compact representation of a proof in Hoare Logic: the assertions surrounding each command specify the Hoare triple to be proved for that part of the program using one of the Hoare Logic rules, and the structure of the program itself shows how to assemble all these individual steps into a proof for the whole program.

Our goal is to verify such decorated programs “mostly automatically.” But, before we can verify anything, we need to be able to *find* a proof for a given specification, and for this we need to discover the right assertions. This can be done in an almost mechanical way, with the exception of finding loop invariants. In the remainder of this section, we explain in detail how to construct decorations for several short programs, all of which are loop free or have simple loop invariants. We’ll return to finding more interesting loop invariants later in the chapter.

6.2.1 Example: Swapping

Consider the following program, which swaps the values of two variables using addition and subtraction, instead of by assigning to a temporary variable.

```
X := X + Y; Y := X - Y; X := X - Y
```

We can give a proof, in the form of decorations, that this program is correct – i.e., it really swaps X and Y – as follows.

```
(1) 15 -> (2) 16 X := X + Y (3) 17; Y := X - Y (4) 18; X := X - Y (5) 19
```

The decorations can be constructed as follows:

```

3True
4m=m
5X=m
6X=m/\p=p
7X=m/\Z=p
8Z-X=p-m
9Z-X=p-m/\X<>0
10(Z-1)-(X-1)=p-m
11Z-(X-1)=p-m
12Z-X=p-m
13Z-X=p-m/\~(X<>0)
14Z=p-m
15X=m/\Y=n
16(X+Y)-((X+Y)-Y)=n/\(X+Y)-Y=m
17X-(X-Y)=n/\X-Y=m
18X-Y=n/\Y=m
19X=n/\Y=m

```

- We begin with the undecorated program (the unnumbered lines).
- We add the specification – i.e., the outer precondition (1) and postcondition (5). In the precondition, we use parameters m and n to remember the initial values of variables X and Y so that we can refer to them in the postcondition (5).
- We work backwards, mechanically, starting from (5) and proceeding until we get to (2). At each step, we obtain the precondition of the assignment from its postcondition by substituting the assigned variable with the right-hand-side of the assignment. For instance, we obtain (4) by substituting X with $X - Y$ in (5), and we obtain (3) by substituting Y with $X - Y$ in (4).
- Finally, we verify that (1) logically implies (2) – i.e., that the step from (1) to (2) is a valid use of the law of consequence – by doing a bit of high-school algebra.

6.2.2 Example: Simple Conditionals

Here is a simple decorated program using conditionals:

(1) ²⁰ if $X \leq Y$ then (2) ²¹ $\rightarrow$ (3) ²² $Z := Y - X$ (4) ²³ else (5) ²⁴ $\rightarrow$ (6) ²⁵ $Z := X - Y$
 (7) ²⁶ end (8) ²⁷

These decorations can be constructed as follows:

- We start with the outer precondition (1) and postcondition (8).
- Following the format dictated by the `hoare_if` rule, we copy the postcondition (8) to (4) and (7). We conjoin the precondition (1) with the guard of the conditional to obtain (2). We conjoin (1) with the negated guard of the conditional to obtain (5).
- In order to use the assignment rule and obtain (3), we substitute Z by $Y - X$ in (4). To obtain (6) we substitute Z by $X - Y$ in (7).
- Finally, we verify that (2) implies (3) and (5) implies (6). Both of these implications crucially depend on the ordering of X and Y obtained from the guard. For instance, knowing that $X \leq Y$ ensures that subtracting X from Y and then adding back X produces Y , as required by the first disjunct of (3). Similarly, knowing that $\neg (X \leq Y)$ ensures that subtracting Y from X and then adding back Y produces X , as needed by the second disjunct of (6). Note that $n - m + m = n$ does *not* hold for arbitrary natural numbers n and m (for example, $3 - 5 + 5 = 5$).

²⁰True

²¹True/ $X \leq Y$

²² $(Y - X) + X = Y \wedge (Y - X) + Y = X$

²³ $Z + X = Y \wedge Z + Y = X$

²⁴True/ $\neg (X \leq Y)$

²⁵ $(X - Y) + X = Y \wedge (X - Y) + Y = X$

²⁶ $Z + X = Y \wedge Z + Y = X$

²⁷ $Z + X = Y \wedge Z + Y = X$

Exercise: 2 stars, standard, optional (if_minus_plus_reloaded) N.b.: Although this exercise is marked optional, it is an excellent warm-up for the (non-optional) `if_minus_plus_correct` exercise below!

Fill in valid decorations for the following program: Briefly justify each use of $\Rightarrow$.

□

6.2.3 Example: Reduce to Zero

Here is a `while` loop that is so simple that `True` suffices as a loop invariant.

(1) ²⁸ while $X \neq 0$ do (2) ²⁹ $\Rightarrow$ (3) ³⁰ $X := X - 1$ (4) ³¹ end (5) ³² $\Rightarrow$ (6) ³³

The decorations can be constructed as follows:

- Start with the outer precondition (1) and postcondition (6).
- Following the format dictated by the `hoare_while` rule, we copy (1) to (4). We conjoin (1) with the guard to obtain (2). We also conjoin (1) with the negation of the guard to obtain (5).
- Because the final postcondition (6) does not syntactically match (5), we add an implication between them.
- Using the assignment rule with assertion (4), we trivially substitute and obtain assertion (3).
- We add the implication between (2) and (3).

Finally we check that the implications do hold; both are trivial.

6.2.4 Example: Division

Let's do one more example of simple reasoning about a loop.

The following Imp program calculates the integer quotient and remainder of parameters `m` and `n`.

$X := m; Y := 0; \text{while } n \leq X \text{ do } X := X - n; Y := Y + 1 \text{ end};$

If we replace `m` and `n` by concrete numbers and execute the program, it will terminate with the variable `X` set to the remainder when `m` is divided by `n` and `Y` set to the quotient.

In order to give a specification to this program we need to remember that dividing `m` by `n` produces a remainder `X` and a quotient `Y` such that $n \times Y + X = m \wedge X < n$.

²⁸`True`
²⁹`True /\ X <> 0`
³⁰`True`
³¹`True`
³²`True /\ ~(X <> 0)`
³³`X = 0`

It turns out that we get lucky with this program and don't have to think very hard about the loop invariant: the loop invariant is just the first conjunct, $n \times Y + X = m$, and we can use this to decorate the program.

(1) ³⁴ $\rightarrow$ (2) ³⁵ $X := m$; (3) ³⁶ $Y := 0$; (4) ³⁷ while $n \leq X$ do (5) ³⁸ $\rightarrow$ (6) ³⁹ $X := X - n$; (7) ⁴⁰ $Y := Y + 1$ (8) ⁴¹ end (9) ⁴² $\rightarrow$ (10) ⁴³

Assertions (4), (5), (8), and (9) are derived mechanically from the loop invariant and the loop's guard. Assertions (8), (7), and (6) are derived using the assignment rule going backwards from (8) to (6). Assertions (4), (3), and (2) are again backwards applications of the assignment rule.

Now that we've decorated the program it only remains to check that the uses of the consequence rule are correct – i.e., that (1) implies (2), that (5) implies (6), and that (9) implies (10). This is indeed the case:

- (1) $\rightarrow$ (2): trivial, by algebra.
- (5) $\rightarrow$ (6): because $n \leq X$, we are guaranteed that the subtraction in (6) does not get zero-truncated. We can therefore rewrite (6) as $n \times Y + n + X - n$ and cancel the n s, which results in the left conjunct of (5).
- (9) $\rightarrow$ (10): if $\neg (n \leq X)$ then $X < n$. That's straightforward from high-school algebra.

So, we have a valid decorated program.

6.2.5 From Decorated Programs to Formal Proofs

From an informal proof in the form of a decorated program, it is “easy in principle” to read off a formal proof using the Rocq theorems corresponding to the Hoare Logic rules, but these proofs can be a bit long and fiddly.

Note that we do *not* unfold the definition of `valid_hoare_triple` anywhere in this proof: the point of the game we're playing now is to use the Hoare rules as a self-contained logic for reasoning about programs.

For example... `Definition reduce_to_zero : com :=`

```
<{ while X ≠ 0 do
  X := X - 1
end }>.
```

```
34 True
35 n*0+m=m
36 n*0+X=m
37 n*Y+X=m
38 n*Y+X=m /\ n<=X
39 n*(Y+1)+(X-n)=m
40 n*(Y+1)+X=m
41 n*Y+X=m
42 n*Y+X=m /\ ~(n<=X)
43 n*Y+X=m /\ X<n
```

```

Theorem reduce_to_zero_correct' :
  {{True}}
  reduce_to_zero
  {{X = 0}}.
Proof.
  unfold reduce_to_zero.
  eapply hoare_consequence_post.
  - apply hoare_while.
    +
      eapply hoare_consequence_pre.
      × apply hoare_asgn.
      ×
        unfold assertion_sub, "-»". simpl. intros.
        exact l.
  -
    intros st [Inv GuardFalse].
    unfold bassertion in GuardFalse. simpl in GuardFalse.
    rewrite not_true_iff_false in GuardFalse.
    rewrite negb_false_iff in GuardFalse.
    apply eqb_eq in GuardFalse.
    apply GuardFalse.
Qed.

```

In **Hoare** we introduced a series of tactics named *assertion_auto* to automate proofs involving assertions.

The following declaration introduces a more sophisticated tactic that will help with proving assertions throughout the rest of this chapter. You don't need to understand the details, but briefly: it uses **split** repeatedly to turn all the conjunctions into separate subgoals, tries to use several theorems about booleans and (in)equalities, then uses **eauto** and *lia* to finish off as many subgoals as possible. What's left after *verify_assertion* does its thing should be just the “interesting parts” of the proof (which, if we're lucky, might be nothing at all!).

```

Ltac verify_assertion :=
  repeat split;
  simpl;
  unfold assert_implies;
  unfold bassertion in *; unfold beval in *; unfold aeval in *;
  unfold assertion_sub; intros;
  repeat (simpl in *;
    rewrite t_update_eq ||
    (try rewrite t_update_neq;
    [| (intro X; inversion X; fail)]));
  simpl in *;

```

```

repeat match goal with [H : _ ∧ _ ⊢ _] ⇒
    destruct H end;
repeat rewrite not_true_iff_false in *;
repeat rewrite not_false_iff_true in *;
repeat rewrite negb_true_iff in *;
repeat rewrite negb_false_iff in *;
repeat rewrite eqb_eq in *;
repeat rewrite eqb_neq in *;
repeat rewrite leb_iff in *;
repeat rewrite leb_iff_conv in *;
try subst;
simpl in *;
repeat
    match goal with
    [st : state ⊢ _] ⇒
        match goal with
        | [H : st _ = _ ⊢ _] ⇒
            rewrite → H in *; clear H
        | [H : _ = st _ ⊢ _] ⇒
            rewrite ← H in *; clear H
        end
    end;
try eauto;
try lia.

```

This makes it pretty easy to verify `reduce_to_zero`:

Theorem `reduce_to_zero_correct`''' :

```

{{True}}
  reduce_to_zero
{{X = 0}}.

```

Proof.

```

unfold reduce_to_zero.
eapply hoare_consequence_post.
- apply hoare_while.
  + eapply hoare_consequence_pre.
    × apply hoare_asgn.
    × verify_assertion.
- verify_assertion.

```

Qed.

This example shows that it is conceptually straightforward to read off the main elements of a formal proof from a decorated program. Indeed, the process is so straightforward that it can be automated, as we will see next.

6.3 Formal Decorated Programs

Our informal conventions for decorated programs amount to a way of “displaying” Hoare triples, in which commands are annotated with enough embedded assertions that checking the validity of a triple is reduced to simple logical and algebraic calculations showing that some assertions imply others.

In this section, we show that this presentation style can be made completely formal – and indeed that checking the validity of decorated programs can be largely automated.

6.3.1 Syntax

The first thing we need to do is to formalize a variant of the syntax of Imp commands that includes embedded assertions, which we’ll call “decorations.” We call the new commands *decorated commands*, or **dcoms**.

The choice of exactly where to put assertions in the definition of **dcom** is a bit subtle. The simplest thing to do would be to annotate every **dcom** with a precondition and postcondition – something like this...

```
Module DCOMFIRSTTRY.
```

```
Inductive dcom : Type :=
```

```
| DCSkip (P : Assertion)
```

```
| DCSeq (P : Assertion) (d1 : dcom) (Q : Assertion)
      (d2 : dcom) (R : Assertion)
```

```
| DCAsgn (X : string) (a : aexp) (Q : Assertion)
```

```
| DCIf (P : Assertion) (b : bexp) (P1 : Assertion) (d1 : dcom)
      (P2 : Assertion) (d2 : dcom) (Q : Assertion)
```

```
| DCWhile (P : Assertion) (b : bexp)
      (P1 : Assertion) (d : dcom) (P2 : Assertion)
      (Q : Assertion)
```

```
| DCPre (P : Assertion) (d : dcom)
```

```
| DCPost (d : dcom) (Q : Assertion).
```

```
End DCOMFIRSTTRY.
```

But this would result in *very* verbose decorated programs with a lot of repeated annotations: a simple program like *skip;skip* would be decorated like this,

44 (45 skip 46) ; (47 skip 48) 49

with pre- and post-conditions around each *skip*, plus identical pre- and post-conditions on the semicolon!

In other words, we don't want both preconditions and postconditions on each command, because a sequence of two commands would contain redundant decorations—the postcondition of the first likely being the same as the precondition of the second.

Instead, our formal syntax of decorated commands will omit preconditions whenever possible and embed just postconditions.

- The *skip* command, for example, is decorated only with its postcondition

skip 50

on the assumption that the precondition will be provided by somebody else.

We carry the same assumption through the other syntactic forms: each decorated command is assumed to carry its own postcondition within itself but take its precondition from its context in which it is used.

- Sequences *d1* ; *d2* need no additional decorations.

Why?

Because inside *d2* there will be a postcondition, which also serves as the postcondition of *d1*;*d2*.

Similarly, inside *d1* there will also be a postcondition, which additionally serves as the *precondition* for *d2*.

- An assignment *X* := *a* is decorated only with its postcondition:

X := *a* 51

- A conditional *if b then d1 else d2* is decorated with a postcondition for the entire statement, as well as preconditions for each branch:

if *b* then 52 *d1* else 53 *d2* end 54

44 P
45 P
46 P
47 P
48 P
49 P
50 Q
51 Q
52 P₁
53 P₂
54 Q

- A loop `while b do d end` is decorated with its final postcondition plus a precondition for the body:

`while b do 55 d end 56`

The postcondition embedded in `d` serves as the loop invariant.

- Implications $\rightarrow$ can be added as decorations either for a precondition...

`-  $\rightarrow$  57 d`

...or for a postcondition:

`d  $\rightarrow$  58`

The former is waiting for another precondition to be supplied by the context; the latter relies on the postcondition already embedded in `d`.

Putting this all together gives us the formal syntax of decorated commands:

```
Inductive dcom : Type :=
| DCSkip (Q : Assertion)

| DCSeq (d1 d2 : dcom)

| DCAsgn (X : string) (a : aexp) (Q : Assertion)

| DCIf (b : bexp) (P1 : Assertion) (d1 : dcom)
      (P2 : Assertion) (d2 : dcom) (Q : Assertion)

| DCWhile (b : bexp) (P : Assertion) (d : dcom)
      (Q : Assertion)

| DCPre (P : Assertion) (d : dcom)

| DCPost (d : dcom) (Q : Assertion)

.
```

To provide the initial precondition that goes at the very top of a decorated program, we introduce a new type **decorated**:

```
Inductive decorated : Type :=
| Decorated : Assertion  $\rightarrow$  dcom  $\rightarrow$  decorated.
```

⁵⁵P
⁵⁶Q
⁵⁷P
⁵⁸Q

To avoid clashing with the existing **Notations** for ordinary commands, we introduce these notations in a new grammar scope called **dcom**.

Declare Scope dcom_scope.

Notation "'skip' '{{\$ P }}'" := (DCSkip P)

(in custom com at level 0,

P custom assn at level 99,

format "'[v' 'skip' '/' '{{\$ P }}']'" : dcom_scope.

Notation "l ':=' a '{{\$ P }}'" := (DCAsgn l a P)

(in custom com at level 0,

l constr at level 0,

a custom com at level 85,

P custom assn at level 99,

no associativity,

format "'[v' l ':=' a '/' '{{\$ P }}']'" : dcom_scope.

Notation "'while' b 'do' '{{\$ Pbody }}' d 'end' '{{\$ Ppost }}'" := (DCWhile b Pbody d Ppost)

(in custom com at level 89,

b custom com at level 99,

Pbody custom assn at level 99,

Ppost custom assn at level 99,

format "'[v' 'while' b 'do' '/' '{{\$ Pbody }}' '/' d '/' 'end' '/' '{{\$ Ppost }}']'" :

dcom_scope.

Notation "'if' b 'then' {{\$ P1 }} d1 'else' {{\$ P2 }} d2 'end' {{\$ Q }}" := (DCIf b P1 d1 P2 d2 Q)

(in custom com at level 89,

b custom com at level 99,

P1 custom assn at level 99,

P2 custom assn at level 99,

Q custom assn at level 99,

format "'[v' 'if' b 'then' '/' '{{\$ P1 }}' '/' d1 '/' 'else' '/' '{{\$ P2 }}' '/' d2 '/' 'end' '/' '{{\$ Q }}']'" : dcom_scope.

Notation "'->' {{\$ P }} d"

:= (DCPre P d)

(in custom com at level 12, right associativity, P custom assn at level 99)

: dcom_scope.

Notation "d '->' {{\$ P }}"

:= (DCPost d P)

(in custom com at level 10, right associativity, P custom assn at level 99)

: dcom_scope.

Notation "x ; y" := (DCSeq x y)

```

(in custom com at level 90,
  right associativity,
  format "[v' x ; '/' y ']'") : dcom_scope.
Notation "{{ P }}" d := (Decorated P d)
(in custom com at level 91,
  P custom assn at level 99,
  format "[v' '{{ P }}' '/' d ']'") : dcom_scope.

```

Local Open Scope dcom_scope.

Example dec0 : **dcom** :=
 <{ skip {{ True }} }>.

Example dec1 : **dcom** :=
 <{ while true do {{ True }} skip {{ True }} end {{ True }} }>.

Recall that you can Set Printing All to see how all that notation is desugared. Set
 Printing All.

Print dec1.

Unset Printing All.

An example **decorated** program that decrements **X** to 0:

```

Example dec_while : decorated :=
  <{
    {{ True }}
    while X ≠ 0
    do
      {{ True ∧ (X ≠ 0) }}
      X := X - 1
      {{ True }}
    end
    {{ True ∧ X = 0 }} ->
    {{ X = 0 }} }>.

```

It is easy to go from a **dcom** to a **com** by erasing all annotations.

```

Fixpoint erase (d : dcom) : com :=
  match d with
  | DCSkip _ => CSkip
  | DCSeq d1 d2 => CSeq (erase d1) (erase d2)
  | DCAsgn X a _ => CAsgn X a
  | DCIf b _ d1 _ d2 _ => CIf b (erase d1) (erase d2)
  | DCWhile b _ d _ => CWhile b (erase d)
  | DCPre _ d => erase d
  | DCPPost d _ => erase d
  end.

```

Definition erase_d (dec : **decorated**) : **com** :=


```

match dec with
| Decorated P d  $\Rightarrow$  erase d
end.

```

Example erase_while_ex :
 erase_d dec_while
 = <{while X $\neq$ 0 do X := X - 1 end}>.

Proof.
 unfold dec_while.
 reflexivity.

Qed.

It is also straightforward to extract the precondition and postcondition from a decorated program.

Definition precondition_from (*dec* : decorated) : Assertion :=
 match *dec* with
 | Decorated P d $\Rightarrow$ P
 end.

Fixpoint post (*d* : dcom) : Assertion :=
 match *d* with
 | DCSkip P $\Rightarrow$ P
 | DCSeq _ d2 $\Rightarrow$ post d2
 | DCAsgn _ _ Q $\Rightarrow$ Q
 | DCIf _ _ _ _ Q $\Rightarrow$ Q
 | DCWhile _ _ _ Q $\Rightarrow$ Q
 | DCPre _ d $\Rightarrow$ post *d*
 | DCPPost _ Q $\Rightarrow$ Q
 end.

Definition postcondition_from (*dec* : decorated) : Assertion :=
 match *dec* with
 | Decorated P d $\Rightarrow$ post d
 end.

Example precondition_from_while : precondition_from dec_while = True.

Proof. reflexivity. Qed.

Example postcondition_from_while : postcondition_from dec_while = {{ X = 0 }}.

Proof. reflexivity. Qed.

We can then express what it means for a decorated program to be correct as follows:

Definition outer_triple_valid (*dec* : decorated) :=
 {{\$(precondition_from *dec*)} } erase_d *dec* {{\$(postcondition_from *dec*)} }.

For example:

Example dec_while_triple_correct :

```

outer_triple_valid dec_while
=
  {{ True }}
  while X ≠ 0 do X := X - 1 end
  {{ X = 0 }}.

```

Proof. reflexivity. Qed.

The outer Hoare triple of a decorated program is just a **Prop**; thus, to show that it is *valid*, we need to produce a proof of this proposition.

We will do this by extracting “proof obligations” from the decorations sprinkled throughout the program.

These obligations are often called *verification conditions*, because they are the facts that must be verified to see that the decorations are locally consistent and thus constitute a proof of validity of the outer triple.

6.3.2 Extracting Verification Conditions

The function `verification_conditions` takes a decorated command d together with a precondition P and returns a *proposition* that, if it can be proved, implies that the triple

⁵⁹ `erase d` ⁶⁰

is valid.

It does this by walking over d and generating a big conjunction that includes

- local consistency checks for each form of command, plus
- uses of $\rightarrow$ to bridge the gap between the assertions found inside a decorated command and the assertions imposed by the external precondition; these uses correspond to applications of the consequence rule.

Local consistency is defined as follows...

- The decorated command

`skip` ⁶¹

is locally consistent with respect to a precondition P if $P \rightarrow Q$.

- The sequential composition of $d1$ and $d2$ is locally consistent with respect to P if $d1$ is locally consistent with respect to P and $d2$ is locally consistent with respect to the postcondition of $d1$.

⁵⁹ P

⁶⁰`postd`

⁶¹ Q

- An assignment

$X := a$ ⁶²

is locally consistent with respect to a precondition P if:

$P \multimap Q \mid X \mid \multimap a$

- A conditional

if b then ⁶³ $d1$ else ⁶⁴ $d2$ end ⁶⁵

is locally consistent with respect to precondition P if

(1) $P \wedge b \multimap P1$

(2) $P \wedge \neg b \multimap P2$

(3) $d1$ is locally consistent with respect to $P1$

(4) $d2$ is locally consistent with respect to $P2$

(5) $\text{post } d1 \multimap Q$

(6) $\text{post } d2 \multimap Q$

- A loop

while b do ⁶⁶ d end ⁶⁷

is locally consistent with respect to precondition P if:

(1) $P \multimap \text{post } d$

(2) $\text{post } d \wedge b \multimap Q$

(3) $\text{post } d \wedge \neg b \multimap R$

(4) d is locally consistent with respect to Q

- A command with an extra assertion at the beginning

$\multimap$ ⁶⁸ d

is locally consistent with respect to a precondition P if:

(1) $P \multimap Q$

(2) d is locally consistent with respect to Q

⁶² Q

⁶³ $P1$

⁶⁴ $P2$

⁶⁵ Q

⁶⁶ Q

⁶⁷ R

⁶⁸ Q

- A command with an extra assertion at the end

$d \multimap^{69}$

is locally consistent with respect to a precondition P if:

- (1) d is locally consistent with respect to P
- (2) $\text{post } d \multimap Q$

With all this in mind, we can write a *verification condition generator* that takes a decorated command and reads off a proposition saying that all its decorations are locally consistent.

Formally, since a decorated command is “waiting for its precondition” the main VC generator takes a **dcom** plus a given precondition as arguments.

```

Fixpoint verification_conditions (P : Assertion) (d : dcom) : Prop :=
  match d with
  | DCSkip Q =>
    (P -> Q)
  | DCSeq d1 d2 =>
    verification_conditions P d1
    ∧ verification_conditions (post d1) d2
  | DCAsgn X a Q =>
    P -> {{ Q [X |-> a] }}
  | DCIf b P1 d1 P2 d2 Q =>
    {{ P ∧ b }} -> P1
    ∧ {{ P ∧ ¬ b }} -> P2
    ∧ (post d1 -> Q) ∧ (post d2 -> Q)
    ∧ verification_conditions P1 d1
    ∧ verification_conditions P2 d2
  | DCWhile b Q d R =>
    (P -> post d)
    ∧ {{ $(post d) ∧ b }} -> Q
    ∧ {{ $(post d) ∧ ¬ b }} -> R
    ∧ verification_conditions Q d
  | DCPre P' d =>
    (P -> P')
    ∧ verification_conditions P' d
  | DCPost d Q =>
    verification_conditions P d
    ∧ (post d -> Q)
  end.

```

⁶⁹ Q

The following key theorem states that `verification_conditions` does its job correctly. Not surprisingly, each of the Hoare Logic rules plays a critical role at some point in the proof.

Theorem `verification_correct` : $\forall d P,$
`verification_conditions` $P d \rightarrow \{\{P\}\}$ erase $d \{\{ \$(post\ d) \}\}$.

Proof.

```
induction d; intros; simpl in *.
-
  eapply hoare_consequence_pre.
  + apply hoare_skip.
  + assumption.
-
  destruct H as [H1 H2].
  eapply hoare_seq.
  + apply IHd2. apply H2.
  + apply IHd1. apply H1.
-
  eapply hoare_consequence_pre.
  + apply hoare_asgn.
  + assumption.
-
  destruct H as [HPre1 [HPre2 [Hd1 [Hd2 [HThen HElse] ] ] ] ].
  apply IHd1 in HThen. clear IHd1.
  apply IHd2 in HElse. clear IHd2.
  apply hoare_if.
  + eapply hoare_consequence; eauto.
  + eapply hoare_consequence; eauto.
-
  destruct H as [Hpre [Hbody1 [Hpost1 Hd] ] ].
  eapply hoare_consequence; eauto.
  apply hoare_while.
  eapply hoare_consequence_pre; eauto.
-
  destruct H as [HP Hd].
  eapply hoare_consequence_pre; eauto.
-
  destruct H as [Hd HQ].
  eapply hoare_consequence_post; eauto.
```

Qed.

Now that all the pieces are in place, we can define what it means to verify an entire program.

Definition `verification_conditions_from`
 $(dec : \text{decorated}) : \text{Prop} :=$

```

match dec with
| Decorated P d  $\Rightarrow$  verification_conditions P d
end.

```

And this brings us to the main theorem of this section:

```

Corollary verification_conditions_correct :  $\forall$  dec,
  verification_conditions_from dec  $\rightarrow$ 
  outer_triple_valid dec.

```

Proof.

```

  intros [P d]. apply verification_correct.
Qed.

```

6.3.3 More Automation

The propositions generated by `verification_conditions` are fairly big and contain many conjuncts that are essentially trivial.

```

Eval simpl in verification_conditions_from dec_while.

```

Fortunately, our `verify_assertion` tactic can generally take care of most (or sometimes all) of them. Example `vc_dec_while` : `verification_conditions_from dec_while`.

Proof. `verify_assertion`. Qed.

To automate the overall process of verification, we can use `verification_correct` to extract the verification conditions, use `verify_assertion` to verify them as much as it can, and finally tidy up any remaining bits by hand. Ltac `verify` :=

```

  intros;
  apply verification_correct;
  verify_assertion.

```

Here's the final, formal proof that `dec_while` is correct.

```

Theorem dec_while_correct :
  outer_triple_valid dec_while.

```

Proof. `verify`. Qed.

Similarly, here is the formal decorated program for the “swapping by adding and subtracting” example that we saw earlier.

```

Definition swap_dec (m n:nat) : decorated :=
  <{
    {{ X = m  $\wedge$  Y = n }} ->
      {{ (X + Y) - ((X + Y) - Y) = n  $\wedge$  (X + Y) - Y = m }}
    X := X + Y
      {{ X - (X - Y) = n  $\wedge$  X - Y = m }};
    Y := X - Y
      {{ X - Y = n  $\wedge$  Y = m }};
    X := X - Y
  }

```

```

  {{ X = n ∧ Y = m }}
}>.

```

Theorem swap_correct : $\forall m\ n,$
 outer_triple_valid (swap_dec $m\ n$).

Proof. verify. Qed.

And here is the formal decorated version of the “positive difference” program from earlier:

Definition positive_difference_dec :=

```

<{
  {{True}}
  if X ≤ Y then
    {{True ∧ X ≤ Y}} ->
    {{(Y - X) + X = Y ∨ (Y - X) + Y = X}}
    Z := Y - X
    {{Z + X = Y ∨ Z + Y = X}}
  else
    {{True ∧ ¬(X ≤ Y)}} ->
    {{(X - Y) + X = Y ∨ (X - Y) + Y = X}}
    Z := X - Y
    {{Z + X = Y ∨ Z + Y = X}}
  end
  {{Z + X = Y ∨ Z + Y = X}}
}>.

```

Theorem positive_difference_correct :
 outer_triple_valid positive_difference_dec.

Proof. verify. Qed.

Exercise: 2 stars, standard, especially useful (if_minus_plus_correct) Here is a skeleton of the formal decorated version of the `if_minus_plus` program that we saw earlier. Replace all occurrences of `FILL_IN_HERE` with appropriate assertions and fill in the proof (which should be just as straightforward as in the examples above).

Definition if_minus_plus_dec :=

```

<{
  {{True}}
  if (X ≤ Y) then
    {{ FILL_IN_HERE }} ->
    {{ FILL_IN_HERE }}
    Z := Y - X
    {{ FILL_IN_HERE }}
  else
    {{ FILL_IN_HERE }} ->
    {{ FILL_IN_HERE }}

```

```

    Y := X + Z
      {{ FILL_IN_HERE }}
end
{{ Y = X + Z }} >.
Theorem if_minus_plus_correct :
  outer_triple_valid if_minus_plus_dec.
Proof.
  Admitted.
□

```

Exercise: 2 stars, standard, optional (div_mod_outer_triple_valid) Fill in appropriate assertions for the division program from above.

Definition div_mod_dec (*a b* : nat) : decorated :=

```

<{
  {{ True }} ->
  {{ FILL_IN_HERE }}
  X := a
      {{ FILL_IN_HERE }};
  Y := 0
      {{ FILL_IN_HERE }};
  while b ≤ X do
    {{ FILL_IN_HERE }} ->
    {{ FILL_IN_HERE }}
    X := X - b
        {{ FILL_IN_HERE }};
    Y := Y + 1
        {{ FILL_IN_HERE }}
  end
  {{ FILL_IN_HERE }} ->
  {{ FILL_IN_HERE }} >.

```

Theorem div_mod_outer_triple_valid : $\forall$ *a b*,
 outer_triple_valid (div_mod_dec *a b*).

Proof.
 Admitted.
 □

6.4 Finding Loop Invariants

Once the outermost precondition and postcondition are chosen, the only creative part of a verifying program using Hoare Logic is finding the right loop invariants. The reason this is difficult is the same as the reason that inductive mathematical proofs are:

- Strengthening a *loop invariant* means that you have a stronger assumption to work with when trying to establish the postcondition of the loop body, but it also means that the loop body's postcondition is harder to prove.
- Similarly, strengthening an *induction hypothesis* means that you have a stronger assumption to work with when trying to complete the induction step of the proof, but it also means that the statement being proved inductively is harder to prove.

This section explains how to approach the challenge of finding loop invariants through a series of examples and exercises.

6.4.1 Example: Slow Subtraction

The following program subtracts the value of X from the value of Y by repeatedly decrementing both X and Y . We want to verify its correctness with respect to the pre- and postconditions shown:

⁷⁰ while $X \neq 0$ do $Y := Y - 1$; $X := X - 1$ end ⁷¹

To verify this program, we need to find an invariant *Inv* for the loop. As a first step we can leave *Inv* as an unknown and build a *skeleton* for the proof by applying the rules for local consistency, working from the end of the program to the beginning, as usual, and without doing any thinking at all yet.

This leads to the following skeleton:

(1) ⁷² $\rightarrow$ (a) (2) ⁷³ while $X \neq 0$ do (3) ⁷⁴ $\rightarrow$ (c) (4) ⁷⁵ $Y := Y - 1$; (5) ⁷⁶ $X := X - 1$ (6) ⁷⁷ end (7) ⁷⁸ $\rightarrow$ (b) (8) ⁷⁹ Examining this skeleton, we can see that any valid *Inv* will have to respect three conditions:

- (a) it must be *weak* enough to be implied by the loop's precondition, i.e., (1) must imply (2);
- (b) it must be *strong* enough to imply the program's postcondition, i.e., (7) must imply (8);
- (c) it must be *preserved* by a single iteration of the loop, assuming that the loop guard also evaluates to true, i.e., (3) must imply (4).

⁷⁰ $X=m \wedge Y=n$

⁷¹ $Y=n-m$

⁷² $X=m \wedge Y=n$

⁷³ *Inv*

⁷⁴ $Inv \wedge X \neq 0$

⁷⁵ $Inv[X \rightarrow X-1] [Y \rightarrow Y-1]$

⁷⁶ $Inv[X \rightarrow X-1]$

⁷⁷ *Inv*

⁷⁸ $Inv \wedge \neg (X \neq 0)$

⁷⁹ $Y=n-m$

These conditions are actually independent of the particular program and specification we are considering: every loop invariant has to satisfy them.

One way to find a loop invariant that simultaneously satisfies these three conditions is by using an iterative process: start with a “candidate” invariant (e.g., a guess or a heuristic choice) and check the three conditions above; if any of the checks fails, try to use the information that we get from the failure to produce another – hopefully better – candidate invariant, and repeat.

For instance, in the reduce-to-zero example above, we saw that, for a very simple loop, choosing **True** as a loop invariant did the job. Maybe it will work here too. To find out, let’s try instantiating *Inv* with **True** in the skeleton above and see what we get...

(1) ⁸⁰ -» (a - OK) (2) ⁸¹ while $X \neq 0$ do (3) ⁸² -» (c - OK) (4) ⁸³ $Y := Y - 1$; (5) ⁸⁴ $X := X - 1$ (6) ⁸⁵ end (7) ⁸⁶ -» (b - WRONG!) (8) ⁸⁷

While conditions (a) and (c) are trivially satisfied, (b) is wrong: it is not the case that **True** $\wedge X = 0$ (7) implies $Y = n - m$ (8). In fact, the two assertions are completely unrelated, so it is very easy to find a counterexample to the implication (say, $Y = X = m = 0$ and $n = 1$).

If we want (b) to hold, we need to strengthen the loop invariant so that it implies the postcondition (8). One simple way to do this is to let the loop invariant *be* the postcondition. So let’s return to our skeleton, instantiate *Inv* with $Y = n - m$, and try checking conditions (a) to (c) again.

(1) ⁸⁸ -» (a - WRONG!) (2) ⁸⁹ while $X \neq 0$ do (3) ⁹⁰ -» (c - WRONG!) (4) ⁹¹ $Y := Y - 1$; (5) ⁹² $X := X - 1$ (6) ⁹³ end (7) ⁹⁴ -» (b - OK) (8) ⁹⁵

This time, condition (b) holds trivially, but (a) and (c) are broken. Condition (a) requires that (1) $X = m \wedge Y = n$ implies (2) $Y = n - m$. If we substitute Y by n we have to show that $n = n - m$ for arbitrary m and n , which is not the case (for instance, when $m = n = 1$). Condition (c) requires that $n - m - 1 = n - m$, which fails, for instance, for $n = 1$ and $m = 0$. So, although $Y = n - m$ holds at the end of the loop, it does not hold from the start, and it doesn’t hold on each iteration; it is not a correct loop invariant.

⁸⁰ $X=m/\backslash Y=n$
⁸¹ **True**
⁸² $\text{True}/\backslash X \neq 0$
⁸³ **True**
⁸⁴ **True**
⁸⁵ **True**
⁸⁶ $\text{True}/\backslash \sim (X \neq 0)$
⁸⁷ $Y=n-m$
⁸⁸ $X=m/\backslash Y=n$
⁸⁹ $Y=n-m$
⁹⁰ $Y=n-m/\backslash X \neq 0$
⁹¹ $Y-1=n-m$
⁹² $Y=n-m$
⁹³ $Y=n-m$
⁹⁴ $Y=n-m/\backslash \sim (X \neq 0)$
⁹⁵ $Y=n-m$

This failure is not very surprising: the variable Y changes during the loop, while m and n are constant, so the assertion we chose didn't have much chance of being a loop invariant!

To do better, we need to generalize (7) to some statement that is equivalent to (8) when X is 0, since this will be the case when the loop terminates, and that “fills the gap” in some appropriate way when X is nonzero. Looking at how the loop works, we can observe that X and Y are decremented together until X reaches 0. So, if $X = 2$ and $Y = 5$ initially, after one iteration of the loop we obtain $X = 1$ and $Y = 4$; after two iterations $X = 0$ and $Y = 3$; and then the loop stops. Notice that the difference between Y and X stays constant between iterations: initially, $Y = n$ and $X = m$, and the difference is always $n - m$. So let's try instantiating *Inv* in the skeleton above with $Y - X = n - m$.

(1) ⁹⁶ $\rightarrow$ (a - OK) (2) ⁹⁷ while $X \neq 0$ do (3) ⁹⁸ $\rightarrow$ (c - OK) (4) ⁹⁹ $Y := Y - 1$; (5) ¹⁰⁰ $X := X - 1$ (6) ¹⁰¹ end (7) ¹⁰² $\rightarrow$ (b - OK) (8) ¹⁰³

Success! Conditions (a), (b) and (c) all hold now. (To verify (c), we need to check that, under the assumption that $X \neq 0$, we have $Y - X = (Y - 1) - (X - 1)$; this holds for all natural numbers X and Y .)

Here is the final version of the decorated program:

```
Example subtract_slowly_dec (m : nat) (n : nat) : decorated :=
  <{
    {{ X = m ∧ Y = n }} ->
    {{ Y - X = n - m }}
    while X ≠ 0 do
      {{ Y - X = n - m ∧ X ≠ 0 }} ->
      {{ (Y - 1) - (X - 1) = n - m }}
      Y := Y - 1
      {{ Y - (X - 1) = n - m }} ;
      X := X - 1
      {{ Y - X = n - m }}
    end
    {{ Y - X = n - m ∧ X = 0 }} ->
    {{ Y = n - m }} >.
```

Theorem subtract_slowly_outer_triple_valid : $\forall m n$,
outer_triple_valid (subtract_slowly_dec m n).

Proof.

verify. Qed.

⁹⁶ $X=m/\backslash Y=n$
⁹⁷ $Y-X=n-m$
⁹⁸ $Y-X=n-m/\backslash X \neq 0$
⁹⁹ $(Y-1)-(X-1)=n-m$
¹⁰⁰ $Y-(X-1)=n-m$
¹⁰¹ $Y-X=n-m$
¹⁰² $Y-X=n-m/\backslash \sim (X \neq 0)$
¹⁰³ $Y=n-m$

6.4.2 Exercise: Slow Assignment

Exercise: 2 stars, standard (slow_assignment) A roundabout way of assigning a number currently stored in X to the variable Y is to start Y at 0, then decrement X until it hits 0, incrementing Y at each step. Here is a program that implements this idea. Fill in decorations and prove the decorated program correct. (The proof should be very simple.)

Example `slow_assignment_dec (m : nat) : decorated :=`

```
<{
  {{ X = m }}
  Y := 0
  {{ FILL_IN_HERE }} ->
  {{ FILL_IN_HERE }} ;
  while X ≠ 0 do
    {{ FILL_IN_HERE }} ->
    {{ FILL_IN_HERE }}
    X := X - 1
    {{ FILL_IN_HERE }} ;
    Y := Y + 1
    {{ FILL_IN_HERE }}
  end
  {{ FILL_IN_HERE }} ->
  {{ Y = m }}
}>.
```

Theorem `slow_assignment : ∀ m,`
`outer_triple_valid (slow_assignment_dec m).`

Proof. Admitted.

□

6.4.3 Example: Parity

Here is a cute way of computing the parity of a value initially stored in X , due to Daniel Cristofani.

¹⁰⁴ while $2 \leq X$ do $X := X - 2$ end ¹⁰⁵

The `parity` function used in the specification is defined in Rocq as follows:

```
Fixpoint parity x :=
  match x with
  | 0 => 0
  | 1 => 1
  | S (S x') => parity x'
  end.
```

¹⁰⁴ $X=m$

¹⁰⁵ $X=parity\,m$

The postcondition does not hold at the beginning of the loop, since $m = \text{parity } m$ does not hold for an arbitrary m , so we cannot hope to use that as a loop invariant. To find a loop invariant that works, let's think a bit about what this loop does. On each iteration it decrements X by 2, which preserves the parity of X . So the parity of X does not change, i.e., it is invariant. The initial value of X is m , so the parity of X is always equal to the parity of m . Using $\text{parity } X = \text{parity } m$ as an invariant we obtain the following decorated program:

```

106 -> (a - OK) 107 while 2 <= X do 108 -> (c - OK) 109 X := X - 2 110 end 111 -> (b - OK)
112

```

With this loop invariant, conditions (a), (b), and (c) are all satisfied. For verifying (b), we observe that, when $X < 2$, we have $\text{parity } X = X$ (we can easily see this in the definition of parity). For verifying (c), we observe that, when $2 \leq X$, we have $\text{parity } X = \text{parity } (X-2)$.

Exercise: 3 stars, standard, optional (parity) Translate the above informal decorated program into a formal one and prove it correct.

Hint: There are actually several possible loop invariants that all lead to good proofs; one that leads to a particularly simple proof is $\text{parity } X = \text{parity } m$ – or more formally, using the $\#$ syntax to lift the application of the parity function into the syntax of assertions, $\{\{ \# \text{parity } X = \# \text{parity } m \} \}$.

Definition `parity_dec (m:nat) : decorated :=`

```

<{
  {{ X = m }} ->
  {{ FILL_IN_HERE }}
  while 2 ≤ X do
    {{ FILL_IN_HERE }} ->
    {{ FILL_IN_HERE }}
    X := X - 2
    {{ FILL_IN_HERE }}
  end
  {{ FILL_IN_HERE }} ->
  {{ X = #parity m }} >.

```

If you use the suggested loop invariant, you may find the following lemmas helpful (as well as *leb_complete* and *leb_correct*).

Lemma `parity_ge_2` : $\forall x,$
 $2 \leq x \rightarrow$
 $\text{parity } (x - 2) = \text{parity } x.$

```

106 X=m
107 parityX=paritym
108 parityX=paritym/\2<=X
109 parity(X-2)=paritym
110 parityX=paritym
111 parityX=paritym/\~(2<=X)
112 X=paritym

```

Proof.

```
destruct x; intros; simpl.
- reflexivity.
- destruct x; simpl.
  + lia.
  + rewrite sub_0_r. reflexivity.
```

Qed.

Lemma parity_lt_2 : $\forall x$,

```
 $\neg 2 \leq x \rightarrow$ 
parity  $x = x$ .
```

Proof.

```
induction x; intros; simpl.
- reflexivity.
- destruct x.
  + reflexivity.
  + lia.
```

Qed.

Theorem parity_outer_triple_valid : $\forall m$,

```
outer_triple_valid (parity_dec  $m$ ).
```

Proof.

```
Admitted.
```

□

6.4.4 Example: Finding Square Roots

The following program computes the integer square root of X by naive iteration:

```
113  $Z := 0$ ; while  $(Z+1)*(Z+1) \leq X$  do  $Z := Z+1$  end 114
```

As we did before, we can try to use the postcondition as a candidate loop invariant, obtaining the following decorated program:

```
(1) 115 -» (a - second conjunct of (2) WRONG!) (2) 116  $Z := 0$  (3) 117; while  $(Z+1)*(Z+1) \leq X$  do (4) 118 -» (c - WRONG!) (5) 119  $Z := Z+1$  (6) 120 end (7) 121 -» (b - OK) (8) 122
```

This didn't work very well: conditions (a) and (c) both failed. Looking at condition (c), we see that the second conjunct of (4) is almost the same as the first conjunct of (5),

¹¹³ $X=m$

¹¹⁴ $Z * Z \leq m \wedge m < (Z+1) * (Z+1)$

¹¹⁵ $X=m$

¹¹⁶ $0 * 0 \leq m \wedge m < (0+1) * (0+1)$

¹¹⁷ $Z * Z \leq m \wedge m < (Z+1) * (Z+1)$

¹¹⁸ $Z * Z \leq m \wedge m < (Z+1) * (Z+1) \wedge (Z+1) * (Z+1) \leq X$

¹¹⁹ $(Z+1) * (Z+1) \leq m \wedge m < ((Z+1)+1) * ((Z+1)+1)$

¹²⁰ $Z * Z \leq m \wedge m < (Z+1) * (Z+1)$

¹²¹ $Z * Z \leq m \wedge m < (Z+1) * (Z+1) \wedge \neg ((Z+1) * (Z+1) \leq X)$

¹²² $Z * Z \leq m \wedge m < (Z+1) * (Z+1)$

except that (4) mentions X while (5) mentions m . But note that X is never assigned in this program, so we should always have $X=m$. We didn't propagate this information from (1) into the loop invariant, but we could!

Also, we don't need the second conjunct of (8), since we can obtain it from the negation of the guard – the third conjunct in (7) – again under the assumption that $X=m$. This allows us to simplify a bit.

So we now try $X=m \wedge Z \times Z \leq m$ as the loop invariant:

¹²³ $\rightarrow$ (a - OK) ¹²⁴ $Z := 0$ ¹²⁵; while $(Z+1) \times (Z+1) \leq X$ do ¹²⁶ $\rightarrow$ (c - OK) ¹²⁷ $Z := Z + 1$ ¹²⁸ end ¹²⁹ $\rightarrow$ (b - OK) ¹³⁰

This works, since conditions (a), (b), and (c) are now all rather trivially satisfied.

Very often, when a variable is used in a loop in a read-only fashion (i.e., it is referred to by the program or by the specification, and it is not changed by the loop), it is necessary to record the *fact* that it doesn't change in the loop invariant.

Exercise: 3 stars, standard, optional (sqrt) Translate the above informal decorated program into a formal one and prove it correct.

Hint: The loop invariant here must ensure that Z^2 is consistently less than or equal to X .

Definition `sqrt_dec (m:nat) : decorated :=`

```
<{
  {{ X = m }}  $\rightarrow$ 
  {{ FILL_IN_HERE }}
  Z := 0
  {{ FILL_IN_HERE }};
  while ((Z+1) × (Z+1) ≤ X) do
    {{ FILL_IN_HERE }}  $\rightarrow$ 
    {{ FILL_IN_HERE }}
    Z := Z + 1
    {{ FILL_IN_HERE }}
  end
  {{ FILL_IN_HERE }}  $\rightarrow$ 
  {{ Z × Z ≤ m ∧ m < (Z+1) × (Z+1) }}
}>.
```

Theorem `sqrt_correct : $\forall m$,
outer_triple_valid (sqrt_dec m).`

¹²³ $X=m$
¹²⁴ $X=m/\wedge 0*0 \leq m$
¹²⁵ $X=m/\wedge Z*Z \leq m$
¹²⁶ $X=m/\wedge Z*Z \leq m/\wedge (Z+1)*(Z+1) \leq X$
¹²⁷ $X=m/\wedge (Z+1)*(Z+1) \leq m$
¹²⁸ $X=m/\wedge Z*Z \leq m$
¹²⁹ $X=m/\wedge Z*Z \leq m/\wedge \sim((Z+1)*(Z+1) \leq X)$
¹³⁰ $Z*Z \leq m/\wedge m < (Z+1)*(Z+1)$

Proof. Admitted.

6.4.5 Example: Squaring

Here is a program that squares X by repeated addition:

¹³¹ $Y := 0; Z := 0; \text{ while } Y \neq X \text{ do } Z := Z + X; Y := Y + 1 \text{ end}$ ¹³²

The first thing to note is that the loop reads X but doesn't change its value. As we saw in the previous example, it can be a good idea in such cases to add $X = m$ to the loop invariant. The other thing that we know is often useful in the loop invariant is the postcondition, so let's add that too, leading to the candidate loop invariant $Z = m \times m \wedge X = m$.

¹³³ $\neg \gg (\text{a} - \text{WRONG})$ ¹³⁴ $Y := 0$ ¹³⁵; $Z := 0$ ¹³⁶; $\text{ while } Y \neq X \text{ do}$ ¹³⁷ $\neg \gg (\text{c} - \text{WRONG})$
¹³⁸ $Z := Z + X$ ¹³⁹; $Y := Y + 1$ ¹⁴⁰ end ¹⁴¹ $\neg \gg (\text{b} - \text{OK})$ ¹⁴²

Conditions (a) and (c) fail because of the $Z = m \times m$ part. While Z starts at 0 and works itself up to $m \times m$, we can't expect Z to be $m \times m$ from the start. If we look at how Z progresses in the loop, after the 1st iteration $Z = m$, after the 2nd iteration $Z = 2 \times m$, and at the end $Z = m \times m$. Since the variable Y tracks how many times we go through the loop, this leads us to derive a new loop invariant candidate: $Z = Y \times m \wedge X = m$.

¹⁴³ $\neg \gg (\text{a} - \text{OK})$ ¹⁴⁴ $Y := 0$ ¹⁴⁵; $Z := 0$ ¹⁴⁶; $\text{ while } Y \neq X \text{ do}$ ¹⁴⁷ $\neg \gg (\text{c} - \text{OK})$ ¹⁴⁸ $Z := Z + X$ ¹⁴⁹; $Y := Y + 1$ ¹⁵⁰ end ¹⁵¹ $\neg \gg (\text{b} - \text{OK})$ ¹⁵²

This new loop invariant makes the proof go through: all three conditions are easy to check.

It is worth comparing the postcondition $Z = m \times m$ and the $Z = Y \times m$ conjunct of the loop invariant. It is often the case that one has to replace parameters with variables – or

¹³¹ $X = m$
¹³² $Z = m * m$
¹³³ $X = m$
¹³⁴ $0 = m * m / \setminus X = m$
¹³⁵ $0 = m * m / \setminus X = m$
¹³⁶ $Z = m * m / \setminus X = m$
¹³⁷ $Z = m * m / \setminus X = m / \setminus Y \neq X$
¹³⁸ $Z + X = m * m / \setminus X = m$
¹³⁹ $Z = m * m / \setminus X = m$
¹⁴⁰ $Z = m * m / \setminus X = m$
¹⁴¹ $Z = m * m / \setminus X = m / \setminus \sim (Y \neq X)$
¹⁴² $Z = m * m$
¹⁴³ $X = m$
¹⁴⁴ $0 = 0 * m / \setminus X = m$
¹⁴⁵ $0 = Y * m / \setminus X = m$
¹⁴⁶ $Z = Y * m / \setminus X = m$
¹⁴⁷ $Z = Y * m / \setminus X = m / \setminus Y \neq X$
¹⁴⁸ $Z + X = (Y + 1) * m / \setminus X = m$
¹⁴⁹ $Z = (Y + 1) * m / \setminus X = m$
¹⁵⁰ $Z = Y * m / \setminus X = m$
¹⁵¹ $Z = Y * m / \setminus X = m / \setminus \sim (Y \neq X)$
¹⁵² $Z = m * m$

with expressions involving both variables and parameters, like `m - Y` – when going from postconditions to loop invariants.

□

6.4.6 Exercise: Factorial

Exercise: 4 stars, advanced (factorial_correct) Recall that `n!` denotes the factorial of `n` (i.e., `n! = 1*2*...*n`). Formally, the factorial function is defined recursively in the Rocq standard library in a way that is equivalent to the following:

Fixpoint fact (n : nat) : nat := match n with | 0 => 1 | S n' => n * (fact n') end.

Compute fact 5.

First, write the Imp program `factorial` that calculates the factorial of the number initially stored in the variable `X` and puts it in the variable `Y`.

Using your definition `factorial` and `slow_assignment_dec` as a guide, write a formal decorated program `factorial_dec` that implements the factorial function. Hint: recall the use of `#` in assertions to apply a function to an Imp variable.

Fill in the blanks and finish the proof of correctness. Bear in mind that we are working with natural numbers, for which both division and subtraction can behave differently than with real numbers. Excluding both operations from your loop invariant is advisable!

Then state a theorem named `factorial_correct` that says `factorial_dec` is correct, and prove the theorem. If all goes well, *verify* will leave you with just two subgoals, each of which requires establishing some mathematical property of `fact`, rather than proving anything about your program.

Hint: if those two subgoals become tedious to prove, give some thought to how you could restate your assertions such that the mathematical operations are more amenable to manipulation in Rocq. For example, recall that `1 + ...` is easier to work with than `... + 1`.

Example factorial_dec (m:nat) : decorated
. Admitted.

Theorem factorial_correct: $\forall m,$
outer_triple_valid (factorial_dec m).

Proof. Admitted.

□

6.4.7 Exercise: Minimum

Exercise: 3 stars, advanced (minimum_correct) Fill in decorations for the following program and prove them correct. As with `factorial`, be careful about mathematical reasoning involving natural numbers, especially subtraction.

Also, remember that applications of Rocq functions in assertions need an *ap* or *ap2* to be parsed correctly. E.g., `min a b` needs to be written *ap2 min a b* in an assertion.

You may find *andb_true_eq* useful (perhaps after using symmetry to get an equality the right way around).

Definition minimum_dec (*a b* : nat) : decorated :=

```
<{
  {{ True }} ->
  {{ FILL_IN_HERE }}
  X := a
      {{ FILL_IN_HERE }};
  Y := b
      {{ FILL_IN_HERE }};
  Z := 0
      {{ FILL_IN_HERE }};
  while X ≠ 0 && Y ≠ 0 do
    {{ FILL_IN_HERE }} ->
    {{ FILL_IN_HERE }}
    X := X - 1
        {{ FILL_IN_HERE }};
    Y := Y - 1
        {{ FILL_IN_HERE }};
    Z := Z + 1
        {{ FILL_IN_HERE }}
  end
  {{ FILL_IN_HERE }} ->
  {{ Z = #min a b }}
}>.
```

Theorem minimum_correct : $\forall a b$,
outer_triple_valid (minimum_dec *a b*).

Proof. Admitted.

□

6.4.8 Exercise: Two Loops

Exercise: 3 stars, standard (two_loops) Here is a pretty inefficient way of adding 3 numbers:

X := 0; Y := 0; Z := c; while X <> a do X := X + 1; Z := Z + 1 end; while Y <> b do Y := Y + 1; Z := Z + 1 end

Show that it does what it should by completing the following decorated program.

Definition two_loops_dec (*a b c* : nat) : decorated :=

```
<{
  {{ True }} ->
  {{ FILL_IN_HERE }}
  X := 0
```

```

                                {{ FILL_IN_HERE }};
Y := 0
                                {{ FILL_IN_HERE }};
Z := c
                                {{ FILL_IN_HERE }};
while X ≠ a do
                                {{ FILL_IN_HERE }} -»
                                {{ FILL_IN_HERE }}
    X := X + 1
                                {{ FILL_IN_HERE }};
    Z := Z + 1
                                {{ FILL_IN_HERE }}
end
                                {{ FILL_IN_HERE }} -»
                                {{ FILL_IN_HERE }};
while Y ≠ b do
                                {{ FILL_IN_HERE }} -»
                                {{ FILL_IN_HERE }}
    Y := Y + 1
                                {{ FILL_IN_HERE }};
    Z := Z + 1
                                {{ FILL_IN_HERE }}
end
{{ FILL_IN_HERE }} -»
{{ Z = a + b + c }}
}>.

```

Theorem two_loops : $\forall a b c,$
 outer_triple_valid (two_loops_dec $a b c$).

Proof.

Admitted.

□

6.4.9 Exercise: Power Series

Exercise: 4 stars, standard, optional (dpow2) Here is a program that computes the series: $1 + 2 + 2^2 + \dots + 2^m = 2^{(m+1)} - 1$

$X := 0; Y := 1; Z := 1; \text{while } X <> m \text{ do } Z := 2 * Z; Y := Y + Z; X := X + 1 \text{ end}$

Turn this into a decorated program and prove it correct.

```

Fixpoint pow2 n :=
  match n with
  | 0 => 1
  | S n' => 2 × (pow2 n')

```

end.

```
Definition dpow2_dec (n : nat) :=
  <{
    {{ True }} ->
    {{ FILL_IN_HERE }}
    X := 0
        {{ FILL_IN_HERE }};
    Y := 1
        {{ FILL_IN_HERE }};
    Z := 1
        {{ FILL_IN_HERE }};
    while X ≠ n do
        {{ FILL_IN_HERE }} ->
        {{ FILL_IN_HERE }}
        Z := 2 × Z
            {{ FILL_IN_HERE }};
        Y := Y + Z
            {{ FILL_IN_HERE }};
        X := X + 1
            {{ FILL_IN_HERE }}
    end
    {{ FILL_IN_HERE }} ->
    {{ Y = #pow2 (n+1) - 1 }}
  >.
```

Some lemmas that you may find useful...

Lemma pow2_plus_1 : $\forall n$,
pow2 (n+1) = pow2 n + pow2 n.

Proof.

```
induction n; simpl.
- reflexivity.
- lia.
```

Qed.

Lemma pow2_le_1 : $\forall n$, pow2 n $\geq$ 1.

Proof.

```
induction n; simpl; [constructor | lia].
```

Qed.

The main correctness theorem:

Theorem dpow2_down_correct : $\forall n$,
outer_triple_valid (dpow2_dec n).

Proof.

Admitted.

□

Exercise: 2 stars, advanced, optional (fib_eqn) The Fibonacci function is usually written like this:

Fixpoint fib n := match n with | 0 => 1 | 1 => 1 | _ => fib (pred n) + fib (pred (pred n)) end.

This doesn't pass Rocq's termination checker, but here is a slightly clunkier definition that does:

```
Fixpoint fib n :=
  match n with
  | 0 => 1
  | S n' => match n' with
    | 0 => 1
    | S n'' => fib n' + fib n''
  end
end.
```

Prove that fib satisfies the following equation. You will need this as a lemma in the next exercise.

```
Lemma fib_eqn : ∀ n,
  n > 0 →
  fib n + fib (pred n) = fib (1 + n).
```

Proof.

Admitted.

□

Exercise: 4 stars, advanced, optional (fib) The following Imp program leaves the value of fib n in the variable Y when it terminates:

X := 1; Y := 1; Z := 1; while X <> 1 + n do T := Z; Z := Z + Y; Y := T; X := 1 + X
end

Fill in the following definition of dfib and prove that it satisfies this specification:

¹⁵³ dfib ¹⁵⁴

You will need many uses of ap in your assertions. If all goes well, your proof will be very brief.

Definition T : string := "T".

Definition dfib (n : nat) : decorated :=

```
<{
  {{ True }} ->
  {{ FILL_IN_HERE }}
```

¹⁵³True

¹⁵⁴Y=fibn

```

X := 1
    {{ FILL_IN_HERE }} ;
Y := 1
    {{ FILL_IN_HERE }} ;
Z := 1
    {{ FILL_IN_HERE }} ;
while X ≠ 1 + n do
    {{ FILL_IN_HERE }} -»
    {{ FILL_IN_HERE }}
    T := Z
        {{ FILL_IN_HERE }};
    Z := Z + Y
        {{ FILL_IN_HERE }};
    Y := T
        {{ FILL_IN_HERE }};
    X := 1 + X
        {{ FILL_IN_HERE }}
end
{{ FILL_IN_HERE }} -»
{{ Y = #fib n }}
}>.

```

Theorem dfib_correct : $\forall n$,
 outer_triple_valid (dfib n).

Proof.

Admitted.

□

Exercise: 5 stars, advanced, optional (improve_dcom) The formal decorated programs defined above are intended to look as similar as possible to the informal ones defined earlier. If we drop this requirement, we can eliminate almost all annotations, just requiring final postconditions and loop invariants to be provided explicitly. Do this – i.e., define a new version of dcom with as few annotations as possible and adapt the rest of the formal development leading up to the `verification_correct` theorem.

6.5 Weakest Preconditions (Optional)

Some preconditions are more interesting than others. For example, the Hoare triple

¹⁵⁵ $X := Y + 1$ ¹⁵⁶

¹⁵⁵False

¹⁵⁶ $X \leq 5$

is *not* very interesting: although it is perfectly valid, it tells us nothing useful. Since the precondition isn't satisfied by any state, it doesn't describe any situations where we can use the command $X := Y + 1$ to achieve the postcondition $X \leq 5$.

By contrast,

¹⁵⁷ $X := Y + 1$ ¹⁵⁸

has a useful precondition: it tells us that, if we can somehow create a situation in which we know that $Y \leq 4 \wedge Z = 0$, then running this command will produce a state satisfying the postcondition. However, this precondition is not as useful as it could be, because the $Z = 0$ clause in the precondition actually has nothing to do with the postcondition $X \leq 5$.

The *most* useful precondition for this command is this one:

¹⁵⁹ $X := Y + 1$ ¹⁶⁰

The assertion $Y \leq 4$ is called the *weakest precondition* of $X := Y + 1$ with respect to the postcondition $X \leq 5$.

Assertion $Y \leq 4$ is a *weakest precondition* of command $X := Y + 1$ with respect to postcondition $X \leq 5$. Think of *weakest* here as meaning “easiest to satisfy”: a weakest precondition is one that as many states as possible can satisfy.

P is a weakest precondition of command c for postcondition Q if

- P is a precondition, that is, $\{\{P\}\} c \{\{Q\}\}$; and
- P is at least as weak as all other preconditions, that is, if $\{\{P'\}\} c \{\{Q\}\}$ then $P' \multimap P$.

Note that weakest preconditions need not be unique. For example, $Y \leq 4$ was a weakest precondition above, but so are the logically equivalent assertions $Y < 5$, $Y \leq 2 \times 2$, etc. It is easy to show that any two weakest preconditions P and P' of a command c with respect to postcondition Q are logically equivalent; that is, $P \multimap P'$.

Definition $\text{is\_wp } P \ c \ Q :=$

$\{\{P\}\} c \{\{Q\}\} \wedge$
 $\forall P', \{\{P'\}\} c \{\{Q\}\} \rightarrow (P' \multimap P).$

Exercise: 1 star, standard, optional (wp) What are weakest preconditions of the following commands for the following postconditions?

- 1) ¹⁶¹ skip ¹⁶²
- 2) ¹⁶³ $X := Y + Z$ ¹⁶⁴

¹⁵⁷ $Y <= 4 \wedge Z = 0$

¹⁵⁸ $X <= 5$

¹⁵⁹ $Y <= 4$

¹⁶⁰ $X <= 5$

¹⁶¹ ?

¹⁶² $X = 5$

¹⁶³ ?

¹⁶⁴ $X = 5$

- 3) ¹⁶⁵ $X := Y$ ¹⁶⁶
- 4) ¹⁶⁷ if $X = 0$ then $Y := Z + 1$ else $Y := W + 2$ end ¹⁶⁸
- 5) ¹⁶⁹ $X := 5$ ¹⁷⁰
- 6) ¹⁷¹ while true do $X := 0$ end ¹⁷²

Exercise: 3 stars, advanced, optional (is_wp) Prove formally, using the definition of `valid_hoare_triple`, that $Y \leq 4$ is indeed a weakest precondition of $X := Y + 1$ with respect to postcondition $X \leq 5$.

Note: we have to put parentheses around the inputs to `is_wp` to prevent Rocq from parsing those three things as a Hoare triple.

Theorem `is_wp_example` :

`is_wp ({Y ≤ 4}) (<X := Y + 1>) ({X ≤ 5}).`

Proof.

Admitted.

□

Exercise: 2 stars, advanced, optional (hoare_asgn_weakest) Show that the precondition in the rule `hoare_asgn` is in fact the weakest precondition.

Theorem `hoare_asgn_weakest` : $\forall Q X a,$

`is_wp ({Q [X ↦ a]}) <X := a> Q.`

Proof.

Admitted.

□

Exercise: 2 stars, advanced, optional (hoare_havoc_weakest) Show that your `havoc_pre` function from the *himp_hoare* exercise in the Hoare chapter returns a weakest precondition. `Module HIMP2.`

`Import HIMP.`

Lemma `hoare_havoc_weakest` : $\forall (P Q : \text{Assertion}) (X : \text{string}),$

`{P} havoc X {Q} →`

`P -> havoc_pre X Q.`

Proof.

Admitted.

`End HIMP2.`

¹⁶⁵ ?
¹⁶⁶ $X=Y$
¹⁶⁷ ?
¹⁶⁸ $Y=5$
¹⁶⁹ ?
¹⁷⁰ $X=0$
¹⁷¹ ?
¹⁷² $X=0$

□

Chapter 7

Library `PLF.HoareAsLogic`

7.1 `HoareAsLogic`: Hoare Logic as a Logic

The presentation of Hoare logic in chapter `Hoare` could be described as “model-theoretic”: the proof rules for each of the constructors were presented as *theorems* about the evaluation behavior of programs, and proofs of program correctness (validity of Hoare triples) were constructed by combining these theorems directly in Rocq.

Another way of presenting Hoare logic is to define a completely separate proof system – a set of axioms and inference rules that talk about commands, Hoare triples, etc. – and then say that a proof of a Hoare triple is a valid derivation in *that* logic. We can do this by giving an inductive definition of *valid derivations* in this new logic.

This chapter is optional. Before reading it, you’ll want to read the *ProofObjects* chapter in *Logical Foundations (Software Foundations, volume 1)*.

`Set Warnings "-deprecated-hint-without-locality,-deprecated-hint-without-locality,-parsing".`

`From PLF Require Import Maps.`

`From PLF Require Import Hoare.`

`Hint Constructors ceval : core.`

7.2 Hoare Logic and Model Theory

In `Hoare` we introduced Hoare triples, which contain a precondition, command, and postcondition. For example (and for the moment deliberately avoiding the notation we previously introduced),

Pre: $X = 0$ Com: $X := X + 1$ Post: $X = 1$

is a Hoare triple, as is

Pre: $X = 0$ Com: `skip` Post: $X = 1$

But there’s an important difference between those two triples: the former expresses a truth about how Imp programs execute, whereas the latter does not.

To capture that difference, we introduced a definition `valid_hoare_triple` that described when a triple expresses such a truth. Let's repeat that definition, but this time we'll call it `valid`:

```
Definition valid (P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st st',
    st =[ c ]=> st' →
    P st →
    Q st'.
```

This notion of *validity* is based on the underlying model of how Imp programs execute. That model itself is based on states. So,

Pre: $X = 0$ Com: $X := X + 1$ Post: $X = 1$

is *valid*, because starting from any state in which X is 0, and executing $X := X + 1$, we are guaranteed to reach a state in which X is 1. But,

Pre: $X = 0$ Com: `skip` Post: $X = 1$

is *invalid*, because starting from any state in which X is 0, we are guaranteed not to change the state, so X cannot be 1.

So far, we have punned between the syntax of a Hoare triple, written $\{\{P\}\} c \{\{Q\}\}$, and its validity, as expressed by `valid`. In essence, we have said that the semantic meaning of that syntax is the proposition returned by `valid`. This way of giving semantic meaning to something syntactic is part of the branch of mathematical logic known as *model theory*.

Our approach to Hoare logic through model theory led us to state proof rules in terms of that same state-based model, and to prove program correctness in it, too. But there is another approach, which is arguably more common in Hoare logic. We turn to it, next.

7.3 Hoare Logic and Proof Theory

Instead of using states and evaluation as the basis for reasoning, let's take the proof rules from `Hoare` as the basis. Those proof rules give us a set of axioms and inference rules that constitute a logic in their own right. We repeat them here:

(hoare_skip)	¹	skip	²

(hoare_asgn)	³	$X := a$	⁴
⁵	c_1	⁶	⁷
	c_2	⁸	

¹ P
² P
³ $Q[X] \rightarrow a$
⁴ Q
⁵ P
⁶ Q
⁷ Q
⁸ R

(hoare_seq) ⁹ c1; c2 ¹⁰
¹¹ c1 ¹² ¹³ c2 ¹⁴

(hoare_if) ¹⁵ if b then c1 else c2 end ¹⁶
¹⁷ c ¹⁸

(hoare_while) while b do c end P / b ¹⁹
²⁰ c ²¹ P -» P' Q' -» Q

(hoare_consequence) ²² c ²³

Read the Hoare triples in those rules as devoid of any meaning other than what the rules give them. Forget about states and evaluations. They are just syntax that the rules tell us how to manipulate in legal ways.

Through this new lens, triple $\{\{X = 0\}\} X := X + 1 \{\{X = 1\}\}$ is *derivable*, because we can derive a proof tree using the rules:

(hoare_asgn) X=0 -» X+1=1 ²⁴ X := X+1 ²⁵

(hoare_consequence) ²⁶ X := X+1 ²⁷

At each step we have either used one of the rules, or we have appealed to reasoning about assertions, which do not involve Hoare triples. (Note that we have left off the trivial part of [hoare_consequence](#) above, namely X=1 -» X=1, only because of horizontal space constraints: it's hard to fit that many characters on a line and have the page still be readable. If you prefer, think of it as using [hoare_consequence_pre](#) instead.)

⁹P
¹⁰R
¹¹P/\b
¹²Q
¹³P/\~b
¹⁴Q
¹⁵P
¹⁶Q
¹⁷P/\b
¹⁸P
¹⁹P
²⁰P',
²¹Q',
²²P
²³Q
²⁴X+1=1
²⁵X=1
²⁶X=0
²⁷X=1

On the other hand, $\{\{X = 0\}\} \text{ skip } \{\{X = 1\}\}$ is *not* derivable, because there is no way to apply the rules to construct a proof tree with this triple at its root.

This approach gives meaning to triples not in terms of a model, but in terms of how they can be used to construct proof trees. It's a different way of giving semantic meaning to something syntactic, and it's part of the branch of mathematical logic known as *proof theory*.

Our goal for the rest of this chapter is to formalize Hoare logic using proof theory, and then to prove that the model-theoretic and proof-theoretic formalizations are consistent with one another.

7.4 Derivability

To formalize derivability of Hoare triples, we introduce inductive type **derivable**, which describes legal proof trees using the Hoare rules.

```
Inductive derivable : Assertion → com → Assertion → Type :=
| H_Skip : ∀ P,
  derivable P <{skip}> P
| H_Asgn : ∀ Q V a,
  derivable ({Q [V |-> a]}) <{V := a}> Q
| H_Seq : ∀ P c Q d R,
  derivable Q d R → derivable P c Q → derivable P <{c; d}> R
| H_If : ∀ P Q b c1 c2,
  derivable (fun st ⇒ P st ∧ bassertion b st) c1 Q →
  derivable (fun st ⇒ P st ∧ ~(bassertion b st)) c2 Q →
  derivable P <{if b then c1 else c2 end}> Q
| H_While : ∀ P b c,
  derivable (fun st ⇒ P st ∧ bassertion b st) c P →
  derivable P <{while b do c end}> (fun st ⇒ P st ∧ ¬ (bassertion b st))
| H_Consequence : ∀ (P Q P' Q' : Assertion) c,
  derivable P' c Q' →
  (∀ st, P st → P' st) →
  (∀ st, Q' st → Q st) →
  derivable P c Q.
```

We don't need to include axioms corresponding to `hoare_consequence_pre` or `hoare_consequence_post`, because these can be proven easily from `H_Consequence`.

```
Lemma H_Consequence_pre : ∀ (P Q P' : Assertion) c,
  derivable P' c Q →
  (∀ st, P st → P' st) →
  derivable P c Q.
```

Proof. `eauto using H_Consequence. Qed.`

```
Lemma H_Consequence_post : ∀ (P Q Q' : Assertion) c,
```

derivable $P \ c \ Q' \rightarrow$
 $(\forall \ st, \ Q' \ st \rightarrow Q \ st) \rightarrow$
derivable $P \ c \ Q$.

Proof. eauto using H_Consequence. Qed.

As an example, let's construct a proof tree for

²⁸ $X := X + 1; X := X + 2$ ²⁹

Example sample_proof :

derivable
 $(\{ \{ \text{\$}(\text{fun } st:state \Rightarrow st \ X = 3) \ [X \mid \rightarrow X + 2] \ [X \mid \rightarrow X + 1] \} \})$
 $\langle \{ X := X + 1; X := X + 2 \} \rangle$
 $(\text{fun } st:state \Rightarrow st \ X = 3).$

Proof.

eapply H_Seq.
- apply H_Asgn.
- apply H_Asgn.

Qed.

You can see how the structure of the proof script mirrors the structure of the proof tree: at the root there is a use of the sequence rule; and at the leaves, the assignment rule.

Exercise: 3 stars, standard (provable_true_post) Show that any Hoare triple whose postcondition is **True** is derivable. Proceed by induction on c .

Theorem provable_true_post : $\forall \ c \ P,$
derivable $P \ c \ \text{True}.$

Proof.

Admitted.

□

Exercise: 3 stars, standard (provable_false_pre) Show that any Hoare triple whose precondition is **False** is derivable. Again, proceed by induction on c .

Theorem provable_false_pre : $\forall \ c \ Q,$
derivable $\text{False} \ c \ Q.$

Proof.

Admitted.

□

7.5 Soundness and Completeness

We now have two approaches to formulating Hoare logic:

²⁸ $(X=3) \ [X \mid \rightarrow X+2] \ [X \mid \rightarrow X+1]$

²⁹ $X=3$

- The model-theoretic approach uses **valid** to characterize when a Hoare triple holds in a model, which is based on states.
- The proof-theoretic approach uses **derivable** to characterize when a Hoare triple is derivable as the end of a proof tree.

Do these two approaches agree? That is, are the valid Hoare triples exactly the derivable ones? This is a standard question investigated in mathematical logic. There are two pieces to answering it:

- A logic is *sound* if everything that is derivable is valid.
- A logic is *complete* if everything that is valid is derivable.

We can prove that Hoare logic is sound and complete.

Exercise: 3 stars, standard (hoare_sound) Prove that if a Hoare triple is derivable, then it is valid. Nearly all the work for this was already done in **Hoare** as theorems **hoare_skip**, **hoare_asgn**, etc.; leverage those proofs. Proceed by induction on the derivation of the triple.

Theorem **hoare_sound** : $\forall P \ c \ Q,$
derivable $P \ c \ Q \rightarrow$ **valid** $P \ c \ Q$.

Proof.

Admitted.

□

The proof of completeness is more challenging. To carry out the proof, we need to invent some intermediate assertions using a technical device known as *weakest preconditions* (which are also discussed in **Hoare2**). Given a command c and a desired postcondition assertion Q , the weakest precondition **wp** $c \ Q$ is an assertion P such that $\{\{P\}\} \ c \ \{\{Q\}\}$ holds, and moreover, for any other assertion P' , if $\{\{P'\}\} \ c \ \{\{Q\}\}$ holds then $P' \multimap P$.

Another way of stating that idea is that **wp** $c \ Q$ is the following assertion:

Definition **wp** (c :**com**) (Q :**Assertion**) : **Assertion** :=
 $\text{fun } s \Rightarrow \forall s', s = [c] \Rightarrow s' \rightarrow Q \ s'.$

Hint **Unfold** **wp** : *core*.

The following two theorems show that the two ways of thinking about **wp** are the same.

Theorem **wp_is_precondition** : $\forall c \ Q,$
 $\{\{(\text{wp } c \ Q)\}\} \ c \ \{\{Q\}\}.$

Proof. **auto.** **Qed.**

Theorem **wp_is_weakest** : $\forall c \ Q \ P',$
 $\{\{P'\}\} \ c \ \{\{Q\}\} \rightarrow$
 $P' \multimap (\text{wp } c \ Q).$

Proof. **eauto.** **Qed.**

Weakest preconditions are useful because they enable us to identify assertions that otherwise would require clever thinking. The next two lemmas show that in action.

Exercise: 1 star, standard (wp_seq) What if we have a sequence $c1; c2$, but not an intermediate assertion for what should hold in between $c1$ and $c2$? No problem. Prove that $\text{wp } c2 \ Q$ suffices as such an assertion.

Lemma `wp_seq` : $\forall P \ Q \ c1 \ c2,$
 $\text{derivable } P \ c1 \ (\text{wp } c2 \ Q) \rightarrow \text{derivable } (\text{wp } c2 \ Q) \ c2 \ Q \rightarrow \text{derivable } P \ \langle\{c1; c2\}\rangle \ Q.$

Proof.

Admitted.

□

Exercise: 2 stars, standard (wp_invariant) What if we have a while loop, but not an invariant for it? No problem. Prove that for any Q , assertion $\text{wp } (\text{while } b \text{ do } c \text{ end}) \ Q$ is a loop invariant of $\text{while } b \text{ do } c \text{ end}$.

Lemma `wp_invariant` : $\forall b \ c \ Q,$
 $\text{valid } (\{\{\$(\text{wp } \langle\{\text{while } b \text{ do } c \text{ end}\}\rangle \ Q) \wedge b\}\}) \ c \ (\text{wp } \langle\{\text{while } b \text{ do } c \text{ end}\}\rangle \ Q).$

Proof.

Admitted.

□

Exercise: 4 stars, standard (hoare_complete) Now we are ready to prove the completeness of Hoare logic. Finish the proof of the theorem below.

Hint: for the *while* case, use the invariant suggested by `wp_invariant`.

Acknowledgment: Our approach to this proof is inspired by:

{<https://www.ps.uni-saarland.de/courses/sem-ws11/script/Hoare.html>}

Theorem `hoare_complete`: $\forall P \ c \ Q,$
 $\text{valid } P \ c \ Q \rightarrow \text{derivable } P \ c \ Q.$

Proof.

`unfold valid. intros P c. generalize dependent P.`

`induction c; intros P Q HT.`

Admitted.

□

7.6 Postscript: Decidability

We might hope that Hoare logic would be *decidable*; that is, that there would be an (terminating) algorithm (a *decision procedure*) that can determine whether or not a given Hoare triple is valid or derivable. Sadly, such a decision procedure cannot exist.

Consider the triple $\{\{\mathbf{True}\}\} \ c \ \{\{\mathbf{False}\}\}$. This triple is valid if and only if c is non-terminating. So any algorithm that could determine validity of arbitrary triples could solve the Halting Problem.

Similarly, the triple $\{\{\mathbf{True}\}\} \text{ skip } \{\{P\}\}$ is valid if and only if $\forall \mathbf{s}, P \mathbf{s}$ is valid, where P is an arbitrary assertion of Rocq's logic. But this logic is far too powerful to be decidable.

Chapter 8

Library `PLF.Smallstep`

8.1 Smallstep: Small-step Operational Semantics

```
Set Warnings "-notation-overridden".
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import Lia.
From Stdlib Require Import List. Import LISTNOTATIONS.
From PLF Require Import Maps.
From PLF Require Import Imp.
Definition FILL_IN_HERE := <{True}>.
```

The evaluators we have seen so far (for `aexps`, `bexps`, commands, ...) have been formulated in a “big-step” style: they specify how a given expression can be evaluated to its final value (or a how command plus a store can be evaluated to a final store) “all in one big step.”

This style is simple and natural for many purposes – indeed, Gilles Kahn, who popularized it, called it *natural semantics*. But there are some things it does not do well. In particular, it does not give us a convenient way of talking about *concurrent* programming languages, where the semantics of a program – the essence of how it behaves – includes not just which input states get mapped to which output states, but also the intermediate states that it passes through along the way; this is crucial, since these states can also be observed by concurrently executing code.

Another shortcoming of the big-step style is more technical but equally critical in many situations.

Suppose we want to define a variant of `Imp` where variables could hold *either* numbers *or* lists of numbers. In the syntax of this extended language, it will be possible to write strange expressions like `2 + nil`, and our semantics for arithmetic expressions will then need to say something about how such expressions behave. One possibility is to maintain the convention that every arithmetic expression evaluates to some number by choosing some way of viewing

a list as a number – e.g., by specifying that a list should be interpreted as 0 when it occurs in a context expecting a number. But this would be a bit of a hack.

A much more natural approach is simply to say that the behavior of the expression `2+nil` is *undefined* – i.e., it doesn’t evaluate to any result at all. And we can easily do this: we just have to formulate `aeval` and `beval` as **Inductive** propositions rather than **Fixpoints**, so that we can make them partial functions instead of total ones.

Now, however, we encounter a serious deficiency. In this language, a command might fail to map a given starting state to any ending state for *two quite different reasons*: either because the execution gets into an infinite loop or because, at some point, the program tries to do an operation that makes no sense, such as adding a number to a list, so that none of the evaluation rules can be applied.

These two outcomes – nontermination vs. getting stuck in an erroneous configuration – should not be confused. In particular, we want to *allow* the first (because permitting the possibility of infinite loops is the price we pay for the convenience of programming with general looping constructs like *while*) but *prevent* the second (which is just wrong), for example by adding some form of *typechecking* to the language. Indeed, this will be a major topic in the rest of the course. As a first step, we need a way of presenting the semantics that allows us to distinguish nontermination from erroneous “stuck states.”

So, for lots of reasons, we’d like to have a finer-grained way of defining and reasoning about program behaviors. This is the topic of the present chapter. Our goal is to replace the “big-step” `eval` relation with a “small-step” relation that specifies, for a given program, how its atomic steps of computation are performed.

8.2 A Toy Language

To save space in the discussion, let’s start with an incredibly simple language containing just constants and addition. (We use single letters – **C** and **P** (for Constant and Plus) – as constructor names, for brevity.) At the end of the chapter, we’ll see how to apply the same techniques to the full Imp language.

```
Inductive tm : Type :=
| C : nat → tm
| P : tm → tm → tm.
```

Here is a standard evaluator for this language, written in the big-step style that we’ve been using up to this point.

```
Fixpoint evalF (t : tm) : nat :=
  match t with
  | C n ⇒ n
  | P t1 t2 ⇒ evalF t1 + evalF t2
  end.
```

Here is the same evaluator, written in exactly the same style, but formulated as an inductively defined relation. We use the notation $t ==> n$ for “ t evaluates to n .”

(E_C) $C\ n ==> n$
 $t1 ==> n1\ t2 ==> n2$

(E_P) $P\ t1\ t2 ==> n1 + n2$

Reserved Notation " $t' ==> n$ " (at level 50, left associativity).

Inductive **eval** : **tm** → **nat** → Prop :=

| E_C : $\forall\ n,$
 $C\ n ==> n$
| E_P : $\forall\ t1\ t2\ n1\ n2,$
 $t1 ==> n1 \rightarrow$
 $t2 ==> n2 \rightarrow$
 $P\ t1\ t2 ==> (n1 + n2)$

where " $t' ==> n$ " := (**eval** $t\ n$).

Module SIMPLEARITH1.

Now, here is the corresponding *small-step* evaluation relation.

(ST_PCC) $P\ (C\ n1)\ (C\ n2) \rightarrow C\ (n1 + n2)$
 $t1 \rightarrow t1'$

(ST_P1) $P\ t1\ t2 \rightarrow P\ t1'\ t2$
 $t2 \rightarrow t2'$

(ST_P2) $P\ (C\ n1)\ t2 \rightarrow P\ (C\ n1)\ t2'$

Reserved Notation " $t' \rightarrow t'$ " (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

| ST_PCC : $\forall\ n1\ n2,$
 $P\ (C\ n1)\ (C\ n2) \rightarrow C\ (n1 + n2)$
| ST_P1 : $\forall\ t1\ t1'\ t2,$
 $t1 \rightarrow t1' \rightarrow$
 $P\ t1\ t2 \rightarrow P\ t1'\ t2$
| ST_P2 : $\forall\ n1\ t2\ t2',$
 $t2 \rightarrow t2' \rightarrow$
 $P\ (C\ n1)\ t2 \rightarrow P\ (C\ n1)\ t2'$

where " $t' \rightarrow t'$ " := (**step** $t\ t'$).

Things to notice:

- We are defining a single reduction step, in which just one P node is replaced by its value.

- Each step finds the *leftmost* P node that is ready to go (both of its operands are constants) and rewrites it in place. The first rule tells how to rewrite this P node itself; the other two rules tell how to find it.
- A term that is just a constant cannot take a step.

Let's pause and check a couple of examples of reasoning with the **step** relation...

If $t1$ can take a step to $t1'$, then $P\ t1\ t2$ steps to $P\ t1'\ t2$:

Example test_step_1 :

```

P
  (P (C 1) (C 3))
  (P (C 2) (C 4))
->
P
  (C 4)
  (P (C 2) (C 4)).

```

Proof.

apply ST_P1. apply ST_PCC. Qed.

Exercise: 1 star, standard (test_step_2) Right-hand sides of sums can take a step only when the left-hand side is finished: if $t2$ can take a step to $t2'$, then $P\ (C\ n)\ t2$ steps to $P\ (C\ n)\ t2'$:

Example test_step_2 :

```

P
  (C 0)
  (P
    (C 2)
    (P (C 1) (C 3)))
->
P
  (C 0)
  (P
    (C 2)
    (C 4)).

```

Proof.

Admitted.

□

End SIMPLEARITH1.

8.3 Relations

We will be working with several different single-step relations, so it is helpful to generalize a bit and state a few definitions and theorems about relations in general. (The optional chapter *Rel.v* in *Logical Foundations* develops some of these ideas in a bit more detail; reviewing that chapter may be useful if the treatment here feels too terse.)

A *binary relation* on a set X is a family of propositions parameterized by two elements of X – i.e., a proposition about pairs of elements of X .

Definition *relation* $(X : \text{Type}) := X \rightarrow X \rightarrow \text{Prop}$.

Our main examples of such relations in this chapter will be the single-step reduction relation, $\rightarrow$, and its multi-step variant, $\rightarrow^*$, defined below, but there are many other examples – e.g., the “equals,” “less than,” “less than or equal to,” and “is the square of” relations on numbers, and the “prefix of” relation on lists and strings.

One simple property of the $\rightarrow$ relation is that, like the big-step evaluation relation for `Imp`, it is *deterministic*.

Theorem: For each t , there is at most one t' such that t steps to t' ($t \rightarrow t'$ is provable).

Proof sketch: We show that if x steps to both $y1$ and $y2$, then $y1$ and $y2$ are equal, by induction on a derivation of `step x y1`. There are several cases to consider, depending on the last rule used in this derivation and the last rule in the given derivation of `step x y2`.

- If both are `ST_PCC`, the result is immediate.
- The cases when both derivations end with `ST_P1` or `ST_P2` follow by the induction hypothesis.
- It cannot happen that one is `ST_PCC` and the other is `ST_P1` or `ST_P2`, since this would imply that x has the form $P \ t1 \ t2$ where both $t1$ and $t2$ are constants (by `ST_PCC`) and one of $t1$ or $t2$ has the form $P \ \_$.
- Similarly, it cannot happen that one is `ST_P1` and the other is `ST_P2`, since this would imply that x has the form $P \ t1 \ t2$ where $t1$ has both the form $P \ t11 \ t12$ and the form $C \ n$. $\square$

Formally:

Definition *deterministic* $\{X : \text{Type}\} (R : \text{relation } X) :=$
 $\forall x \ y1 \ y2 : X, R \ x \ y1 \rightarrow R \ x \ y2 \rightarrow y1 = y2$.

`Module SIMPLEARITH2.`

`Import SIMPLEARITH1.`

Theorem *step_deterministic:*

deterministic `step`.

Proof.

`unfold deterministic. intros x y1 y2 Hy1.`

`generalize dependent y2.`

```

induction Hy1; intros y2 Hy2.
- inversion Hy2; subst.
  + reflexivity.
  + inversion H2.
  + inversion H2.
- inversion Hy2; subst.
  +
    inversion Hy1.
  +
    apply IHHy1 in H2. rewrite H2. reflexivity.
  +
    inversion Hy1.
- inversion Hy2; subst.
  +
    inversion Hy1.
  + inversion H2.
  +
    apply IHHy1 in H2. rewrite H2. reflexivity.
Qed.
End SIMPLEARITH2.

```

There is some annoying repetition in this proof. Each use of `inversion Hy2` results in three subcases, only one of which is relevant (the one that matches the current case in the induction on `Hy1`). The other two subcases need to be dismissed by finding the contradiction among the hypotheses and doing inversion on it.

The following custom tactic, called `solve_by_inverts`, can be helpful in such cases. It will solve the goal if it can be solved by inverting some hypothesis; otherwise, it fails.

```

Ltac solve_by_inverts n :=
  match goal with | H : ?T ⊢ _ ⇒
  match type of T with Prop ⇒
  solve [
    inversion H;
    match n with S (S (?n')) ⇒ subst; solve_by_inverts (S n') end ]
  end end.

```

The details of how this works are not important, but it illustrates the power of Rocq's `Ltac` language for programmatically defining special-purpose tactics. It looks through the current proof state for a hypothesis `H` (the first `match`) of type `Prop` (the second `match`) such that performing inversion on `H` (followed by a recursive invocation of the same tactic, if its argument `n` is greater than one) completely solves the current goal. If no such hypothesis exists, it fails.

We will usually want to call `solve_by_inverts` with argument 1 (especially as larger arguments can lead to very slow proof checking), so we define `solve_by_invert` as a shorthand for

this case.

```
Ltac solve_by_invert :=  
  solve_by_inverts 1.
```

The proof of the previous theorem can now be simplified...

```
Module SIMPLEARITH3.
```

```
Import SIMPLEARITH1.
```

```
Theorem step_deterministic_alt: deterministic step.
```

```
Proof.
```

```
  intros x y1 y2 Hy1.  
  generalize dependent y2.  
  induction Hy1; intros y2 Hy2;  
    inversion Hy2; subst; try solve_by_invert.  
- reflexivity.  
-  
  apply IHHy1 in H2. rewrite H2. reflexivity.  
-  
  apply IHHy1 in H2. rewrite H2. reflexivity.
```

```
Qed.
```

```
End SIMPLEARITH3.
```

8.3.1 Values

Next, it will be useful to slightly reformulate the definition of single-step reduction by stating it in terms of “values.”

It can be useful to think of the $\rightarrow$ relation as defining an *abstract machine*:

- At any moment, the *state* of the machine is a term.
- A *step* of the machine is an atomic unit of computation – here, a single “add” operation.
- The *halting states* of the machine are ones where there is no more computation to be done.

We can then *execute* a term t as follows:

- Take t as the starting state of the machine.
- Repeatedly use the $\rightarrow$ relation to find a sequence of machine states, starting with t , where each state steps to the next.
- When no more reduction is possible, “read out” the final state of the machine as the result of execution.

Intuitively, it is clear that the final states of our machine are always terms of the form $C\ n$ for some n . We call such terms *values*.

Inductive value : $\mathbf{tm} \rightarrow \mathbf{Prop} :=$
 $| \mathbf{v\_C} : \forall n, \mathbf{value} (C\ n).$

Having introduced the idea of values, we can use it in the definition of the $\rightarrow$ relation to write **ST_P2** rule in a slightly more elegant way:

$(\mathbf{ST\_PCC})\ P\ (C\ n1)\ (C\ n2) \rightarrow C\ (n1 + n2)$ $t1 \rightarrow t1'$

$(\mathbf{ST\_P1})\ P\ t1\ t2 \rightarrow P\ t1'\ t2$ $\mathbf{value}\ v1\ t2 \rightarrow t2'$

$(\mathbf{ST\_P2})\ P\ v1\ t2 \rightarrow P\ v1\ t2'$

Again, the variable names in the informal presentation carry important information: by convention, $v1$ ranges only over values, while $t1$ and $t2$ range over arbitrary terms.

(Given this convention, the explicit **value** hypothesis is arguably redundant, since the naming convention tells us where to add it when translating the informal rule to Rocq. We'll keep it for now, to maintain a close correspondence between the informal and Rocq versions of the rules, but later on we'll drop it in informal rules for brevity.)

Here are the formal rules:

Reserved Notation " $t \rightarrow t'$ " (at level 40).

Inductive step : $\mathbf{tm} \rightarrow \mathbf{tm} \rightarrow \mathbf{Prop} :=$
 $| \mathbf{ST\_PCC} : \forall n1\ n2,$
 $\quad P\ (C\ n1)\ (C\ n2)$
 $\quad \rightarrow C\ (n1 + n2)$
 $| \mathbf{ST\_P1} : \forall t1\ t1'\ t2,$
 $\quad t1 \rightarrow t1' \rightarrow$
 $\quad P\ t1\ t2 \rightarrow P\ t1'\ t2$
 $| \mathbf{ST\_P2} : \forall v1\ t2\ t2',$
 $\quad \mathbf{value}\ v1 \rightarrow$
 $\quad t2 \rightarrow t2' \rightarrow$
 $\quad P\ v1\ t2 \rightarrow P\ v1\ t2'$

where " $t \rightarrow t'$ " := (**step** $t\ t'$).

Exercise: 3 stars, standard, especially useful (redo_determinism) As a sanity check on this change, let's re-verify determinism. Here's an informal proof:

Proof sketch: We must show that if x steps to both $y1$ and $y2$, then $y1$ and $y2$ are equal. Consider the final rules used in the derivations of **step** $x\ y1$ and **step** $x\ y2$.

- If both are **ST_PCC**, the result is immediate.

- The cases when both derivations end with **ST_P1** or **ST_P2** follow by the induction hypothesis.
- It cannot happen that one is **ST_PCC** and the other is **ST_P1** or **ST_P2**, since this would imply that x has the form $P \ t1 \ t2$ where both $t1$ and $t2$ are constants (by **ST_PCC**) and one of $t1$ or $t2$ has the form $P \ \_$.
- Similarly, it cannot happen that one is **ST_P1** and the other is **ST_P2**, since this would imply that x has the form $P \ t1 \ t2$ where $t1$ both has the form $P \ t11 \ t12$ and is a value (hence has the form **C n**). $\square$

Most of this proof is the same as the one above. But to get maximum benefit from the exercise you should try to write your formal version from scratch and just use the earlier one if you get stuck.

Theorem step_deterministic :
deterministic step.

Proof.

Admitted.

$\square$

8.3.2 Strong Progress and Normal Forms

The definition of single-step reduction for our toy language is fairly simple, but for a larger language it would be easy to forget one of the rules and accidentally create a situation where some term cannot take a step even though it has not been completely reduced to a value. The following theorem shows that we did not, in fact, make such a mistake here.

Theorem (Strong Progress): If t is a term, then either t is a value or else there exists a term t' such that $t \rightarrow t'$.

Proof: By induction on t .

- Suppose $t = \mathbf{C} \ n$. Then t is a value.
- Suppose $t = P \ t1 \ t2$, where (by the IH) $t1$ either is a value or can step to some $t1'$, and where $t2$ is either a value or can step to some $t2'$. We must show $P \ t1 \ t2$ is either a value or steps to some t' .
 - If $t1$ and $t2$ are both values, then t can take a step, by **ST_PCC**.
 - If $t1$ is a value and $t2$ can take a step, then so can t , by **ST_P2**.
 - If $t1$ can take a step, then so can t , by **ST_P1**. $\square$

Or, formally:

Theorem strong_progress : $\forall \ t,$
value $t \vee (\exists \ t', \ t \rightarrow t')$.

Proof.

```

induction t.
- left. apply v_C.
- right. destruct IHt1 as [IHt1 | [t1' Ht1] ].
  + destruct IHt2 as [IHt2 | [t2' Ht2] ].
    × inversion IHt1. inversion IHt2.
      ∃ (C (n + n0)).
      apply ST_PCC.
    ×
      ∃ (P t1 t2').
      apply ST_P2; auto.
  +
    ∃ (P t1' t2).
    apply ST_P1. apply Ht1.

```

Qed.

This important property is called *strong progress*, because every term either is a value or can “make progress” by stepping to some other term. (The qualifier “strong” distinguishes it from a more refined version that we’ll see in later chapters, called simply *progress*.)

The idea of “making progress” can be extended to tell us something interesting about values in this language: they are exactly the terms that do *not* make progress in this sense.

To state this observation formally, let’s begin by giving a name to “terms that cannot make progress.” We’ll call them *normal forms*.

Definition `normal_form` { $X : \text{Type}$ }
 $(R : \text{relation } X) (t : X) : \text{Prop} :=$
 $\neg \exists t', R \ t \ t'.$

Note that this definition specifies what it is to be a normal form for an *arbitrary* relation R over an arbitrary set X , not just for the particular single-step reduction relation over terms that we are interested in at the moment. We’ll re-use the same terminology for talking about other relations later in the course.

We can use this terminology to generalize the observation we made in the strong progress theorem: in this language (though not necessarily, in general), normal forms and values are actually the same thing.

Lemma `value_is_nf` : $\forall v,$
`value` $v \rightarrow$ `normal_form` `step` v .

Proof.

```

unfold normal_form. intros v H contra.
destruct contra. destruct H. inversion H0.

```

Qed.

Lemma `nf_is_value` : $\forall t,$
`normal_form` `step` $t \rightarrow$ `value` t .

Proof. `unfold normal_form. intros t H.`

```

assert (G : value t ∨ ∃ t', t -> t').
{ apply strong_progress. }
destruct G as [G | G].
- apply G.
- contradiction.

```

Qed.

Corollary nf_same_as_value : $\forall t$,
 normal_form step $t \leftrightarrow$ value t .

Proof.

```

split.
- apply nf_is_value.
- apply value_is_nf.

```

Qed.

Why is this interesting?

Because **value** is a syntactic concept – it is defined by looking at the way a term is written – while **normal_form** is a semantic one – it is defined by looking at how the term steps.

It is not obvious that these concepts should characterize the same set of terms!

Indeed, we could easily have written the definitions (incorrectly) so that they would *not* coincide...

Exercise: 3 stars, standard, optional (value_not_same_as_normal_form1) We might, for example, define **value** so that it includes some terms that are not finished reducing:

(Even if you don't work this exercise and the following ones in Rocq, make sure you can think of an example of such a term.)

Module TEMP1.

```

Inductive value : tm → Prop :=
| v_C : ∀ n, value (C n)
| v_funny : ∀ t1 n,
    value (P t1 (C n)).

```

Reserved Notation "t '→' t'" (at level 40).

```

Inductive step : tm → tm → Prop :=
| ST_PCC : ∀ n1 n2,
    P (C n1) (C n2) -> C (n1 + n2)
| ST_P1 : ∀ t1 t1' t2,
    t1 -> t1' →
    P t1 t2 -> P t1' t2
| ST_P2 : ∀ v1 t2 t2',
    value v1 →
    t2 -> t2' →
    P v1 t2 -> P v1 t2'

```

```

where " t '→' t' " := (step t t').
Lemma value_not_same_as_normal_form :
  ∃ v, value v ∧ ¬ normal_form step v.
Proof.
  Admitted.
End TEMP1.
□

```

Exercise: 2 stars, standard, optional (value_not_same_as_normal_form2) Or we might (again, wrongly) define **step** so that it permits something designated as a value to reduce further.

```

Module TEMP2.
Inductive value : tm → Prop :=
| v_C : ∀ n, value (C n).
Reserved Notation " t '→' t' " (at level 40).
Inductive step : tm → tm → Prop :=
| ST_Funny : ∀ n,
  C n -> P (C n) (C 0)
| ST_PCC : ∀ n1 n2,
  P (C n1) (C n2) -> C (n1 + n2)
| ST_P1 : ∀ t1 t1' t2,
  t1 -> t1' →
  P t1 t2 -> P t1' t2
| ST_P2 : ∀ v1 t2 t2',
  value v1 →
  t2 -> t2' →
  P v1 t2 -> P v1 t2'

```

```

where " t '→' t' " := (step t t').
Lemma value_not_same_as_normal_form :
  ∃ v, value v ∧ ¬ normal_form step v.
Proof.
  Admitted.
End TEMP2.
□

```

Exercise: 3 stars, standard, optional (value_not_same_as_normal_form3) Finally, we might define **value** and **step** so that there is some term that is not a value but that cannot take a step in the **step** relation. Such terms are said to be *stuck*. In this case,

this is caused by a mistake in the semantics, but we will also see situations where, even in a correct language definition, it makes sense to allow some terms to be stuck.

Module TEMP3.

```
Inductive value : tm → Prop :=
| v_C : ∀ n, value (C n).
```

Reserved Notation "t '→' t'" (at level 40).

```
Inductive step : tm → tm → Prop :=
| ST_PCC : ∀ n1 n2,
  P (C n1) (C n2) → C (n1 + n2)
| ST_P1 : ∀ t1 t1' t2,
  t1 → t1' →
  P t1 t2 → P t1' t2
```

where "t '→' t'" := (step t t').

(Note that ST_P2 is missing.)

Lemma value_not_same_as_normal_form :
 $\exists t, \neg \text{value } t \wedge \text{normal\_form step } t$.

Proof.

Admitted.

End TEMP3.

□

Additional Exercises

Module TEMP4.

Here is another very simple language whose terms, instead of being just addition expressions and numbers, are just the booleans true and false plus a conditional expression...

```
Inductive tm : Type :=
| tru : tm
| fls : tm
| test : tm → tm → tm → tm.
```

```
Inductive value : tm → Prop :=
| v_tru : value tru
| v_fls : value fls.
```

Reserved Notation "t '→' t'" (at level 40).

```
Inductive step : tm → tm → Prop :=
| ST_IfTrue : ∀ t1 t2,
  test tru t1 t2 → t1
| ST_IfFalse : ∀ t1 t2,
```

```

    test fls t1 t2 -> t2
| ST_If :  $\forall t1\ t1'\ t2\ t3,$ 
     $t1 \rightarrow t1' \rightarrow$ 
    test t1 t2 t3 -> test t1' t2 t3

where " t '→' t' " := (step t t').

```

Exercise: 1 star, standard, optional (smallstep_bools) Which of the following propositions are provable? (This is just a thought exercise, but for an extra challenge feel free to prove your answers in Rocq.)

Definition bool_step_prop1 :=
 fls -> fls.

Definition bool_step_prop2 :=
 test
 tru
 (test tru tru tru)
 (test fls fls fls)
->
 tru.

Definition bool_step_prop3 :=
 test
 (test tru tru tru)
 (test tru tru tru)
 fls
->
 test
 tru
 (test tru tru tru)
 fls.

Definition manual_grade_for_smallstep_bools : option (nat × string) := None.

□

Exercise: 3 stars, standard, optional (strong_progress_bool) Just as we proved a progress theorem for plus expressions, we can do so for boolean expressions as well.

Theorem strong_progress_bool : $\forall t,$
 value $t \vee (\exists t', t \rightarrow t').$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (step_deterministic) *Theorem* `step_deterministic` : `deterministic step`.

Proof.

Admitted.

□

Module `TEMP5`.

Exercise: 2 stars, standard (smallstep_bool_shortcut) Suppose we want to add a “short circuit” to the step relation for boolean expressions, so that it can recognize when the `then` and `else` branches of a conditional are the same value (either `tru` or `fls`) and reduce the whole conditional to this value in a single step, even if the guard has not yet been reduced to a value. For example, we would like this proposition to be provable:

`test (test tru tru tru) fls fls > fls.`

Write an extra clause for the step relation that achieves this effect and prove `bool_step_prop4`.

Reserved Notation " `t '→' t'` " (at level 40).

Inductive `step` : `tm` → `tm` → Prop :=

```
| ST_IfTrue : ∀ t1 t2,
  test tru t1 t2 -> t1
| ST_IfFalse : ∀ t1 t2,
  test fls t1 t2 -> t2
| ST_If : ∀ t1 t1' t2 t3,
  t1 -> t1' →
  test t1 t2 t3 -> test t1' t2 t3
```

where " `t '→' t'` " := (`step t t'`).

Definition `bool_step_prop4` :=

```
test
  (test tru tru tru)
  fls
  fls
->
  fls.
```

Example `bool_step_prop4_holds` :

`bool_step_prop4`.

Proof.

Admitted.

□

Exercise: 3 stars, standard, optional (properties_of_altered_step) It can be shown that the determinism and strong progress theorems for the step relation in the lecture notes also hold for the definition of **step** given above.

After we add the clause *ST_ShortCircuit*...

- Is the **step** relation still deterministic? Write yes or no and briefly (1 sentence) explain your answer.

Optional: prove your answer correct in Rocq.

Definition *manual_grade_for_properties_of_altered_step* : **option** (nat×string) := None.

□

End TEMP5.

End TEMP4.

8.4 Multi-Step Reduction

We’ve been working so far with the *single-step reduction* relation $\rightarrow$, which formalizes the individual steps of an abstract machine for executing programs.

We can use the same machine to reduce programs to completion – to find out what final result they yield. This can be formalized as follows:

- First, we define a *multi-step reduction relation* $\rightarrow^*$, which relates terms t and t' if t can reach t' by any number (including zero) of single reduction steps.
- Then we define a “result” of a term t as a normal form that t can reach by multi-step reduction.

Since we’ll want to reuse the idea of multi-step reduction many times with many different single-step relations, let’s pause and define the concept generically.

Given a relation **R** (e.g., the step relation $\rightarrow$), we define a new relation **multi R**, called the *multi-step closure of R* as follows.

```
Inductive multi {X : Type} (R : relation X) : relation X :=
| multi_refl : ∀ (x : X), multi R x x
| multi_step : ∀ (x y z : X),
    R x y →
    multi R y z →
    multi R x z.
```

(In the *Rel* chapter of *Logical Foundations* and in the Rocq standard library, this relation is called *clos_refl_trans_1n*. We give it a shorter name here for the sake of readability.)

The effect of this definition is that **multi R** relates two elements **x** and **y** if

- $x = y$, or

- $R\ x\ y$, or
- there is some nonempty sequence $z1, z2, \dots, zn$ such that
 $R\ x\ z1\ R\ z1\ z2\ \dots\ R\ zn\ y$.

Intuitively, if R describes a single-step of computation, then $z1 \dots zn$ are the intermediate steps of computation that get us from x to y .

We write $\rightarrow^*$ for the **multi step** relation on terms.

Notation " $t \rightarrow^* t'$ " := (**multi step** $t\ t'$) (at level 40).

The relation **multi** R has several crucial properties.

First, it is obviously *reflexive* (that is, $\forall x, \text{multi } R\ x\ x$). In the case of the $\rightarrow^*$ (i.e., **multi step**) relation, the intuition is that a term can execute to itself by taking zero steps of reduction.

Second, it contains R – that is, single-step reductions are a particular case of multi-step executions. (It is this fact that justifies the word “closure” in the term “multi-step closure of R .”)

Theorem multi_R : $\forall (X : \text{Type}) (R : \text{relation } X) (x\ y : X),$
 $R\ x\ y \rightarrow (\text{multi } R)\ x\ y.$

Proof.

```
intros X R x y H.
apply multi_step with y.
- apply H.
- apply multi_refl.
```

Qed.

Third, **multi** R is *transitive*.

Theorem multi_trans :
 $\forall (X : \text{Type}) (R : \text{relation } X) (x\ y\ z : X),$
 $\text{multi } R\ x\ y \rightarrow$
 $\text{multi } R\ y\ z \rightarrow$
 $\text{multi } R\ x\ z.$

Proof.

```
intros X R x y z G H.
induction G.
- assumption.
-
  apply multi_step with y.
  + assumption.
  + apply IHG. assumption.
```

Qed.

In particular, for the **multi step** relation on terms, if $t1 \rightarrow^* t2$ and $t2 \rightarrow^* t3$, then $t1 \rightarrow^* t3$.

8.4.1 Examples

Here's a specific instance of the **multi step** relation:

Lemma `test_multistep_1`:

```
P
  (P (C 0) (C 3))
  (P (C 2) (C 4))
->*
  C ((0 + 3) + (2 + 4)).
```

Proof.

```
apply multi_step with
  (P (C (0 + 3))
    (P (C 2) (C 4))).
{ apply ST_P1. apply ST_PCC. }
apply multi_step with
  (P (C (0 + 3))
    (C (2 + 4))).
{ apply ST_P2.
  - apply v_C.
  - apply ST_PCC. }
apply multi_R.
apply ST_PCC.
```

Qed.

Here's an alternate proof of the same fact that uses `eapply` to avoid explicitly constructing all the intermediate terms.

Lemma `test_multistep_1'`:

```
P
  (P (C 0) (C 3))
  (P (C 2) (C 4))
->*
  C ((0 + 3) + (2 + 4)).
```

Proof.

```
eapply multi_step. { apply ST_P1. apply ST_PCC. }
eapply multi_step. { apply ST_P2.
  - apply v_C.
  - apply ST_PCC. }
eapply multi_step. { apply ST_PCC. }
apply multi_refl.
```

Qed.

Exercise: 1 star, standard, optional (`test_multistep_2`) Lemma `test_multistep_2`:

```
C 3 ->* C 3.
```

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (test_multistep_3) *Lemma test_multistep_3:*

```
P (C 0) (C 3)
->*
P (C 0) (C 3).
```

Proof.

Admitted.

□

Exercise: 2 stars, standard (test_multistep_4) *Lemma test_multistep_4:*

```
P
(C 0)
(P
(C 2)
(P (C 0) (C 3)))
->*
P
(C 0)
(C (2 + (0 + 3))).
```

Proof.

Admitted.

□

8.4.2 Normal Forms Again

If t reduces to t' in zero or more steps and t' is a normal form, we say that “ t' is a normal form of t .”

Definition `normal_form_of` $\{X : \text{Type}\} (R : \text{relation } X) (t\ t' : X) :=$
 $((\text{multi } R) t\ t' \wedge \text{normal\_form } R\ t').$

We have already seen that, for our language, single-step reduction is deterministic – i.e., a given term can take a single step in at most one way. It follows from this that, if t can reach a normal form, then this normal form is unique.

In other words, we can actually pronounce `normal_form t t'` as “ t' is *the* normal form of t .”

Exercise: 3 stars, standard, optional (normal_forms_unique) *Theorem normal_forms_unique:*
`deterministic (normal_form_of step).`

Proof.

```
unfold deterministic. unfold normal_form_of.
```

```

intros x y1 y2 P1 P2.
destruct P1 as [P11 P12].
destruct P2 as [P21 P22].
Admitted.

```

□

Indeed, something stronger is true for this language (though not for all the languages we will see): the reduction of *any* term t will eventually reach a normal form in a finite number of steps—i.e., `normal_form_of` is a *total* function. We say the **step** relation is *normalizing*.

Definition `normalizing` $\{X : \text{Type}\}$ ($R : \text{relation } X$) :=
 $\forall t, \exists t', \text{normal\_form\_of } R t t'.$

To prove that **step** is normalizing, we need a couple of lemmas. First, we observe that, if t reduces to t' in many steps, then the same sequence of reduction steps within t is also possible when t appears as the first argument to P , and similarly when t appears as the second argument to P (and the first argument is a value).

Lemma `multistep_congr_1` : $\forall t1\ t1'\ t2,$
 $t1 \rightarrow^* t1' \rightarrow$
 $P\ t1\ t2 \rightarrow^* P\ t1'\ t2.$

Proof.

```

intros t1 t1' t2 H. induction H.
- apply multi_refl.
- apply multi_step with (P y t2).
  + apply ST_P1. apply H.
  + apply IHmulti.

```

Qed.

Exercise: 2 stars, standard (`multistep_congr_2`) **Lemma** `multistep_congr_2` : $\forall v1\ t2\ t2',$

```

value v1 →
t2 →* t2' →
P v1 t2 →* P v1 t2'.

```

Proof.

Admitted.

□

With these lemmas in hand, the main proof is a straightforward induction.

Theorem: The **step** function is normalizing — i.e., for every t there exists some t' such that t reduces to t' and t' is a normal form.

Proof sketch: By induction on terms. There are two cases to consider:

- $t = \text{C } n$ for some n . Here t doesn't take a step, and we have $t' = t$. We can derive the left-hand side by reflexivity and the right-hand side by observing (a) that values are normal forms (by `nf_same_as_value`) and (b) that t is a value (by `v_C`).

- $t = P \ t1 \ t2$ for some $t1$ and $t2$. By the IH, $t1$ and $t2$ reduce to normal forms $t1'$ and $t2'$. Recall that normal forms are values (by `nf_same_as_value`); we therefore know that $t1' = C \ n1$ and $t2' = C \ n2$, for some $n1$ and $n2$. We can combine the $\rightarrow^*$ derivations for $t1$ and $t2$ using `multi_congr_1` and `multi_congr_2` to prove that $P \ t1 \ t2$ reduces in many steps to $t' = C \ (n1 + n2)$.

Finally, $C \ (n1 + n2)$ is a value, which is in turn a normal form by `nf_same_as_value`.
 $\square$

Theorem `step_normalizing` :
`normalizing step`.

Proof.

`unfold normalizing.`

`induction t.`

-

$\exists (C \ n).$

`split.`

+ `apply multi_refl.`

+

`apply nf_same_as_value. apply v_C.`

-

`destruct IHt1 as [t1' [Hsteps1 Hnormal1]].`

`destruct IHt2 as [t2' [Hsteps2 Hnormal2]].`

`apply nf_same_as_value in Hnormal1.`

`apply nf_same_as_value in Hnormal2.`

`destruct Hnormal1 as [n1].`

`destruct Hnormal2 as [n2].`

$\exists (C \ (n1 + n2)).$

`split.`

+

`apply multi_trans with (P (C n1) t2).`

$\times$ `apply multistep_congr_1. apply Hsteps1.`

$\times$ `apply multi_trans with (P (C n1) (C n2)).`

$\{$ `apply multistep_congr_2.`

$-$ `apply v_C.`

$-$ `apply Hsteps2. }`

`apply multi_R. apply ST_PCC.`

+

`apply nf_same_as_value. apply v_C.`

Qed.

8.4.3 Equivalence of Big-Step and Small-Step

Having defined the operational semantics of our tiny programming language in two different ways (big-step and small-step), it makes sense to ask whether these definitions actually define the same thing!

They do, though it takes a little work to show it. The details are left as an exercise.

We consider the two implications separately.

Exercise: 3 stars, standard (eval__multistep) *Theorem* `eval__multistep` : $\forall t\ n,$
 $t \implies n \rightarrow t \rightarrow^* C\ n.$

The key ideas in the proof can be seen in the following picture:

$P\ t1\ t2 \rightarrow (\text{by ST\_P1})\ P\ t1'\ t2 \rightarrow (\text{by ST\_P1})\ P\ t1''\ t2 \rightarrow (\text{by ST\_P1})\ \dots\ P\ (C\ n1)\ t2$
 $\rightarrow (\text{by ST\_P2})\ P\ (C\ n1)\ t2' \rightarrow (\text{by ST\_P2})\ P\ (C\ n1)\ t2'' \rightarrow (\text{by ST\_P2})\ \dots\ P\ (C\ n1)\ (C\ n2)$
 $\rightarrow (\text{by ST\_PCC})\ C\ (n1 + n2)$

That is, the multi-step reduction of a term of the form $P\ t1\ t2$ proceeds in three phases:

- First, we use `ST_P1` some number of times to reduce $t1$ to a normal form, which must (by `nf_same_as_value`) be a term of the form $C\ n1$ for some $n1$.
- Next, we use `ST_P2` some number of times to reduce $t2$ to a normal form, which must again be a term of the form $C\ n2$ for some $n2$.
- Finally, we use `ST_PCC` one time to reduce $P\ (C\ n1)\ (C\ n2)$ to $C\ (n1 + n2)$.

To formalize this intuition, you'll need to use the congruence lemmas from above (you might want to review them now, so that you'll be able to recognize when they are useful), plus some basic properties of $\rightarrow^*$ (that it is reflexive, transitive, and includes $\rightarrow$).

Proof.

Admitted.

□

Exercise: 3 stars, advanced, optional (eval__multistep_inf) Write a detailed informal version of the proof of `eval__multistep`.

Definition `manual_grade_for_eval__multistep_inf` : `option (nat×string)` := `None`.

□

For the other direction, we need one lemma, which establishes a relation between single-step reduction and big-step evaluation.

Exercise: 3 stars, standard (step__eval) *Lemma* `step__eval` : $\forall t\ t'\ n,$
 $t \rightarrow t' \rightarrow$
 $t' \implies n \rightarrow$
 $t \implies n.$

Proof.

```
intros t t' n Hs. generalize dependent n.
```

Admitted.

□

The fact that small-step reduction implies big-step evaluation is now straightforward to prove.

The proof proceeds by induction on the multi-step reduction sequence that is buried in the hypothesis `normal_form_of t t'`.

Make sure you understand the statement before you start to work on the proof.

Exercise: 3 stars, standard (multistep__eval) `Theorem multistep__eval :  $\forall t t', \text{normal\_form\_of step } t t' \rightarrow \exists n, t' = \text{C } n \wedge t ==> n.$`

Proof.

Admitted.

□

8.4.4 Additional Exercises

Exercise: 3 stars, standard, optional (interp_tm) Remember that we also defined big-step evaluation of terms as a function `evalF`. Prove that it is equivalent to the existing semantics. (Hint: we just proved that `eval` and `multistep` are equivalent, so logically it doesn't matter which you choose. One will be easier than the other, though!)

`Theorem evalF_eval :  $\forall t n,$`

`evalF t = n  $\leftrightarrow$  t ==> n.`

Proof.

Admitted.

□

We've considered arithmetic and conditional expressions separately. This exercise explores how the two interact. `Module COMBINED.`

`Inductive tm : Type :=`

`| C : nat  $\rightarrow$  tm`

`| P : tm  $\rightarrow$  tm  $\rightarrow$  tm`

`| tru : tm`

`| fls : tm`

`| test : tm  $\rightarrow$  tm  $\rightarrow$  tm  $\rightarrow$  tm.`

`Inductive value : tm  $\rightarrow$  Prop :=`

`| v_C :  $\forall n,$  value (C n)`

`| v_tru : value tru`

`| v_fls : value fls.`

`Reserved Notation " t '=>' t' " (at level 40).`

`Inductive step : tm  $\rightarrow$  tm  $\rightarrow$  Prop :=`

`| ST_PCC :  $\forall n1 n2,$`


```

    P (C n1) (C n2) -> C (n1 + n2)
| ST_P1 : ∀ t1 t1' t2,
    t1 -> t1' →
    P t1 t2 -> P t1' t2
| ST_P2 : ∀ v1 t2 t2',
    value v1 →
    t2 -> t2' →
    P v1 t2 -> P v1 t2'
| ST_IfTrue : ∀ t1 t2,
    test tru t1 t2 -> t1
| ST_IfFalse : ∀ t1 t2,
    test fls t1 t2 -> t2
| ST_If : ∀ t1 t1' t2 t3,
    t1 -> t1' →
    test t1 t2 t3 -> test t1' t2 t3

```

where "t' -> t'" := (step t t').

Earlier, we separately proved for both plus- and if-expressions...

- that the step relation was deterministic, and
- a strong progress lemma, stating that every term is either a value or can take a step.

Formally prove or disprove these two properties for the combined language.

Exercise: 3 stars, standard (combined_step_deterministic) Theorem combined_step_deterministic
(deterministic step) ∨ ¬ (deterministic step).

Proof.

Admitted.

□

Exercise: 3 stars, standard (combined_strong_progress) Theorem combined_strong_progress

:

```

(∀ t, value t ∨ (∃ t', t -> t'))
∨ ¬ (∀ t, value t ∨ (∃ t', t -> t')).

```

Proof.

Admitted.

□

End COMBINED.

8.5 Small-Step Imp

Now for a more serious example: a small-step version of the Imp operational semantics.

The small-step reduction relations for arithmetic and boolean expressions are straightforward extensions of the tiny language we've been working up to now. To make them easier to read, we introduce the symbolic notations $\rightarrow_a$ and $\rightarrow_b$ for the arithmetic and boolean step relations.

```
Inductive aval : aexp → Prop :=
| av_num : ∀ n, aval (ANum n).
```

We are not actually going to bother to define boolean values, since they aren't needed in the definition of $\rightarrow_b$ below (why?), though they might be if our language were a bit more complicated (why?).

```
Reserved Notation "a '/' st '→a' a' "
(at level 40, st at next level, left associativity).
```

```
Inductive astep (st : state) : aexp → aexp → Prop :=
| AS_Id : ∀ (i : string),
  i / st →a ANum (st i)
| AS_P1 : ∀ a1 a1' a2,
  a1 / st →a a1' →
  <{ a1 + a2 }> / st →a <{ a1' + a2 }>
| AS_P2 : ∀ v1 a2 a2',
  aval v1 →
  a2 / st →a a2' →
  <{ v1 + a2 }> / st →a <{ v1 + a2' }>
| AS_P : ∀ (v1 v2 : nat),
  <{ v1 + v2 }> / st →a ANum (v1 + v2)
| AS_Minus1 : ∀ a1 a1' a2,
  a1 / st →a a1' →
  <{ a1 - a2 }> / st →a <{ a1' - a2 }>
| AS_Minus2 : ∀ v1 a2 a2',
  aval v1 →
  a2 / st →a a2' →
  <{ v1 - a2 }> / st →a <{ v1 - a2' }>
| AS_Minus : ∀ (v1 v2 : nat),
  <{ v1 - v2 }> / st →a ANum (v1 - v2)
| AS_Mult1 : ∀ a1 a1' a2,
  a1 / st →a a1' →
  <{ a1 × a2 }> / st →a <{ a1' × a2 }>
| AS_Mult2 : ∀ v1 a2 a2',
  aval v1 →
  a2 / st →a a2' →
  <{ v1 × a2 }> / st →a <{ v1 × a2' }>
```

```

| AS_Mult :  $\forall (v1\ v2 : \mathbf{nat}),$ 
    <{  $v1 \times v2$  }> /  $st \rightarrow_a \mathbf{ANum}\ (v1 \times v2)$ 

    where " a '/' st '→' a' a' " := (astep  $st\ a\ a'$ ).

Reserved Notation " b '/' st '→' b' b' "
    (at level 40,  $st$  at next level, left associativity).

Inductive bstep ( $st : \mathbf{state}$ ) : bexp → bexp → Prop :=
| BS_Eq1 :  $\forall\ a1\ a1'\ a2,$ 
     $a1 / st \rightarrow_a a1' \rightarrow$ 
    <{  $a1 = a2$  }> /  $st \rightarrow_b$  <{  $a1' = a2$  }>
| BS_Eq2 :  $\forall\ v1\ a2\ a2',$ 
    aval  $v1 \rightarrow$ 
     $a2 / st \rightarrow_a a2' \rightarrow$ 
    <{  $v1 = a2$  }> /  $st \rightarrow_b$  <{  $v1 = a2'$  }>
| BS_Eq :  $\forall (v1\ v2 : \mathbf{nat}),$ 
    <{  $v1 = v2$  }> /  $st \rightarrow_b$ 
    (if ( $v1 =? v2$ ) then BTrue else BFalse)
| BS_LtEq1 :  $\forall\ a1\ a1'\ a2,$ 
     $a1 / st \rightarrow_a a1' \rightarrow$ 
    <{  $a1 \leq a2$  }> /  $st \rightarrow_b$  <{  $a1' \leq a2$  }>
| BS_LtEq2 :  $\forall\ v1\ a2\ a2',$ 
    aval  $v1 \rightarrow$ 
     $a2 / st \rightarrow_a a2' \rightarrow$ 
    <{  $v1 \leq a2$  }> /  $st \rightarrow_b$  <{  $v1 \leq a2'$  }>
| BS_LtEq :  $\forall (v1\ v2 : \mathbf{nat}),$ 
    <{  $v1 \leq v2$  }> /  $st \rightarrow_b$ 
    (if ( $v1 \leq? v2$ ) then BTrue else BFalse)
| BS_NotStep :  $\forall\ b1\ b1',$ 
     $b1 / st \rightarrow_b b1' \rightarrow$ 
    <{  $\neg b1$  }> /  $st \rightarrow_b$  <{  $\neg b1'$  }>
| BS_NotTrue : <{  $\neg \mathbf{true}$  }> /  $st \rightarrow_b$  <{  $\mathbf{false}$  }>
| BS_NotFalse : <{  $\neg \mathbf{false}$  }> /  $st \rightarrow_b$  <{  $\mathbf{true}$  }>
| BS_AndStep :  $\forall\ b1\ b1'\ b2,$ 
     $b1 / st \rightarrow_b b1' \rightarrow$ 
    <{  $b1 \ \&\&\ b2$  }> /  $st \rightarrow_b$  <{  $b1' \ \&\&\ b2$  }>
| BS_AndTrueStep :  $\forall\ b2\ b2',$ 
     $b2 / st \rightarrow_b b2' \rightarrow$ 
    <{  $\mathbf{true} \ \&\&\ b2$  }> /  $st \rightarrow_b$  <{  $\mathbf{true} \ \&\&\ b2'$  }>
| BS_AndFalse :  $\forall\ b2,$ 
    <{  $\mathbf{false} \ \&\&\ b2$  }> /  $st \rightarrow_b$  <{  $\mathbf{false}$  }>
| BS_AndTrueTrue : <{  $\mathbf{true} \ \&\&\ \mathbf{true}$  }> /  $st \rightarrow_b$  <{  $\mathbf{true}$  }>
| BS_AndTrueFalse : <{  $\mathbf{true} \ \&\&\ \mathbf{false}$  }> /  $st \rightarrow_b$  <{  $\mathbf{false}$  }>

```

where " $b \text{ '/' } st \text{ '}\rightarrow b' b' \text{ " := (bstep } st \text{ } b \text{ } b')$.

The semantics of commands is the interesting part. We need two small tricks to make it work:

- We use *skip* as a “command value” – i.e., a command that has reached a normal form.
 - An assignment command reduces to *skip* (and an updated state).
 - The sequencing command waits until its left-hand subcommand has reduced to *skip*, then throws it away so that reduction can continue with the right-hand subcommand.
- We reduce a *while* command a single step by transforming it into a conditional followed by the same *while*.

(There are other ways of achieving the effect of the latter trick, but they all share the feature that the original *while* command is stashed away somewhere while a single copy of the loop body is being reduced.)

Reserved Notation " $t \text{ '/' } st \text{ '}\rightarrow t' \text{ '/' } st' \text{ " }$

(at level 40, *st* at *next* level, *t'* at *next* level, left associativity).

Inductive **cstep** : (**com** $\times$ state) $\rightarrow$ (**com** $\times$ state) $\rightarrow$ Prop :=

- | CS_AsgnStep : $\forall st \ i \ a1 \ a1',$
 $a1 \ / \ st \rightarrow a \ a1' \rightarrow$
 $\langle \{ i := a1 \} \rangle / st \rightarrow \langle \{ i := a1' \} \rangle / st$
- | CS_Asgn : $\forall st \ i \ (n : \mathbf{nat}),$
 $\langle \{ i := n \} \rangle / st \rightarrow \langle \{ \mathbf{skip} \} \rangle / (i \mathbf{!} \rightarrow n ; st)$
- | CS_SeqStep : $\forall st \ c1 \ c1' \ st' \ c2,$
 $c1 \ / \ st \rightarrow c1' \ / \ st' \rightarrow$
 $\langle \{ c1 ; c2 \} \rangle / st \rightarrow \langle \{ c1' ; c2 \} \rangle / st'$
- | CS_SeqFinish : $\forall st \ c2,$
 $\langle \{ \mathbf{skip} ; c2 \} \rangle / st \rightarrow c2 \ / \ st$
- | CS_IfStep : $\forall st \ b1 \ b1' \ c1 \ c2,$
 $b1 \ / \ st \rightarrow b \ b1' \rightarrow$
 $\langle \{ \mathbf{if } b1 \mathbf{ then } c1 \mathbf{ else } c2 \mathbf{ end} \} \rangle / st$
 $\rightarrow$
 $\langle \{ \mathbf{if } b1' \mathbf{ then } c1 \mathbf{ else } c2 \mathbf{ end} \} \rangle / st$
- | CS_IfTrue : $\forall st \ c1 \ c2,$
 $\langle \{ \mathbf{if true then } c1 \mathbf{ else } c2 \mathbf{ end} \} \rangle / st \rightarrow c1 \ / \ st$
- | CS_IfFalse : $\forall st \ c1 \ c2,$
 $\langle \{ \mathbf{if false then } c1 \mathbf{ else } c2 \mathbf{ end} \} \rangle / st \rightarrow c2 \ / \ st$
- | CS_While : $\forall st \ b1 \ c1,$
 $\langle \{ \mathbf{while } b1 \mathbf{ do } c1 \mathbf{ end} \} \rangle / st$

```

->
<{ if b1 then c1; while b1 do c1 end else skip end }> / st

where " t '/' st '->' t' '/' st' " := (cstep (t, st) (t', st')).

```

8.6 Concurrent Imp

Finally, to show the power of this definitional style, let's enrich Imp with a new form of command that runs two subcommands in parallel and terminates when both have terminated. To reflect the unpredictability of scheduling, the actions of the subcommands may be interleaved in any order, but they share the same memory and can communicate by reading and writing the same variables.

For example:

- This program sets *X* to 0 in one thread and 1 in another, leaving it set to either 0 or 1 at the end:
 $X := 0 \parallel X := 1$
- This one leaves *X* set to one of 0, 1, 2, or 3 at the end:
 $X := 0; (X := X+2 \parallel X := X+1 \parallel X := 0)$

Module CIMP.

```

Inductive com : Type :=
| CSkip : com
| CAsgn : string → aexp → com
| CSeq : com → com → com
| Clf : bexp → com → com → com
| CWhile : bexp → com → com
| CPar : com → com → com.

```

Notation "*x* '||' *y*" := (CPar *x* *y*)
(in *custom com* at level 100, right associativity,
format "[*v* *x* '/' '||' '/' *y* ']'").

Notation "'skip'" := CSkip
(in *custom com* at level 0) : *com_scope*.

Notation "*x* := *y*" := (CAsgn *x* *y*)
(in *custom com* at level 0, *x* constr at level 0, *y* at level 85, no associativity,
format "*x* := *y*") : *com_scope*.

Notation "*x* ; *y*" := (CSeq *x* *y*)
(in *custom com* at level 90,
right associativity,
format "[*v* *x* ; '/' *y* ']'") : *com_scope*.

Notation "if' x 'then' y 'else' z 'end'" := (Clf x y z)
 (in *custom com* at level 89, x at level 99, y at level 99, z at level 99,
format "[v' 'if' x 'then' '/' y '/' 'else' '/' z '/' 'end' ']'") : *com_scope*.
 Notation "while' x 'do' y 'end'" := (CWhile x y)
 (in *custom com* at level 89, x at level 99, y at level 99,
format "[v' 'while' x 'do' '/' y '/' 'end' ']'") : *com_scope*.
 Inductive **cstep** : (**com** $\times$ state) $\rightarrow$ (**com** $\times$ state) $\rightarrow$ Prop :=

 | CS_AsgnStep : $\forall st\ i\ a1\ a1'$,
 $a1 / st \rightarrow a\ a1' \rightarrow$
 $\langle \{ i := a1 \} \rangle / st \rightarrow \langle \{ i := a1' \} \rangle / st$
 | CS_Asgn : $\forall st\ i\ (n : \mathbf{nat})$,
 $\langle \{ i := n \} \rangle / st \rightarrow \langle \{ \text{skip} \} \rangle / (i !\rightarrow n ; st)$
 | CS_SeqStep : $\forall st\ c1\ c1'\ st'\ c2$,
 $c1 / st \rightarrow c1' / st' \rightarrow$
 $\langle \{ c1 ; c2 \} \rangle / st \rightarrow \langle \{ c1' ; c2 \} \rangle / st'$
 | CS_SeqFinish : $\forall st\ c2$,
 $\langle \{ \text{skip} ; c2 \} \rangle / st \rightarrow c2 / st$
 | CS_IfStep : $\forall st\ b1\ b1'\ c1\ c2$,
 $b1 / st \rightarrow b\ b1' \rightarrow$
 $\langle \{ \text{if } b1 \text{ then } c1 \text{ else } c2 \text{ end} \} \rangle / st$
 $\rightarrow$
 $\langle \{ \text{if } b1' \text{ then } c1 \text{ else } c2 \text{ end} \} \rangle / st$
 | CS_IfTrue : $\forall st\ c1\ c2$,
 $\langle \{ \text{if true then } c1 \text{ else } c2 \text{ end} \} \rangle / st \rightarrow c1 / st$
 | CS_IfFalse : $\forall st\ c1\ c2$,
 $\langle \{ \text{if false then } c1 \text{ else } c2 \text{ end} \} \rangle / st \rightarrow c2 / st$
 | CS_While : $\forall st\ b1\ c1$,
 $\langle \{ \text{while } b1 \text{ do } c1 \text{ end} \} \rangle / st$
 $\rightarrow$
 $\langle \{ \text{if } b1 \text{ then } c1 ; \text{while } b1 \text{ do } c1 \text{ end else skip end} \} \rangle / st$

 | CS_Par1 : $\forall st\ c1\ c1'\ c2\ st'$,
 $c1 / st \rightarrow c1' / st' \rightarrow$
 $\langle \{ c1 \parallel c2 \} \rangle / st \rightarrow \langle \{ c1' \parallel c2 \} \rangle / st'$
 | CS_Par2 : $\forall st\ c1\ c2\ c2'\ st'$,
 $c2 / st \rightarrow c2' / st' \rightarrow$
 $\langle \{ c1 \parallel c2 \} \rangle / st \rightarrow \langle \{ c1 \parallel c2' \} \rangle / st'$
 | CS_ParDone : $\forall st$,
 $\langle \{ \text{skip} \parallel \text{skip} \} \rangle / st \rightarrow \langle \{ \text{skip} \} \rangle / st$

 where " $t' / st' \rightarrow t' / st'$ " := (**cstep** (t, st) (t', st')).

Notation " $t \text{ '/' } st \text{ '}\rightarrow^*$ " $t' \text{ '/' } st'$ " :=
 (multi cstep (t, st) (t', st'))
 (at level 40, st at next level, t' at next level, left associativity).

Among the many interesting properties of this language is the fact that the following program can terminate with the variable X set to any value.

Definition par_loop : com :=
 <{
 $Y := 1$
 ||
 while ($Y = 0$) do $X := X + 1$ end
 }>.

In particular, it can terminate with X set to 0:

Example par_loop_example_0:
 $\exists st',$
 par_loop / empty_st $\rightarrow^*$ <{skip}> / st'
 $\wedge st' X = 0$.

Proof.

```

unfold par_loop.
eexists. split.
- eapply multi_step.
  + apply CS_Par1. apply CS_Asgn.
  + eapply multi_step.
    × apply CS_Par2. apply CS_While.
    × eapply multi_step.
      - apply CS_Par2. apply CS_IfStep.
        apply BS_Eq1. apply AS_Id.
      - eapply multi_step.
        ++ apply CS_Par2. apply CS_IfStep.
          apply BS_Eq.
        ++ simpl. eapply multi_step.
          ** apply CS_Par2. apply CS_IfFalse.
          ** eapply multi_step.
            — apply CS_ParDone.
            — eapply multi_refl.
- reflexivity.
```

Qed.

It can also terminate with X set to 2:

The following proofs are particularly “deep” – they require following the small step semantics in a particular strategy to exhibit the desired behavior. For that reason, they are a bit awkward to write with “forced bullets”. Nevertheless, we keep them because they emphasize that the witness for an evaluation by small-step semantics has a size that is proportional

to the number of steps taken. It would be an interesting exercise to write Rocq tactics that can help automate the construction of such proofs, but such a tactic would need to “search” among the many possibilities.

Example `par_loop_example_2`:

```

 $\exists st',$ 
  par_loop / empty_st ->* <{skip}> / st'
 $\wedge st' X = 2.$ 

```

Proof.

```

unfold par_loop.
eexists. split.
- eapply multi_step.
  + apply CS_Par2. apply CS_While.
  + eapply multi_step.
    × apply CS_Par2. apply CS_IfStep.
      apply BS_Eq1. apply AS_Id.
    × eapply multi_step.
      - apply CS_Par2. apply CS_IfStep.
        apply BS_Eq.
      - simpl. eapply multi_step.
        ++ apply CS_Par2. apply CS_IfTrue.
        ++ eapply multi_step.
          ** apply CS_Par2. apply CS_SeqStep.
            apply CS_AsgnStep. apply AS_P1. apply AS_Id.
          ** eapply multi_step.
            — apply CS_Par2. apply CS_SeqStep.
              apply CS_AsgnStep. apply AS_P.
            — eapply multi_step.
              +++ apply CS_Par2. apply CS_SeqStep.
                apply CS_Asgn.
              +++ eapply multi_step.
                *** apply CS_Par2. apply CS_SeqFinish.
                *** eapply multi_step.
                  — apply CS_Par2. apply CS_While.
                  — eapply multi_step.
                    +++++ apply CS_Par2. apply CS_IfStep.
                      apply BS_Eq1. apply AS_Id.
                    +++++ eapply multi_step.
                      **** apply CS_Par2. apply CS_IfStep.
                        apply BS_Eq.
                      **** simpl.
                        eapply multi_step.
                        — apply CS_Par2. apply CS_IfTrue.

```


AS_Id.

```

—— eapply multi_step.
+++++ apply CS_Par2. apply CS_SeqStep.
apply CS_AsgnStep. apply AS_P1. apply

+++++ eapply multi_step.
***** apply CS_Par2. apply CS_SeqStep.
apply CS_AsgnStep. apply AS_P.
***** eapply multi_step.
—— apply CS_Par2. apply CS_SeqStep.
apply CS_Asgn.
—— eapply multi_step.
+++++++ apply CS_Par1. apply CS_Asgn.
+++++++ eapply multi_step.
***** apply CS_Par2. apply CS_SeqFinish.
***** eapply multi_step.
——- apply CS_Par2. apply CS_While.
——- eapply multi_step.
+++++++ apply CS_Par2. apply CS_IfStep.
apply BS_Eq1. apply AS_Id.
+++++++ eapply multi_step.
***** apply CS_Par2. apply CS_IfStep.
apply BS_Eq.
***** simpl. eapply multi_step.
—— apply CS_Par2. apply CS_IfFalse.
—— eapply multi_step.
+++++++ apply CS_ParDone.
+++++++ eapply multi_refl.

```

- reflexivity. Qed.

More generally...

Exercise: 3 stars, standard, optional (par_body_n__Sn) Lemma par_body_n__Sn :

$\forall n \ st,$
 $st \ X = n \wedge st \ Y = 0 \rightarrow$
 $par\_loop \ / \ st \ \rightarrow^* \ par\_loop \ / \ (X \ !\rightarrow S \ n \ ; \ st).$

Proof.

Admitted.

□

Exercise: 3 stars, standard, optional (par_body_n) Lemma par_body_n : $\forall n \ st,$

$st \ X = 0 \wedge st \ Y = 0 \rightarrow$

$\exists st',$

$par\_loop \ / \ st \ \rightarrow^* \ par\_loop \ / \ st' \wedge st' \ X = n \wedge st' \ Y = 0.$

Proof.

Admitted.

□

... the above loop can exit with X having any value whatsoever.

Theorem `par_loop_any_X`:

$\forall n, \exists st',$
 `par_loop / empty_st ->* <{skip}> / st'`
 $\wedge st' X = n.$

Proof.

```
intros n.
destruct (par_body_n n empty_st).
- split; reflexivity.
- rename x into st.
inversion H as [H' [HX HY] ]; clear H.
 $\exists (Y \rightarrow 1 ; st).$  split.
+ eapply multi_trans with (par_loop, st).
   $\times$  apply H'.
   $\times$  eapply multi_step.
    - apply CS_Par1. apply CS_Asgn.
    - eapply multi_step.
      ++ apply CS_Par2. apply CS_While.
      ++ eapply multi_step.
        ** apply CS_Par2. apply CS_IfStep.
          apply BS_Eq1. apply AS_Id.
        ** rewrite t_update_eq.
          eapply multi_step.
          — apply CS_Par2. apply CS_IfStep.
            apply BS_Eq.
          — simpl. eapply multi_step.
            +++ apply CS_Par2. apply CS_IfFalse.
            +++ eapply multi_step.
              *** apply CS_ParDone.
              *** apply multi_refl.
      + rewrite t_update_neq.
         $\times$  assumption.
         $\times$  intro X; inversion X.
```

Qed.

End CIMP.

8.7 A Small-Step Stack Machine

Our last example is a small-step semantics for the stack machine example from the `Imp` chapter of *Logical Foundations*.

Definition `stack` := `list nat`.

Definition `prog` := `list sinstr`.

Inductive `stack_step` (`st` : state) : prog × stack → prog × stack → Prop :=
 | `SS_Push` : ∀ `stk n p`,
 `stack_step st` (`SPush n :: p, stk`) (`p, n :: stk`)
 | `SS_Load` : ∀ `stk i p`,
 `stack_step st` (`SLoad i :: p, stk`) (`p, st i :: stk`)
 | `SS_P` : ∀ `stk n m p`,
 `stack_step st` (`SPlus :: p, n :: m :: stk`) (`p, (m+n) :: stk`)
 | `SS_Minus` : ∀ `stk n m p`,
 `stack_step st` (`SMinus :: p, n :: m :: stk`) (`p, (m-n) :: stk`)
 | `SS_Mult` : ∀ `stk n m p`,
 `stack_step st` (`SMult :: p, n :: m :: stk`) (`p, (m×n) :: stk`).

Theorem `stack_step_deterministic` : ∀ `st`,
 deterministic (`stack_step st`).

Proof.

 unfold deterministic. intros `st x y1 y2 H1 H2`.
 induction `H1`; inversion `H2`; reflexivity.

Qed.

Definition `stack_multistep st` := `multi (stack_step st)`.

Exercise: 3 stars, advanced (compiler_is_correct) Remember the definition of `compile` for `aexp` given in the `Imp` chapter of *Logical Foundations*. We want now to prove `s_compile` correct with respect to the stack machine.

Copy your definition of `s_compile` from `Imp` here, then state what it means for the compiler to be correct according to the stack machine small step semantics, and then prove it.

Definition `compiler_is_correct_statement` : Prop
 . *Admitted*.

Theorem `compiler_is_correct` : `compiler_is_correct_statement`.

Proof.

Admitted.

□

8.8 Aside: A *normalize* Tactic

When experimenting with definitions of programming languages in Rocq, we often want to see what a particular concrete term steps to – i.e., we want to find proofs for goals of the form $t \rightarrow^* t'$, where t is a completely concrete term and t' is unknown. These proofs are quite tedious to do by hand. Consider, for example, reducing an arithmetic expression using the small-step relation `astep`.

Example `step_example1` :

```
(P (C 3) (P (C 3) (C 4)))  
->* (C 10).
```

Proof.

```
apply multi_step with (P (C 3) (C 7)).  
- apply ST_P2.  
  + apply v_C.  
  + apply ST_PCC.  
- apply multi_step with (C 10).  
  + apply ST_PCC.  
  + apply multi_refl.
```

Qed.

Proofs that one term normalizes to another must repeatedly apply `multi_step` until the term reaches a normal form. Fortunately, the sub-proofs for the intermediate steps are simple enough that `auto`, with appropriate hints, can solve them.

Hint Constructors `step value` : *core*.

Example `step_example1'` :

```
(P (C 3) (P (C 3) (C 4)))  
->* (C 10).
```

Proof.

```
eapply multi_step; auto. simpl.  
eapply multi_step; auto. simpl.  
apply multi_refl.
```

Qed.

The following custom `Tactic Notation` definition captures this pattern. In addition, before each step, we print out the current goal, so that we can follow how the term is being reduced.

```
Tactic Notation "print_goal" :=  
  match goal with  $\vdash ?x \Rightarrow$  idtac  $x$  end.
```

```
Tactic Notation "normalize" :=  
  repeat (print_goal; eapply multi_step ;  
    [ (eauto 10; fail) | simpl]);  
  apply multi_refl.
```

Example `step_example1''` :

```
(P (C 3) (P (C 3) (C 4)))
->* (C 10).
```

Proof.

normalize.

Qed.

The *normalize* tactic also provides a simple way to calculate the normal form of a term, by starting with a goal with an existentially bound variable.

```
Example step_example1''' :  $\exists e'$ ,
  (P (C 3) (P (C 3) (C 4)))
  ->*  $e'$ .
```

Proof.

eexists. normalize.

Qed.

This time, the trace is:

```
(P (C 3) (P (C 3) (C 4)) ->* ?e') (P (C 3) (C 7) ->* ?e') (C 10 ->* ?e')
where ?e' is the variable "guessed" by eapply.
```

Exercise: 1 star, standard (normalize_ex) *Theorem normalize_ex : $\exists e'$,*
 (P (C 3) (P (C 2) (C 1)))
 ->* $e' \wedge$ **value** e' .

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (normalize_ex') For comparison, prove it using *apply* instead of *eapply*.

```
Theorem normalize_ex' :  $\exists e'$ ,  

  (P (C 3) (P (C 2) (C 1)))  

  ->*  $e' \wedge$  value  $e'$ .
```

Proof.

Admitted.

□

Chapter 9

Library `PLF.Types`

9.1 Types: Type Systems

Our next major topic is *type systems* – static program analyses that classify expressions according to the “shapes” of their results. We’ll begin with a typed version of the simplest imaginable language, to introduce the basic ideas of types and typing rules and the fundamental theorems about type systems: *type preservation* and *progress*. In chapter `Stlc` we’ll move on to the *simply typed lambda-calculus*, which lives at the core of every modern functional programming language (including Rocq!).

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import Arith.
From PLF Require Import Maps.
From PLF Require Import Smallstep.
Set Default Goal Selector "!".
Hint Constructors multi : core.
```

9.2 Typed Arithmetic Expressions

To motivate the discussion of type systems, let’s begin as usual with a tiny toy language. We want it to have the potential for programs to go wrong because of runtime type errors, so we need something a tiny bit more complex than the language of constants and addition that we used in chapter `Smallstep`: a single kind of data (e.g., numbers) is too simple, but just two kinds (numbers and booleans) gives us enough material to tell an interesting story.

The language definition is completely routine.

9.2.1 Syntax

Here is the syntax, informally:

$t ::= \text{true} \mid \text{false} \mid \text{if } t \text{ then } t \text{ else } t \mid 0 \mid \text{succ } t \mid \text{pred } t \mid \text{iszero } t$ And here it is formally:

```
Module TM.
```

```

Inductive tm : Type :=
| tru : tm
| fls : tm
| ite : tm → tm → tm → tm
| zro : tm
| scc : tm → tm
| prd : tm → tm
| iszro : tm → tm.

Declare Custom Entry tm.
Declare Scope tm_scope.
Notation "'true'" := true (at level 1): tm_scope.
Notation "'true'" := (tru) (in custom tm at level 0): tm_scope.
Notation "'false'" := false (at level 1): tm_scope.
Notation "'false'" := (fls) (in custom tm at level 0): tm_scope.
Notation "<{ e }>" := e (e custom tm at level 99): tm_scope.
Notation "( x )" := x (in custom tm, x at level 99): tm_scope.
Notation "x" := x (in custom tm at level 0, x constr at level 0): tm_scope.
Notation "'0'" := (zro) (in custom tm at level 0): tm_scope.
Notation "'0'" := 0 (at level 1): tm_scope.
Notation "'succ' x" := (scc x) (in custom tm at level 90, x custom tm at level 80):
tm_scope.
Notation "'pred' x" := (prd x) (in custom tm at level 90, x custom tm at level 80):
tm_scope.
Notation "'iszero' x" := (iszro x) (in custom tm at level 80, x custom tm at level
70): tm_scope.
Notation "'if' c 'then' t 'else' e" := (ite c t e)
      (in custom tm at level 90, c custom tm at level 80,
      t custom tm at level 80, e custom tm at level 80): tm_scope.
Local Open Scope tm_scope.

Values are true, false, and numeric values... Inductive bvalue : tm → Prop :=
| bv_true : bvalue <{ true }>
| bv_false : bvalue <{ false }>.

Inductive nvalue : tm → Prop :=
| nv_0 : nvalue <{ 0 }>
| nv_succ : ∀ t, nvalue t → nvalue <{ succ t }>.

Definition value (t : tm) := bvalue t ∨ nvalue t.

Hint Constructors bvalue nvalue : core.
Hint Unfold value : core.

```

9.2.2 Operational Semantics

Here is the single-step relation, first informally...

(ST_IfTrue) if true then t1 else t2 -> t1
(ST_IfFalse) if false then t1 else t2 -> t2 t1 -> t1'
(ST_If) if t1 then t2 else t3 -> if t1' then t2 else t3 t1 -> t1'
(ST_Succ) succ t1 -> succ t1'
(ST_Pred0) pred 0 -> 0 numeric value v
(ST_PredSucc) pred (succ v) -> v t1 -> t1'
(ST_Pred) pred t1 -> pred t1'
(ST_IsZero0) iszero 0 -> true numeric value v
(ST_IsZeroSucc) iszero (succ v) -> false t1 -> t1'
(ST_IsZero) iszero t1 -> iszero t1' ... and then formally:

Reserved Notation "t '->' t'" (at level 40).

```

Inductive step : tm → tm → Prop :=
| ST_IfTrue : ∀ t1 t2,
  <{ if true then t1 else t2 }> -> t1
| ST_IfFalse : ∀ t1 t2,
  <{ if false then t1 else t2 }> -> t2
| ST_If : ∀ c c' t2 t3,
  c -> c' →
  <{ if c then t2 else t3 }> -> <{ if c' then t2 else t3 }>
| ST_Succ : ∀ t1 t1',
  t1 -> t1' →
  <{ succ t1 }> -> <{ succ t1' }>

```



```

| ST_Pred0 :
  <{ pred 0 }> -> <{ 0 }>
| ST_PredSucc : ∀ v,
  nvalue v →
  <{ pred (succ v) }> -> v
| ST_Pred : ∀ t1 t1',
  t1 -> t1' →
  <{ pred t1 }> -> <{ pred t1' }>
| ST_IsZero0 :
  <{ iszero 0 }> -> <{ true }>
| ST_IsZeroSucc : ∀ v,
  nvalue v →
  <{ iszero (succ v) }> -> <{ false }>
| ST_IsZero : ∀ t1 t1',
  t1 -> t1' →
  <{ iszero t1 }> -> <{ iszero t1' }>

```

where "t '→' t'" := (step t t').

Hint Constructors **step** : core.

The **nvalue** conditions in **ST_PredSucc** and *ST_IszeroSucc* are needed for determinism (will be proved in an optional exercise below).

Notice that the **step** relation doesn't care about whether the expression being stepped makes global sense – it just checks that the operation in the *next* reduction step is being applied to the right kinds of operands. For example, the term *succ true* cannot take a step, but the almost as obviously nonsensical term

succ (if true then true else true)
can take a step (once, before becoming stuck).

9.2.3 Normal Forms and Values

The first interesting thing to notice about this **step** relation is that the strong progress theorem from the **Smallstep** chapter fails here. That is, there are terms that are normal forms (they can't take a step) but not values (they are not included in our definition of possible "results of reduction").

Such terms are *stuck*.

Notation **step_normal_form** := (normal_form **step**).

Definition **stuck** (t : **tm**) : Prop :=
step_normal_form t ∧ ¬ value t.

Hint Unfold **stuck** : core.

Exercise: 2 stars, standard (some_term_is_stuck) Example **some_term_is_stuck** :

$\exists t, \text{stuck } t.$

Proof.

Admitted.

□

However, although values and normal forms are *not* the same in this language, the set of values is a subset of the set of normal forms.

This is important because it shows we did not accidentally define things so that some value could still take a step.

Exercise: 3 stars, standard (value_is_nf) Lemma `value_is_nf` : $\forall t,$
`value` $t \rightarrow \text{step\_normal\_form } t.$

Proof.

Admitted.

(Hint: You will reach a point in this proof where you need to use an induction to reason about a term that is known to be a numeric value. This induction can be performed either over the term itself or over the evidence that it is a numeric value. The proof goes through in either case, but you will find that one way is quite a bit shorter than the other. For the sake of the exercise, try to complete the proof both ways.)

□

Exercise: 3 stars, standard, optional (step_deterministic) Use `value_is_nf` to show that the `step` relation is also deterministic.

Theorem `step_deterministic`:

deterministic `step`.

Proof with `eauto`.

Admitted.

□

9.2.4 Typing

The next critical observation is that, although this language has stuck terms, they are always nonsensical, mixing booleans and numbers in a way that we don't even *want* to have a meaning. We can easily exclude such ill-typed terms by defining a *typing relation* that relates terms to the types (either numeric or boolean) of their final results.

Inductive `ty` : Type :=

| `Bool` : `ty`

| `Nat` : `ty`.

In informal notation, the typing relation is often written $\vdash t \text{ in } T$ and pronounced “ t has type T .” The $\vdash$ symbol is called a “turnstile.” Below, we’re going to see richer typing relations where one or more additional “context” arguments are written to the left of the turnstile. For the moment, the context is always empty.

(T_True) |- true \in Bool

(T_False) |- false \in Bool
|- t1 \in Bool |- t2 \in T |- t3 \in T

(T_If) |- if t1 then t2 else t3 \in T

(T_0) |- 0 \in Nat
|- t1 \in Nat

(T_Succ) |- succ t1 \in Nat
|- t1 \in Nat

(T_Pred) |- pred t1 \in Nat
|- t1 \in Nat

(T_IsZero) |- iszero t1 \in Bool

Declare Custom Entry ty.

Notation "'Nat'" := Nat (in custom ty).

Notation "'Bool'" := Bool (in custom ty).

Notation "x" := x (in custom ty, x global).

Reserved Notation "<{ ' |- ' t ' \in ' T }>"
(at level 0, t custom tm, T custom ty).

Inductive **has_type** : **tm** → **ty** → Prop :=

| T_True :
 <{ |- true \in Bool }>
| T_False :
 <{ |- false \in Bool }>
| T_If : ∀ *t1 t2 t3 T*,
 <{ |- *t1* \in Bool }> →
 <{ |- *t2* \in T }> →
 <{ |- *t3* \in T }> →
 <{ |- if *t1* then *t2* else *t3* \in T }>
| T_0 :
 <{ |- 0 \in Nat }>
| T_Succ : ∀ *t1*,
 <{ |- *t1* \in Nat }> →
 <{ |- succ *t1* \in Nat }>
| T_Pred : ∀ *t1*,
 <{ |- *t1* \in Nat }> →
 <{ |- pred *t1* \in Nat }>

```
| T_!szero : ∀ t1 ,
  <{ |- t1 \in Nat }> →
  <{ |- iszero t1 \in Bool }>
```

where "<{ ' |- ' t '\in' T }>" := (**has_type** t T).

Hint Constructors **has_type** : core.

Example **has_type_1** :

```
<{ |- if false then 0 else (succ 0) \in Nat }>.
```

Proof.

```
apply T_!f.
- apply T_!False.
- apply T_!0.
- apply T_!Succ. apply T_!0.
```

Qed.

(Since we've included all the constructors of the typing relation in the hint database, the **auto** tactic can actually find this proof automatically.)

It's important to realize that the typing relation is a *conservative* (or *static*) approximation: it does not consider what happens when the term is reduced – in particular, it does not calculate the type of its normal form.

Example **not_has_type** :

```
¬ <{ |- if false then 0 else true \in Bool }>.
```

Proof.

```
intros Contra. solve_by_inverts 2. Qed.
```

Example **not_has_type'** :

```
¬ <{ |- if iszero (succ 0) then succ false else true \in Bool }>.
```

Proof.

```
intros Contra. solve_by_inverts 2. Qed.
```

Exercise: 1 star, standard, optional (succ_hastype_nat__hastype_nat) Example

succ_hastype_nat__hastype_nat : ∀ *t*,

```
<{ |- succ t \in Nat }> →
```

```
<{ |- t \in Nat }>.
```

Proof.

```
Admitted.
```

□

Canonical forms

The following two lemmas capture the fundamental fact that the definitions of boolean and numeric values agree with the typing relation.

Lemma **bool_canonical** : ∀ *t*,

$\langle \{ \mid - t \text{ \texttt{in Bool}} \} \rangle \rightarrow \text{value } t \rightarrow \text{bvalue } t.$

Proof.

```
intros t HT [Hb | Hn].
- assumption.
- destruct Hn as [ | Hs].
  + inversion HT.
  + inversion HT.
```

Qed.

Lemma nat_canonical : $\forall t,$
 $\langle \{ \mid - t \text{ \texttt{in Nat}} \} \rangle \rightarrow \text{value } t \rightarrow \text{nvalue } t.$

Proof.

```
intros t HT [Hb | Hn].
- inversion Hb; subst; inversion HT.
- assumption.
```

Qed.

9.2.5 Progress

The typing relation enjoys two critical properties.

The first is that well-typed normal forms are not stuck – or conversely, if a term is well typed, then either it is a value or it can take at least one step. We call this *progress*.

Exercise: 3 stars, standard (finish_progress) Theorem progress : $\forall t \ T,$

$\langle \{ \mid - t \text{ \texttt{in T}} \} \rangle \rightarrow$
 $\text{value } t \vee \exists t', t \rightarrow t'.$

Complete the formal proof of the *progress* property. (Make sure you understand the parts we've given of the informal proof in the following exercise before starting – this will save you a lot of time.) Proof.

```
intros t T HT.
induction HT; auto.
-
  right. destruct IHHT1.
  +
    apply (bool_canonical t1 HT1) in H.
    destruct H.
    ×  $\exists t2.$  auto.
    ×  $\exists t3.$  auto.
  +
    destruct H as [t1' H1].
     $\exists \langle \text{if } t1' \text{ then } t2 \text{ else } t3 \rangle.$  auto.
```

Admitted.

□

Exercise: 3 stars, advanced, optional (finish_progress_informal) Complete the corresponding informal proof:

Theorem: If $\vdash t \text{ \texttt{in} } T$, then either t is a value or else $t \rightarrow t'$ for some t' .

Proof: By induction on a derivation of $\vdash t \text{ \texttt{in} } T$.

- If the last rule in the derivation is T_If , then $t = \text{if } t1 \text{ then } t2 \text{ else } t3$, with $\vdash t1 \text{ \texttt{in} } Bool$, $\vdash t2 \text{ \texttt{in} } T$ and $\vdash t3 \text{ \texttt{in} } T$. By the IH, either $t1$ is a value or else $t1$ can step to some $t1'$.
 - If $t1$ is a value, then by the canonical forms lemmas and the fact that $\vdash t1 \text{ \texttt{in} } Bool$ we have that $t1$ is a **bvalue** – i.e., it is either **true** or **false**. If $t1 = \text{true}$, then t steps to $t2$ by ST_IfTrue , while if $t1 = \text{false}$, then t steps to $t3$ by $ST\_IfFalse$. Either way, t can step, which is what we wanted to show.
 - If $t1$ itself can take a step, then, by ST_If , so can t .

•

Definition `manual_grade_for_finish_progress_informal` : `option (nat×string)` := `None`.

□

This theorem is more interesting than the strong progress theorem that we saw in the `Smallstep` chapter, where *all* normal forms were values. Here a term can be stuck, but only if it is ill typed.

9.2.6 Type Preservation

The second critical property of typing is that, when a well-typed term takes a step, the result is a well-typed term (of the same type).

Exercise: 2 stars, standard (finish_preservation) **Theorem** `preservation` : $\forall t t' T,$
 $\langle \{ \vdash t \text{ \texttt{in} } T \} \rangle \rightarrow$
 $t \rightarrow t' \rightarrow$
 $\langle \{ \vdash t' \text{ \texttt{in} } T \} \rangle.$

Complete the formal proof of the `preservation` property. (Again, make sure you understand the informal proof fragment in the following exercise first.)

Proof.

```

intros t t' T HT HE.
generalize dependent t'.
induction HT;

    intros t' HE;

    try solve_by_invert.

```

```

- inversion HE; subst; clear HE.
+ assumption.
+ assumption.
+ apply T_lf; try assumption.
  apply IHHT1; assumption.
Admitted.
□

```

Exercise: 3 stars, advanced, optional (finish_preservation_informal) Complete the following informal proof:

Theorem: If $\vdash t \text{ \texttt{in} } T$ and $t \rightarrow t'$, then $\vdash t' \text{ \texttt{in} } T$.

Proof: By induction on a derivation of $\vdash t \text{ \texttt{in} } T$.

- If the last rule in the derivation is T_lf , then $t = \text{if } t1 \text{ then } t2 \text{ else } t3$, with $\vdash t1 \text{ \texttt{in} } Bool$, $\vdash t2 \text{ \texttt{in} } T$ and $\vdash t3 \text{ \texttt{in} } T$.

Inspecting the rules for the small-step reduction relation and remembering that t has the form `if ...`, we see that the only ones that could have been used to prove $t \rightarrow t'$ are ST_lfTrue , $ST\_lfFalse$, or ST_lf .

- If the last rule was ST_lfTrue , then $t' = t2$. But we know that $\vdash t2 \text{ \texttt{in} } T$, so we are done.
- If the last rule was $ST\_lfFalse$, then $t' = t3$. But we know that $\vdash t3 \text{ \texttt{in} } T$, so we are done.
- If the last rule was ST_lf , then $t' = \text{if } t1' \text{ then } t2 \text{ else } t3$, where $t1 \rightarrow t1'$. We know $\vdash t1 \text{ \texttt{in} } Bool$ so, by the IH, $\vdash t1' \text{ \texttt{in} } Bool$. The T_lf rule then gives us $\vdash \text{if } t1' \text{ then } t2 \text{ else } t3 \text{ \texttt{in} } T$, as required.

•

Definition `manual_grade_for_finish_preservation_informal` : `option (nat × string)` := `None`.

□

Exercise: 3 stars, standard (preservation_alternate_proof) Now prove the same property again by induction on the *evaluation* derivation instead of on the typing derivation. Begin by carefully reading and thinking about the first few lines of the above proofs to make sure you understand what each one is doing. The set-up for this proof is similar, but not exactly the same.

Theorem `preservation'` : $\forall t t' T,$

```

<{  $\vdash t \text{ \texttt{in} } T$  }>  $\rightarrow$ 
 $t \rightarrow t' \rightarrow$ 
<{  $\vdash t' \text{ \texttt{in} } T$  }>.

```

Proof with `eauto`.

Admitted.

□

The preservation theorem is often called *subject reduction*, because it tells us what happens when the “subject” of the typing relation is reduced. This terminology comes from thinking of typing statements as sentences, where the term is the subject and the type is the predicate.

9.2.7 Type Soundness

Putting progress and preservation together, we see that a well-typed term can never reach a stuck state.

Definition `multistep` := (`multi step`).

Notation "`t1 '→*' t2`" := (`multistep t1 t2`) (at level 40).

Corollary `soundness` : $\forall t\ t'\ T,$

$\langle\{ \mid -\ t \ \text{in } T \}\rangle \rightarrow$

$t \rightarrow^* t' \rightarrow$

$\sim(\text{stuck } t').$

Proof.

intros `t t' T HT P`. induction `P`; intros [`R S`].

- apply `progress` in `HT`. destruct `HT`; auto.

- apply `IHP`.

+ apply `preservation` with (`t := x`); auto.

+ unfold `stuck`. split; auto.

Qed.

9.2.8 Additional Exercises

Exercise: 3 stars, standard, especially useful (`subject_expansion`) Having seen the subject reduction property, one might wonder whether the opposite property – *subject expansion* – also holds. That is, is it always the case that, if $t \rightarrow t'$ and $\mid -\ t' \ \text{in } T$, then $\mid -\ t \ \text{in } T$? If so, prove it. If not, give a counter-example.

Theorem `subject_expansion`:

$(\forall t\ t'\ T, t \rightarrow t' \wedge \langle\{ \mid -\ t' \ \text{in } T \}\rangle \rightarrow \langle\{ \mid -\ t \ \text{in } T \}\rangle)$

$\vee$

$\neg (\forall t\ t'\ T, t \rightarrow t' \wedge \langle\{ \mid -\ t' \ \text{in } T \}\rangle \rightarrow \langle\{ \mid -\ t \ \text{in } T \}\rangle).$

Proof.

Admitted.

□

Exercise: 2 stars, standard (`variation1`) Suppose that we add this new rule to the typing relation:

$\mid T\_SuccBool : \text{forall } t, \langle\{ \mid -\ t \ \text{in } Bool \}\rangle \rightarrow \langle\{ \mid -\ succ\ t \ \text{in } Bool \}\rangle$

Which of the following properties remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

Definition `manual_grade_for_variation1` : **option** (**nat**×**string**) := **None**.

□

Exercise: 2 stars, standard (variation2) Suppose, instead, that we add this new rule to the **step** relation:

| ST_Funny1 : forall t2 t3, (<{ if true then t2 else t3 }>) -> t3

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

Definition `manual_grade_for_variation2` : **option** (**nat**×**string**) := **None**.

□

Exercise: 2 stars, standard, optional (variation3) Suppose instead that we add this rule:

| ST_Funny2 : forall t1 t2 t2' t3, t2 -> t2' -> (<{ if t1 then t2 else t3 }>) -> (<{ if t1 then t2' else t3 }>)

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example. □

Exercise: 2 stars, standard, optional (variation4) Suppose instead that we add this rule:

| ST_Funny3 : (<{ pred false }>) -> (<{ pred (pred false) }>)

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example. □

Exercise: 2 stars, standard, optional (variation5) Suppose instead that we add this rule:

| T_Funny4 : |- <{ 0 }> \in Bool

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example. □

Exercise: 2 stars, standard, optional (variation6) Suppose instead that we add this rule:

| T_Funny5 : |- pred 0 }> \in Bool

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example. $\square$

Exercise: 3 stars, standard, optional (more_variations) Make up some exercises of your own along the same lines as the ones above. Try to find ways of selectively breaking properties – i.e., ways of changing the definitions that break just one of the properties and leave the others alone.

Exercise: 1 star, standard (remove_pred0) The reduction rule `ST_Pred0` is a bit counter-intuitive: we might feel that it makes more sense for the predecessor of 0 to be undefined, rather than being defined to be 0. Can we achieve this simply by removing the rule from the definition of `step`? Would doing so create any problems elsewhere?

Definition `manual_grade_for_remove_pred0` : `option (nat×string)` := `None`.

$\square$

Exercise: 4 stars, advanced (prog_pres_bigstep) Suppose our evaluation relation is defined in the big-step style. State appropriate analogs of the progress and preservation properties. (You do not need to prove them.)

Can you see any limitations of either of your properties? Do they allow for nonterminating programs? Why might we prefer the small-step semantics for stating preservation and progress?

Definition `manual_grade_for_prog_pres_bigstep` : `option (nat×string)` := `None`.

$\square$ End TM.

Chapter 10

Library `PLF.Stlc`

10.1 Stlc: The Simply Typed Lambda-Calculus

The simply typed lambda-calculus (STLC) is a tiny core calculus embodying the key concept of *functional abstraction*. This concept shows up in pretty much every real-world programming language in some form (functions, procedures, methods, etc.).

We will follow exactly the same pattern as in the previous chapter when formalizing this calculus (syntax, small-step semantics, typing rules) and its main properties (progress and preservation). The new technical challenges arise from the mechanisms of *variable binding* and *substitution*. It will take some work to deal with these.

The STLC lives in the lower-left front corner of the famous *lambda cube* (also called the *Barendregt Cube*), which visualizes three sets of features that can be added to its simple core:

Calculus of Constructions type operators +——+ /| /| /| /| polymorphism +——+
|| || || +——|+ | /| /| /| /| dependent types STLC

Moving from bottom to top in the cube corresponds to adding *polymorphic types* like $\forall X, X \rightarrow X$. Adding *just* polymorphism gives us the famous Girard-Reynolds calculus, System F.

Moving from front to back corresponds to adding *type operators* like `list`.

Moving from left to right corresponds to adding *dependent types* like $\forall n, \text{array-of-size } n$.

The top right corner on the back, which combines all three features, is called the *Calculus of Constructions*. First studied by Coquand and Huet, it forms the foundation of Rocq's logic.

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
```

```
From Stdlib Require Import Strings.String.
```

```
From PLF Require Import Maps.
```

```
From PLF Require Import Smallstep.
```

```
Set Default Goal Selector "!".
```

10.2 Overview

The STLC is built on some collection of *base types*: booleans, numbers, strings, etc. The exact choice of base types doesn't matter much – the definition of the language as well as its theoretical properties work out the same no matter what we choose – so for the sake of brevity let's take just `Bool` for the moment. In the next chapter we'll see how to add more base types, and in later chapters we'll enrich the pure STLC with other useful constructs like pairs, records, subtyping, and mutable state.

Starting from boolean constants and conditionals, we add three things:

- variables
- function abstractions
- application

This gives us the following collection of abstract syntax constructors (written out first in informal BNF notation – we'll formalize it below).

$t ::= x$ (variable) | $\backslash x:T, t$ (abstraction) | $t_1 t_2$ (application) | `true` (constant true) | `false` (constant false) | `if t1 then t2 else t3` (conditional) The $\backslash$ symbol in a function abstraction $\backslash x:T, t$ is usually written as a Greek letter “lambda” (hence the name of the calculus). The variable x is called the *parameter* to the function; the term t is its *body*. The annotation $:T$ specifies the type of arguments that the function can be applied to.

If you've seen lambda-calculus notation elsewhere, you might be wondering why abstraction is written here as $\backslash x:T, t$ instead of the usual “ $\backslash x:T. t$ ”. The reason is that some user interfaces for interacting with Rocq use periods to separate a file into “sentences” to be passed separately to the Rocq top level.

Some examples:

- $\backslash x:\text{Bool}, x$

The identity function for booleans.

- $(\backslash x:\text{Bool}, x) \text{ true}$

The identity function for booleans, applied to the boolean `true`.

- $\backslash x:\text{Bool}, \text{if } x \text{ then false else true}$

The boolean “not” function.

- $\backslash x:\text{Bool}, \text{true}$

The constant function that takes every (boolean) argument to `true`.

- $\backslash x:\text{Bool}, \backslash y:\text{Bool}, x$

A two-argument function that takes two booleans and returns the first one.

(As in Rocq, a two-argument function in the lambda-calculus is really a one-argument function whose body is also a one-argument function.)

- $(\lambda x:\text{Bool}, \lambda y:\text{Bool}, x) \text{ false true}$

A two-argument function that takes two booleans and returns the first one, applied to the booleans `false` and `true`.

(As in Rocq, application associates to the left – i.e., this expression is parsed as $((\lambda x:\text{Bool}, \lambda y:\text{Bool}, x) \text{ false}) \text{ true}$.)

- $\lambda f:\text{Bool} \rightarrow \text{Bool}, f (f \text{ true})$

A higher-order function that takes a *function* f (from booleans to booleans) as an argument, applies f to `true`, and applies f again to the result.

- $(\lambda f:\text{Bool} \rightarrow \text{Bool}, f (f \text{ true})) (\lambda x:\text{Bool}, \text{false})$

The same higher-order function, applied to the constantly `false` function.

As the last several examples show, the STLC is a language of *higher-order* functions: we can write down functions that take other functions as arguments and/or return other functions as results.

The STLC doesn’t provide any primitive syntax for defining *named* functions: i.e., all functions are “anonymous.” We’ll see in chapter `MoreStlc` that it is easy to add named functions – indeed, the fundamental naming and binding mechanisms are exactly the same.

The *types* of the STLC include `Bool`, which classifies the boolean constants `true` and `false` as well as more complex computations that yield booleans, plus *arrow types* that classify functions.

$T ::= \text{Bool} \mid T \rightarrow T$

For example:

- $\lambda x:\text{Bool}, \text{false}$ has type $\text{Bool} \rightarrow \text{Bool}$
- $\lambda x:\text{Bool}, x$ has type $\text{Bool} \rightarrow \text{Bool}$
- $(\lambda x:\text{Bool}, x) \text{ true}$ has type Bool
- $\lambda x:\text{Bool}, \lambda y:\text{Bool}, x$ has type $\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$ (i.e., $\text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$)
- $(\lambda x:\text{Bool}, \lambda y:\text{Bool}, x) \text{ false}$ has type $\text{Bool} \rightarrow \text{Bool}$
- $(\lambda x:\text{Bool}, \lambda y:\text{Bool}, x) \text{ false true}$ has type Bool

10.3 Syntax

We next formalize the syntax of the STLC.

`Module STLC.`

10.3.1 Types

```
Inductive ty : Type :=  
  | Ty_Bool : ty  
  | Ty_Arrow : ty → ty → ty.
```

10.3.2 Terms

```
Inductive tm : Type :=  
  | tm_var : string → tm  
  | tm_app : tm → tm → tm  
  | tm_abs : string → ty → tm → tm  
  | tm_true : tm  
  | tm_false : tm  
  | tm_if : tm → tm → tm → tm.
```

We need some notation magic to set up the concrete syntax, as we did in the `Imp` chapter...

```
Declare Scope stlc_scope.  
Delimit Scope stlc_scope with stlc.  
Open Scope stlc_scope.  
  
Declare Custom Entry stlc_ty.  
Declare Custom Entry stlc_tm.  
  
Notation "x" := x (in custom stlc_ty at level 0, x global) : stlc_scope.  
Notation "<{{ x }}>" := x (x custom stlc_ty).  
  
Notation "( t )" := t (in custom stlc_ty at level 0, t custom stlc_ty) : stlc_scope.  
Notation "S -> T" := (Ty_Arrow S T) (in custom stlc_ty at level 99, right associativity)  
: stlc_scope.  
  
Notation "$ ( t )" := t (in custom stlc_ty at level 0, t constr) : stlc_scope.  
Notation "'Bool'" := Ty_Bool (in custom stlc_ty at level 0) : stlc_scope.  
Notation "'if' x 'then' y 'else' z" :=  
  (tm_if x y z) (in custom stlc_tm at level 200,  
    x custom stlc_tm,  
    y custom stlc_tm,  
    z custom stlc_tm at level 200,  
    left associativity).  
Notation "'true'" := true (at level 1).  
Notation "'true'" := tm_true (in custom stlc_tm at level 0).  
Notation "'false'" := false (at level 1).  
Notation "'false'" := tm_false (in custom stlc_tm at level 0).
```

We'll write types inside of `<{{ ... }}>` brackets:

```

Check <{{ Bool }}>.
Check <{{ Bool → Bool }}>.
Check <{{ (Bool → Bool) → Bool }}>.

Notation "$ ( x )" := x (in custom stlc_tm at level 0, x constr, only parsing) :
stlc_scope.
Notation "x" := x (in custom stlc_tm at level 0, x constr at level 0) : stlc_scope.
Notation "<{ e }>" := e (e custom stlc_tm at level 200) : stlc_scope.
Notation "( x )" := x (in custom stlc_tm at level 0, x custom stlc_tm) : stlc_scope.
Notation "x y" := (tm_app x y) (in custom stlc_tm at level 10, left associativity)
: stlc_scope.
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc_tm at level 200, x global,
                  t custom stlc_ty,
                  y custom stlc_tm at level 200,
                  left associativity).

Coercion tm_var : string >-> tm.
Arguments tm_var _%_string.

Definition x : string := "x".
Definition y : string := "y".
Definition z : string := "z".
Hint Unfold x : core.
Hint Unfold y : core.
Hint Unfold z : core.

```

The upshot of these notation definitions is that we can write STLC terms in these brackets: $\langle \{ \dots \} \rangle$ (similar to how we wrote Imp commands) and STLC types in these brackets: $\langle \{ \{ \dots \} \} \rangle$.

As before, we can use $\$(\dots)$ to “escape” to arbitrary Rocq notation.

And terms inside of $\langle \{ \dots \} \rangle$ brackets:

Examples...

```

Notation idB :=
  <{ \x:Bool, x }>.

Notation idBB :=
  <{ \x:Bool→Bool, x }>.

Notation idBBBB :=
  <{ \x: (Bool→Bool)→(Bool→Bool), x }>.

Notation k := <{ \x:Bool, \y:Bool, x }>.

Notation notB := <{ \x:Bool, if x then false else true }>.

```

Note that an abstraction $\backslash x:T, t$ (formally, `tm_abs x T t`) is always annotated with the type T of its parameter, in contrast to Rocq (and other functional languages like ML, Haskell,

etc.), which use type inference to fill in missing annotations. We're not considering type inference at all here.

(We write these as **Notations** rather than **Definitions** to make things easier for **auto**.)

10.4 Operational Semantics

To define the small-step semantics of STLC terms, we begin, as always, by defining the set of values. Next, we define the critical notions of *free variables* and *substitution*, which are used in the reduction rule for application expressions. And finally we give the small-step relation itself.

10.4.1 Values

To define the values of the STLC, we have a few cases to consider.

First, for the boolean part of the language, the situation is clear: **true** and **false** are the only values. An **if** expression is never a value.

Second, an application is not a value: it represents a function being invoked on some argument, which clearly still has work left to do.

Third, for abstractions, we have a choice:

- We can say that $\lambda x:T. t$ is a value only when t is a value – i.e., only if the function's body has been reduced (as much as it can be without knowing what argument it is going to be applied to).
- Or we can say that $\lambda x:T. t$ is always a value, no matter whether t is one or not – in other words, we can say that reduction stops at abstractions.

Our usual way of evaluating expressions in Gallina makes the first choice – for example, Compute (fun x:bool => 3 + 4)
yields:

fun x:bool => 7

But Gallina is rather unusual in this respect. Most functional programming languages make the second choice – reduction of a function's body only begins when the function is actually applied to an argument.

We also make the second choice here.

```
Inductive value : tm → Prop :=
| v_abs : ∀ x T2 t1,
  value <{\x:T2, t1}>
| v_true :
  value <{true}>
| v_false :
  value <{false}>.
```


Hint Constructors **value** : core.

Finally, we must consider what constitutes a *complete* program.

Intuitively, a “complete program” must not refer to any undefined variables. We’ll see shortly how to define the *free* variables in a STLC term. A complete program, then, is one that is *closed* – that is, that contains no free variables.

(Conversely, a term that may contain free variables is often called an *open term*.)

Having made the choice not to reduce under abstractions, we don’t need to worry about whether variables are values, since we’ll always be reducing programs “from the outside in,” and that means the **step** relation will always be working with closed terms.

10.4.2 Substitution

Now we come to the heart of the STLC: the operation of substituting one term for a variable in another term. This operation is used below to define the operational semantics of function application, where we will need to substitute the argument term for the function parameter in the function’s body. For example, we reduce

(\x:Bool, if x then true else x) false

to

if false then true else false

by substituting **false** for the parameter **x** in the body of the function.

In general, we need to be able to substitute some given term **s** for occurrences of some variable **x** in another term **t**. Informally, this is written $[x:=s]t$ and pronounced “substitute **s** for **x** in **t**.”

Here are some examples:

x:=true (if x then true else false) yields if true then true else false

x:=true x yields true

x:=true (if x then x else y) yields if true then true else y

x:=true y yields y

x:=true false yields false (vacuous substitution)

x:=true (\y:Bool, if y then x else false) yields \y:Bool, if y then true else false

x:=true (\y:Bool, x) yields \y:Bool, true

x:=true (\y:Bool, y) yields \y:Bool, y

x:=true (\x:Bool, x) yields \x:Bool, x

The last example is key: substituting x with `true` in $\backslash x:\text{Bool}$, x does *not* yield $\backslash x:\text{Bool}$, `true`! The reason for this is that the x in the body of $\backslash x:\text{Bool}$, x is *bound* by the abstraction: it is a new, local name that just happens to be spelled the same as some global name x .

Here is the definition, informally...

$x:=sx = s$ $x:=sy = y$ if $x \neq y$ $x:=s(\backslash x:T, t) = \backslash x:T, t$ $x:=s(\backslash y:T, t) = \backslash y:T, x:=st$ if $x \neq y$ $x:=s(t1\ t2) = (x:=st1)\ (x:=st2)$ $x:=strue = true$ $x:=sfalse = false$ $x:=s(\text{if } t1 \text{ then } t2 \text{ else } t3) = \text{if } x:=st1 \text{ then } x:=st2 \text{ else } x:=st3$

... and formally:

Reserved Notation "'[' x ' := ' s ']' t" (in custom stlc_tm at level 5, x global, s custom stlc_tm,

t custom stlc_tm at next level, right associativity).

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=

```
match t with
| tm_var y =>
  if String.eqb x y then s else t
| <{\y:T, t1}> =>
  if String.eqb x y then t else <{\y:T, [x:=s] t1}>
| <{t1 t2}> =>
  <{[x:=s] t1 [x:=s] t2}>
| <{true}> =>
  <{true}>
| <{false}> =>
  <{false}>
| <{if t1 then t2 else t3}> =>
  <{if [x:=s] t1 then [x:=s] t2 else [x:=s] t3}>
end
```

where "'[' x ' := ' s ']' t" := (subst x s t) (in custom stlc_tm).

Technical note: Substitution also becomes trickier to define if we consider the case where s , the term being substituted for a variable in some other term, may itself contain free variables.

Here is an example of how things would become trickier if one were to substitute *open* terms. Using the simple definition of substitution above to substitute the open term

$s = \backslash x:\text{Bool}, r$

(where r is a *free* reference to some global resource) for the free variable z in the term

$t = \backslash r:\text{Bool}, z$

where r is a bound variable, we would get

$\backslash r:\text{Bool}, \backslash x:\text{Bool}, r$

where the free reference to r in s has been “captured” by the binder at the beginning of t .

Why would this be bad? Because it violates the principle that the names of bound variables do not matter. For example, if we rename the bound variable in t , e.g., let

$t' = \backslash w:\text{Bool}, z$
 then $[z:=s]t'$ is
 $\backslash w:\text{Bool}, \backslash x:\text{Bool}, r$
 which does not behave the same as the substituting in the original t :
 $z:=st = \backslash r:\text{Bool}, \backslash x:\text{Bool}, r$

That is, renaming a bound variable in t would change how t behaves under our simple substitution. So substitution gets more complicated in that setting, but fortunately we don't have that problem in our STLC variant.

See, for example, *Aydemir* 2008 (in Bib.v) for further discussion of this issue.

Fortunately, since we are only interested here in defining the **step** relation on *closed* terms (i.e., terms like $\backslash x:\text{Bool}, x$ that include binders for all of the variables they mention), we can sidestep this extra complexity, but it must be dealt with when formalizing richer languages.

Exercise: 3 stars, standard (substi_correct) The definition that we gave above uses Rocq's **Fixpoint** facility to define substitution as a *function*. Suppose, instead, we wanted to define substitution as an inductive *relation* **substi**. We've begun the definition by providing the **Inductive** header and one of the constructors; your job is to fill in the rest of the constructors and prove that the relation you've defined coincides with the function given above.

```

Inductive substi (s : tm) (x : string) : tm → tm → Prop :=
| s_var1 :
    substi s x (tm_var x) s
.

Hint Constructors substi : core.

Theorem substi_correct : ∀ s x t t',
  <{ [x:=s] t }> = t' ↔ substi s x t t'.
Proof.
  Admitted.
□

```

10.4.3 Reduction

The small-step reduction relation for STLC now follows the same pattern as the ones we have seen before. Intuitively, to reduce a function application, we first reduce its left-hand side (the function) until it becomes an abstraction; then we reduce its right-hand side (the argument) until it is also a value; and finally we substitute the argument for the bound variable in the body of the abstraction. This last rule, written informally as

$(\backslash x:T, t12) \ v2 \rightarrow x:=v2 \ t12$
 is traditionally called *beta-reduction*.
 value v

(ST_AppAbs) ($\backslash x:T, t$) $v \rightarrow x:=vt$
 $t1 \rightarrow t1'$

(ST_App1) $t1\ t2 \rightarrow t1'\ t2$
 $\text{value } v1\ t2 \rightarrow t2'$

(ST_App2) $v1\ t2 \rightarrow v1\ t2'$... plus the usual rules for conditionals:

(ST_IfTrue) (if true then $t1$ else $t2$) $\rightarrow t1$

(ST_IfFalse) (if false then $t1$ else $t2$) $\rightarrow t2$
 $t1 \rightarrow t1'$

(ST_If) (if $t1$ then $t2$ else $t3$) $\rightarrow$ (if $t1'$ then $t2$ else $t3$)

This is *call by value* reduction: to reduce an application ($t1\ t2$), we

- first reduce $t1$ to a value: a function $\backslash x:T, t$
- then reduce the argument $t2$ to a value v
- then reduce the application itself by substituting v for the bound variable x in the body t .

Formally:

Reserved Notation " $t \rightarrow t$ " (at level 40).

Inductive **step** : **tm** $\rightarrow$ **tm** $\rightarrow$ Prop :=

```
| ST_AppAbs :  $\forall x\ T\ t\ v$ ,
  value  $v \rightarrow$ 
   $\langle \{ (\backslash x:T, t)\ v \} \rangle \rightarrow \langle \{ [x:=v] t \} \rangle$ 
| ST_App1 :  $\forall t1\ t1'\ t2$ ,
   $t1 \rightarrow t1' \rightarrow$ 
   $\langle \{ t1\ t2 \} \rangle \rightarrow \langle \{ t1'\ t2 \} \rangle$ 
| ST_App2 :  $\forall v1\ t2\ t2'$ ,
  value  $v1 \rightarrow$ 
   $t2 \rightarrow t2' \rightarrow$ 
   $\langle \{ v1\ t2 \} \rangle \rightarrow \langle \{ v1\ t2' \} \rangle$ 
| ST_IfTrue :  $\forall t1\ t2$ ,
   $\langle \{ \text{if true then } t1 \text{ else } t2 \} \rangle \rightarrow t1$ 
| ST_IfFalse :  $\forall t1\ t2$ ,
   $\langle \{ \text{if false then } t1 \text{ else } t2 \} \rangle \rightarrow t2$ 
| ST_If :  $\forall t1\ t1'\ t2\ t3$ ,
   $t1 \rightarrow t1' \rightarrow$ 
   $\langle \{ \text{if } t1 \text{ then } t2 \text{ else } t3 \} \rangle \rightarrow \langle \{ \text{if } t1' \text{ then } t2 \text{ else } t3 \} \rangle$ 
```

```

where "t '→' t'" := (step t t').
Hint Constructors step : core.
Notation multistep := (multi step).
Notation "t1 '→*' t2" := (multistep t1 t2) (at level 40).

```

10.4.4 Examples

Example:

```

(\x:Bool->Bool, x) (\x:Bool, x) ->* \x:Bool, x
i.e.,
idBB idB ->* idB

```

Lemma step_example1 :
 <{idBB idB}> ->* idB.

Proof.

```

eapply multi_step.
- apply ST_AppAbs.
  apply v_abs.
- simpl.
  apply multi_refl. Qed.

```

Example:

```

(\x:Bool->Bool, x) ((\x:Bool->Bool, x) (\x:Bool, x)) ->* \x:Bool, x
i.e.,
(idBB (idBB idB)) ->* idB.

```

Lemma step_example2 :
 <{idBB (idBB idB)}> ->* idB.

Proof.

```

eapply multi_step.
- apply ST_App2.
  + auto.
  + apply ST_AppAbs. auto.
- eapply multi_step.
  + apply ST_AppAbs. simpl. auto.
  + simpl. apply multi_refl. Qed.

```

Example:

```

(\x:Bool->Bool, x) (\x:Bool, if x then false else true) true ->* false
i.e.,
(idBB notB) true ->* false.

```

Lemma step_example3 :
 <{idBB notB true}> ->* <{false}>.

Proof.

```

eapply multi_step.
- apply ST_App1. apply ST_AppAbs. auto.
- simpl. eapply multi_step.
  + apply ST_AppAbs. auto.
  + simpl. eapply multi_step.
    × apply ST_IfTrue.
    × apply multi_refl. Qed.

```

Example:

$(\lambda x:\text{Bool} \rightarrow \text{Bool}, x) ((\lambda x:\text{Bool}, \text{if } x \text{ then false else true}) \text{ true}) \rightarrow^* \text{false}$

i.e.,

$\text{idBB} (\text{notB true}) \rightarrow^* \text{false}.$

(Note that this term doesn't actually typecheck; even so, we can ask how it reduces.)

Lemma step_example4 :

$\langle \{\text{idBB} (\text{notB true})\} \rangle \rightarrow^* \langle \{\text{false}\} \rangle.$

Proof.

```

eapply multi_step.
- apply ST_App2; auto.
- eapply multi_step.
  + apply ST_App2; auto.
  + apply ST_IfTrue.
  + eapply multi_step.
    × apply ST_AppAbs. auto.
    × simpl. apply multi_refl. Qed.

```

We can use the *normalize* tactic defined in the *Smallstep* chapter to simplify these proofs.

Lemma step_example1' :

$\langle \{\text{idBB idB}\} \rangle \rightarrow^* \text{idB}.$

Proof. *normalize*. Qed.

Lemma step_example2' :

$\langle \{\text{idBB} (\text{idBB idB})\} \rangle \rightarrow^* \text{idB}.$

Proof. *normalize*. Qed.

Lemma step_example3' :

$\langle \{\text{idBB notB true}\} \rangle \rightarrow^* \langle \{\text{false}\} \rangle.$

Proof. *normalize*. Qed.

Lemma step_example4' :

$\langle \{\text{idBB} (\text{notB true})\} \rangle \rightarrow^* \langle \{\text{false}\} \rangle.$

Proof. *normalize*. Qed.

Exercise: 2 stars, standard (step_example5) Try to do this one both with and without *normalize*.

Lemma step_example5 :

```

      <{idBBBB idBB idB}>
    ->* idB.
Proof.
  Admitted.

Lemma step_example5_with_normalize :
  <{idBBBB idBB idB}>
  ->* idB.
Proof.
  Admitted.
□

```

10.5 Typing

Next we consider the typing relation of the STLC, which is meant to prevent reduction from getting stuck.

For instance, the following two STLC terms are both stuck `if \x:Bool,x then true else false` (where we branch on a function as a boolean) and `true false` (where we apply a boolean as a function).

10.5.1 Contexts

Although we are primarily interested in the binary relation $\vdash t \text{ in } T$, relating a closed term t to its type T , we need to generalize a bit to make the definitions work.

Consider checking that $\backslash x:T11, t12$ has type $T11 \rightarrow T12$. Intuitively, we need to check that $t12$ has type $T12$. However, we have removed the binder $\backslash x$, so x may occur free in $t12$ (that is, $t12$ may be *open*). While checking that $t12$ has type $T12$, we must remember that x has type $T11$, in order to deal with these free occurrences of x . Similarly, $t12$ itself could contain abstractions, and typechecking their bodies could require looking up the declared types of yet more free variables.

To keep track of all this, we add a third element to the relation, a *typing context* Γ , which records the types of the variables that may occur free in a term – that is, Γ is a partial map from variables to types.

The new *typing judgment* is written $\Gamma \vdash t \text{ in } T$ and informally read as “term t has type T , given the types of free variables in t as specified by Γ ”.

We’ll also write $x \mapsto T ; \Gamma$ for “update the partial map Γ so that it maps x to T ,” following the notation from the [Maps](#) chapter.

With these refinements, we are ready to give informal and formal specifications of the typing relation.

Definition `context := partial_map ty`.

10.5.2 Typing Relation

Gamma $x = T1$

(T_Var) Gamma $\vdash x \backslash \text{in } T1$
 $x \mapsto T2$; Gamma $\vdash t1 \backslash \text{in } T1$

(T_Abs) Gamma $\vdash \backslash x:T2, t1 \backslash \text{in } T2 \rightarrow T1$
 Gamma $\vdash t1 \backslash \text{in } T2 \rightarrow T1$ Gamma $\vdash t2 \backslash \text{in } T2$

(T_App) Gamma $\vdash t1 t2 \backslash \text{in } T1$

(T_True) Gamma $\vdash \text{true} \backslash \text{in Bool}$

(T_False) Gamma $\vdash \text{false} \backslash \text{in Bool}$
 Gamma $\vdash t1 \backslash \text{in Bool}$ Gamma $\vdash t2 \backslash \text{in } T$ Gamma $\vdash t3 \backslash \text{in } T$

(T_If) Gamma $\vdash \text{if } t1 \text{ then } t2 \text{ else } t3 \backslash \text{in } T$

We can read the three-place relation $\text{Gamma} \vdash t \backslash \text{in } T$ as: “under the assumptions in Gamma, the term t has the type T .”

In the formal development, we write this judgment in $\langle \{ .. \} \rangle$ brackets, as introduced by the following notational conventions.

Notation " $x \mapsto v$; m " := (update m x v)

(in custom stlc_tm at level 0, x constr at level 0, v custom stlc_ty, right associativity) : stlc_scope.

Notation " $x \mapsto v$ " := (update empty x v)

(in custom stlc_tm at level 0, x constr at level 0, v custom stlc_ty) : stlc_scope.

Notation "'empty'" := empty (in custom stlc_tm) : stlc_scope.

Reserved Notation "<{ Gamma $\vdash$ $t \backslash \text{in } T$ }>"

(at level 0, Gamma custom stlc_tm at level 200, t custom stlc_tm, T custom stlc_ty).

Inductive **has_type** : context $\rightarrow$ tm $\rightarrow$ ty $\rightarrow$ Prop :=

| T_Var : $\forall$ Gamma x $T1$,
 Gamma $x = \text{Some } T1 \rightarrow$
 <{ Gamma $\vdash x \backslash \text{in } T1$ }>
 | T_Abs : $\forall$ Gamma x $T1$ $T2$ $t1$,
 <{ $x \mapsto T2$; Gamma $\vdash t1 \backslash \text{in } T1$ }> $\rightarrow$
 <{ Gamma $\vdash \backslash x:T2, t1 \backslash \text{in } T2 \rightarrow T1$ }>
 | T_App : $\forall$ $T1$ $T2$ Gamma $t1$ $t2$,
 <{ Gamma $\vdash t1 \backslash \text{in } T2 \rightarrow T1$ }> $\rightarrow$
 <{ Gamma $\vdash t2 \backslash \text{in } T2$ }> $\rightarrow$


```

    <{ Gamma |- t1 t2 \in T1 }>
| T_True : ∀ Gamma,
    <{ Gamma |- true \in Bool }>
| T_False : ∀ Gamma,
    <{ Gamma |- false \in Bool }>
| T_If : ∀ t1 t2 t3 T1 Gamma,
    <{ Gamma |- t1 \in Bool }> →
    <{ Gamma |- t2 \in T1 }> →
    <{ Gamma |- t3 \in T1 }> →
    <{ Gamma |- if t1 then t2 else t3 \in T1 }>

```

where "`<{ Gamma |- ' t '\in' T }>`" := (`has_type` Gamma t T) : `stlc_scope`.
Hint Constructors `has_type` : `core`.

10.5.3 Examples

Example `typing_example_1` :
`<{ empty |- \x:Bool, x \in Bool → Bool }>`.
Proof. `eauto`. Qed.

Note that, since we added the `has_type` constructors to the hints database, `eauto` can solve this one immediately.

For what it's worth, in this case `auto` works too, since our term contains no application nodes: `Example typing_example_1'` :
`<{ empty |- \x:Bool, x \in Bool → Bool }>`.
Proof. `auto`. Qed.

More examples:

`empty |- \x:Bool, \y:Bool->Bool, y (y x) \in Bool -> (Bool->Bool) -> Bool`.

Example `typing_example_2` :
`<{ empty |-`
`\x:Bool,`
`\y:Bool→Bool,`
`(y (y x)) \in`
`Bool → (Bool → Bool) → Bool }>`.
Proof. `eauto` 20. Qed.

Exercise: 2 stars, standard, optional (`typing_example_2_full`) Prove the same result without using `auto`, `eauto`, or `eapply` (or ...).

Example `typing_example_2_full` :
`<{ empty |-`
`\x:Bool,`
`\y:Bool→Bool,`

```

      (y (y x)) \in
    Bool → (Bool → Bool) → Bool }>.
Proof.
  Admitted.
□

```

Exercise: 2 stars, standard (typing_example_3) Formally prove the following typing derivation holds:

exists T, empty |- \x:Bool->Bool, \y:Bool->Bool, \z:Bool, y (x z) \in T.

Example typing_example_3 :

```

  ∃ T,
  <{ empty |-
    \x:Bool→Bool,
    \y:Bool→Bool,
    \z:Bool,
    (y (x z)) \in
    T }>.

```

```

Proof.
  Admitted.
□

```

We can also show that some terms are *not* typable. For example, we can check that there is no typing derivation assigning a type to the term \x:Bool, \y:Bool, x y – i.e.,

~ exists T, empty |- \x:Bool, \y:Bool, x y \in T.

Example typing_nonexample_1 :

```

  ¬ ∃ T,
  <{ empty |-
    \x:Bool,
    \y:Bool,
    (x y) \in
    T }>.

```

```

Proof.
  intros Hc. destruct Hc as [T Hc].
  inversion Hc; subst; clear Hc.
  inversion H4; subst; clear H4.
  inversion H5; subst; clear H5 H4.
  inversion H2; subst; clear H2.
  discriminate H1.
Qed.

```

Exercise: 3 stars, standard, optional (typing_nonexample_3) Another nonexample:

~ (exists S T, empty |- \x:S, x x \in T).

```

Example typing_nonexample_3 :
  ¬ (∃ S T,
    <{ empty |-
      \x:S, x x \in T }>).

```

Proof.

Admitted.

□

End STLC.

Chapter 11

Library `PLF.StlcProp`

11.1 `StlcProp`: Properties of STLC

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From PLF Require Import Maps.
From PLF Require Import Types.
From PLF Require Import Stlc.
From PLF Require Import Smallstep.
Set Default Goal Selector "!".
Module STLCProp.
Import STLC.
```

In this chapter, we develop the fundamental theory of the Simply Typed Lambda Calculus – in particular, the type safety theorem.

11.2 Canonical Forms

As we saw for the very simple language in the `Types` chapter, the first step in establishing basic properties of reduction and types is to identify the possible *canonical forms* (i.e., well-typed values) belonging to each type. For `Bool`, these are again the boolean values `true` and `false`; for arrow types, they are lambda-abstractions.

Formally, we will need these lemmas only for terms that are not only well typed but *closed* – i.e., well typed in the empty context.

```
Lemma canonical_forms_bool : ∀ t,
  <{ empty |- t \in Bool }> →
  value t →
  (t = <{true}>) ∨ (t = <{false}>).
```

Proof.

```
  intros t HT HVal.
  destruct HVal; auto.
```

```

    inversion HT.
Qed.

Lemma canonical_forms_fun :  $\forall t \ T1 \ T2,$ 
  <{ empty |-  $t \ \text{in } T1 \rightarrow T2$  }>  $\rightarrow$ 
  value  $t \rightarrow$ 
   $\exists x \ u, \ t = \langle \{x:T1, \ u\} \rangle$ .
Proof.
  intros t T1 T2 HT HVal.
  destruct HVal as [x ? t1 | |] ; inversion HT; subst.
   $\exists x, t1$ . reflexivity.
Qed.

```

11.3 Progress

The *progress* theorem tells us that closed, well-typed terms are not stuck: either a well-typed term is a value, or it can take a reduction step. The proof is a relatively straightforward extension of the progress proof we saw in the `Types` chapter. We give the proof in English first, then the formal version.

```

Theorem progress :  $\forall t \ T,$ 
  <{ empty |-  $t \ \text{in } T$  }>  $\rightarrow$ 
  value  $t \vee \exists t', t \rightarrow t'$ .

```

Proof: By induction on the derivation of $\text{|- } t \ \text{in } T$.

- The last rule of the derivation cannot be `T_Var`, since a variable is never well typed in an empty context.
- The `T_True`, `T_False`, and `T_Abs` cases are trivial, since in each of these cases we can see by inspecting the rule that t is a value.
- If the last rule of the derivation is `T_App`, then t has the form $t1 \ t2$ for some $t1$ and $t2$, where $\text{|- } t1 \ \text{in } T2 \rightarrow T$ and $\text{|- } t2 \ \text{in } T2$ for some type $T2$. The induction hypothesis for the first subderivation says that either $t1$ is a value or else it can take a reduction step.
 - If $t1$ is a value, then consider $t2$, which by the induction hypothesis for the second subderivation must also either be a value or take a step.
 - * Suppose $t2$ is a value. Since $t1$ is a value with an arrow type, it must be a lambda abstraction; hence $t1 \ t2$ can take a step by `ST_AppAbs`.
 - * Otherwise, $t2$ can take a step, and hence so can $t1 \ t2$ by `ST_App2`.
 - If $t1$ can take a step, then so can $t1 \ t2$ by `ST_App1`.

- If the last rule of the derivation is T_lf , then $t = \text{if } t1 \text{ then } t2 \text{ else } t3$, where $t1$ has type $Bool$. The first IH says that $t1$ either is a value or takes a step.
 - If $t1$ is a value, then since it has type $Bool$ it must be either $true$ or $false$. If it is $true$, then t steps to $t2$; otherwise it steps to $t3$.
 - Otherwise, $t1$ takes a step, and therefore so does t (by ST_lf).

Proof with eauto.

```

intros t T Ht.
remember empty as Gamma.
induction Ht; subst Gamma; auto.
-
  discriminate H.
-
  right. destruct IHht1...
  +
    destruct IHht2...
    ×
      eapply canonical_forms_fun in Ht1; [|assumption].
      destruct Ht1 as [x [t0 H1]]. subst.
      ∃ (<{ [x:=t2]t0 }>)...
    ×
      destruct H0 as [t2' Hstp]. ∃ (<{t1 t2'}>)...
  +
    destruct H as [t1' Hstp]. ∃ (<{t1' t2}>)...
-
  right. destruct IHht1...
  +
    destruct (canonical_forms_bool t1); subst; eauto.
  +
    destruct H as [t1' Hstp]. ∃ <{if t1' then t2 else t3}>...
```

Qed.

Exercise: 3 stars, advanced (progress_from_term_ind) Show that progress can also be proved by induction on terms instead of induction on typing derivations.

Theorem `progress'` : $\forall t\ T,$
 $\langle\{ \text{empty} \mid -\ t \ \backslash\text{in } T \}\rangle \rightarrow$
 $\text{value } t \vee \exists t',\ t \rightarrow t'.$

Proof.

```

intros t.
induction t; intros T Ht; auto.
Admitted.
□

```

11.4 Preservation

The other half of the type soundness property is the preservation of types during reduction. For this part, we'll need to develop some technical machinery for reasoning about variables and substitution. Working from top to bottom (from the high-level property we are actually interested in to the lowest-level technical lemmas that are needed by various cases of the more interesting proofs), the story goes like this:

- The *preservation theorem* is proved by induction on a typing derivation and case analysis on the step relation, pretty much as we did in the `Types` chapter. The one case that is significantly different is the one for the `ST_AppAbs` rule, whose definition uses the substitution operation. To see that this step preserves typing, we need to know that the substitution itself does. So we prove a...
- *substitution lemma*, stating that substituting a (closed, well-typed) term `s` for a variable `x` in a term `t` preserves the type of `t`. The proof goes by induction on the form of `t` and requires looking at all the different cases in the definition of substitution. This time, for the variables case, we discover that we need to deduce from the fact that a term `s` has type `S` in the empty context the fact that `s` has type `S` in every context. For this we prove a...
- *weakening lemma*, showing that typing is preserved under “extensions” to the context *Gamma*.

To make Rocq happy, though, we need to formalize the story in the opposite order, starting with weakening...

11.4.1 The Weakening Lemma

First, we show that typing is preserved under “extensions” to the context *Gamma*. (Recall the definition of “includedin” from Maps.v.)

```

Lemma weakening : ∀ Gamma Gamma' t T,
  includedin Gamma Gamma' →
  <{ Gamma |- t \in T }> →
  <{ Gamma' |- t \in T }>.

```

Proof.

```

intros Gamma Gamma' t T H Ht.
generalize dependent Gamma'.

```

```

induction Ht; eauto using includedin_update.
Qed.

```

The following simple corollary is what we actually need below.

```

Lemma weakening_empty :  $\forall \text{Gamma } t \text{ T},$ 
   $\langle \{ \text{empty} \mid - t \text{ in T} \} \rangle \rightarrow$ 
   $\langle \{ \text{Gamma} \mid - t \text{ in T} \} \rangle.$ 

```

Proof.

```

intros Gamma t T.
eapply weakening.
discriminate.
Qed.

```

11.4.2 The Substitution Lemma

Now we come to the conceptual heart of the proof that reduction preserves types – namely, the observation that *substitution* preserves types.

Formally, the so-called *substitution lemma* says this: Suppose we have a term t with a free variable x , and suppose we’ve assigned a type T to t under the assumption that x has some type U . Also, suppose that we have some other term v and that we’ve shown that v has type U . Then, since v satisfies the assumption we made about x when typing t , we can substitute v for each of the occurrences of x in t and obtain a new term that still has type T .

```

Lemma substitution_preserves_typing :  $\forall \text{Gamma } x \text{ U } t \text{ v } T,$ 
   $\langle \{ x \mapsto U ; \text{Gamma} \mid - t \text{ in T} \} \rangle \rightarrow$ 
   $\langle \{ \text{empty} \mid - v \text{ in U} \} \rangle \rightarrow$ 
   $\langle \{ \text{Gamma} \mid - [x:=v]t \text{ in T} \} \rangle.$ 

```

The substitution lemma can be viewed as a kind of “commutation property.” Intuitively, it says that substitution and typing can be done in either order: we can either assign types to the terms t and v separately (under suitable contexts) and then combine them using substitution, or we can substitute first and then assign a type to $[x:=v]t$; the result is the same either way.

Proof: We show, by induction on t , that for all T and Gamma , if $x \mapsto U$; $\text{Gamma} \mid - t \text{ in T}$ and $- v \text{ in U}$, then $\text{Gamma} \mid - [x:=v]t \text{ in T}$.

- If t is a variable there are two cases to consider, depending on whether t is x or some other variable.
 - If $t = x$, then from the fact that $x \mapsto U$; $\text{Gamma} \mid - x \text{ in T}$ we conclude that $U = T$. We must show that $[x:=v]x = v$ has type T under Gamma , given the assumption that v has type $U = T$ under the empty context. This follows from the weakening lemma.

- If t is some variable y that is not equal to x , then we need only note that y has the same type under $x|->U; \textit{Gamma}$ as under $\textit{Gamma}$.
- If t is an abstraction $\backslash y:S, t0$, then $T = S \rightarrow T1$ and the IH tells us, for all $\textit{Gamma}'$ and $T0$, that if $x|->U; \textit{Gamma}' \vdash t0 \text{ in } T0$, then $\textit{Gamma}' \vdash [x:=v]t0 \text{ in } T0$. Moreover, by inspecting the typing rules we see it must be the case that $y|->S; x|->U; \textit{Gamma} \vdash t0 \text{ in } T1$.

The substitution in the conclusion behaves differently depending on whether x and y are the same variable.

First, suppose $x = y$. Then, by the definition of substitution, $[x:=v]t = t$, so we just need to show $\textit{Gamma} \vdash t \text{ in } T$. Using T_Abs , we need to show that $y|->S; \textit{Gamma} \vdash t0 \text{ in } T1$. But we know $y|->S; x|->U; \textit{Gamma} \vdash t0 \text{ in } T1$, and the claim follows since $x = y$.

Second, suppose $x \neq y$. Again, using T_Abs , we need to show that $y|->S; \textit{Gamma} \vdash [x:=v]t0 \text{ in } T1$. Since $x \neq y$, we have $y|->S; x|->U; \textit{Gamma} = x|->U; y|->S; \textit{Gamma}$. So we have $x|->U; y|->S; \textit{Gamma} \vdash t0 \text{ in } T1$. Then, the IH applies (taking $\textit{Gamma}' = y|->S; \textit{Gamma}$), giving us $y|->S; \textit{Gamma} \vdash [x:=v]t0 \text{ in } T1$, as required.

- If t is an application $t1\ t2$, the result follows straightforwardly from the definition of substitution and the induction hypotheses.
- The remaining cases are similar to the application case.

Proof.

```

intros Gamma x U t v T Ht Hv.
generalize dependent Gamma. generalize dependent T.
induction t; intros T Gamma H;

  inversion H; clear H; subst; simpl; eauto.
-
  rename s into y. destruct (eqb_spec x y); subst.
  +
    rewrite update_eq in H2.
    injection H2 as H2; subst.
    apply weakening_empty. assumption.
  +
    apply T_Var. rewrite update_neq in H2; auto.
-
  rename s into y, t into S.
  destruct (eqb_spec x y); subst; apply T_Abs.
  +

```

```

      rewrite update_shadow in H5. assumption.
+
      apply IHt.
      rewrite update_permute; auto.
Qed.

```

One technical subtlety in the statement of the above lemma is that we assume v has type U in the *empty* context – in other words, we assume v is closed. (Since we are using a simple definition of substitution that is not capture-avoiding, it doesn't make sense to substitute non-closed terms into other terms. Fortunately, closed terms are all we need!)

Exercise: 3 stars, advanced (substitution_preserves_typing_from_typing_ind)

Show that `substitution_preserves_typing` can also be proved by induction on typing derivations instead of induction on terms. `Lemma substitution_preserves_typing_from_typing_ind`

```

: ∀ Gamma x U t v T,
  <{ x |-> U ; Gamma |- t \in T }> →
  <{ empty |- v \in U }> →
  <{ Gamma |- [x:=v]t \in T }>.

```

Proof.

```

intros Gamma x U t v T Ht Hv.
remember (x |-> U; Gamma) as Gamma'.
generalize dependent Gamma.
induction Ht; intros Gamma' G; simpl; eauto.
Admitted.
□

```

11.4.3 Main Theorem

We now have the ingredients we need to prove preservation: if a closed, well-typed term t has type T and takes a step to t' , then t' is also a closed term with type T . In other words, the small-step reduction relation preserves types.

`Theorem preservation : ∀ t t' T,`

```

  <{ empty |- t \in T }> →
  t -> t' →
  <{ empty |- t' \in T }>.

```

Proof: By induction on the derivation of $\vdash t \in T$.

- We can immediately rule out `T_Var`, `T_Abs`, `T_True`, and `T_False` as final rules in the derivation, since in each of these cases t cannot take a step.
- If the last rule in the derivation is `T_App`, then $t = t1\ t2$, and there are subderivations showing that $\vdash t1 \in T2 \rightarrow T$ and $\vdash t2 \in T2$ plus two induction hypotheses: (1) $t1 \rightarrow t1'$ implies $\vdash t1' \in T2 \rightarrow T$ and (2) $t2 \rightarrow t2'$ implies $\vdash t2' \in T2$. There

are now three subcases to consider, one for each rule that could be used to show that $t1\ t2$ takes a step to t' .

- If $t1\ t2$ takes a step by `ST_App1`, with $t1$ stepping to $t1'$, then, by the first IH, $t1'$ has the same type as $t1$ ($\vdash t1' \text{ in } T2 \rightarrow T$), and hence by `T_App` $t1'\ t2$ has type T .
- The `ST_App2` case is similar, using the second IH.
- If $t1\ t2$ takes a step by `ST_AppAbs`, then $t1 = \backslash x:T0, t0$ and $t1\ t2$ steps to $[x0:=t2]t0$; the desired result now follows from the substitution lemma.
- If the last rule in the derivation is `T_If`, then $t = \text{if } t1 \text{ then } t2 \text{ else } t3$, with $\vdash t1 \text{ in Bool}$, $\vdash t2 \text{ in } T1$, and $\vdash t3 \text{ in } T1$, and with three induction hypotheses: (1) $t1 \rightarrow t1'$ implies $\vdash t1' \text{ in Bool}$, (2) $t2 \rightarrow t2'$ implies $\vdash t2' \text{ in } T1$, and (3) $t3 \rightarrow t3'$ implies $\vdash t3' \text{ in } T1$.

There are again three subcases to consider, depending on how t steps.

- If t steps to $t2$ or $t3$ by `ST_IfTrue` or `ST_IfFalse`, the result is immediate, since $t2$ and $t3$ have the same type as t .
- Otherwise, t steps by `ST_If`, and the desired conclusion follows directly from the first induction hypothesis.

Proof with `eauto`.

```
intros t t' T HT. generalize dependent t'.
remember empty as Gamma.
induction HT;
  intros t' HE; subst;
  try solve [inversion HE; subst; auto].
-
  inversion HE; subst...
+
  apply substitution_preserves_typing with T2...
  inversion HT1...
```

Qed.

Exercise: 2 stars, standard, especially useful (subject_expansion_stlc) An exercise in the `Types` chapter asked about the *subject expansion* property for the simple language of arithmetic and boolean expressions. This property did not hold for that language, and it also fails for STLC. That is, it is not always the case that, if $t \rightarrow t'$ and $\text{empty} \vdash t' \text{ in } T$, then $\text{empty} \vdash t \text{ in } T$. Show this by giving a counter-example that does *not involve conditionals*.

Theorem `not_subject_expansion`:

```

  ∃ t t' T, t -> t' ∧ <{ empty |- t' \in T }> ∧ ¬ <{ empty |- t \in T }>.
Proof.
  Admitted.
Definition manual_grade_for_subject_expansion_stlc : option (nat×string) := None.
□

```

11.5 Type Soundness

Exercise: 2 stars, standard, optional (type_soundness) Put progress and preservation together and show that a well-typed term can *never* reach a stuck state.

```

Definition stuck (t:tm) : Prop :=
  (normal_form step) t ∧ ¬ value t.
Corollary type_soundness : ∀ t t' T,
  <{ empty |- t \in T }> →
  t ->* t' →
  ~(stuck t').

```

```

Proof.
  intros t t' T Hhas_type Hmulti. unfold stuck.
  intros [Hnf Hnot_val]. unfold normal_form in Hnf.
  induction Hmulti.
  Admitted.
□

```

11.6 Uniqueness of Types

Exercise: 3 stars, standard (unique_types) Another nice property of the STLC is that types are unique: a given term (in a given context) has at most one type.

```

Theorem unique_types : ∀ Gamma e T T',
  <{ Gamma |- e \in T }> →
  <{ Gamma |- e \in T' }> →
  T = T'.
Proof.
  Admitted.
□

```

11.7 Context Invariance (Optional)

Another standard technical lemma associated with typed languages is *context invariance*. It states that typing is preserved under “inessential changes” to the context *Gamma* – in particular, changes that do not affect any of the free variables of the term. In this section,

we establish this property for our system, introducing some other standard terminology on the way.

First, we need to define the *free variables* in a term – i.e., variables that are used in the term in positions that are *not* in the scope of an enclosing function abstraction binding a variable of the same name.

More technically, a variable x *appears free in* a term t if t contains some occurrence of x that is not under an abstraction labeled x . For example:

- y appears free, but x does not, in $\lambda x:T \rightarrow U, x y$
- both x and y appear free in $(\lambda x:T \rightarrow U, x y) x$
- no variables appear free in $\lambda x:T \rightarrow U, \lambda y:T, x y$

Formally:

```
Inductive appears_free_in (x : string) : tm → Prop :=
| afi_var : appears_free_in x <{x}>
| afi_app1 : ∀ t1 t2,
  appears_free_in x t1 →
  appears_free_in x <{t1 t2}>
| afi_app2 : ∀ t1 t2,
  appears_free_in x t2 →
  appears_free_in x <{t1 t2}>
| afi_abs : ∀ y T1 t1,
  y ≠ x →
  appears_free_in x t1 →
  appears_free_in x <{\y:T1, t1}>
| afi_if1 : ∀ t1 t2 t3,
  appears_free_in x t1 →
  appears_free_in x <{if t1 then t2 else t3}>
| afi_if2 : ∀ t1 t2 t3,
  appears_free_in x t2 →
  appears_free_in x <{if t1 then t2 else t3}>
| afi_if3 : ∀ t1 t2 t3,
  appears_free_in x t3 →
  appears_free_in x <{if t1 then t2 else t3}>.
```

Hint Constructors `appears_free_in` : core.

The *free variables* of a term are just the variables that appear free in it. This gives us another way to define *closed* terms – arguably a better one, since it applies even to ill-typed terms. Indeed, this is the standard definition of the term “closed.”

```
Definition closed (t:tm) :=
  ∀ x, ¬ appears_free_in x t.
```

Conversely, an *open* term is one that may contain free variables. (I.e., every term is an open term; the closed terms are a subset of the open ones. “Open” precisely means “possibly containing free variables.”)

Exercise: 1 star, standard, optional (afi) (Officially optional, but strongly recommended!) In the space below, write out the rules of the `appears_free_in` relation in informal inference-rule notation. (Use whatever notational conventions you like – the point of the exercise is just for you to think a bit about the meaning of each rule.) Although this is a rather low-level, technical definition, understanding it is crucial to understanding substitution and its properties, which are really the crux of the lambda-calculus.

Definition `manual_grade_for_afi` : `option (nat × string)` := `None`.

□

Next, we show that if a variable `x` appears free in a term `t`, and if we know `t` is well typed in context `Gamma`, then it must be the case that `Gamma` assigns a type to `x`.

Lemma `free_in_context` : $\forall x\ t\ T\ \textit{Gamma},$
`appears_free_in x t` $\rightarrow$
 $\langle \{ \textit{Gamma} \mid -\ t\ \text{in}\ T \} \rangle \rightarrow$
 $\exists T',\ \textit{Gamma}\ x = \text{Some}\ T'.$

Proof: We show, by induction on the proof that `x` appears free in `t`, that, for all contexts `Gamma`, if `t` is well typed under `Gamma`, then `Gamma` assigns some type to `x`.

- If the last rule used is `afi_var`, then `t` = `x`, and from the assumption that `t` is well typed under `Gamma` we have immediately that `Gamma` assigns a type to `x`.
- If the last rule used is `afi_app1`, then `t` = `t1 t2` and `x` appears free in `t1`. Since `t` is well typed under `Gamma`, we can see from the typing rules that `t1` must also be, and the IH then tells us that `Gamma` assigns `x` a type.
- Almost all the other cases are similar: `x` appears free in a subterm of `t`, and since `t` is well typed under `Gamma`, we know the subterm of `t` in which `x` appears is well typed under `Gamma` as well, and the IH gives us exactly the conclusion we want.
- The only remaining case is `afi_abs`. In this case `t` = `\y:T1,t1` and `x` appears free in `t1`, and we also know that `x` is different from `y`. The difference from the previous cases is that, whereas `t` is well typed under `Gamma`, its body `t1` is well typed under `y|->T1; Gamma`, so the IH allows us to conclude that `x` is assigned some type by the extended context `y|->T1; Gamma`. To conclude that `Gamma` assigns a type to `x`, we appeal to lemma `update_neq`, noting that `x` and `y` are different variables.

Exercise: 2 stars, standard (free_in_context) Complete the following proof.

Proof.

`intros x t T Gamma H H0. generalize dependent Gamma.`

```

generalize dependent T.
induction H as [| | y T1 t1 H H0 IHappears_free_in| | ];
  intros; try solve [inversion H0; eauto].
Admitted.
□

```

From the `free_in_context` lemma, it immediately follows that any term t that is well typed in the empty context is closed (it has no free variables).

Exercise: 2 stars, standard, optional (typable_empty__closed) Corollary `typable_empty__closed`

```

: ∀ t T,
  <{ empty |- t \in T }> →
  closed t.

```

Proof.

```

Admitted.
□

```

Finally, we establish *context_invariance*. It is useful in cases when we have a proof of some typing relation $\Gamma \vdash t \text{ in } T$, and we need to replace Γ by a different context Γ' . When is it safe to do this? Intuitively, it must at least be the case that Γ' assigns the same types as Γ to all the variables that appear free in t . In fact, this is the only condition that is needed.

```

Lemma context_invariance : ∀ Γ Γ' t T,
  <{ Γ |- t \in T }> →
  (∀ x, appears_free_in x t → Γ x = Γ' x) →
  <{ Γ' |- t \in T }>.

```

Proof: By induction on the derivation of $\Gamma \vdash t \text{ in } T$.

- If the last rule in the derivation was `T_Var`, then $t = x$ and $\Gamma x = T$. By assumption, $\Gamma' x = T$ as well, and hence $\Gamma' \vdash t \text{ in } T$ by `T_Var`.
- If the last rule was `T_Abs`, then $t = \lambda y:T2. t1$, with $T = T2 \rightarrow T1$ and $y|->T2; \Gamma \vdash t1 \text{ in } T1$. The induction hypothesis states that for any context Γ'' , if $y|->T2; \Gamma$ and Γ'' assign the same types to all the free variables in $t1$, then $t1$ has type $T1$ under Γ'' . Let Γ' be a context which agrees with Γ on the free variables in t ; we must show $\Gamma' \vdash \lambda y:T2. t1 \text{ in } T2 \rightarrow T1$.

By `T_Abs`, it suffices to show that $y|->T2; \Gamma' \vdash t1 \text{ in } T1$. By the IH (setting $\Gamma'' = y|->T2; \Gamma'$), it suffices to show that $y|->T2; \Gamma$ and $y|->T2; \Gamma'$ agree on all the variables that appear free in $t1$.

Any variable occurring free in $t1$ must be either y or some other variable. $y|->T2; \Gamma$ and $y|->T2; \Gamma'$ clearly agree on y . Otherwise, note that any variable other than y that occurs free in $t1$ also occurs free in $t = \lambda y:T2. t1$, and by assumption Γ and Γ' agree on all such variables; hence so do $y|->T2; \Gamma$ and $y|->T2; \Gamma'$.

- If the last rule was `T_App`, then $t = t1\ t2$, with $\Gamma \vdash t1 \text{ in } T2 \rightarrow T$ and $\Gamma \vdash t2 \text{ in } T2$. One induction hypothesis states that for all contexts Γ' , if Γ' agrees with Γ on the free variables in $t1$, then $t1$ has type $T2 \rightarrow T$ under Γ' ; there is a similar IH for $t2$. We must show that $t1\ t2$ also has type T under Γ' , given the assumption that Γ' agrees with Γ on all the free variables in $t1\ t2$. By `T_App`, it suffices to show that $t1$ and $t2$ each have the same type under Γ' as under Γ . But all free variables in $t1$ are also free in $t1\ t2$, and similarly for $t2$; hence the desired result follows from the induction hypotheses.

Exercise: 3 stars, standard, optional (context_invariance) Complete the following proof. *Proof.*

```
intros.
generalize dependent Gamma'.
induction H as [| ? x0 ????? || |]; intros; auto.
Admitted.
```

□

The context invariance lemma can actually be used in place of the weakening lemma to prove the crucial substitution lemma stated earlier.

11.8 Additional Exercises

Exercise: 1 star, standard, optional (progress_preservation_statement) (Officially optional, but strongly recommended!) Without peeking at their statements above, write down the progress and preservation theorems for the simply typed lambda-calculus (as Rocq theorems). You can write *Admitted* for the proofs.

Definition `manual_grade_for_progress_preservation_statement` : `option (nat × string)` := `None`.

□

Exercise: 2 stars, standard (stlc_variation1) Suppose we add a new term *zap* with the following reduction rule

(ST_Zap) $t \rightarrow \text{zap}$
and the following typing rule:

(T_Zap) $\Gamma \vdash \text{zap} \text{ in } T$

Which of the following properties of the STLC remain true in the presence of these rules? For each property, write either “remains true” or “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**

- Progress
- Preservation

Definition manual_grade_for_stlc_variation1 : option (nat×string) := None.

□

Exercise: 2 stars, standard (stlc_variation2) Suppose instead that we add a new term `foo` with the following reduction rules:

(ST_Foo1) $(\lambda x:A, x) \rightarrow \text{foo}$

(ST_Foo2) $\text{foo} \rightarrow \text{true}$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

Definition manual_grade_for_stlc_variation2 : option (nat×string) := None.

□

Exercise: 2 stars, standard (stlc_variation3) Suppose instead that we remove the rule `ST_App1` from the **step** relation. Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

Definition manual_grade_for_stlc_variation3 : option (nat×string) := None.

□

Exercise: 2 stars, standard, optional (stlc_variation4) Suppose instead that we add the following new rule to the reduction relation:

(ST_FunnyIfTrue) (if true then t1 else t2) $\rightarrow$ true

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

Exercise: 2 stars, standard, optional (stlc_variation5) Suppose instead that we add the following new rule to the typing relation:

$\Gamma \vdash t1 \text{ \textit{in} Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \quad \Gamma \vdash t2 \text{ \textit{in} Bool}$

(T_FunnyApp) $\Gamma \vdash t1 \ t2 \text{ \textit{in} Bool}$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

Exercise: 2 stars, standard, optional (stlc_variation6) Suppose instead that we add the following new rule to the typing relation:

$\Gamma \vdash t1 \text{ \textit{in} Bool} \quad \Gamma \vdash t2 \text{ \textit{in} Bool}$

(T_FunnyApp') $\Gamma \vdash t1 \ t2 \text{ \textit{in} Bool}$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

Exercise: 2 stars, standard, optional (stlc_variation7) Suppose we add the following new rule to the typing relation of the STLC:

$(T\_FunnyAbs) \vdash \lambda x:Bool.t \text{ in } Bool$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

End STLCPROP.

11.8.1 Exercise: STLC with Arithmetic

To see how the STLC might function as the core of a real programming language, let’s extend it with a concrete base type of numbers and some constants and primitive operators.

Module STLCARITH.

Import STLC.

To types, we add a base type of natural numbers (and remove booleans, for brevity).

```
Inductive ty : Type :=
| Ty_Arrow : ty → ty → ty
| Ty_Nat : ty.
```

To terms, we add natural number constants, along with successor, predecessor, multiplication, and zero-testing.

```
Inductive tm : Type :=
| tm_var : string → tm
| tm_app : tm → tm → tm
| tm_abs : string → ty → tm → tm
| tm_const : nat → tm
| tm_succ : tm → tm
| tm_pred : tm → tm
| tm_mult : tm → tm → tm
| tm_if0 : tm → tm → tm → tm.
```

Notation "<{ x }>" := x (x custom stlc_ty).

Notation "(t)" := t (in custom stlc_ty at level 0, t custom stlc_ty) : stlc_scope.

```

Notation "S -> T" := (Ty_Arrow S T) (in custom stlc_ty at level 99, right associativity)
: stlc_scope.

Notation "$ ( t )" := t (in custom stlc_ty at level 0, t constr) : stlc_scope.

Notation "$ ( x )" := x (in custom stlc_tm at level 0, x constr, only parsing) :
stlc_scope.

Notation "x" := x (in custom stlc_tm at level 0, x constr at level 0) : stlc_scope.

Notation "<{ e }>" := e (e custom stlc_tm at level 200) : stlc_scope.

Notation "( x )" := x (in custom stlc_tm at level 0, x custom stlc_tm) : stlc_scope.

Notation "x y" := (tm_app x y) (in custom stlc_tm at level 10, left associativity)
: stlc_scope.

Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc_tm at level 200, x global,
                  t custom stlc_ty,
                  y custom stlc_tm at level 200,
                  left associativity).

Coercion tm_var : string >-> tm.
Arguments tm_var _%-string.

Notation "'Nat'" := Ty_Nat (in custom stlc_ty at level 0).

Notation "'succ' x" := (tm_succ x) (in custom stlc_tm at level 10,
                                   x custom stlc_tm at level 0) : stlc_scope.

Notation "'pred' x" := (tm_pred x) (in custom stlc_tm at level 10,
                                   x custom stlc_tm at level 0) : stlc_scope.

Notation "x * y" := (tm_mult x y) (in custom stlc_tm at level 95,
                                   right associativity) : stlc_scope.

Notation "'if0' x 'then' y 'else' z" :=
  (tm_if0 x y z) (in custom stlc_tm at level 0,
                  x custom stlc_tm at level 0,
                  y custom stlc_tm at level 0,
                  z custom stlc_tm at level 0) : stlc_scope.

Coercion tm_const : nat >-> tm.

```

In this extended exercise, your job is to finish formalizing the definition and properties of the STLC extended with arithmetic. Specifically:

Fill in the core definitions for STLCArith, by starting with the rules and terms which are the same as STLC. Then prove the key lemmas and theorems we provide. You will need to define and prove helper lemmas, as before.

It will be necessary to also fill in “Reserved Notation”, “Notation”, and “Hint Constructors”.

Hint: If you get an error “STLC.tm” found instead of term “tm” then Rocq is picking up the old notation for ie: subst instead of the new notation for STLCArith, so you need to overwrite the old with the notation before you can use it.

Make sure Rocq accepts the whole file before submitting.

Exercise: 5 stars, standard (STLCArith.subst) `Fixpoint subst (x : string) (s : tm) (t : tm) : tm`
`. Admitted.`

(You'll need to remove the period at the end of this definition and add where "`x := s`" `t`" := (subst x s t) (in custom stlc_tm). when you fill it in.)

Inductive `value` : `tm` → Prop :=

.

Hint Constructors `value` : core.

Reserved Notation "`t -> t`" (at level 40).

Inductive `step` : `tm` → `tm` → Prop :=

where "`t -> t`" := (`step t t`).

Notation `multistep` := (`multi step`).

Notation "`t1 -> * t2`" := (`multistep t1 t2`) (at level 40).

Hint Constructors `step` : core.

Example `Nat_step_example` : ∃ `t`,
`<{(\x: Nat, \y: Nat, x × y) $(3) $(2) }> -> * t`.

Proof. `Admitted.`

Definition `context` := `partial_map ty`.

Inductive `has_type` : `context` → `tm` → `ty` → Prop :=

where "`<{ Gamma |- t \in T }>`" := (`has_type Gamma t T`).

Hint Constructors `has_type` : core.

Example `Nat_typing_example` :

`<{ empty |- (\x: Nat, \y: Nat, x × y) $(3) $(2) \in Nat }>`.

Proof.

`Admitted.`

□

11.8.2 The Technical Theorems

The next lemmas are proved *exactly* as before.

Exercise: 4 stars, standard (STLCArith.weakening) `Lemma weakening : ∀ Gamma Gamma' t T,`
`includedin Gamma Gamma' →`

$\langle \{ \textit{Gamma} \mid - t \text{ \textit{in T} } \} \rangle \rightarrow$
 $\langle \{ \textit{Gamma}' \mid - t \text{ \textit{in T} } \} \rangle.$

Proof. Admitted.

□

Exercise: 4 stars, standard (STLCArith.preservation) **Theorem preservation :** $\forall t \textit{ T},$

$\langle \{ \text{empty} \mid - t \text{ \textit{in T} } \} \rangle \rightarrow$
 $t \rightarrow t' \rightarrow$
 $\langle \{ \text{empty} \mid - t' \text{ \textit{in T} } \} \rangle.$

Proof with eauto. Admitted.

□

Exercise: 4 stars, standard (STLCArith.progress) **Theorem progress :** $\forall t \textit{ T},$

$\langle \{ \text{empty} \mid - t \text{ \textit{in T} } \} \rangle \rightarrow$
value $t \vee \exists t', t \rightarrow t'.$

Proof with eauto. *Admitted.*

□

End STLCARITH.

Chapter 12

Library `PLF.MoreStlc`

12.1 MoreStlc: More on the Simply Typed Lambda-Calculus

```
Set Warnings "-notation-overridden,-parsing".
From PLF Require Import Maps.
From PLF Require Import Types.
From PLF Require Import Smallstep.
From PLF Require Import Stlc.
```

12.2 Simple Extensions to STLC

The simply typed lambda-calculus has a rich enough structure to make its theoretical properties interesting, but it is not much of a programming language!

In this chapter, we begin to close the gap with real-world languages by introducing a number of familiar features that have straightforward treatments at the level of typing.

12.2.1 Numbers

As we saw in the `STLCARITH` exercises at the end of the `StlcProp` chapter, adding types, constants, and primitive operations for natural numbers is easy – basically just a matter of combining the `Types` and `Stlc` chapters. Adding more realistic numeric types like machine integers and floats is also straightforward, though of course the specifications of the numeric primitives become more fiddly.

12.2.2 Let Bindings

When writing a complex expression, it is useful to be able to give names to some of its subexpressions to avoid repetition and increase readability. Most languages provide one or more ways of doing this. In OCaml (and Rocq), for example, we can write `let x=t1 in`

$t2$ to mean “reduce the expression $t1$ to a value and bind the name x to this value while reducing $t2$.”

Our `let`-binder follows OCaml in choosing a standard *call-by-value* evaluation order, where the `let`-bound term must be fully reduced before reduction of the `let`-body can begin. The typing rule T_Let tells us that the type of a `let` can be calculated by calculating the type of the `let`-bound term, extending the context with a binding with this type, and in this enriched context calculating the type of the body (which is then the type of the whole `let` expression).

At this point in the book, it’s probably easier simply to look at the rules defining this new feature than to wade through a lot of English text conveying the same information. Here they are:

Syntax:

$t ::= \text{Terms} \mid \dots \text{ (other terms same as before)} \mid \text{let } x=t1 \text{ in } t2 \text{ let-binding}$

Reduction:

$t1 \rightarrow t1'$

(ST_Let1) $\text{let } x=t1 \text{ in } t2 \rightarrow \text{let } x=t1' \text{ in } t2$

(ST_LetValue) $\text{let } x=v1 \text{ in } t2 \rightarrow x:=v1\ t2$

Typing:

$\Gamma \vdash t1 \text{ \textbackslash in } T1 \ x \vdash T1; \Gamma \vdash t2 \text{ \textbackslash in } T2$

(T_Let) $\Gamma \vdash \text{let } x=t1 \text{ in } t2 \text{ \textbackslash in } T2$

12.2.3 Pairs

Our functional programming examples in Rocq have made frequent use of *pairs* of values. The type of such a pair is called a *product type*.

The formalization of pairs is almost too simple to be worth discussing. However, let’s look briefly at the various parts of the definition to emphasize the common pattern.

In Rocq, the primitive way of extracting the components of a pair is *pattern matching*. An alternative is to take `fst` and `snd` – the first- and second-projection operators – as primitives. Just for fun, let’s do our pairs this way. For example, here’s how we’d write a function that takes a pair of numbers and returns the pair of their sum and difference:

$\backslash x : \text{Nat} * \text{Nat}, \text{let sum} = x.\text{fst} + x.\text{snd} \text{ in let diff} = x.\text{fst} - x.\text{snd} \text{ in (sum,diff)}$

Adding pairs to the simply typed lambda-calculus, then, involves adding two new forms of term – pairing, written $(t1, t2)$, and projection, written $t.\text{fst}$ for the first projection from t and $t.\text{snd}$ for the second projection – plus one new type constructor, $T1 \times T2$, called the *product* of $T1$ and $T2$.

Syntax:

$t ::= \text{Terms} \mid \dots \mid (t1, t2) \text{ pair} \mid t0.\text{fst} \text{ first projection} \mid t0.\text{snd} \text{ second projection}$

$v ::= \text{Values} \mid \dots \mid (v1, v2) \text{ pair value}$

$T ::= \text{Types} \mid \dots \mid T1 * T2 \text{ product type}$

For reduction, we need several new rules specifying how pairs and projection behave.

$t1 \rightarrow t1'$

(ST_Pair1) $(t1, t2) \rightarrow (t1', t2)$
 $t2 \rightarrow t2'$

(ST_Pair2) $(v1, t2) \rightarrow (v1, t2')$
 $t0 \rightarrow t0'$

(ST_Fst1) $t0.fst \rightarrow t0'.fst$

(ST_FstPair) $(v1, v2).fst \rightarrow v1$
 $t0 \rightarrow t0'$

(ST_Snd1) $t0.snd \rightarrow t0'.snd$

(ST_SndPair) $(v1, v2).snd \rightarrow v2$

Rules **ST_FstPair** and **ST_SndPair** say that, when a fully reduced pair meets a first or second projection, the result is the appropriate component. The congruence rules **ST_Fst1** and **ST_Snd1** allow reduction to proceed under projections, when the term being projected from has not yet been fully reduced. **ST_Pair1** and **ST_Pair2** reduce the parts of pairs: first the left part, and then – when a value appears on the left – the right part. The ordering arising from the use of the metavariables v and t in these rules enforces a left-to-right evaluation strategy for pairs. (Note the implicit convention that metavariables like v and $v1$ can only denote values.) We’ve also added a clause to the definition of values, above, specifying that $(v1, v2)$ is a value. The fact that the components of a pair value must themselves be values ensures that a pair passed as an argument to a function will be fully reduced before the function body starts executing.

The typing rules for pairs and projections are straightforward.

$\Gamma \vdash t1 \text{ \textit{in} } T1 \quad \Gamma \vdash t2 \text{ \textit{in} } T2$

(T_Pair) $\Gamma \vdash (t1, t2) \text{ \textit{in} } T1 * T2$
 $\Gamma \vdash t0 \text{ \textit{in} } T1 * T2$

(T_Fst) $\Gamma \vdash t0.fst \text{ \textit{in} } T1$
 $\Gamma \vdash t0 \text{ \textit{in} } T1 * T2$

(T_Snd) $\Gamma \vdash t0.snd \text{ \textit{in} } T2$

T_Pair says that $(t1, t2)$ has type $T1 \times T2$ if $t1$ has type $T1$ and $t2$ has type $T2$. Conversely, **T_Fst** and **T_Snd** tell us that, if $t0$ has a product type $T1 \times T2$ (i.e., if it will reduce to a pair), then the types of the projections from this pair are $T1$ and $T2$.

12.2.4 Unit

Another handy base type, found especially in functional languages, is the singleton type *Unit*.

It has a single element – the term constant *unit* (with a small *u*) – and a typing rule making *unit* an element of *Unit*. We also add *unit* to the set of possible values – indeed, *unit* is the *only* possible result of reducing an expression of type *Unit*.

Syntax:

$t ::= \text{Terms} \mid \dots \text{ (other terms same as before)} \mid \text{unit unit}$

$v ::= \text{Values} \mid \dots \mid \text{unit unit value}$

$T ::= \text{Types} \mid \dots \mid \text{Unit unit type}$

Typing:

$(T\_Unit) \text{ Gamma} \vdash \text{unit} \text{ in Unit}$

It may seem a little strange to bother defining a type that has just one element – after all, wouldn’t every computation living in such a type be trivial?

This is a fair question, and indeed in the STLC the *Unit* type is not especially critical (though we’ll see two uses for it below). Where *Unit* really comes in handy is in richer languages with *side effects* – e.g., assignment statements that mutate variables or pointers, exceptions and other sorts of nonlocal control structures, etc. In such languages, it is convenient to have a type for the (trivial) result of an expression that is evaluated only for its effect.

No! For instance $(\lambda x:\text{Unit},x) \text{ unit}$ is also a *term* of type *unit*.

12.2.5 Sums

Many programs need to deal with values that can take two distinct forms. For example, we might identify students in a university database using *either* their name *or* their id number. A search function might return *either* a matching value *or* an error code.

These are specific examples of a binary *sum type* (sometimes called a *disjoint union*), which describes a set of values drawn from one of two given types, e.g.:

Nat + Bool We create elements of these types by *tagging* elements of the component types. For example, if *n* is a **Nat** then *inl n* is an element of **Nat+Bool**; similarly, if *b* is a **Bool** then *inr b* is a **Nat+Bool**. The names of the tags *inl* and *inr* arise from thinking of them as functions

$\text{inl} \text{ in Nat} \rightarrow \text{Nat} + \text{Bool}$ $\text{inr} \text{ in Bool} \rightarrow \text{Nat} + \text{Bool}$

that “inject” elements of **Nat** or **Bool** into the left and right components of the sum type **Nat+Bool**. (But note that we don’t actually treat them as functions in the way we formalize them: *inl* and *inr* are keywords, and *inl t* and *inr t* are primitive syntactic forms, not function applications.)

In general, the elements of a type $T1 + T2$ consist of the elements of *T1* tagged with the token *inl*, plus the elements of *T2* tagged with *inr*.

As we’ve seen in Rocq programming, one important use of sums is signaling errors:

$\text{div} \setminus \text{in Nat} \rightarrow \text{Nat} \rightarrow (\text{Nat} + \text{Unit})$ $\text{div} = \setminus x:\text{Nat}, \setminus y:\text{Nat}, \text{if iszero } y \text{ then inr unit else inl } \dots$ The type $\text{Nat} + \text{Unit}$ above is in fact isomorphic to **option nat** in Rocq – i.e., it’s easy to write functions that translate back and forth.

To *use* elements of sum types, we introduce a **case** construct (a very simplified form of Rocq’s **match**) to destruct them. For example, the following procedure converts a $\text{Nat} + \text{Bool}$ into a **Nat**:

```
getNat \in Nat+Bool -> Nat
getNat = \x:Nat+Bool, case x of inl n => n | inr b => if
b then 1 else 0
```

More formally...

Syntax:

$t ::= \text{Terms} \mid \dots$ (other terms same as before) $\mid \text{inl } T2 \ t1 \text{ tagging (left)} \mid \text{inr } T1 \ t2 \text{ tagging (right)} \mid \text{case } t0 \text{ of case analysis inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2$

$v ::= \text{Values} \mid \dots \mid \text{inl } T2 \ v1 \text{ tagged value (left)} \mid \text{inr } T1 \ v2 \text{ tagged value (right)}$

$T ::= \text{Types} \mid \dots \mid T1 + T2 \text{ sum type}$

Reduction:

$t1 \rightarrow t1'$

(ST_Inl) $\text{inl } T2 \ t1 \rightarrow \text{inl } T2 \ t1'$
 $t2 \rightarrow t2'$

(ST_Inr) $\text{inr } T1 \ t2 \rightarrow \text{inr } T1 \ t2'$
 $t0 \rightarrow t0'$

(ST_Case) $\text{case } t0 \text{ of inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \rightarrow \text{case } t0' \text{ of inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2$

(ST_CaseInl) $\text{case (inl } T2 \ v1) \text{ of inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 > x1 := v1 t1$

(ST_CaseInr) $\text{case (inr } T1 \ v2) \text{ of inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 > x2 := v2 t2$

Typing:

$\Gamma \vdash t1 \setminus \text{in } T1$

(T_Inl) $\Gamma \vdash \text{inl } T2 \ t1 \setminus \text{in } T1 + T2$
 $\Gamma \vdash t2 \setminus \text{in } T2$

(T_Inr) $\Gamma \vdash \text{inr } T1 \ t2 \setminus \text{in } T1 + T2$
 $\Gamma \vdash t0 \setminus \text{in } T1 + T2 \ x1 \mapsto T1; \Gamma \vdash t1 \setminus \text{in } T3 \ x2 \mapsto T2; \Gamma \vdash t2 \setminus \text{in } T3$

(T_Case) $\Gamma \vdash \text{case } t0 \text{ of inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \setminus \text{in } T3$

We use the type annotations on *inl* and *inr* to make the typing relation deterministic (each term has at most one type), as we did for functions.

Without this extra information, the typing rule **T_Inl**, for example, would have to say that, once we have shown that *t1* is an element of type *T1*, we can derive that *inl t1* is an

element of $T1 + T2$ for *any* type $T2$. For example, we could derive both $\text{inl } 5 : \text{Nat} + \text{Nat}$ and $\text{inl } 5 : \text{Nat} + \text{Bool}$ (and infinitely many other types). This peculiarity (technically, a failure of uniqueness of types) would mean that we cannot build a typechecking algorithm simply by “reading the rules from bottom to top” as we could for all the other features seen so far.

There are various ways to deal with this difficulty. One simple one – which we’ve adopted here – forces the programmer to explicitly annotate the “other side” of a sum type when performing an injection. This is a bit heavy for programmers (so real languages adopt other solutions), but it is easy to understand and formalize.

12.2.6 Lists

The typing features we have seen can be classified into *base types* like `Bool`, and *type constructors* like $\rightarrow$ and $\times$ that build new types from old ones. Another useful type constructor is `List`. For every type T , the type `List T` describes finite-length lists whose elements are drawn from T .

In principle, we could encode lists using pairs, sums, unit, and *recursive* types. But giving semantics to recursive types is non-trivial. Instead, we’ll just discuss the special case of lists directly.

Below we give the syntax, semantics, and typing rules for lists. Except for the fact that explicit type annotations are mandatory on `nil` and cannot appear on `cons`, these lists are essentially identical to those we built in Rocq. We use `case`, rather than `head` and `tail` operators, to destruct lists, to avoid dealing with questions like “what is the *head* of the empty list?”

For example, here is a function that calculates the sum of the first two elements of a list of numbers:

```
\x:List Nat, case x of nil => 0 | a::x' => case x' of nil => a | b::x'' => a+b
```

Syntax:

```
t ::= Terms | ... | nil T empty list | t1 :: t2 cons | case t1 of case analysis nil => t2 |
xh::xt => t3
```

```
v ::= Values | ... | nil T nil value | v1 :: v2 cons value
```

```
T ::= Types | ... | List T list of Ts
```

Reduction:

```
t1 -> t1'
```

```
(ST_Cons1) t1 :: t2 -> t1' :: t2
          t2 -> t2'
```

```
(ST_Cons2) v1 :: t2 -> v1 :: t2'
          t1 -> t1'
```

```
(ST_Lcase1) (case t1 of nil => t2 | xh::xt => t3) -> (case t1' of nil => t2 | xh::xt => t3)
```

(ST_LcaseNil) (case nil T1 of nil => t2 | xh::xt => t3) > t2

(ST_LcaseCons) (case (vh::vt) of nil => t2 | xh::xt => t3) > *xh:=vht:=vtt3*
 Typing:

(T_Nil) Gamma |- nil T1 \in List T1
 Gamma |- t1 \in T1 Gamma |- t2 \in List T1

(T_Cons) Gamma |- t1 :: t2 \in List T1
 Gamma |- t1 \in List T1 Gamma |- t2 \in T2 (xh|->T1; xt|->List T1; Gamma) |- t3
 \in T2

(T_Lcase) Gamma |- (case t1 of nil => t2 | xh::xt => t3) \in T2

12.2.7 General Recursion

Another facility found in most programming languages (including Rocq) is the ability to define recursive functions. For example, we would like to be able to define and use the factorial function like this:

let fact = \x:Nat, if x=0 then 1 else x * (fact (pred x))) in fact 3.

Note that the right-hand side of this binder mentions *fact*, the variable being bound – something that is not allowed according to the way we defined *let* above.

(The body of a *let* is typechecked in the same context as the *let* itself, which means that the recursive occurrence of *fact* in the body will not have a type in the context when it is looked up by the *T_Var* rule.)

Changing the *let* rule to handle “recursive definitions” like this is possible, but it requires some extra effort – e.g., passing around an extra “environment” of recursive function definitions in the definition of the *step* relation. We’re going to take a simpler path here.

Here is another way of presenting recursive functions that is a bit more verbose but equally powerful and much more straightforward to formalize: instead of writing recursive definitions, we will define a *fixed-point operator* called *fix* that performs the “unfolding” of the recursive definition in the right-hand side as needed, during reduction.

For example, instead of

fact = \x:Nat, if x=0 then 1 else x * (fact (pred x)))

we will write:

fact = fix (\f:Nat->Nat, \x:Nat, if x=0 then 1 else x * (f (pred x)))

We can derive the latter from the former as follows:

- In the right-hand side of the definition of *fact*, replace recursive references to *fact* by a fresh variable *f*.
- Add an abstraction binding *f* at the front, with an appropriate type annotation. (Since we are using *f* in place of *fact*, which had type *Nat*→*Nat*, we should require *f* to have the same type.) The new abstraction has type (*Nat*→*Nat*) → (*Nat*→*Nat*).

- Apply `fix` to this abstraction. This application has type $\text{Nat} \rightarrow \text{Nat}$.
- Use all of this as the right-hand side of an ordinary `let`-binding for `fact`.

For the mathematically inclined, the intuition here is that the higher-order function `f` passed to `fix` is a *generator* for the `fact` function: if `f` is applied to a function that “approximates” the desired behavior of `fact` up to some number `n` (that is, a function that returns correct results on inputs less than or equal to `n` but we don’t care what it does on inputs greater than `n`), then `f` returns a slightly better approximation to `fact` – a function that returns correct results for inputs up to `n+1`. Applying `fix` to this generator returns its *fixed point*, which is a function that gives the desired behavior for all inputs `n`.

(The term “fixed point” is used here in exactly the same sense as in ordinary mathematics, where a fixed point of a function `f` is an input `x` such that `f(x) = x`. Here, a fixed point of a function `F` of type $(\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat})$ is a function `f` of type $\text{Nat} \rightarrow \text{Nat}$ such that `F f` behaves the same as `f`.)

Syntax:

`t ::= Terms | ... | fix t1` fixed-point operator

Reduction:

`t1 -> t1'`

(ST_Fix1) `fix t1 -> fix t1'`

(ST_FixAbs) `fix (\xf:T1.t1) -> xf:=fix (\xf:T1.t1) t1`

Typing:

$\Gamma \vdash t1 \text{ in } T1 \rightarrow T1$

(T_Fix) $\Gamma \vdash \text{fix } t1 \text{ in } T1$

Let’s see how `ST_FixAbs` works by reducing `fact 3 = fix F 3`, where

`F = (\f. \x. if x=0 then 1 else x * (f (pred x)))`

(type annotations are omitted for brevity).

`fix F 3`

$\rightarrow \text{ST\_FixAbs} + \text{ST\_App1}$

`(\x. if x=0 then 1 else x * (fix F (pred x))) 3`

$\rightarrow \text{ST\_AppAbs}$

`if 3=0 then 1 else 3 * (fix F (pred 3))`

$\rightarrow \text{ST\_If0\_Nonzero}$

`3 * (fix F (pred 3))`

$\rightarrow \text{ST\_FixAbs} + \text{ST\_Mult2} + \text{ST\_App1}$

`3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 3))`

$\rightarrow \text{ST\_PredNat} + \text{ST\_Mult2} + \text{ST\_App2}$

`3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 2)`

$\rightarrow \text{ST\_AppAbs} + \text{ST\_Mult2}$

`3 * (if 2=0 then 1 else 2 * (fix F (pred 2)))`

```

-> ST_If0_Nonzero + ST_Mult2
3 * (2 * (fix F (pred 2)))
-> ST_FixAbs + 2 x ST_Mult2 + ST_App1
3 * (2 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 2)))
-> ST_PredNat + 2 x ST_Mult2 + ST_App2
3 * (2 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 1))
-> ST_AppAbs + 2 x ST_Mult2
3 * (2 * (if 1=0 then 1 else 1 * (fix F (pred 1))))
-> ST_If0_Nonzero + 2 x ST_Mult2
3 * (2 * (1 * (fix F (pred 1))))
-> ST_FixAbs + 3 x ST_Mult2 + ST_App1
3 * (2 * (1 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 1))))
-> ST_PredNat + 3 x ST_Mult2 + ST_App2
3 * (2 * (1 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 0)))
-> ST_AppAbs + 3 x ST_Mult2
3 * (2 * (1 * (if 0=0 then 1 else 0 * (fix F (pred 0)))))
-> ST_If0Zero + 3 x ST_Mult2
3 * (2 * (1 * 1))
-> ST_MultNats + 2 x ST_Mult2
3 * (2 * 1)
-> ST_MultNats + ST_Mult2
3 * 2
-> ST_MultNats
6

```

The simply typed lambda-calculus with fixed points is a famous and extensively studied system. It is often called *PCF* because it is a simple language of “partial computable functions”.

Exercise: 1 star, standard, optional (halve_fix) Translate this informal recursive definition into one using **fix**:

halve = $\lambda x:\text{Nat}, \text{ if } x=0 \text{ then } 0 \text{ else if } (\text{pred } x)=0 \text{ then } 0 \text{ else } 1 + (\text{halve } (\text{pred } (\text{pred } x)))$
 $\square$

Exercise: 1 star, standard, optional (fact_steps) Write down the sequence of steps that the term **fact** 1 goes through to reduce to a normal form (assuming the usual reduction rules for arithmetic operations).
 $\square$

The ability to form the fixed point of a function of type $\mathbf{T} \rightarrow \mathbf{T}$ for any $\mathbf{T}$ has some surprising consequences. In particular, it implies that *every* type is inhabited by some term. To see this, observe that, for every type $\mathbf{T}$, we can define the term

$\text{fix } (\lambda x:\mathbf{T}, x)$

By T_Fix and T_Abs , this term has type T . By ST_FixAbs it reduces to itself, over and over again. Thus it is a *diverging element* of T .

More usefully, here's an example using fix to define a two-argument recursive function:

$equal = fix (\backslash eq:Nat \rightarrow Nat \rightarrow Bool, \backslash m:Nat, \backslash n:Nat, \text{if } m=0 \text{ then iszero } n \text{ else if } n=0 \text{ then false else eq (pred } m) \text{ (pred } n))$ And finally, here is an example where fix is used to define a *pair* of recursive functions (illustrating the fact that the type $T1$ in the rule T_Fix need not be a function type):

$let \text{ evenodd} = fix (\backslash eo: ((Nat \rightarrow Nat) * (Nat \rightarrow Nat)), (\backslash n:Nat, \text{if0 } n \text{ then } 1 \text{ else } (eo.snd (\text{pred } n))), \backslash n:Nat, \text{if0 } n \text{ then } 0 \text{ else } (eo.fst (\text{pred } n))))$ in $let \text{ even} = \text{evenodd.fst}$ in $let \text{ odd} = \text{evenodd.snd}$ in $(\text{even } 3, \text{even } 4)$

12.2.8 Records

As a final example of a basic extension of the STLC, let's look briefly at how to define *records* and their types. Intuitively, records can be obtained from pairs by two straightforward generalizations: they are n -ary (rather than just binary) and their fields are accessed by *label* (rather than position).

Syntax:

$t ::= \text{Terms} \mid \dots \mid \{i1=t1, \dots, in=tn\} \text{ record} \mid t0.i \text{ projection}$

$v ::= \text{Values} \mid \dots \mid \{i1=v1, \dots, in=vn\} \text{ record value}$

$T ::= \text{Types} \mid \dots \mid \{i1:T1, \dots, in:Tn\} \text{ record type}$

The generalization from products should be pretty obvious. But it's worth noticing the ways in which what we've actually written is even *more* informal than the informal syntax we've used in previous sections and chapters: we've used “...” in several places to mean “any number of these,” and we've omitted explicit mention of the usual side condition that the labels of a record should not contain any repetitions.

Reduction:

$ti \rightarrow ti'$

(ST_Rcd) $\{i1=v1, \dots, im=vm, in=ti, \dots\} > \{i1=v1, \dots, im=vm, in=ti', \dots\}$
 $t0 \rightarrow t0'$

(ST_Proj1) $t0.i \rightarrow t0'.i$

(ST_ProjRcd) $\{\dots, i=vi, \dots\}.i \rightarrow vi$

Again, these rules are a bit informal. For example, the first rule is intended to be read “if ti is the leftmost field that is not a value and if ti steps to ti' , then the whole record steps...” In the last rule, the intention is that there should be only one field called i , and that all the other fields must contain values.

The typing rules are also simple:

$\Gamma \vdash t1 \backslash in T1 \dots \Gamma \vdash tn \backslash in Tn$

(T_Rcd) $\Gamma \vdash \{i1=t1, \dots, in=tn\} \backslash in \{i1:T1, \dots, in:Tn\}$

$\Gamma \vdash t_0 \in \{ \dots, i:Ti, \dots \}$

(T_Proj) $\Gamma \vdash t_0.i \in Ti$

There are several ways to approach formalizing the above definitions.

- We can directly formalize the syntactic forms and inference rules, staying as close as possible to the form we've given them above. This is conceptually straightforward, and it's probably what we'd want to do if we were building a real compiler (in particular, it will allow us to print error messages in the form that programmers will find easy to understand). But the formal versions of the rules will not be very pretty or easy to work with, because all the ...s above will have to be replaced with explicit quantifications or comprehensions. For this reason, records are not included in the extended exercise at the end of this chapter. (It is still useful to discuss them informally here because they will help motivate the addition of subtyping to the type system when we get to the [Sub](#) chapter.)
- Alternatively, we could look for a smoother way of presenting records – for example, a binary presentation with one constructor for the empty record and another constructor for adding a single field to an existing record, instead of a single monolithic constructor that builds a whole record at once. This is the right way to go if we are primarily interested in studying the metatheory of the calculi with records, since it leads to clean and elegant definitions and proofs. Chapter [Records](#) shows how this can be done.
- Finally, if we like, we can avoid formalizing records altogether, by stipulating that record notations are just informal shorthands for more complex expressions involving pairs and product types. We sketch this approach in the next section.

Encoding Records (Optional)

Let's see how records can be encoded using just pairs and *unit*. (This clever encoding, as well as the observation that it also extends to systems with subtyping, is due to Luca Cardelli.)

First, observe that we can encode arbitrary-size *tuples* using nested pairs and the *unit* value. To avoid overloading the pair notation (t_1, t_2) , we'll use curly braces without labels to write down tuples, so $\{\}$ is the empty tuple, $\{5\}$ is a singleton tuple, $\{5,6\}$ is a 2-tuple (morally the same as a pair), $\{5,6,7\}$ is a triple, etc.

$\{\} \longrightarrow \text{unit}$ $\{t_1, t_2, \dots, t_n\} \longrightarrow (t_1, \text{trest})$ where $\{t_2, \dots, t_n\} \longrightarrow \text{trest}$

Similarly, we can encode tuple types using nested product types:

$\{\} \longrightarrow \text{Unit}$ $\{T_1, T_2, \dots, T_n\} \longrightarrow T_1 * \text{TRest}$ where $\{T_2, \dots, T_n\} \longrightarrow \text{TRest}$

The operation of projecting a field from a tuple can be encoded using a sequence of second projections followed by a first projection:

$t.0 \longrightarrow t.\text{fst}$ $t.(n+1) \longrightarrow (t.\text{snd}).n$

Next, suppose that there is some total ordering on record labels, so that we can associate each label with a unique natural number. This number is called the *position* of the label. For example, we might assign positions like this:

LABEL POSITION a 0 b 1 c 2 bar 1395 foo 4460

We use these positions to encode record values as tuples (i.e., as nested pairs) by sorting the fields according to their positions. For example:

$\{a=5, b=6\} \longrightarrow \{5, 6\}$ $\{a=5, c=7\} \longrightarrow \{5, \text{unit}, 7\}$ $\{c=7, a=5\} \longrightarrow \{5, \text{unit}, 7\}$ $\{c=5, b=3\} \longrightarrow \{\text{unit}, 3, 5\}$ $\{f=8, c=5, a=7\} \longrightarrow \{7, \text{unit}, 5, \text{unit}, \text{unit}, 8\}$ $\{f=8, c=5\} \longrightarrow \{\text{unit}, \text{unit}, 5, \text{unit}, \text{unit}, 8\}$

Note that each field appears in the position associated with its label, that the size of the tuple is determined by the label with the highest position, and that we fill in unused positions with *unit*.

We do exactly the same thing with record types:

$\{a:\text{Nat}, b:\text{Nat}\} \longrightarrow \{\text{Nat}, \text{Nat}\}$ $\{c:\text{Nat}, a:\text{Nat}\} \longrightarrow \{\text{Nat}, \text{Unit}, \text{Nat}\}$ $\{f:\text{Nat}, c:\text{Nat}\} \longrightarrow \{\text{Unit}, \text{Unit}, \text{Nat}, \text{Unit}, \text{Unit}, \text{Nat}\}$

Finally, record projection is encoded as a tuple projection from the appropriate position: $t.l \longrightarrow t.\text{(position of } l\text{)}$

It is not hard to check that all the typing rules for the original “direct” presentation of records are validated by this encoding. (The reduction rules are “almost validated” – not quite, because the encoding reorders fields.)

Of course, this encoding will not be very efficient if we happen to use a record with label *foo*! But things are not actually as bad as they might seem: for example, if we assume that our compiler can see the whole program at the same time, we can *choose* the numbering of labels so that we assign small positions to the most frequently used labels. Indeed, there are industrial compilers that essentially do this!

Variants (Optional)

Just as products can be generalized to records, sums can be generalized to n-ary labeled types called *variants*. Instead of $T1 + T2$, we can write something like $\langle l1:T1, l2:T2, \dots, ln:Tn \rangle$ where $l1, l2, \dots$ are field labels which are used both to build instances and as case arm labels.

These n-ary variants give us almost enough mechanism to build arbitrary inductive data types like lists and trees from scratch – the only thing missing is a way to allow *recursion* in type definitions. We won’t cover this here, but detailed treatments can be found in many textbooks – e.g., *Types and Programming Languages* *Pierce* 2002 (in Bib.v).

12.3 Exercise: Formalizing the Extensions

Module **STLC**EXTENDED.

In this series of exercises, you will formalize some of the extensions described in this chapter. We’ve provided the necessary additions to the syntax of terms and types, and we’ve included a few examples that you can test your definitions with to make sure they are working as expected. You’ll fill in the rest of the definitions and extend all the proofs accordingly.

To get you started, we’ve provided implementations for:

- numbers
- sums
- lists
- unit

You need to complete the implementations for:

- pairs
- let (which involves binding)
- fix

A good strategy is to work on the extensions one at a time (first pairs, then let, then fix), in separate passes, rather than trying to do all three at once in a single pass. For each definition or proof, begin by reading carefully through the parts that are provided for you, referring to the text in the [Stlc](#) chapter for high-level intuitions and the embedded comments for detailed mechanics.

Syntax

```
Inductive ty : Type :=
| Ty_Arrow : ty → ty → ty
| Ty_Nat : ty
| Ty_Sum : ty → ty → ty
| Ty_List : ty → ty
| Ty_Unit : ty
| Ty_Prod : ty → ty → ty.

Inductive tm : Type :=

| tm_var : string → tm
| tm_app : tm → tm → tm
| tm_abs : string → ty → tm → tm

| tm_const : nat → tm
| tm_succ : tm → tm
| tm_pred : tm → tm
| tm_mult : tm → tm → tm
| tm_if0 : tm → tm → tm → tm

| tm_inl : ty → tm → tm
```

```
| tm_inr : ty → tm → tm
| tm_case : tm → string → tm → string → tm → tm
```

```
| tm_nil : ty → tm
| tm_cons : tm → tm → tm
| tm_lcase : tm → tm → string → string → tm → tm
```

```
| tm_unit : tm
```

```
| tm_pair : tm → tm → tm
| tm_fst : tm → tm
| tm_snd : tm → tm
```

```
| tm_let : string → tm → tm → tm
```

```
| tm_fix : tm → tm.
```

Note that, for brevity, we've omitted booleans and instead provided a single *if0* form combining a zero test and a conditional. That is, instead of writing

if $x = 0$ then ... else ...

we'll write this:

if0 x then ... else ...

```
Definition w : string := "w".
```

```
Definition x : string := "x".
```

```
Definition y : string := "y".
```

```
Definition z : string := "z".
```

```
Hint Unfold x : core.
```

```
Hint Unfold y : core.
```

```
Hint Unfold z : core.
```

```
Delimit Scope stlc_scope with stlc.
```

```
Open Scope stlc_scope.
```

```
Notation "x" := x (in custom stlc_ty at level 0, x global) : stlc_scope.
```

```
Notation "<{ x }>" := x (x custom stlc_ty).
```

```
Notation "( t )" := t (in custom stlc_ty at level 0, t custom stlc_ty) : stlc_scope.
```

```
Notation "S -> T" := (Ty_Arrow S T) (in custom stlc_ty at level 99, right associativity)
: stlc_scope.
```

```

Notation "$ ( t )" := t (in custom stlc_ty at level 0, t constr) : stlc_scope.
Notation "x" := x (in custom stlc_tm at level 0, x global) : stlc_scope.
Notation "'_'" := _ (in custom stlc_tm at level 0) : stlc_scope.
Notation "'$' x" := x (in custom stlc_tm at level 0, x constr at level 0) : stlc_scope.
Notation "'$( ' x ' )'" := x (in custom stlc_tm at level 0, x constr, only parsing)
: stlc_scope.
Notation "<{ e }>" := e (e custom stlc_tm at level 200) : stlc_scope.
Notation "( x )" := x (in custom stlc_tm at level 0, x custom stlc_tm) : stlc_scope.
Notation "x y" := (tm_app x y) (in custom stlc_tm at level 10, left associativity)
: stlc_scope.
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc_tm at level 200, x global,
                  t custom stlc_ty,
                  y custom stlc_tm at level 200,
                  left associativity).
Coercion tm_var : string >-> tm.
Arguments tm_var _%string.
Notation "'Nat'" := Ty_Nat (in custom stlc_ty at level 0).
Notation "'succ' x" := (tm_succ x) (in custom stlc_tm at level 10,
                                   x custom stlc_tm at level 0) : stlc_scope.
Notation "'pred' x" := (tm_pred x) (in custom stlc_tm at level 10,
                                   x custom stlc_tm at level 0) : stlc_scope.
Notation "x * y" := (tm_mult x y) (in custom stlc_tm at level 95,
                                   right associativity) : stlc_scope.
Notation "'if0' x 'then' y 'else' z" :=
  (tm_if0 x y z) (in custom stlc_tm at level 0,
                  x custom stlc_tm at level 0,
                  y custom stlc_tm at level 0,
                  z custom stlc_tm at level 0) : stlc_scope.
Coercion tm_const : nat >-> tm.
Notation "S + T" :=
  (Ty_Sum S T) (in custom stlc_ty at level 3, left associativity).
Notation "'inl' T t" := (tm_inl T t) (in custom stlc_tm at level 10,
                                     T custom stlc_ty,
                                     t custom stlc_tm at level 0).
Notation "'inr' T t" := (tm_inr T t) (in custom stlc_tm at level 10,
                                     T custom stlc_ty,
                                     t custom stlc_tm at level 0).
Notation "'case' t0 'of' '|' 'inl' x1 '=>' t1 '|' 'inr' x2 '=>' t2" :=
  (tm_case t0 x1 t1 x2 t2) (in custom stlc_tm at level 200,
                           t0 custom stlc_tm at level 200,

```

```

x1 global,
t1 custom stlc_tm at level 200,
x2 global,
t2 custom stlc_tm at level 200,
left associativity).

Notation "X * Y" :=
  (Ty_Prod X Y) (in custom stlc_ty at level 2, X custom stlc_ty, Y custom stlc_ty
at level 0) : stlc_scope.

Notation "( x ', ' y )" := (tm_pair x y) (in custom stlc_tm at level 0,
x custom stlc_tm,
y custom stlc_tm) : stlc_scope.

Notation "t '.fst'" := (tm_fst t) (in custom stlc_tm at level 1) : stlc_scope.
Notation "t '.snd'" := (tm_snd t) (in custom stlc_tm at level 1) : stlc_scope.

Notation "'List' T" :=
  (Ty_List T) (in custom stlc_ty at level 4) : stlc_scope.
Notation "'List' T" := (Ty_List T) (at level 0) : stlc_scope.
Notation "'nil' T" := (tm_nil T) (in custom stlc_tm at level 0, T custom stlc_ty)
: stlc_scope.
Notation "h '::' t" := (tm_cons h t) (in custom stlc_tm at level 2, right associativity)
: stlc_scope.
Notation "'case' t1 'of' '|' 'nil' '=>' t2 '|' x '::' y '=>' t3" :=
  (tm_lcase t1 t2 x y t3) (in custom stlc_tm at level 200,
t1 custom stlc_tm at level 200,
t2 custom stlc_tm at level 0,
x global,
y global,
t3 custom stlc_tm at level 0,
left associativity) : stlc_scope.

Notation "'Unit'" :=
  (Ty_Unit) (in custom stlc_ty at level 0) : stlc_scope.
Notation "'unit'" := tm_unit (in custom stlc_tm at level 0) : stlc_scope.

Notation "'let' x '=' t1 'in' t2" :=
  (tm_let x t1 t2) (in custom stlc_tm at level 200) : stlc_scope.
Notation "'fix' t" := (tm_fix t) (in custom stlc_tm at level 200) : stlc_scope.

```

Substitution

```

Reserved Notation "'[' x ':=' s ']' t" (in custom stlc_tm at level 5, x global,
s custom stlc_tm,
t custom stlc_tm at next level, right associativity).

```

Exercise: 3 stars, standard (STLCExtended.subst) Fixpoint subst (*x* : string)

(*s* : tm) (*t* : tm) : tm :=

```

match t with

| tm_var y ⇒
  if String.eqb x y then s else t
| <{\y:T, t1}> ⇒
  if String.eqb x y then t else <{\y:T, [x:=s] t1}>
| <{t1 t2}> ⇒
  <{([x:=s] t1) ([x:=s] t2)}>

| tm_const _ ⇒
  t
| <{succ t1}> ⇒
  <{succ ([x := s] t1)}>
| <{pred t1}> ⇒
  <{pred ([x := s] t1)}>
| <{t1 × t2}> ⇒
  <{ ([x := s] t1) × ([x := s] t2) }>
| <{if0 t1 then t2 else t3}> ⇒
  <{if0 [x := s] t1 then [x := s] t2 else [x := s] t3}>

| <{inl T2 t1}> ⇒
  <{inl T2 ( [x:=s] t1) }>
| <{inr T1 t2}> ⇒
  <{inr T1 ( [x:=s] t2) }>
| <{case t0 of | inl y1 ⇒ t1 | inr y2 ⇒ t2}> ⇒
  <{case ([x:=s] t0) of
    | inl y1 ⇒ $(if String.eqb x y1 then t1 else <{ [x:=s] t1 }> )
    | inr y2 ⇒ $(if String.eqb x y2 then t2 else <{ [x:=s] t2 }> ) }>

| <{nil T}> ⇒
  t
| <{t1 :: t2}> ⇒
  <{ ([x:=s] t1) :: [x:=s] t2 }>
| <{case t1 of | nil ⇒ t2 | y1 :: y2 ⇒ t3}> ⇒
  <{case ( [x:=s] t1 ) of
    | nil ⇒ [x:=s] t2
    | y1 :: y2 ⇒
      $(if String.eqb x y1 then
        t3
      else if String.eqb x y2 then t3

```

```

      else <{ [x:=s] t3 }>> }>

| <{unit}>> ⇒ <{unit}>>

```

```

| _ ⇒ t
end

```

where "'[x ' := ' s ']' t" := (subst x s t) (in custom stlc_tm) : stlc_scope.

Example substeg1 :

```
<{ [z:=$0] (let w = z in z) }> = <{ let w = $0 in $0 }>.
```

Proof.

Admitted.

Example substeg2 :

```
<{ [z:=$0] (let w = z in w) }> = <{ let w = $0 in w }>.
```

Proof.

Admitted.

Example substeg3 :

```
<{ [z:=$0] (let y = succ $0 in z) }> = <{ let y = succ $0 in $0 }>.
```

Proof.

Admitted.

□

Reduction

Next we define the values of our language.

Inductive **value** : **tm** → Prop :=

```

| v_abs : ∀ x T2 t1,
  value <{\x:T2, t1}>

| v_nat : ∀ n : nat,
  value <{n}>

```



```

| v_inl :  $\forall v\ T1,$ 
  value  $v \rightarrow$ 
  value  $\langle\{inl\ T1\ v\}\rangle$ 
| v_inr :  $\forall v\ T1,$ 
  value  $v \rightarrow$ 
  value  $\langle\{inr\ T1\ v\}\rangle$ 

| v_nil :  $\forall T1,$  value  $\langle\{nil\ T1\}\rangle$ 
| v_cons :  $\forall v1\ v2,$ 
  value  $v1 \rightarrow$ 
  value  $v2 \rightarrow$ 
  value  $\langle\{v1 :: v2\}\rangle$ 

| v_unit : value  $\langle\{unit\}\rangle$ 

| v_pair :  $\forall v1\ v2,$ 
  value  $v1 \rightarrow$ 
  value  $v2 \rightarrow$ 
  value  $\langle\{(v1, v2)\}\rangle$ .

```

Hint Constructors **value** : core.

Reserved Notation "t '->' t'" (at level 40).

Exercise: 3 stars, standard (STLCExtended.step) Inductive **step** : **tm** $\rightarrow$ **tm** $\rightarrow$ Prop :=

```

| ST_AppAbs :  $\forall x\ T2\ t1\ v2,$ 
  value  $v2 \rightarrow$ 
   $\langle\{(\backslash x:T2, t1)\ v2\}\rangle \rightarrow \langle\{[x:=v2]\ t1\}\rangle$ 
| ST_App1 :  $\forall t1\ t1'\ t2,$ 
   $t1 \rightarrow t1' \rightarrow$ 
   $\langle\{t1\ t2\}\rangle \rightarrow \langle\{t1'\ t2\}\rangle$ 
| ST_App2 :  $\forall v1\ t2\ t2',$ 
  value  $v1 \rightarrow$ 
   $t2 \rightarrow t2' \rightarrow$ 
   $\langle\{v1\ t2\}\rangle \rightarrow \langle\{v1\ t2'\}\rangle$ 

| ST_Succ :  $\forall t1\ t1',$ 
   $t1 \rightarrow t1' \rightarrow$ 
   $\langle\{succ\ t1\}\rangle \rightarrow \langle\{succ\ t1'\}\rangle$ 
| ST_SuccNat :  $\forall n : nat,$ 
   $\langle\{succ\ n\}\rangle \rightarrow \langle\{\$(S\ n)\}\rangle$ 
| ST_Pred :  $\forall t1\ t1',$ 

```

```

     $t1 \rightarrow t1' \rightarrow$ 
     $\langle \{ \text{pred } t1 \} \rangle \rightarrow \langle \{ \text{pred } t1' \} \rangle$ 
| ST_PredNat :  $\forall n : \text{nat},$ 
     $\langle \{ \text{pred } n \} \rangle \rightarrow \langle \{ \$ (n - 1) \} \rangle$ 
| ST_Mulconsts :  $\forall n1 \ n2 : \text{nat},$ 
     $\langle \{ n1 \times n2 \} \rangle \rightarrow \langle \{ \$ (n1 \times n2) \} \rangle$ 
| ST_Mult1 :  $\forall t1 \ t1' \ t2,$ 
     $t1 \rightarrow t1' \rightarrow$ 
     $\langle \{ t1 \times t2 \} \rangle \rightarrow \langle \{ t1' \times t2 \} \rangle$ 
| ST_Mult2 :  $\forall v1 \ t2 \ t2',$ 
    value  $v1 \rightarrow$ 
     $t2 \rightarrow t2' \rightarrow$ 
     $\langle \{ v1 \times t2 \} \rangle \rightarrow \langle \{ v1 \times t2' \} \rangle$ 
| ST_If0 :  $\forall t1 \ t1' \ t2 \ t3,$ 
     $t1 \rightarrow t1' \rightarrow$ 
     $\langle \{ \text{if0 } t1 \text{ then } t2 \text{ else } t3 \} \rangle \rightarrow \langle \{ \text{if0 } t1' \text{ then } t2 \text{ else } t3 \} \rangle$ 
| ST_If0_Zero :  $\forall t2 \ t3,$ 
     $\langle \{ \text{if0 } \$0 \text{ then } t2 \text{ else } t3 \} \rangle \rightarrow t2$ 
| ST_If0_Nonzero :  $\forall n \ t2 \ t3,$ 
     $\langle \{ \text{if0 } \$ (S \ n) \text{ then } t2 \text{ else } t3 \} \rangle \rightarrow t3$ 

| ST_Inl :  $\forall t1 \ t1' \ T2,$ 
     $t1 \rightarrow t1' \rightarrow$ 
     $\langle \{ \text{inl } T2 \ t1 \} \rangle \rightarrow \langle \{ \text{inl } T2 \ t1' \} \rangle$ 
| ST_Inr :  $\forall t2 \ t2' \ T1,$ 
     $t2 \rightarrow t2' \rightarrow$ 
     $\langle \{ \text{inr } T1 \ t2 \} \rangle \rightarrow \langle \{ \text{inr } T1 \ t2' \} \rangle$ 
| ST_Case :  $\forall t0 \ t0' \ x1 \ t1 \ x2 \ t2,$ 
     $t0 \rightarrow t0' \rightarrow$ 
     $\langle \{ \text{case } t0 \text{ of } | \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \} \rangle \rightarrow$ 
     $\langle \{ \text{case } t0' \text{ of } | \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \} \rangle$ 
| ST_CaseInl :  $\forall v0 \ x1 \ t1 \ x2 \ t2 \ T2,$ 
    value  $v0 \rightarrow$ 
     $\langle \{ \text{case inl } T2 \ v0 \text{ of } | \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \} \rangle \rightarrow \langle \{ [x1 := v0] t1 \} \rangle$ 
| ST_CaseInr :  $\forall v0 \ x1 \ t1 \ x2 \ t2 \ T1,$ 
    value  $v0 \rightarrow$ 
     $\langle \{ \text{case inr } T1 \ v0 \text{ of } | \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \} \rangle \rightarrow \langle \{ [x2 := v0] t2 \} \rangle$ 

| ST_Cons1 :  $\forall t1 \ t1' \ t2,$ 
     $t1 \rightarrow t1' \rightarrow$ 
     $\langle \{ t1 :: t2 \} \rangle \rightarrow \langle \{ t1' :: t2 \} \rangle$ 
| ST_Cons2 :  $\forall v1 \ t2 \ t2',$ 

```

```

    value v1 →
    t2 -> t2' →
    <{v1 :: t2}> -> <{v1 :: t2'}>
| ST_Lcase1 : ∀ t1 t1' t2 x1 x2 t3,
    t1 -> t1' →
    <{case t1 of | nil ⇒ t2 | x1 :: x2 ⇒ t3}> ->
    <{case t1' of | nil ⇒ t2 | x1 :: x2 ⇒ t3}>
| ST_LcaseNil : ∀ T1 t2 x1 x2 t3,
    <{case nil T1 of | nil ⇒ t2 | x1 :: x2 ⇒ t3}> -> t2
| ST_LcaseCons : ∀ v1 vl t2 x1 x2 t3,
    value v1 →
    value vl →
    <{case v1 :: vl of | nil ⇒ t2 | x1 :: x2 ⇒ t3}>
    -> <{ [x2 := vl] ([x1 := v1] t3) }>

```

where "t '→' t'" := (step t t').

□

Notation multistep := (multi step).

Notation "t1 '→*' t2" := (multistep t1 t2) (at level 40).

Hint Constructors step : core.

Typing

Definition context := partial_map ty.

Next we define the typing rules. These are nearly direct transcriptions of the inference rules shown above.

Notation "x '|->' v ';' m " := (update m x v)

(in custom stlc_tm at level 0, x global, v custom stlc_ty, right associativity)
: stlc_scope.

Notation "x '|->' v " := (update empty x v)

(in custom stlc_tm at level 0, x global, v custom stlc_ty) : stlc_scope.

Notation "'empty'" := empty (in custom stlc_tm) : stlc_scope.

Reserved Notation "<{ Gamma ' |- ' t '\in' T }>"

(at level 0, Gamma custom stlc_tm at level 200, t custom stlc_tm, T custom stlc_ty).

Exercise: 3 stars, standard (STLCExtended.has_type) Inductive **has_type** : context $\rightarrow$ **tm** $\rightarrow$ **ty** $\rightarrow$ Prop :=

```
| T_Var :  $\forall$  Gamma x T1,
  Gamma x = Some T1  $\rightarrow$ 
  <{ Gamma |- x \in T1 }>
| T_Abs :  $\forall$  Gamma x T1 T2 t1,
  <{ x |-> T2 ; Gamma |- t1 \in T1 }>  $\rightarrow$ 
  <{ Gamma |- \x:T2, t1 \in T2  $\rightarrow$  T1 }>
| T_App :  $\forall$  T1 T2 Gamma t1 t2,
  <{ Gamma |- t1 \in T2  $\rightarrow$  T1 }>  $\rightarrow$ 
  <{ Gamma |- t2 \in T2 }>  $\rightarrow$ 
  <{ Gamma |- t1 t2 \in T1 }>

| T_Nat :  $\forall$  Gamma (n : nat),
  <{ Gamma |- n \in Nat }>
| T_Succ :  $\forall$  Gamma t,
  <{ Gamma |- t \in Nat }>  $\rightarrow$ 
  <{ Gamma |- succ t \in Nat }>
| T_Pred :  $\forall$  Gamma t,
  <{ Gamma |- t \in Nat }>  $\rightarrow$ 
  <{ Gamma |- pred t \in Nat }>
| T_Mult :  $\forall$  Gamma t1 t2,
  <{ Gamma |- t1 \in Nat }>  $\rightarrow$ 
  <{ Gamma |- t2 \in Nat }>  $\rightarrow$ 
  <{ Gamma |- t1  $\times$  t2 \in Nat }>
| T_If0 :  $\forall$  Gamma t1 t2 t3 T0,
  <{ Gamma |- t1 \in Nat }>  $\rightarrow$ 
  <{ Gamma |- t2 \in T0 }>  $\rightarrow$ 
  <{ Gamma |- t3 \in T0 }>  $\rightarrow$ 
  <{ Gamma |- if0 t1 then t2 else t3 \in T0 }>

| T_Inl :  $\forall$  Gamma t1 T1 T2,
  <{ Gamma |- t1 \in T1 }>  $\rightarrow$ 
  <{ Gamma |- (inl T2 t1) \in T1 + T2 }>
| T_Inr :  $\forall$  Gamma t2 T1 T2,
  <{ Gamma |- t2 \in T2 }>  $\rightarrow$ 
  <{ Gamma |- (inr T1 t2) \in T1 + T2 }>
```

```

| T_Case : ∀ Gamma t0 x1 T1 t1 x2 T2 t2 T3,
  <{ Gamma |- t0 \in T1 + T2 }> →
  <{ x1 |-> T1 ; Gamma |- t1 \in T3 }> →
  <{ x2 |-> T2 ; Gamma |- t2 \in T3 }> →
  <{ Gamma |- case t0 of | inl x1 ⇒ t1 | inr x2 ⇒ t2 \in T3 }>

| T_Nil : ∀ Gamma T1,
  <{ Gamma |- nil T1 \in List T1 }>
| T_Cons : ∀ Gamma t1 t2 T1,
  <{ Gamma |- t1 \in T1 }> →
  <{ Gamma |- t2 \in List T1 }> →
  <{ Gamma |- t1 :: t2 \in List T1 }>
| T_Lcase : ∀ Gamma t1 T1 t2 x1 x2 t3 T2,
  <{ Gamma |- t1 \in List T1 }> →
  <{ Gamma |- t2 \in T2 }> →
  <{ x1 |-> T1 ; x2 |-> List T1 ; Gamma |- t3 \in T2 }> →
  <{ Gamma |- case t1 of | nil ⇒ t2 | x1 :: x2 ⇒ t3 \in T2 }>

| T_Unit : ∀ Gamma,
  <{ Gamma |- unit \in Unit }>

```

where "`<{ Gamma ' |- ' t '\in' T }>`" := (`has_type` *Gamma* *t* *T*).

□

Hint Constructors `has_type` : core.

12.3.1 Examples

Exercise: 5 stars, standard, optional (STLCExtended_examples) This section presents formalized versions of the examples from above (plus several more).

For each example, uncomment proofs and replace *Admitted* by `Qed` once you've implemented enough of the definitions for the tests to pass.

The examples at the beginning focus on specific features; you can use these to make sure your definition of a given feature is reasonable before moving on to extending the proofs later

in the file with the cases relating to this feature. The later examples require all the features together, so you'll need to come back to these when you've got all the definitions filled in.

Module `EXAMPLES`.

Preliminaries

First, let's define a few variable names:

```
Open Scope string_scope.
Notation x := "x".
Notation y := "y".
Notation a := "a".
Notation f := "f".
Notation g := "g".
Notation l := "l".
Notation k := "k".
Notation i1 := "i1".
Notation i2 := "i2".
Notation processSum := "processSum".
Notation n := "n".
Notation eq := "eq".
Notation m := "m".
Notation evenodd := "evenodd".
Notation even := "even".
Notation odd := "odd".
Notation eo := "eo".
```

Next, a bit of Rocq hackery to automate searching for typing derivations. You don't need to understand this bit in detail – just have a look over it so that you'll know what to look for if you ever find yourself needing to make custom extensions to `auto`.

The following `Hint` declarations say that, whenever `auto` arrives at a goal of the form $(\Gamma \vdash (tm\_app\ e1\ e2) \text{ in } T)$, it should consider `eapply T_App`, leaving an existential variable for the middle type $T1$, and similar for `lcase`. That variable will then be filled in during the search for type derivations for `e1` and `e2`. We also include a hint to “try harder” when solving equality goals; this is useful to automate uses of `T_Var` (which includes an equality as a precondition).

```
Hint Extern 2 (has_type _ (tm_app _ _) _) =>
  eapply T_App; auto : core.
Hint Extern 2 (has_type _ (tm_lcase _ _ _ _) _) =>
  eapply T_Lcase; auto : core.
Hint Extern 2 (_ = _) => compute; reflexivity : core.
```

Numbers

```
Module NUMTEST.
Definition tm_test :=
  <{if0
    (pred
      (succ
        (pred
          ( $2 \times \$0$ ))))
    then $5
    else $6}>.
Example typechecks :
  <{ empty |- tm_test \in Nat }>.
Proof.
  unfold tm_test.
  auto 10.
  Admitted.
Example reduces :
  tm_test ->* 5.
Proof.
  Admitted.
End NUMTEST.
```

Products

```
Module PRODTTEST.
Definition tm_test :=
  <{((($5,$6),$7).fst.snd)>.
Example typechecks :
  <{ empty |- tm_test \in Nat }>.
Proof. unfold tm_test. eauto. Admitted.
Example reduces :
  tm_test ->* 6.
Proof.
  Admitted.
End PRODTTEST.
```

let

```
Module LETTEST.
```

```

Definition tm_test :=
  <{let x = (pred $6) in
    (succ x)}>.

Example typechecks :
  <{ empty |- tm_test \in Nat }>.
Proof. unfold tm_test. eauto.
  Admitted.

Example reduces :
  tm_test ->* 6.
Proof.
  Admitted.

End LETTEST.

Module LETTEST1.

Definition tm_test :=
  <{ let z = pred $6 in
    (succ z) }>.

Example typechecks :
  <{ empty |- tm_test \in Nat }>.
Proof. unfold tm_test. eauto.
  Admitted.

Example reduces :
  tm_test ->* 6.
Proof.
  Admitted.

End LETTEST1.

```

Sums

```

Module SUMTEST1.

Definition tm_test :=
  <{ case (inl Nat $5) of
    | inl x => x
    | inr y => y }>.

Example typechecks :
  <{ empty |- tm_test \in Nat }>.
Proof. unfold tm_test. eauto. Admitted.

Example reduces :
  tm_test ->* 5.
Proof.

```



```

    Admitted.
End SUMTEST1.
Module SUMTEST2.

Definition tm_test :=
  <{ let processSum =
    (\x:Nat + Nat,
     case x of
     | inl n => n
     | inr n => (if0 n then $1 else $0)) in
    (processSum (inl Nat $5), processSum (inr Nat $5)) }>.

Example typechecks :
  <{ empty |- tm_test \in Nat × Nat }>.
Proof. unfold tm_test. eauto 10. Admitted.

Example reduces :
  tm_test ->* <{ ($5, $0) }>.
Proof.
  Admitted.
End SUMTEST2.

```

Lists

```

Module LISTTEST.

Definition tm_test :=
  <{ let l = ($5 :: $6 :: (nil Nat)) in
    case l of
    | nil => $0
    | x :: y => (x × x) }>.

Example typechecks :
  <{ empty |- tm_test \in Nat }>.
Proof. unfold tm_test. eauto. Admitted.

Example reduces :
  tm_test ->* 25.
Proof.
  Admitted.
End LISTTEST.

fix

Module FIXTEST1.

```

Definition fact :=

```
<{ fix
  (\f:Nat→Nat,
   \a:Nat,
   if0 a then $1 else (a × (f (pred a)))) }>.
```

(Warning: you may be able to typecheck `fact` but still have some rules wrong!)

Example typechecks :

```
<{ empty |- fact \in Nat → Nat }>.
```

Proof. unfold fact. auto 10. Admitted.

Example reduces :

```
<{ fact $4 }> ->* 24.
```

Proof.

Admitted.

End FIXTEST1.

Module FIXTEST2.

Definition map :=

```
<{ \g:Nat→Nat,
  fix
    (\f:(List Nat)→(List Nat),
     \l:List Nat,
     case l of
     | nil ⇒ nil Nat
     | x::l ⇒ ((g x)::(f l))) }>.
```

Example typechecks :

```
<{ empty |- map \in
  (Nat → Nat) → (List Nat) → (List Nat) }>.
```

Proof. unfold map. auto 10. Admitted.

Example reduces :

```
<{ map (\a:Nat, succ a) ($1 :: $2 :: (nil Nat)) }>
->* <{ $2 :: $3 :: (nil Nat) }>.
```

Proof.

Admitted.

End FIXTEST2.

Module FIXTEST3.

Definition equal :=

```
<{ fix
  (\eq:Nat→Nat→Nat,
   \m:Nat, \n:Nat,
   if0 m then (if0 n then $1 else $0)
```

```

      else (if0 n
        then $0
        else (eq (pred m) (pred n)))) }>.

```

Example typechecks :

```

<{ empty |- equal \in Nat → Nat → Nat }>.

```

Proof. unfold equal. auto 10. Admitted.

Example reduces :

```

<{ equal $4 $4 }> ->* 1.

```

Proof.

Admitted.

Example reduces2 :

```

<{ equal $4 $5 }> ->* 0.

```

Proof.

Admitted.

End FIXTEST3.

Module FIXTEST4.

Definition eotest :=

```

<{ let evenodd =
  fix
    (\eo: (Nat → Nat) × (Nat → Nat),
      (\n:Nat, if0 n then $1 else (eo.snd (pred n)),
        \n:Nat, if0 n then $0 else (eo.fst (pred n)))) in
  let even = evenodd.fst in
  let odd = evenodd.snd in
  (even $3, even $4) }>.

```

Example typechecks :

```

<{ empty |- eotest \in Nat × Nat }>.

```

Proof. unfold eotest. eauto 30. Admitted.

Example reduces :

```

eotest ->* <{ ($0, $1) }>.

```

Proof.

Admitted.

End FIXTEST4.

End EXAMPLES.

□

12.3.2 Properties of Typing

The proofs of progress and preservation for this enriched system are essentially the same (though of course longer) as for the pure STLC.

Progress

Exercise: 3 stars, standard (STLCExtended.progress) Complete the proof of `progress`.

Theorem: Suppose $\text{empty} \vdash t \text{ in } T$. Then either 1. t is a value, or 2. $t \rightarrow t'$ for some t' .

Proof: By induction on the given typing derivation. `Theorem progress :  $\forall t\ T,$`

`<{ empty  $\vdash t \text{ in } T$  }> \rightarrow`

`value  $t \vee \exists t', t \rightarrow t'$ .`

Proof with `eauto`.

```
intros t T Ht.
remember empty as Gamma.
generalize dependent HeqGamma.
induction Ht; intros HeqGamma; subst.
-
  discriminate H.
-
  left...
-
  right.
  destruct IHHT1; subst...
  +
    destruct IHHT2; subst...
    ×
    destruct H; try solve_by_invert.
     $\exists <\{ [x0 := t2] t1 \}> \dots$ 
    ×
    destruct H0 as [t2' Hstp].  $\exists <\{ t1\ t2' \}> \dots$ 
  +
    destruct H as [t1' Hstp].  $\exists <\{ t1'\ t2 \}> \dots$ 
-
  left...
-
  right.
  destruct IHHT...
  +
    destruct H; try solve_by_invert.
     $\exists <\{ \$(S\ n) \}> \dots$ 
```

```

+
  destruct H as [t' Hstp].
   $\exists$  <{succ t'}>...
-
right.
destruct IHHT...
+
  destruct H; try solve_by_invert.
   $\exists$  <{$(n - 1)$}>...
+
  destruct H as [t' Hstp].
   $\exists$  <{pred t'}>...
-
right.
destruct IHHT1...
+
  destruct IHHT2...
   $\times$ 
  destruct H; try solve_by_invert.
  destruct H0; try solve_by_invert.
   $\exists$  <{$(n \times n0)$}>...
   $\times$ 
  destruct H0 as [t2' Hstp].
   $\exists$  <{t1  $\times$  t2'}>...
+
  destruct H as [t1' Hstp].
   $\exists$  <{t1'  $\times$  t2}>...
-
right.
destruct IHHT1...
+
  destruct H; try solve_by_invert.
  destruct n as [|n'].
   $\times$ 
   $\exists$  t2...
   $\times$ 
   $\exists$  t3...
+
  destruct H as [t1' H0].
   $\exists$  <{if0 t1' then t2 else t3}>...
-
destruct IHHT...

```

```

+
  right. destruct H as [t1' Hstp]...
-
destruct IHHT...
+
  right. destruct H as [t1' Hstp]...
-
right.
destruct IHHT1...
+
  destruct H; try solve_by_invert.
  ×
  ∃ <{ [x1:=v] t1 }>...
  ×
  ∃ <{ [x2:=v] t2 }>...
+
  destruct H as [t0' Hstp].
  ∃ <{case t0' of | inl x1 ⇒ t1 | inr x2 ⇒ t2}>...
-
left...
-
destruct IHHT1...
+
  destruct IHHT2...
  ×
  right. destruct H0 as [t2' Hstp].
  ∃ <{t1 :: t2'}>...
+
  right. destruct H as [t1' Hstp].
  ∃ <{t1' :: t2}>...
-
right.
destruct IHHT1...
+
  destruct H; try solve_by_invert.
  ×
  ∃ t2...
  ×
  ∃ <{ [x2:=v2] ([x1:=v1] t3) }>...
+
  destruct H as [t1' Hstp].
  ∃ <{case t1' of | nil ⇒ t2 | x1 :: x2 ⇒ t3}>...

```

```

-
  left...

  Admitted.

  □

```

12.3.3 Weakening

The weakening claim and (automated) proof are exactly the same as for the original STLC. (We only need to increase the search depth of eauto to 7.)

```

Lemma weakening : ∀ Gamma Gamma' t T,
  includedin Gamma Gamma' →
  <{ Gamma |- t \in T }> →
  <{ Gamma' |- t \in T }>.

```

Proof.

```

  intros Gamma Gamma' t T H Ht.
  generalize dependent Gamma'.
  induction Ht; eauto 7 using includedin_update.

```

Qed.

```

Lemma weakening_empty : ∀ Gamma t T,
  <{ empty |- t \in T }> →
  <{ Gamma |- t \in T }>.

```

Proof.

```

  intros Gamma t T.
  eapply weakening.
  discriminate.

```

Qed.

Substitution

Exercise: 2 stars, standard (STLCExtended.substitution_preserves_typing) Complete the proof of substitution_preserves_typing.

```

Lemma substitution_preserves_typing : ∀ Gamma x U t v T,
  <{ x |-> U ; Gamma |- t \in T }> →
  <{ empty |- v \in U }> →
  <{ Gamma |- [x:=v]t \in T }>.

```

Proof with eauto.

```

  intros Gamma x U t v T Ht Hv.
  generalize dependent Gamma. generalize dependent T.
  induction t; intros T Gamma H;

```

```

    inversion H; clear H; subst; simpl; eauto.

```

```

-
  rename s into y. destruct (eqb_spec x y); subst.
+
  rewrite update_eq in H2.
  injection H2 as H2; subst.
  apply weakening_empty. assumption.
+
  apply T_Var. rewrite update_neq in H2; auto.
-
  rename s into y, t into S.
  destruct (eqb_spec x y); subst; apply T_Abs.
+
  rewrite update_shadow in H5. assumption.
+
  apply IHt.
  rewrite update_permute; auto.
-
  rename s into x1, s0 into x2.
  eapply T_Case...
+
  destruct (eqb_spec x x1); subst.
  ×
  rewrite update_shadow in H8. assumption.
  ×
  apply IHt2.
  rewrite update_permute; auto.
+
  destruct (eqb_spec x x2); subst.
  ×
  rewrite update_shadow in H9. assumption.
  ×
  apply IHt3.
  rewrite update_permute; auto.
-
  rename s into y1, s0 into y2.
  eapply T_Lcase...
  destruct (eqb_spec x y1); subst.
+
  destruct (eqb_spec y2 y1); subst.
  ×
  repeat rewrite update_shadow in H9.
  rewrite update_shadow.

```



```

    assumption.
  × rewrite update_permute in H9; [|assumption|.
    rewrite update_shadow in H9.
    rewrite update_permute; assumption.
+
destruct (eqb_spec x y2); subst.
×
  rewrite update_shadow in H9.
  assumption.
×
  apply IHt3.
  rewrite (update_permute _ _ _ _ _ n0) in H9.
  rewrite (update_permute _ _ _ _ _ n) in H9.
  assumption.

Admitted.
□

```

Preservation

Exercise: 3 stars, standard (STLCExtended.preservation) Complete the proof of preservation.

Theorem preservation : $\forall t\ t'\ T,$
 $\langle\{\text{empty} \mid -\ t \ \backslash\text{in } T\}\rangle \rightarrow$
 $t \rightarrow t' \rightarrow$
 $\langle\{\text{empty} \mid -\ t' \ \backslash\text{in } T\}\rangle.$

Proof with eauto.

```

intros t t' T HT. generalize dependent t'.
remember empty as Gamma.
induction HT;
  intros t' HE; subst; inversion HE; subst...
-
  inversion HE; subst...
+
  apply substitution_preserves_typing with T2...
  inversion HT1...
-
  inversion HT1; subst.
  eapply substitution_preserves_typing...
-
  inversion HT1; subst.
  eapply substitution_preserves_typing...
-

```

+
 inversion HT1; subst.
 apply substitution_preserves_typing with <{{List T1}}>...
 apply *substitution_preserves_typing* with *T1*...

Admitted.

□

End STLCEXTENDED.

Chapter 13

Library **PLF.Sub**

13.1 Sub: Subtyping

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import Strings.String.
From PLF Require Import Maps.
From PLF Require Import Types.
From PLF Require Import Smallstep.
Set Default Goal Selector "!".
```

13.2 Concepts

We now turn to *subtyping*, a key feature of – in particular – object-oriented programming languages.

13.2.1 A Motivating Example

Suppose we are writing a program involving two record types defined as follows:

Person = {name:String, age:Nat} Student = {name:String, age:Nat, gpa:Nat}

In the simply typed lambda-calculus with records, the term

(\r:Person, (r.age)+1) {name="Pat",age=21,gpa=1}

is not typable, since it applies a function that wants a two-field record to an argument that actually provides three fields, while the **T_App** rule demands that the domain type of the function being applied must match the type of the argument precisely.

But this is silly: we're passing the function a *better* argument than it needs! The only thing the body of the function can possibly do with its record argument **r** is project the field *age* from it: nothing else is allowed by the type, and the presence or absence of an extra *gpa* field makes no difference at all. So, intuitively, it seems that this function should be applicable to any record value that has at least an *age* field.

More generally, a record with more fields is “at least as good in any context” as one with just a subset of these fields, in the sense that any value belonging to the longer record type can be used *safely* in any context expecting the shorter record type. If the context expects something with the shorter type but we actually give it something with the longer type, nothing bad will happen (formally, the program will not get stuck).

The principle at work here is called *subtyping*. We say that “ S is a subtype of T ”, written $S <: T$, if a value of type S can safely be used in any context where a value of type T is expected. The idea of subtyping applies not only to records, but to all of the type constructors in the language – functions, pairs, etc.

Safe substitution principle:

- S is a subtype of T , written $S <: T$, if a value of type S can safely be used in any context where a value of type T is expected.

13.2.2 Subtyping and Object-Oriented Languages

Subtyping plays a fundamental role in many programming languages – in particular, it is central to the design of object-oriented languages and their libraries.

An *object* in Java, C#, etc. can be thought of as a record, some of whose fields are functions (“methods”) and some of whose fields are data values (“fields” or “instance variables”). Invoking a method m of an object o on some arguments $a_1..a_n$ roughly consists of projecting out the m field of o and applying it to $a_1..a_n$.

The type of an object is called a *class* – or, in some languages, an *interface*. It describes which methods and which data fields the object offers. Classes and interfaces are related by the *subclass* and *subinterface* relations. An object belonging to a subclass (or subinterface) is required to provide all the methods and fields of one belonging to a superclass (or superinterface), plus possibly some more.

The fact that an object from a subclass can be used in place of one from a superclass provides a degree of flexibility that is extremely handy for organizing complex libraries. For example, a GUI toolkit like Java’s Swing framework might define an abstract interface *Component* that collects together the common fields and methods of all objects having a graphical representation that can be displayed on the screen and interact with the user, such as the buttons, checkboxes, and scrollbars of a typical GUI. A method that relies only on this common interface can now be applied to any of these objects.

Of course, real object-oriented languages include many other features besides these. For example, fields can be updated. Fields and methods can be declared “private”. Classes can give *initializers* that are used when constructing objects. Code in subclasses can cooperate with code in superclasses via *inheritance*. Classes can have static methods and fields. Etc., etc.

To keep things simple here, we won’t deal with any of these issues – in fact, we won’t even talk any more about objects or classes. (There is a lot of discussion in *Pierce* 2002 (in Bib.v), if you are interested.) Instead, we’ll study the core concepts behind the subclass / subinterface relation in the simplified setting of the STLC.

13.2.3 The Subsumption Rule

Our goal for this chapter is to add subtyping to the simply typed lambda-calculus (with some of the basic extensions from [MoreStlc](#)). This involves two steps:

- Defining a binary *subtype relation* between types.
- Enriching the typing relation to take subtyping into account.

The second step is actually very simple. We add just a single rule to the typing relation: the so-called *rule of subsumption*:

$$\Gamma \vdash t_1 \text{ in } T_1 \quad T_1 <: T_2$$

(T-Sub) $\Gamma \vdash t_1 \text{ in } T_2$

This rule says, intuitively, that it is OK to “forget” some of what we know about a term.

For example, we may know that t_1 is a record with two fields (e.g., $T_1 = \{x:A \rightarrow A, y:B \rightarrow B\}$), but choose to forget about one of the fields ($T_2 = \{y:B \rightarrow B\}$) so that we can pass t_1 to a function that requires just a single-field record.

13.2.4 The Subtype Relation

The first step – the definition of the relation $S <: T$ – is where all the action is. Let’s look at each of the clauses of its definition.

Structural Rules

To start off, we impose two “structural rules” that are independent of any particular type constructor: a rule of *transitivity*, which says intuitively that, if S is better (richer, safer) than U and U is better than T , then S is better than T ...

$$S <: U \quad U <: T$$

(S-Trans) $S <: T$

... and a rule of *reflexivity*, since certainly any type T is as good as itself:

(S-Refl) $T <: T$

Products

Now we consider the individual type constructors, one by one, beginning with product types. We consider one pair to be a subtype of another if each of its components is.

$$S_1 <: T_1 \quad S_2 <: T_2$$

(S-Prod) $S_1 * S_2 <: T_1 * T_2$

Arrows

The subtyping rule for arrows is a little less intuitive. Suppose we have functions f and g with these types:

$$f : C \rightarrow \text{Student} \quad g : (C \rightarrow \text{Person}) \rightarrow D$$

That is, f is a function that yields a record of type `Student`, and g is a (higher-order) function that expects its argument to be a function yielding a record of type `Person`. Also suppose that `Student` is a subtype of `Person`. Then the application $g f$ is safe even though their types do not match up precisely, because the only thing g can do with f is to apply it to some argument (of type `C`); the result will actually be a `Student`, while g will be expecting a `Person`, but this is safe because the only thing g can then do is to project out the two fields that it knows about (*name* and *age*), and these will certainly be among the fields that are present.

This example suggests that the subtyping rule for arrow types should say that two arrow types are in the subtype relation if their results are:

$$S2 <: T2$$

$$(S\_Arrow\_Co) \quad S1 \rightarrow S2 <: S1 \rightarrow T2$$

We can generalize this to allow the arguments of the two arrow types to be in the subtype relation as well:

$$T1 <: S1 \quad S2 <: T2$$

$$(S\_Arrow) \quad S1 \rightarrow S2 <: T1 \rightarrow T2$$

But notice that the argument types are subtypes “the other way round”: in order to conclude that $S1 \rightarrow S2$ to be a subtype of $T1 \rightarrow T2$, it must be the case that $T1$ is a subtype of $S1$. The arrow constructor is said to be *contravariant* in its first argument and *covariant* in its second.

Here is an example that illustrates this:

$$f : \text{Person} \rightarrow C \quad g : (\text{Student} \rightarrow C) \rightarrow D$$

The application $g f$ is safe, because the only thing the body of g can do with f is to apply it to some argument of type `Student`. Since f requires records having (at least) the fields of a `Person`, this will always work. So $\text{Person} \rightarrow C$ is a subtype of $\text{Student} \rightarrow C$ since `Student` is a subtype of `Person`.

The intuition is that, if we have a function f of type $S1 \rightarrow S2$, then we know that f accepts elements of type $S1$; clearly, f will also accept elements of any subtype $T1$ of $S1$. The type of f also tells us that it returns elements of type $S2$; we can also view these results belonging to any supertype $T2$ of $S2$. That is, any function f of type $S1 \rightarrow S2$ can also be viewed as having type $T1 \rightarrow T2$.

Records

What about subtyping for record types?

The basic intuition is that it is always safe to use a “bigger” record in place of a “smaller” one. That is, given a record type, adding extra fields will always result in a subtype. If some code is expecting a record with fields x and y , it is perfectly safe for it to receive a record with fields x , y , and z ; the z field will simply be ignored. For example,

$\{\text{name:String, age:Nat, gpa:Nat}\} <: \{\text{name:String, age:Nat}\} \{\text{name:String, age:Nat}\} <: \{\text{name:String}\} \{\text{name:String}\} <: \{\}$

This is known as “width subtyping” for records.

We can also create a subtype of a record type by replacing the type of one of its fields with a subtype. If some code is expecting a record with a field x of type T , it will be happy with a record having a field x of type S as long as S is a subtype of T . For example,

$\{x:\text{Student}\} <: \{x:\text{Person}\}$

This is known as “depth subtyping”.

Finally, although the fields of a record type are written in a particular order, the order does not really matter. For example,

$\{\text{name:String, age:Nat}\} <: \{\text{age:Nat, name:String}\}$

This is known as “permutation subtyping”.

We *could* formalize these requirements in a single subtyping rule for records as follows:

forall jk in $j1..jn$, exists ip in $i1..im$, such that $jk=ip$ and $Sp <: Tk$

(S_Rcd) $\{i1:S1...im:Sm\} <: \{j1:T1...jn:Tn\}$

That is, the record on the left should have all the field labels of the one on the right (and possibly more), while the types of the common fields should be in the subtype relation.

However, this rule is rather heavy and hard to read, so it is often decomposed into three simpler rules, which can be combined using S_Trans to achieve all the same effects.

First, adding fields to the end of a record type gives a subtype:

$n > m$

(S_RcdWidth) $\{i1:T1...in:Tn\} <: \{i1:T1...im:Tm\}$

We can use $S\_RcdWidth$ to drop later fields of a multi-field record while keeping earlier fields, showing for example that $\{\text{age:Nat, name:String}\} <: \{\text{age:Nat}\}$.

Second, subtyping can be applied inside the components of a compound record type:

$S1 <: T1 \dots Sn <: Tn$

(S_RcdDepth) $\{i1:S1...in:Sn\} <: \{i1:T1...in:Tn\}$

For example, we can use $S\_RcdDepth$ and $S\_RcdWidth$ together to show that $\{y:\text{Student}, x:\text{Nat}\} <: \{y:\text{Person}\}$.

Third, subtyping can reorder fields. For example, we want $\{\text{name:String, gpa:Nat, age:Nat}\} <: \text{Person}$, but we haven’t quite achieved this yet: using just $S\_RcdDepth$ and $S\_RcdWidth$ we can only drop fields from the *end* of a record type. So we add:

$\{i1:S1...in:Sn\}$ is a permutation of $\{j1:T1...jn:Tn\}$

(S_RcdPerm) $\{i1:S1...in:Sn\} <: \{j1:T1...jn:Tn\}$

It is worth noting that full-blown language designs may choose not to adopt all of these subtyping rules. For example, in Java:

- Each class member (field or method) can be assigned a single index, adding new indices “on the right” as more members are added in subclasses (i.e., no permutation for classes).
- A class may implement multiple interfaces – so-called “multiple inheritance” of interfaces (i.e., permutation is allowed for interfaces).
- In early versions of Java, a subclass could not change the argument or result types of a method of its superclass (i.e., no depth subtyping or no arrow subtyping, depending how you look at it).

Exercise: 2 stars, standard, especially useful (arrow_sub_wrong) Suppose we had incorrectly defined subtyping as covariant on both the right and the left of arrow types:

$S1 <: T1 \quad S2 <: T2$

(S_Arrow_wrong) $S1 \rightarrow S2 <: T1 \rightarrow T2$

Give a concrete example of functions f and g with the following types...

$f : \text{Student} \rightarrow \text{Nat} \quad g : (\text{Person} \rightarrow \text{Nat}) \rightarrow \text{Nat}$

... such that the application $g \ f$ will get stuck during execution. (Use informal syntax. No need to prove formally that the application gets stuck.)

Definition `manual_grade_for_arrow_sub_wrong` : `option (nat × string)` := `None`.

□

Top

Finally, it is convenient to give the subtype relation a maximum element – a type that lies above every other type and is inhabited by all (well-typed) values. We do this by adding to the language one new type constant, called *Top*, together with a subtyping rule that places it above every other type in the subtype relation:

(S_Top) $S <: \text{Top}$

The *Top* type is an analog of the *Object* type in Java and C

Summary

In summary, we form the STLC with subtyping by starting with the pure STLC (over some set of base types) and then...

- adding a base type *Top*,

- $$\text{Gamma} \vdash t_1 \text{ in } T_1 \quad T_1 <: T_2$$

to the typing relation, and

- $$S \leq U \quad U \leq T$$

* — (S_Refl)

* — (S_Top)

S <: Top

— ———— (S_Prod) S1 * S2 <: T1 * T2

— (S_Arrow)

$$n > m$$

— (S_RcdWidth)

$$\{i_1:T_1 \dots i_n:T_n\} <: \{i_1:T_1 \dots i_m:T_m\}$$

— (S_RcdDepth)

$$\{i1:S1...in:Sn\} <: \{i1:T1...in:Tn\}$$

$$\{i1:S1...in:Sn\} \text{ is a permutation of } \{j1:T1...jn:Tn\}$$

$$- \text{—————} (\text{S\_RcdPerm}) \{i1:S1...in:Sn\} <: \{j1:T1...jn:Tn\}$$

13.2.5 Exercises

The following “thought exercises” are repeated later as formal exercises.

Exercise: 1 star, standard, optional (subtype_instances_tf_1) Suppose we have types S , T , U , and V with $S <: T$ and $U <: V$. Which of the following subtyping assertions are then true? Write *true* or *false* after each one. (A , B , and C here are base types like Bool , Nat , etc.)

- $T \rightarrow S <: T \rightarrow S$
- $\text{Top} \rightarrow U <: S \rightarrow \text{Top}$
- $(C \rightarrow C) \rightarrow (A \times B) <: (C \rightarrow C) \rightarrow (\text{Top} \times B)$
- $T \rightarrow T \rightarrow U <: S \rightarrow S \rightarrow V$
- $(T \rightarrow T) \rightarrow U <: (S \rightarrow S) \rightarrow V$
- $((T \rightarrow S) \rightarrow T) \rightarrow U <: ((S \rightarrow T) \rightarrow S) \rightarrow V$
- $S \times V <: T \times U$

□

Exercise: 2 stars, standard (subtype_order) The following types happen to form a linear order with respect to subtyping:

- Top
- $\text{Top} \rightarrow \text{Student}$
- $\text{Student} \rightarrow \text{Person}$
- $\text{Student} \rightarrow \text{Top}$
- $\text{Person} \rightarrow \text{Student}$

Write these types in order from the most specific to the most general.

Where does the type $\text{Top} \rightarrow \text{Top} \rightarrow \text{Student}$ fit into this order? That is, state how $\text{Top} \rightarrow (\text{Top} \rightarrow \text{Student})$ compares with each of the five types above. It may be unrelated to some of them.

Definition `manual_grade_for_subtype_order` : `option (nat × string)` := `None`.

□

Exercise: 1 star, standard (subtype_instances_tf_2) Which of the following statements are true? Write *true* or *false* after each one.

forall S T, S <: T -> S->S <: T->T
forall S, S <: A->A -> exists T, S = T->T /\ T <: A
forall S T1 T2, (S <: T1 -> T2) -> exists S1 S2, S = S1 -> S2 /\ T1 <: S1 /\ S2 <: T2
exists S, S <: S->S
exists S, S->S <: S
forall S T1 T2, S <: T1*T2 -> exists S1 S2, S = S1*S2 /\ S1 <: T1 /\ S2 <: T2

Definition manual_grade_for_subtype_instances_tf_2 : option (nat×string) := None.

□

Exercise: 1 star, standard (subtype_concepts_tf) Which of the following statements are true, and which are false?

- There exists a type that is a supertype of every other type.
- There exists a type that is a subtype of every other type.
- There exists a pair type that is a supertype of every other pair type.
- There exists a pair type that is a subtype of every other pair type.
- There exists an arrow type that is a supertype of every other arrow type.
- There exists an arrow type that is a subtype of every other arrow type.
- There is an infinite descending chain of distinct types in the subtype relation—that is, an infinite sequence of types *S*0, *S*1, etc., such that all the *S**i*'s are different and each *S*(*i*+1) is a subtype of *S**i*.
- There is an infinite *ascending* chain of distinct types in the subtype relation—that is, an infinite sequence of types *S*0, *S*1, etc., such that all the *S**i*'s are different and each *S*(*i*+1) is a supertype of *S**i*.

Definition manual_grade_for_subtype_concepts_tf : option (nat×string) := None.

□

Exercise: 2 stars, standard (proper_subtypes) Is the following statement true or false? Briefly explain your answer. (Here *Base* *n* stands for a base type, where *n* is a string standing for the name of the base type. See the Syntax section below.)

forall T, ~(T = Bool /\ exists n, T = Base n) -> exists S, S <: T /\ S <> T

Definition manual_grade_for_proper_subtypes : option (nat×string) := None.

□

Exercise: 2 stars, standard (small_large_1)

- What is the *smallest* type T (“smallest” in the subtype relation) that makes the following assertion true? (Assume we have *Unit* among the base types and *unit* as a constant of this type.)

$\text{empty} \vdash (\backslash p:T*\text{Top}, p.\text{fst}) ((\backslash z:A,z), \text{unit}) \backslash \text{in } A \rightarrow A$

- What is the *largest* type T that makes the same assertion true?

Definition `manual_grade_for_small_large_1` : `option (nat×string)` := `None`.

□

Exercise: 2 stars, standard (small_large_2)

- What is the *smallest* type T that makes the following assertion true?

$\text{empty} \vdash (\backslash p:(A \rightarrow A * B \rightarrow B), p) ((\backslash z:A,z), (\backslash z:B,z)) \backslash \text{in } T$

- What is the *largest* type T that makes the same assertion true?

Definition `manual_grade_for_small_large_2` : `option (nat×string)` := `None`.

□

Exercise: 2 stars, standard, optional (small_large_3)

- What is the *smallest* type T that makes the following assertion true?

$a:A \vdash (\backslash p:(A*T), (p.\text{snd}) (p.\text{fst})) (a, \backslash z:A,z) \backslash \text{in } A$

- What is the *largest* type T that makes the same assertion true?

□

Exercise: 2 stars, standard (small_large_4)

- What is the *smallest* type T (if one exists) that makes the following assertion true?

$\text{exists } S, \text{empty} \vdash (\backslash p:(A*T), (p.\text{snd}) (p.\text{fst})) \backslash \text{in } S$

- What is the *largest* type T that makes the same assertion true?

Definition `manual_grade_for_small_large_4` : `option (nat×string)` := `None`.

□

Exercise: 2 stars, standard (smallest_1) What is the *smallest* type T (if one exists) that makes the following assertion true?

$\text{exists } S \ t, \text{empty} \vdash (\backslash x:T, x \ x) \ t \backslash \text{in } S$

Definition `manual_grade_for_smallest_1` : `option (nat×string)` := `None`.

□

Exercise: 2 stars, standard (smallest_2) What is the *smallest* type T that makes the following assertion true?

$\text{empty} \vdash (\lambda x:\text{Top}, x) ((\lambda z:A, z), (\lambda z:B, z)) \text{ in } T$

Definition `manual_grade_for_smallest_2` : `option (nat×string)` := `None`.

□

Exercise: 3 stars, standard, optional (count_supertypes) How many supertypes does the record type $\{x:A, y:C \rightarrow C\}$ have? That is, how many different types T are there such that $\{x:A, y:C \rightarrow C\} <: T$? (We consider two types to be different if they are written differently, even if each is a subtype of the other. For example, $\{x:A, y:B\}$ and $\{y:B, x:A\}$ are different.)

□

Exercise: 2 stars, standard (pair_permutation) The subtyping rule for product types

$S1 <: T1 \quad S2 <: T2$

$(S\_Prod) \quad S1 * S2 <: T1 * T2$

intuitively corresponds to the “depth” subtyping rule for records. Extending the analogy, we might consider adding a “permutation” rule

$T1 * T2 <: T2 * T1$

for products. Is this a good idea? Briefly explain why or why not.

Definition `manual_grade_for_pair_permutation` : `option (nat×string)` := `None`.

□

13.3 Formal Definitions

Module `STLCSUB`.

Most of the definitions needed to formalize what we’ve discussed above – in particular, the syntax and operational semantics of the language – are identical to what we saw in the last chapter. We just need to extend the typing relation with the subsumption rule and add a new **Inductive** definition for the subtyping relation. Let’s first do the identical bits.

We include products in the syntax of types and terms, but not, for the moment, anywhere else; the *products* exercise below will ask you to extend the definitions of the value relation, operational semantics, subtyping relation, and typing relation and to extend the proofs of progress and preservation to fully support products.

13.3.1 Core Definitions

Syntax

In the rest of the chapter, we formalize just base types, booleans, arrow types, *Unit*, and *Top*, omitting record types and leaving product types as an exercise. For the sake of more interesting examples, we'll add an arbitrary set of base types like *String*, *Float*, etc. (Since they are just for examples, we won't bother adding any operations over these base types, but we could easily do so.)

```
Inductive ty : Type :=
```

```
| Ty_Top : ty
| Ty_Bool : ty
| Ty_Base : string → ty
| Ty_Arrow : ty → ty → ty
| Ty_Unit : ty
| Ty_Prod : ty → ty → ty
```

```
.
```

```
Inductive tm : Type :=
```

```
| tm_var : string → tm
| tm_app : tm → tm → tm
| tm_abs : string → ty → tm → tm
| tm_true : tm
| tm_false : tm
| tm_if : tm → tm → tm → tm
| tm_unit : tm
| tm_pair : tm → tm → tm
| tm_fst : tm → tm
| tm_snd : tm → tm
```

```
.
```

```
Declare Custom Entry stlc_ty.
```

```
Declare Custom Entry stlc_tm.
```

```
Declare Scope stlc_scope.
```

```
Open Scope stlc_scope.
```

```
Notation "x" := x (in custom stlc_ty at level 0, x global) : stlc_scope.
```

```
Notation "<{{ x }}>" := x (x custom stlc_ty).
```

```
Notation "( t )" := t (in custom stlc_ty at level 0, t custom stlc_ty) : stlc_scope.
```

```
Notation "S -> T" := (Ty_Arrow S T) (in custom stlc_ty at level 99, right associativity) : stlc_scope.
```

```
Notation "$ ( t )" := t (in custom stlc_ty at level 0, t constr) : stlc_scope.
```

```
Notation "$ ( x )" := x (in custom stlc_tm at level 0, x constr, only parsing) : stlc_scope.
```

```

Notation "x" := x (in custom stlc_tm at level 0, x constr at level 0) : stlc_scope.
Notation "<{ e }>" := e (e custom stlc_tm at level 200) : stlc_scope.
Notation "( x )" := x (in custom stlc_tm at level 0, x custom stlc_tm) : stlc_scope.
Notation "x y" := (tm_app x y) (in custom stlc_tm at level 10, left associativity)
: stlc_scope.
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc_tm at level 200, x global,
                  t custom stlc_ty,
                  y custom stlc_tm at level 200,
                  left associativity).
Coercion tm_var : string >-> tm.
Arguments tm_var _%string.
Notation "'Bool'" := Ty_Bool (in custom stlc_ty at level 0) : stlc_scope.
Notation "'if' x 'then' y 'else' z" :=
  (tm_if x y z) (in custom stlc_tm at level 200,
                 x custom stlc_tm,
                 y custom stlc_tm,
                 z custom stlc_tm at level 200,
                 left associativity).
Notation "'true'" := true (at level 1).
Notation "'true'" := tm_true (in custom stlc_tm at level 0).
Notation "'false'" := false (at level 1).
Notation "'false'" := tm_false (in custom stlc_tm at level 0).
Notation "'Base' x" := (Ty_Base x) (in custom stlc_ty at level 0, x constr at level
0) : stlc_scope.
Notation "'Unit'" :=
  (Ty_Unit) (in custom stlc_ty at level 0) : stlc_scope.
Notation "'unit'" := tm_unit (in custom stlc_tm at level 0) : stlc_scope.
Notation "X * Y" :=
  (Ty_Prod X Y) (in custom stlc_ty at level 2, X custom stlc_ty, Y custom stlc_ty
at level 0).
Notation "( x ', ' y )" := (tm_pair x y) (in custom stlc_tm at level 0,
                                           x custom stlc_tm,
                                           y custom stlc_tm) : stlc_scope.
Notation "t '.fst'" := (tm_fst t) (in custom stlc_tm at level 1) : stlc_scope.
Notation "t '.snd'" := (tm_snd t) (in custom stlc_tm at level 1) : stlc_scope.
Notation "'Top'" := (Ty_Top) (in custom stlc_ty at level 0) : stlc_scope.

```

Substitution

The definition of substitution remains exactly the same as for the pure STLC.

Reserved Notation "'[' x ' := ' s ']' t" (in custom stlc_tm at level 5, x global,
s custom stlc_tm,
t custom stlc_tm at next level, right associativity).

```
Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  | tm_var y =>
    if String.eqb x y then s else t
  | <{\y:T, t1}> =>
    if String.eqb x y then t else <{\y:T, [x:=s] t1}>
  | <{t1 t2}> =>
    <{([x:=s] t1) ([x:=s] t2)}>
  | <{true}> =>
    <{true}>
  | <{false}> =>
    <{false}>
  | <{if t1 then t2 else t3}> =>
    <{if ([x:=s] t1) then ([x:=s] t2) else ([x:=s] t3)}>
  | <{unit}> =>
    <{unit}>
  | <{(t1, t2)}> =>
    <{([x:=s] t1, [x:=s] t2)}>
  | <{t0.fst}> =>
    <{([x:=s] t0).fst}>
  | <{t0.snd}> =>
    <{([x:=s] t0).snd}>
  end
where "'[' x ' := ' s ']' t" := (subst x s t) (in custom stlc_tm) : stlc_scope.
```

Reduction

Likewise the definitions of **value** and **step**.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T2 t1,
    value <{\x:T2, t1}>
  | v_true :
    value <{true}>
  | v_false :
    value <{false}>
  | v_unit :
    value <{unit}>
```

Hint Constructors value : core.

Reserved Notation "t '→' t'" (at level 40).

```

Inductive step : tm → tm → Prop :=
| ST_AppAbs : ∀ x T2 t1 v2,
  value v2 →
  <{(\x:T2, t1) v2}> -> <{ [x:=v2] t1 }>
| ST_App1 : ∀ t1 t1' t2,
  t1 -> t1' →
  <{t1 t2}> -> <{t1' t2}>
| ST_App2 : ∀ v1 t2 t2',
  value v1 →
  t2 -> t2' →
  <{v1 t2}> -> <{v1 t2'}>
| ST_IfTrue : ∀ t1 t2,
  <{if true then t1 else t2}> -> t1
| ST_IfFalse : ∀ t1 t2,
  <{if false then t1 else t2}> -> t2
| ST_If : ∀ t1 t1' t2 t3,
  t1 -> t1' →
  <{if t1 then t2 else t3}> -> <{if t1' then t2 else t3}>
where "t '→' t'" := (step t t').

Hint Constructors step : core.

```

13.3.2 Subtyping

Now we come to the interesting part. We begin by defining the subtyping relation and developing some of its important technical properties.

The definition of subtyping is just what we sketched in the motivating discussion.

Reserved Notation "T '<:' U" (at level 40).

```

Inductive subtype : ty → ty → Prop :=
| S_Refl : ∀ T,
  T <: T
| S_Trans : ∀ S U T,
  S <: U →
  U <: T →
  S <: T
| S_Top : ∀ S,
  S <: <{{ Top }}>
| S_Arrow : ∀ S1 S2 T1 T2,
  T1 <: S1 →
  S2 <: T2 →
  <{{ S1→S2 }}> <: <{{ T1→T2 }}>
where "T '<:' U" := (subtype T U).

```

Note that we don't need any special rules for base types (`Bool` and `Base`): they are automatically subtypes of themselves (by `S_Refl`) and `Top` (by `S_Top`), and that's all we want.

Hint Constructors `subtype` : core.

Module `EXAMPLES`.

Open Scope `string_scope`.

Notation `x` := "x".

Notation `y` := "y".

Notation `z` := "z".

Notation `A` := <{{ Base "A" }}>.

Notation `B` := <{{ Base "B" }}>.

Notation `C` := <{{ Base "C" }}>.

Notation `String` := <{{ Base "String" }}>.

Notation `Float` := <{{ Base "Float" }}>.

Notation `Integer` := <{{ Base "Integer" }}>.

Example `subtyping_example_0` :

<{{ C → Bool }}> <: <{{ C → Top }}>.

Proof. auto. Qed.

Exercise: 2 stars, standard, optional (subtyping-judgements) (Leave this exercise *Admitted* until after you have finished adding product types to the language – see exercise *products* – at least up to this point in the file).

Recall that, in chapter `MoreStlc`, the optional section “Encoding Records” describes how records can be encoded as pairs. Using this encoding, define pair types representing the following record types:

`Person` := { name : String } `Student` := { name : String ; gpa : Float } `Employee` := { name : String ; ssn : Integer }

Definition `Person` : **ty**

. Admitted.

Definition `Student` : **ty**

. Admitted.

Definition `Employee` : **ty**

. Admitted.

Now use the definition of the subtype relation to prove the following:

Example `sub_student_person` :

Student <: *Person*.

Proof.

Admitted.

Example `sub_employee_person` :

Employee <: *Person*.

Proof.

Admitted.

□

The following facts are mostly easy to prove in Rocq. To get full benefit from the exercises, make sure you also understand how to prove them on paper!

Exercise: 1 star, standard, optional (subtyping_example_1) [Example subtyping_example_1](#)

:

$\langle\{\{ \text{Top} \rightarrow \text{Student} \}\}\rangle <: \langle\{\{ (C \rightarrow C) \rightarrow \text{Person} \}\}\rangle.$

Proof with eauto.

Admitted.

□

Exercise: 1 star, standard, optional (subtyping_example_2) [Example subtyping_example_2](#)

:

$\langle\{\{ \text{Top} \rightarrow \text{Person} \}\}\rangle <: \langle\{\{ \text{Person} \rightarrow \text{Top} \}\}\rangle.$

Proof with eauto.

Admitted.

□

End EXAMPLES.

13.3.3 Typing

The only change to the typing relation is the addition of the rule of subsumption, T_Sub .

Definition context := partial_map ty.

Notation "x '|->' v ';' m " := (update m x v)

(in custom stlc_tm at level 0, x constr at level 0, v custom stlc_ty, right associativity) : stlc_scope.

Notation "x '|->' v " := (update empty x v)

(in custom stlc_tm at level 0, x constr at level 0, v custom stlc_ty) : stlc_scope.

Notation "'empty'" := empty (in custom stlc_tm) : stlc_scope.

Reserved Notation "<{ Gamma '|- ' t '\in' T }>"

(at level 0, Gamma custom stlc_tm at level 200, t custom stlc_tm, T custom stlc_ty).

Inductive has_type : context → tm → ty → Prop :=

| T_Var : ∀ Gamma x T1,
 Gamma x = Some T1 →
 <{ Gamma |- x \in T1 }>
| T_Abs : ∀ Gamma x T1 T2 t1,

```

      <{ x |-> T2 ; Gamma |- t1 \in T1 }> →
      <{ Gamma |- \x:T2, t1 \in T2 → T1 }>
| T_App : ∀ T1 T2 Gamma t1 t2 ,
      <{ Gamma |- t1 \in T2 → T1 }> →
      <{ Gamma |- t2 \in T2 }> →
      <{ Gamma |- t1 t2 \in T1 }>
| T_True : ∀ Gamma ,
      <{ Gamma |- true \in Bool }>
| T_False : ∀ Gamma ,
      <{ Gamma |- false \in Bool }>
| T_If : ∀ t1 t2 t3 T1 Gamma ,
      <{ Gamma |- t1 \in Bool }> →
      <{ Gamma |- t2 \in T1 }> →
      <{ Gamma |- t3 \in T1 }> →
      <{ Gamma |- if t1 then t2 else t3 \in T1 }>
| T_Unit : ∀ Gamma ,
      <{ Gamma |- unit \in Unit }>

| T_Sub : ∀ Gamma t1 T1 T2 ,
      <{ Gamma |- t1 \in T1 }> →
      T1 <: T2 →
      <{ Gamma |- t1 \in T2 }>

```

where "<{ Gamma ' |- ' t '\in' T }>" := (has_type Gamma t T) : stlc_scope.

Hint Constructors has_type : core.

Module EXAMPLES2.

Import EXAMPLES.

Do the following exercises after you have added product types to the language. For each informal typing judgement, write it as a formal statement in Rocq and prove it.

Exercise: 1 star, standard, optional (typing_example_0)

Exercise: 2 stars, standard, optional (typing_example_1)

Exercise: 2 stars, standard, optional (typing_example_2) End EXAMPLES2.

13.4 Properties

The fundamental properties of the system that we want to check are the same as always: progress and preservation. Unlike the extension of the STLC with references (chapter Ref-

erences), we don't need to change the *statements* of these properties to take subtyping into account. However, their proofs do become a little bit more involved.

13.4.1 Inversion Lemmas for Subtyping

Before we look at the properties of the typing relation, we need to establish a couple of critical structural properties of the subtype relation:

- `Bool` is the only subtype of `Bool`, and
- every subtype of an arrow type is itself an arrow type.

These are called *inversion lemmas* because they play a similar role in proofs as the built-in `inversion` tactic: given a hypothesis that there exists a derivation of some subtyping statement $S <: T$ and some constraints on the shape of S and/or T , each inversion lemma reasons about what this derivation must look like to tell us something further about the shapes of S and T and the existence of subtype relations between their parts.

Exercise: 2 stars, standard, optional (sub_inversion_Bool) `Lemma sub_inversion_Bool :  $\forall U,$`

`$U <: <\{\{ \text{Bool} \}\}> \rightarrow$`
 `$U = <\{\{ \text{Bool} \}\}>.$`

Proof with auto.

```
intros U Hs.
remember <\{\{ Bool \}\}> as V.
Admitted.
□
```

Exercise: 3 stars, standard (sub_inversion_arrow) `Lemma sub_inversion_arrow :  $\forall$`
 `$U V1 V2,$`

`$U <: <\{\{ V1 \rightarrow V2 \}\}> \rightarrow$`
 `$\exists U1 U2,$`
 `$U = <\{\{ U1 \rightarrow U2 \}\}> \wedge V1 <: U1 \wedge U2 <: V2.$`

Proof with eauto.

```
intros U V1 V2 Hs.
remember <\{\{ V1 \rightarrow V2 \}\}> as V.
generalize dependent V2. generalize dependent V1.
Admitted.
□
```

There are additional *inversion lemmas* for the other types:

- `Unit` is the only subtype of `Unit`, and
- `Base n` is the only subtype of `Base n`, and
- `Top` is the only supertype of `Top`.

Exercise: 2 stars, standard, optional (sub_inversion_Unit) *Lemma sub_inversion_Unit*

```
: ∀ U,
  U <: <{{ Unit }}> →
  U = <{{ Unit }}>.
```

Proof with auto.

```
intros U Hs.
remember <{{ Unit }}> as V.
Admitted.
□
```

Exercise: 2 stars, standard, optional (sub_inversion_Base) *Lemma sub_inversion_Base*

```
: ∀ U s,
  U <: <{{ Base s }}> →
  U = <{{ Base s }}>.
```

Proof with auto.

```
intros U s Hs.
remember <{{ Base s }}> as V.
Admitted.
□
```

Exercise: 2 stars, standard, optional (sub_inversion_Top) *Lemma sub_inversion_Top*

```
: ∀ U,
  <{{ Top }}> <: U →
  U = <{{ Top }}>.
```

Proof with auto.

```
intros U Hs.
remember <{{ Top }}> as V.
Admitted.
□
```

13.4.2 Canonical Forms

The proof of the progress theorem – that a well-typed non-value can always take a step – doesn’t need to change too much: we just need one small refinement. When we’re considering the case where the term in question is an application $t1\ t2$ where both $t1$ and $t2$ are values, we need to know that $t1$ has the *form* of a lambda-abstraction, so that we can apply the *ST_AppAbs* reduction rule. In the ordinary STLC, this is obvious: we know that $t1$ has a function type $T11 \rightarrow T12$, and there is only one rule that can be used to give a function type to a value – rule *T_Abs* – and the form of the conclusion of this rule forces $t1$ to be an abstraction.

In the STLC with subtyping, this reasoning doesn’t quite work because there’s another rule that can be used to show that a value has a function type: subsumption. Fortunately,

this possibility doesn't change things much: if the last rule used to show $\Gamma \vdash t_1 \text{ in } T_{11} \rightarrow T_{12}$ is subsumption, then there is some *sub*-derivation whose subject is also t_1 , and we can reason by induction until we finally bottom out at a use of T_Abs .

This bit of reasoning is packaged up in the following lemma, which tells us the possible “canonical forms” (i.e., values) of function type.

Exercise: 3 stars, standard, optional (canonical_forms_of_arrow_types) *Lemma*

canonical_forms_of_arrow_types : $\forall \Gamma s T_1 T_2,$

$\langle \{ \Gamma \vdash s \text{ in } T_1 \rightarrow T_2 \} \rangle \rightarrow$

value $s \rightarrow$

$\exists x S_1 s_2,$

$s = \langle \{ \lambda x : S_1, s_2 \} \rangle.$

Proof with eauto.

Admitted.

□

Similarly, the canonical forms of type **Bool** are the constants **tm_true** and **tm_false**.

Lemma canonical_forms_of_Bool : $\forall \Gamma s,$

$\langle \{ \Gamma \vdash s \text{ in Bool } \} \rangle \rightarrow$

value $s \rightarrow$

$s = \text{tm\_true} \vee s = \text{tm\_false}.$

Proof with eauto.

intros Γs Hty Hv.

remember $\langle \{ \text{Bool} \} \rangle$ as T.

induction Hty; try solve_by_invert...

-

subst. apply *sub_inversion_Bool* in H. subst...

Qed.

13.4.3 Progress

The proof of progress now proceeds just like the one for the pure STLC, except that in several places we invoke canonical forms lemmas...

Theorem (Progress): For any term t and type T , if $\text{empty} \vdash t \text{ in } T$ then t is a value or $t \rightarrow t'$ for some term t' .

Proof: Let t and T be given, with $\text{empty} \vdash t \text{ in } T$. Proceed by induction on the typing derivation.

The cases for T_Abs , T_Unit , T_True and T_False are immediate because abstractions, *unit*, **true**, and **false** are already values. The T_Var case is vacuous because variables cannot be typed in the empty context. The remaining cases are more interesting:

- If the last step in the typing derivation uses rule T_App , then there are terms $t_1 t_2$ and types T_1 and T_2 such that $t = t_1 t_2$, $T = T_2$, $\text{empty} \vdash t_1 \text{ in } T_1 \rightarrow T_2$, and

$\text{empty} \vdash t2 \text{ \texttt{\textbackslash in } } T1$. Moreover, by the induction hypothesis, either $t1$ is a value or it steps, and either $t2$ is a value or it steps. There are three possibilities to consider:

- First, suppose $t1 \rightarrow t1'$ for some term $t1'$. Then $t1 \ t2 \rightarrow t1' \ t2$ by **ST_App1**.
 - Second, suppose $t1$ is a value and $t2 \rightarrow t2'$ for some term $t2'$. Then $t1 \ t2 \rightarrow t1 \ t2'$ by rule **ST_App2** because $t1$ is a value.
 - Third, suppose $t1$ and $t2$ are both values. By the canonical forms lemma for arrow types, we know that $t1$ has the form $\text{\texttt{\textbackslash x:S1,s2}}$ for some x , $S1$, and $s2$. But then $(\text{\texttt{\textbackslash x:S1,s2}}) \ t2 \rightarrow [x:=t2]s2$ by **ST_AppAbs**, since $t2$ is a value.
- If the final step of the derivation uses rule **T_If**, then there are terms $t1$, $t2$, and $t3$ such that $t = \text{if } t1 \text{ then } t2 \text{ else } t3$, with $\text{empty} \vdash t1 \text{ \texttt{\textbackslash in } } \text{Bool}$ and with $\text{empty} \vdash t2 \text{ \texttt{\textbackslash in } } T$ and $\text{empty} \vdash t3 \text{ \texttt{\textbackslash in } } T$. Moreover, by the induction hypothesis, either $t1$ is a value or it steps.
 - If $t1$ is a value, then by the canonical forms lemma for booleans, either $t1 = \text{true}$ or $t1 = \text{false}$. In either case, t can step, using rule **ST_IfTrue** or **ST_IfFalse**.
 - If $t1$ can step, then so can t , by rule **ST_If**.
 - If the final step of the derivation is by **T_Sub**, then there is a type $T2$ such that $T1 <: T2$ and $\text{empty} \vdash t1 \text{ \texttt{\textbackslash in } } T1$. The desired result is exactly the induction hypothesis for the typing subderivation.

Formally:

Theorem progress : $\forall t \ T,$
 $\langle \{ \text{empty} \vdash t \text{ \texttt{\textbackslash in } } T \} \rangle \rightarrow$
 $\text{value } t \vee \exists t', t \rightarrow t'.$

Proof with eauto.

```

intros t T Ht.
remember empty as Gamma.
induction Ht; subst Gamma; auto.
-
  discriminate.
-
  right.
  destruct IHHT1; subst...
+
  destruct IHHT2; subst...
×
  eapply canonical_forms_of_arrow_types in Ht1; [|assumption].
  destruct Ht1 as [x [S1 [s2 H1]]]. subst.
  ∃ (<{ [x:=t2]s2 }>)...

```



```

      ×
      destruct H0 as [t2' Hstp]. ∃ <{ t1 t2' }>...
+
      destruct H as [t1' Hstp]. ∃ <{ t1' t2 }>...
-
      right.
      destruct IHht1.
      + eauto.
      + apply canonical_forms_of_Bool in Ht1; [|assumption].
        destruct Ht1; subst...
      + destruct H. rename x into t1'. eauto.
Qed.

```

13.4.4 Inversion Lemmas for Typing

The proof of the preservation theorem also becomes a little more complex with the addition of subtyping. The reason is that, as with the “inversion lemmas for subtyping” above, there are a number of facts about the typing relation that are immediate from the definition in the pure STLC (formally: that can be obtained directly from the `inversion` tactic) but that require real proofs in the presence of subtyping because there are multiple ways to derive the same `has_type` statement.

The following inversion lemma tells us that, if we have a derivation of some typing statement $\Gamma \vdash \lambda x:S1, t2 \text{ in } T$ whose subject is an abstraction, then there must be some subderivation giving a type to the body $t2$.

Lemma: If $\Gamma \vdash \lambda x:S1, t2 \text{ in } T$, then there is a type $S2$ such that $x \vdash S1$; $\Gamma \vdash t2 \text{ in } S2$ and $S1 \rightarrow S2 <: T$.

Notice that the lemma does *not* say, “then T itself is an arrow type” – this is tempting, but false! (Why?)

Proof: Let Γ , x , $S1$, $t2$ and T be given as described. Proceed by induction on the derivation of $\Gamma \vdash \lambda x:S1, t2 \text{ in } T$. The cases for `T_Var` and `T_App` are vacuous as those rules cannot be used to give a type to a syntactic abstraction.

- If the last step of the derivation is a use of `T_Abs` then there is a type $T12$ such that $T = S1 \rightarrow T12$ and $x:S1$; $\Gamma \vdash t2 \text{ in } T12$. Picking $T12$ for $S2$ gives us what we need, since $S1 \rightarrow T12 <: S1 \rightarrow T12$ follows from `S_Refl`.
- If the last step of the derivation is a use of `T_Sub` then there is a type S such that $S <: T$ and $\Gamma \vdash \lambda x:S1, t2 \text{ in } S$. The IH for the typing subderivation tells us that there is some type $S2$ with $S1 \rightarrow S2 <: S$ and $x:S1$; $\Gamma \vdash t2 \text{ in } S2$. Picking type $S2$ gives us what we need, since $S1 \rightarrow S2 <: T$ then follows by `S_Trans`.

Formally:

`Lemma typing_inversion_abs : ∀ Γ x S1 t2 T ,`

```

<{ Gamma |- \x:S1,t2 \in T }> →
∃ S2,
  <{{ S1→S2 }}> <: T
  ∧ <{ x |-> S1 ; Gamma |- t2 \in S2 }>.

```

Proof with eauto.

```

intros Gamma x S1 t2 T H.
remember <\x:S1,t2> as t.
induction H;
  inversion Heqt; subst; intros; try solve_by_invert.
-
  ∃ T1...
-
  destruct IHhas_type as [S2 [Hsub Hty]]...
Qed.

```

Exercise: 3 stars, standard, optional (typing_inversion_var) Lemma typing_inversion_var

```

: ∀ Gamma (x:string) T,
  <{ Gamma |- x \in T }> →
  ∃ S,
    Gamma x = Some S ∧ S <: T.

```

Proof with eauto.

```

Admitted.
□

```

Exercise: 3 stars, standard, optional (typing_inversion_app) Lemma typing_inversion_app

```

: ∀ Gamma t1 t2 T2,
  <{ Gamma |- t1 t2 \in T2 }> →
  ∃ T1,
    <{ Gamma |- t1 \in T1→T2 }> ∧
    <{ Gamma |- t2 \in T1 }>.

```

Proof with eauto.

```

Admitted.
□

```

Lemma typing_inversion_unit : ∀ *Gamma* *T*,

```

  <{ Gamma |- unit \in T }> →
  <{{ Unit }}> <: T.

```

Proof with eauto.

```

intros Gamma T Htyp. remember <{ unit }> as tu.
induction Htyp;
  inversion Heqtu; subst; intros...

```

Qed.

The inversion lemmas for typing and for subtyping between arrow types can be packaged up as a useful “combination lemma” telling us exactly what we’ll actually require below.

```
Lemma abs_arrow : ∀ x S1 s2 T1 T2 ,
  <{ empty |- \x:S1,s2 \in T1→T2 }> →
  T1 <: S1
  ∧ <{ x |-> S1 |- s2 \in T2 }>.
```

Proof with eauto.

```
intros x S1 s2 T1 T2 Hty.
apply typing_inversion_abs in Hty.
destruct Hty as [S2 [Hsub Hty1]].
apply sub_inversion_arrow in Hsub.
destruct Hsub as [U1 [U2 [Heq [Hsub1 Hsub2]]]].
injection Heq as Heq; subst... Qed.
```

13.4.5 Weakening

The weakening lemma is proved as in pure STLC.

```
Lemma weakening : ∀ Gamma Gamma' t T ,
  includedin Gamma Gamma' →
  <{ Gamma |- t \in T }> →
  <{ Gamma' |- t \in T }>.
```

Proof.

```
intros Gamma Gamma' t T H Ht.
generalize dependent Gamma'.
induction Ht; eauto using includedin_update.
```

Qed.

```
Corollary weakening_empty : ∀ Gamma t T ,
  <{ empty |- t \in T }> →
  <{ Gamma |- t \in T }>.
```

Proof.

```
intros Gamma t T.
eapply weakening.
discriminate.
```

Qed.

13.4.6 Substitution

When subtyping is involved proofs are generally easier when done by induction on typing derivations, rather than on terms. The *substitution lemma* is proved as for pure STLC, but using induction on the typing derivation this time (see Exercise `substitution_preserves_typing_from_typing` in `StlcProp.v`).

```
Lemma substitution_preserves_typing : ∀ Gamma x U t v T ,
```

```

<{  $x \mapsto U$  ;  $\Gamma \vdash t \text{ in } T$  }>  $\rightarrow$ 
<{ empty  $\vdash v \text{ in } U$  }>  $\rightarrow$ 
<{  $\Gamma \vdash [x:=v]t \text{ in } T$  }>.

```

Proof.

```

intros  $\Gamma$   $x$   $U$   $t$   $v$   $T$   $Ht$   $Hv$ .
remember ( $x \mapsto U$ ;  $\Gamma$ ) as  $\Gamma'$ .
generalize dependent  $\Gamma$ .
induction  $Ht$ ; intros  $\Gamma'$   $G$ ; simpl; eauto.
Admitted.

```

13.4.7 Preservation

The proof of preservation now proceeds pretty much as in earlier chapters, using the substitution lemma at the appropriate point and the inversion lemma from above to extract structural information from typing assumptions.

Theorem (Preservation): If t , t' are terms and T is a type such that $\text{empty} \vdash t \text{ in } T$ and $t \rightarrow t'$, then $\text{empty} \vdash t' \text{ in } T$.

Proof: Let t and T be given such that $\text{empty} \vdash t \text{ in } T$. We proceed by induction on the structure of this typing derivation. The T_Abs , T_Unit , T_True , and T_False cases are vacuous because abstractions and constants don't step. Case T_Var is vacuous as well, since the context is empty.

- If the final step of the derivation is by T_App , then there are terms $t1$ and $t2$ and types $T1$ and $T2$ such that $t = t1\ t2$, $T = T2$, $\text{empty} \vdash t1 \text{ in } T1 \rightarrow T2$, and $\text{empty} \vdash t2 \text{ in } T1$.

By the definition of the step relation, there are three ways $t1\ t2$ can step. Cases ST_App1 and ST_App2 follow immediately by the induction hypotheses for the typing subderivations and a use of T_App .

Suppose instead $t1\ t2$ steps by ST_AppAbs . Then $t1 = \lambda x:S.t12$ for some type S and term $t12$, and $t' = [x:=t2]t12$.

By lemma `abs_arrow`, we have $T1 <: S$ and $x:S1 \vdash s2 \text{ in } T2$. It then follows by the substitution lemma (`substitution_preserves_typing`) that $\text{empty} \vdash [x:=t2]t12 \text{ in } T2$ as desired.

- If the final step of the derivation uses rule T_If , then there are terms $t1$, $t2$, and $t3$ such that $t = \text{if } t1 \text{ then } t2 \text{ else } t3$, with $\text{empty} \vdash t1 \text{ in } \text{Bool}$ and with $\text{empty} \vdash t2 \text{ in } T$ and $\text{empty} \vdash t3 \text{ in } T$. Moreover, by the induction hypothesis, if $t1$ steps to $t1'$ then $\text{empty} \vdash t1' : \text{Bool}$. There are three cases to consider, depending on which rule was used to show $t \rightarrow t'$.

- If $t \rightarrow t'$ by rule ST_If , then $t' = \text{if } t1' \text{ then } t2 \text{ else } t3$ with $t1 \rightarrow t1'$. By the induction hypothesis, $\text{empty} \vdash t1' \text{ in } \text{Bool}$, and so $\text{empty} \vdash t' \text{ in } T$ by T_If .

- If $t \rightarrow t'$ by rule `ST_IfTrue` or `ST_IfFalse`, then either $t' = t2$ or $t' = t3$, and $\text{empty} \vdash t' \text{ in } T$ follows by assumption.
- If the final step of the derivation is by `T_Sub`, then there is a type S such that $S <: T$ and $\text{empty} \vdash t \text{ in } S$. The result is immediate by the induction hypothesis for the typing subderivation and an application of `T_Sub`. $\square$

Theorem preservation : $\forall t t' T,$
 $\langle \{ \text{empty} \vdash t \text{ in } T \} \rangle \rightarrow$
 $t \rightarrow t' \rightarrow$
 $\langle \{ \text{empty} \vdash t' \text{ in } T \} \rangle.$

Proof with eauto.

```
intros t t' T HT. generalize dependent t'.
remember empty as Gamma.
induction HT;
  intros t' HE; subst;
  try solve [inversion HE; subst; eauto].
-
  inversion HE; subst...
+
  destruct (abs_arrow _ _ _ _ HT1) as [HA1 HA2].
  apply substitution_preserves_typing with T0...
```

Qed.

13.4.8 Records, via Products and Top

This formalization of the STLC with subtyping omits record types for brevity. If we want to deal with them more seriously, we have two choices.

First, we can treat them as part of the core language, writing down proper syntax, typing, and subtyping rules for them. Chapter `RECORDSUB` shows how this extension works.

On the other hand, if we are treating them as a derived form that is desugared in the parser, then we shouldn't need any new rules: we should just check that the existing rules for subtyping product and *Unit* types give rise to reasonable rules for record subtyping via this encoding. To do this, we just need to make one small change to the encoding described earlier: instead of using *Unit* as the base case in the encoding of tuples and the “don't care” placeholder in the encoding of records, we use *Top*. So:

$\{a:\text{Nat}, b:\text{Nat}\} \longrightarrow \{\text{Nat}, \text{Nat}\}$ i.e., $(\text{Nat}, (\text{Nat}, \text{Top}))$ $\{c:\text{Nat}, a:\text{Nat}\} \longrightarrow \{\text{Nat}, \text{Top}, \text{Nat}\}$ i.e., $(\text{Nat}, (\text{Top}, (\text{Nat}, \text{Top})))$

The encoding of record values doesn't change at all. It is easy (and instructive) to check that the subtyping rules above are validated by the encoding.

13.4.9 Exercises

Exercise: 2 stars, standard (variations) Each part of this problem suggests a different way of changing the definition of the STLC with Unit and subtyping. (These changes are not cumulative: each part starts from the original language.) In each part, list which properties (Progress, Preservation, both, or neither) become false. If a property becomes false, give a counterexample.

- Suppose we add the following typing rule:

$$\frac{}{\langle \{ \Gamma \vdash t \text{ in } S1 \rightarrow S2 \} \mid S1 <: T1 \mid T1 <: S1 \mid S2 <: T2 \rangle} \text{ (T\_Funny1) } \langle \{ \Gamma \vdash t \text{ in } T1 \rightarrow T2 \} \rangle$$

- Suppose we add the following reduction rule:

$$\frac{}{\text{unit} \rightarrow (\lambda x:\text{Top}. x)} \text{ (ST\_Funny2)}$$

- Suppose we add the following subtyping rule:

$$\frac{}{\text{Unit} <: \text{Top} \rightarrow \text{Top}} \text{ (S\_Funny3)}$$

- Suppose we add the following subtyping rule:

$$\frac{}{\text{Top} \rightarrow \text{Top} <: \text{Unit}} \text{ (S\_Funny4)}$$

- Suppose we add the following reduction rule:

$$\frac{}{(\text{unit } t) \rightarrow (t \text{ unit})} \text{ (ST\_Funny5)}$$

- Suppose we add the same reduction rule *and* a new typing rule:

$$\frac{}{(\text{unit } t) \rightarrow (t \text{ unit})} \text{ (ST\_Funny5)}$$

$$\frac{}{\text{empty} \vdash \text{unit} \text{ in } \text{Top} \rightarrow \text{Top}} \text{ (T\_Funny6)}$$

- Suppose we *change* the arrow subtyping rule to:

$$S1 <: T1 \quad S2 <: T2$$

$$- \text{—————} (S\_Arrow')$$

$$S1 \rightarrow S2 <: T1 \rightarrow T2$$

Definition manual_grade_for_variations : option (nat×string) := None.

□

13.5 Exercise: Adding Products

Exercise: 5 stars, standard (products) Adding pairs, projections, and product types to the system we have defined is a relatively straightforward matter. Carry out this extension by modifying the definitions and proofs above:

- Constructors for pairs, first and second projections, and product types have already been added to the definitions of **ty** and **tm**. Also, the definition of substitution has been extended.
- Extend the surrounding definitions accordingly (refer to chapter *MoreSTLC*):
 - value relation
 - operational semantics
 - typing relation

- Extend the subtyping relation with this rule:

$$S1 <: T1 \quad S2 <: T2$$

$$- \text{—————} (S\_Prod) \quad S1 * S2 <: T1 * T2$$

- Extend the proofs of progress, preservation, and all their supporting lemmas to deal with the new constructs. (You'll also need to add a couple of completely new lemmas.)

Definition manual_grade_for_products_value_step : option (nat×string) := None.

Definition manual_grade_for_products_subtype_has_type : option (nat×string) := None.

Definition manual_grade_for_products_progress : option (nat×string) := None.

Definition manual_grade_for_products_preservation : option (nat×string) := None.

□

13.5.1 Formalized “Thought Exercises”

The following are formal exercises based on the previous “thought exercises.”

Module FORMALTHOUGHTEXERCISES.

Import EXAMPLES.

Notation $p := \text{"p"}$.

Notation $a := \text{"a"}$.

Definition TF $P := P \vee \neg P$.

Exercise: 1 star, standard, optional (formal_subtype_instances_tf_1a) Theorem

formal_subtype_instances_tf_1a:

TF $(\forall S\ T\ U\ V, S <: T \rightarrow U <: V \rightarrow$
 $\langle\{\{ T \rightarrow S \}\}\rangle <: \langle\{\{ T \rightarrow S \}\}\rangle).$

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (formal_subtype_instances_tf_1b) Theorem

formal_subtype_instances_tf_1b:

TF $(\forall S\ T\ U\ V, S <: T \rightarrow U <: V \rightarrow$
 $\langle\{\{ \text{Top} \rightarrow U \}\}\rangle <: \langle\{\{ S \rightarrow \text{Top} \}\}\rangle).$

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (formal_subtype_instances_tf_1c) Theorem

formal_subtype_instances_tf_1c:

TF $(\forall S\ T\ U\ V, S <: T \rightarrow U <: V \rightarrow$
 $\langle\{\{ (C \rightarrow C) \rightarrow (A \times B) \}\}\rangle <: \langle\{\{ (C \rightarrow C) \rightarrow (\text{Top} \times B) \}\}\rangle).$

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (formal_subtype_instances_tf_1d) Theorem

formal_subtype_instances_tf_1d:

TF $(\forall S\ T\ U\ V, S <: T \rightarrow U <: V \rightarrow$
 $\langle\{\{ T \rightarrow (T \rightarrow U) \}\}\rangle <: \langle\{\{ S \rightarrow (S \rightarrow V) \}\}\rangle).$

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (formal_subtype_instances_tf_1e) Theorem

formal_subtype_instances_tf_1e:

$$\text{TF } (\forall S \ T \ U \ V, S <: T \rightarrow U <: V \rightarrow \\ <\{\{ (T \rightarrow T) \rightarrow U \}\} > <: <\{\{ (S \rightarrow S) \rightarrow V \}\} >).$$

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (formal_subtype_instances_tf_1f) Theorem

formal_subtype_instances_tf_1f:

$$\text{TF } (\forall S \ T \ U \ V, S <: T \rightarrow U <: V \rightarrow \\ <\{\{ ((T \rightarrow S) \rightarrow T) \rightarrow U \}\} > <: <\{\{ ((S \rightarrow T) \rightarrow S) \rightarrow V \}\} >).$$

Proof.

Admitted.

□

Exercise: 1 star, standard, optional (formal_subtype_instances_tf_1g) Theorem

formal_subtype_instances_tf_1g:

$$\text{TF } (\forall S \ T \ U \ V, S <: T \rightarrow U <: V \rightarrow \\ <\{\{ S \times V \}\} > <: <\{\{ T \times U \}\} >).$$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_instances_tf_2a) Theorem

formal_subtype_instances_tf_2a:

$$\text{TF } (\forall S \ T, \\ S <: T \rightarrow \\ <\{\{ S \rightarrow S \}\} > <: <\{\{ T \rightarrow T \}\} >).$$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_instances_tf_2b) Theorem

formal_subtype_instances_tf_2b:

$$\text{TF } (\forall S, \\ S <: <\{\{ A \rightarrow A \}\} > \rightarrow \\ \exists T, \\ S = <\{\{ T \rightarrow T \}\} > \wedge T <: A).$$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_instances_tf_2d) Hint: Assert a generalization of the statement to be proved and use induction on a type (rather than on a subtyping derivation). **Theorem** formal_subtype_instances_tf_2d:

TF $(\exists S,$
 $S <: \langle \{ \{ S \rightarrow S \} \} \rangle).$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_instances_tf_2e) **Theorem** formal_subtype_instances_tf_2e:

TF $(\exists S,$
 $\langle \{ \{ S \rightarrow S \} \} \rangle <: S).$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tfa) **Theorem** formal_subtype_concepts_tfa:

TF $(\exists T, \forall S, S <: T).$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tfb) **Theorem** formal_subtype_concepts_tfb:

TF $(\exists T, \forall S, T <: S).$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tfc) **Theorem** formal_subtype_concepts_tfc:

TF $(\exists T1\ T2, \forall S1\ S2, \langle \{ \{ S1 \times S2 \} \} \rangle <: \langle \{ \{ T1 \times T2 \} \} \rangle).$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tfd) **Theorem** formal_subtype_concepts_tfd:

TF $(\exists T1\ T2, \forall S1\ S2, \langle \{ \{ T1 \times T2 \} \} \rangle <: \langle \{ \{ S1 \times S2 \} \} \rangle).$

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tfe) *Theorem*
formal_subtype_concepts_tfe:

TF ($\exists T1\ T2, \forall S1\ S2, \langle\{\{ S1 \rightarrow S2 \}\}\rangle <: \langle\{\{ T1 \rightarrow T2 \}\}\rangle$).

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tff) *Theorem*
formal_subtype_concepts_tff:

TF ($\exists T1\ T2, \forall S1\ S2, \langle\{\{ T1 \rightarrow T2 \}\}\rangle <: \langle\{\{ S1 \rightarrow S2 \}\}\rangle$).

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tfg) *Theorem*
formal_subtype_concepts_tfg:

TF ($\exists f : \mathbf{nat} \rightarrow \mathbf{ty},$
 $(\forall i\ j, i \neq j \rightarrow f\ i \neq f\ j) \wedge$
 $(\forall i, f\ (S\ i) <: f\ i)$).

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (formal_subtype_concepts_tfh) *Theorem*
formal_subtype_concepts_tfh:

TF ($\exists f : \mathbf{nat} \rightarrow \mathbf{ty},$
 $(\forall i\ j, i \neq j \rightarrow f\ i \neq f\ j) \wedge$
 $(\forall i, f\ i <: f\ (S\ i))$).

Proof.

Admitted.

□

Exercise: 3 stars, standard, optional (formal_proper_subtypes) *Theorem* formal_proper_subtypes

TF ($\forall T,$
 $\sim(T = \langle\{\{ \mathbf{Bool} \}\}\rangle \vee (\exists n, T = \langle\{\{ \mathbf{Base}\ n \}\}\rangle) \vee T = \langle\{\{ \mathbf{Unit} \}\}\rangle) \rightarrow$
 $\exists S,$
 $S <: T \wedge S \neq T$).

Proof.

Admitted.

□

End FORMALTHOUGHTEXERCISES.

End STLCSUB.

Chapter 14

Library `PLF.Typechecking`

14.1 Typechecking: A Typechecker for STLC

The `has_type` relation of the STLC defines what it means for a term to belong to a type (in some context). But it doesn't, by itself, give us an algorithm for *checking* whether or not a term is well typed.

Fortunately, the rules defining `has_type` are *syntax directed* – that is, for every syntactic form of the language, there is just one rule that can be used to give a type to terms of that form. This makes it straightforward to translate the typing rules into clauses of a typechecking *function* that takes a term and a context and either returns the term's type or else signals that the term is not typable.

This short chapter constructs such a function and proves it correct.

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
Set Warnings "-non-recursive".
From Stdlib Require Import Bool.
From PLF Require Import Maps.
From PLF Require Import Smallstep.
From PLF Require Import Stlc.
From PLF Require Import MoreStlc.

Module STLC_TYPES.
Export STLC.
```

14.2 Comparing Types

First, we need a function to compare two types for equality...

```
Fixpoint eqb_ty (T1 T2:ty) : bool :=
  match T1, T2 with
  | <{{ Bool }}> , <{{ Bool }}> =>
    true
```

```

| <{{ T11→T12 }}>, <{{ T21→T22 }}> ⇒
    andb (eqb_ty T11 T21) (eqb_ty T12 T22)
| -, - ⇒
    false
end.

```

... and we need to establish the usual two-way connection between the boolean result returned by `eqb_ty` and the logical proposition that its inputs are equal.

```

Lemma eqb_ty_refl : ∀ T,
  eqb_ty T T = true.

```

Proof.

```

  intros T. induction T; simpl.
    reflexivity.
    rewrite IHT1. rewrite IHT2. reflexivity. Qed.

```

```

Lemma eqb_ty__eq : ∀ T1 T2,
  eqb_ty T1 T2 = true → T1 = T2.

```

Proof with auto.

```

  intros T1. induction T1; intros T2 Hbeq; destruct T2; inversion Hbeq.
  -
    reflexivity.
  -
    rewrite andb_true_iff in H0. inversion H0 as [Hbeq1 Hbeq2].
    apply IHT1_1 in Hbeq1. apply IHT1_2 in Hbeq2. subst... Qed.

```

End STLCTYPES.

14.3 The Typechecker

The typechecker works by walking over the structure of the given term, returning either `Some T` or `None`. Each time we make a recursive call to find out the types of the subterms, we need to pattern-match on the results to make sure that they are not `None`. Also, in the `app` case, we use pattern matching to extract the left- and right-hand sides of the function's arrow type (and fail if the type of the function is not $T11 \rightarrow T12$ for some $T11$ and $T12$).

```

Module FIRSTTRY.

```

```

Import STLCTYPES.

```

```

Fixpoint type_check (Gamma : context) (t : tm) : option ty :=
  match t with
  | tm_var x ⇒
      Gamma x
  | <{\x:T2, t1}> ⇒
      match type_check (x |-> T2 ; Gamma) t1 with
      | Some T1 ⇒ Some <{{ T2→T1 }}>
      | _ ⇒ None

```

```

    end
  | <{t1 t2}> =>
    match type_check Gamma t1, type_check Gamma t2 with
    | Some <{{ T11→T12 }}>, Some T2 =>
      if eqb_ty T11 T2 then Some T12 else None
    | _,- => None
    end
  | <{true}> =>
    Some <{{ Bool }}>
  | <{false}> =>
    Some <{{ Bool }}>
  | <{if guard then t else f}> =>
    match type_check Gamma guard with
    | Some <{{ Bool }}> =>
      match type_check Gamma t, type_check Gamma f with
      | Some T1, Some T2 =>
        if eqb_ty T1 T2 then Some T1 else None
      | _,- => None
      end
    | _ => None
    end
end.
End FIRSTTRY.

```

14.4 Digression: Improving the Notation

Before we consider the properties of this algorithm, let's write it out again in a cleaner way, using “monadic” notations in the style of Haskell to streamline the plumbing of options. First, we define a notation for composing two potentially failing (i.e., option-returning) computations:

```

Notation " x <- e1 ;; e2" := (match e1 with
                              | Some x => e2
                              | None => None
                              end)
(right associativity, at level 60).

```

Second, we define `return` and `fail` as synonyms for `Some` and `None`:

```

Notation " 'return' e "
:= (Some e) (at level 60).
Notation " 'fail' "
:= None.

```

```
Module STLCHECKER.
```

```
Import STLCTYPES.
```

Now we can write the same type-checking function in a more imperative-looking style using these notations.

```
Fixpoint type_check (Gamma : context) (t : tm) : option ty :=
  match t with
  | tm_var x =>
    match Gamma x with
    | Some T => return T
    | None => fail
    end
  | <{\x:T2, t1}> =>
    T1 ← type_check (x |-> T2 ; Gamma) t1 ;;
    return <{{ T2→T1 }}>
  | <{t1 t2}> =>
    T1 ← type_check Gamma t1 ;;
    T2 ← type_check Gamma t2 ;;
    match T1 with
    | <{{ T11→T12 }}> =>
      if eqb_ty T11 T2 then return T12 else fail
    | _ => fail
    end
  | <{ true }> =>
    return <{{ Bool }}>
  | <{ false }> =>
    return <{{ Bool }}>
  | <{ if guard then t1 else t2 }> =>
    Tguard ← type_check Gamma guard ;;
    T1 ← type_check Gamma t1 ;;
    T2 ← type_check Gamma t2 ;;
    match Tguard with
    | <{{ Bool }}> =>
      if eqb_ty T1 T2 then return T1 else fail
    | _ => fail
    end
  end.
end.
```

14.5 Properties

To verify that the typechecking algorithm is correct, we show that it is *sound* and *complete* for the original **has_type** relation – that is, **type_check** and **has_type** define the same partial

function.

Theorem type_checking_sound : $\forall$ *Gamma t T*,
type_check *Gamma t* = Some *T* $\rightarrow$ has_type *Gamma t T*.

Proof with eauto.

```
intros Gamma t. generalize dependent Gamma.
induction t; intros Gamma T Htc; inversion Htc.
- rename s into x. destruct (Gamma x) eqn:H.
  rename t into T'. inversion H0. subst. eauto. solve_by_invert.
-
  remember (type_check Gamma t1) as T01.
  destruct T01 as [T1|]; try solve_by_invert;
  destruct T1 as [|T11 T12]; try solve_by_invert;
  remember (type_check Gamma t2) as T02;
  destruct T02 as [T2|]; try solve_by_invert.
  destruct (eqb_ty T11 T2) eqn: Heqb.
  apply eqb_ty__eq in Heqb.
  inversion H0; subst...
  inversion H0.
-
  rename s into x, t into T1.
  remember (x |-> T1 ; Gamma) as G'.
  remember (type_check G' t0) as T02.
  destruct T02; try solve_by_invert.
  inversion H0; subst...
- eauto.
- eauto.
-
  remember (type_check Gamma t1) as T0c.
  remember (type_check Gamma t2) as T01.
  remember (type_check Gamma t3) as T02.
  destruct T0c as [Tc|]; try solve_by_invert.
  destruct Tc; try solve_by_invert;
  destruct T01 as [T1|]; try solve_by_invert;
  destruct T02 as [T2|]; try solve_by_invert.
  destruct (eqb_ty T1 T2) eqn:Heqb;
  try solve_by_invert.
  apply eqb_ty__eq in Heqb.
  inversion H0. subst. subst...
```

Qed.

Theorem type_checking_complete : $\forall$ *Gamma t T*,
has_type *Gamma t T* $\rightarrow$ type_check *Gamma t* = Some *T*.

Proof with auto.

```

intros Gamma t T Hty.
induction Hty; simpl.
- destruct (Gamma _) eqn:H0; assumption.
- rewrite IHHty...
-
  rewrite IHHty1. rewrite IHHty2.
  rewrite (eqb_ty_refl T2)...
- eauto.
- eauto.
- rewrite IHHty1. rewrite IHHty2.
  rewrite IHHty3. rewrite (eqb_ty_refl T1)...
Qed.
End STLCHECKER.

```

14.6 Exercises

In this exercise we'll extend the typechecker to deal with the extended features discussed in chapter [MoreStlc](#). Your job is to fill in the omitted cases in the following.

```

Module TYPECHECKEREXTENSIONS.
Import MORESTLC.
Import STLCEXTENDED.

Fixpoint eqb_ty (T1 T2 : ty) : bool :=
  match T1, T2 with
  | <{{Nat}}>, <{{Nat}}> =>
    true
  | <{{Unit}}>, <{{Unit}}> =>
    true
  | <{{T11 → T12}}>, <{{T21 → T22}}> =>
    andb (eqb_ty T11 T21) (eqb_ty T12 T22)
  | <{{T11 × T12}}>, <{{T21 × T22}}> =>
    andb (eqb_ty T11 T21) (eqb_ty T12 T22)
  | <{{T11 + T12}}>, <{{T21 + T22}}> =>
    andb (eqb_ty T11 T21) (eqb_ty T12 T22)
  | <{{List T11}}>, <{{List T21}}> =>
    eqb_ty T11 T21
  | -, - =>
    false
  end.

Lemma eqb_ty_refl : ∀ T,
  eqb_ty T T = true.
Proof.

```

```

intros T.
induction T; simpl; auto;
  rewrite IHT1; rewrite IHT2; reflexivity. Qed.
Lemma eqb_ty_eq :  $\forall T1\ T2$ ,
  eqb_ty  $T1\ T2$  = true  $\rightarrow T1 = T2$ .
Proof.
  intros T1.
  induction T1; intros T2 Hbeq; destruct T2; inversion Hbeq;
    try reflexivity;
    try (rewrite andb_true_iff in H0; inversion H0 as [Hbeq1 Hbeq2];
      apply IHT1_1 in Hbeq1; apply IHT1_2 in Hbeq2; subst; auto);
    try (apply IHT1 in Hbeq; subst; auto).
  Qed.

Exercise: 4 stars, standard (type_check_defn) Fixpoint type_check ( $\Gamma$  :
context) ( $t$  : tm) : option ty :=
  match  $t$  with
  | tm_var x  $\Rightarrow$ 
    match  $\Gamma$  x with
    | Some T  $\Rightarrow$  return T
    | None  $\Rightarrow$  fail
    end
  | <{ \ x1 : T1, t2 }>  $\Rightarrow$ 
     $T2 \leftarrow$  type_check (x1  $\mapsto$  T1 ;  $\Gamma$ ) t2 ;;
    return <{{T1  $\rightarrow$  T2}}>
  | <{ t1 t2 }>  $\Rightarrow$ 
     $T1 \leftarrow$  type_check  $\Gamma$  t1 ;;
     $T2 \leftarrow$  type_check  $\Gamma$  t2 ;;
    match  $T1$  with
    | <{{T11  $\rightarrow$  T12}}>  $\Rightarrow$ 
      if eqb_ty T11  $T2$  then return T12 else fail
    | _  $\Rightarrow$  fail
    end
  | tm_const _  $\Rightarrow$ 
    return <{{Nat}}>
  | <{ succ t1 }>  $\Rightarrow$ 
     $T1 \leftarrow$  type_check  $\Gamma$  t1 ;;
    match  $T1$  with
    | <{{Nat}}>  $\Rightarrow$  return <{{Nat}}>
    | _  $\Rightarrow$  fail
    end
  | <{ pred t1 }>  $\Rightarrow$ 

```

```

    T1 ← type_check Gamma t1 ;;
  match T1 with
  | <{{Nat}}> ⇒ return <{{Nat}}>
  | _ ⇒ fail
  end
| <{ t1 × t2 }> ⇒
  T1 ← type_check Gamma t1 ;;
  T2 ← type_check Gamma t2 ;;
  match T1, T2 with
  | <{{Nat}}>, <{{Nat}}> ⇒ return <{{Nat}}>
  | _, _ ⇒ fail
  end
| <{ if0 guard then t else f }> ⇒
  Tguard ← type_check Gamma guard ;;
  T1 ← type_check Gamma t ;;
  T2 ← type_check Gamma f ;;
  match Tguard with
  | <{{Nat}}> ⇒ if eqb_ty T1 T2 then return T1 else fail
  | _ ⇒ fail
  end

| <{ case t0 of | nil ⇒ t1 | x21 :: x22 ⇒ t2 }> ⇒
  T0 ← type_check Gamma t0 ;;
  match T0 with
  | <{{List T}}> ⇒
    T1 ← type_check Gamma t1 ;;
    T2 ← type_check (x21 |-> T ; x22 |-> <{{List T}}> ; Gamma) t2 ;;
    if eqb_ty T1 T2 then return T1 else fail
  | _ ⇒ fail
  end

```

```
| _ ⇒ None
end.
```

Definition manual_grade_for_type_check_defn : option (nat×string) := None.

□

Just for fun, we'll do the soundness proof with just a bit more automation than above, using these “mega-tactics”:

```
Ltac invert_typecheck Gamma t T :=
  remember (type_check Gamma t) as T0;
  destruct T0 as [T|];
  try solve_by_invert; try (inversion H0; eauto); try (subst; eauto).
```

```
Ltac analyze T T1 T2 :=
  destruct T as [T1 T2| |T1 T2|T1| |T1 T2]; try solve_by_invert.
```

```
Ltac fully_invert_typecheck Gamma t T T1 T2 :=
  let TX := fresh T in
  remember (type_check Gamma t) as T0;
  destruct T0 as [TX|]; try solve_by_invert;
  destruct TX as [T1 T2| |T1 T2|T1| |T1 T2];
  try solve_by_invert; try (inversion H0; eauto); try (subst; eauto).
```

```
Ltac case_equality S T :=
  destruct (eqb_ty S T) eqn: Heqb;
  inversion H0; apply eqb_ty__eq in Heqb; subst; subst; eauto.
```

Exercise: 2 stars, standard (ext_type_checking_sound) Theorem type_checking_sound
 : $\forall \text{Gamma } t \text{ } T$,
 type_check Gamma t = Some T $\rightarrow$
 has_type Gamma t T.

Proof with eauto.

```
intros Gamma t. generalize dependent Gamma.
induction t; intros Gamma T Htc; inversion Htc.
- rename s into x. destruct (Gamma x) eqn:H.
  rename t into T'. inversion H0. subst. eauto. solve_by_invert.
-
  invert_typecheck Gamma t1 T1.
  invert_typecheck Gamma t2 T2.
  analyze T1 T11 T12.
  case_equality T11 T2.
-
  rename s into x, t into T1.
  remember (x |-> T1 ; Gamma) as Gamma'.
```

```

    invert_typecheck Gamma' t0 T0.
- eauto.
-
    rename t into t1.
    fully_invert_typecheck Gamma t1 T1 T11 T12.
-
    rename t into t1.
    fully_invert_typecheck Gamma t1 T1 T11 T12.
-
    invert_typecheck Gamma t1 T1.
    invert_typecheck Gamma t2 T2.
    analyze T1 T11 T12; analyze T2 T21 T22.
    inversion H0. subst. eauto.
-
    invert_typecheck Gamma t1 T1.
    invert_typecheck Gamma t2 T2.
    invert_typecheck Gamma t3 T3.
    destruct T1; try solve_by_invert.
    case_equality T2 T3.
-
    rename s into x31, s0 into x32.
    fully_invert_typecheck Gamma t1 T1 T11 T12.
    invert_typecheck Gamma t2 T2.
    remember (x31 |-> T11 ; x32 |-> <{{List T11}}> ; Gamma) as Gamma'2.
    invert_typecheck Gamma'2 t3 T3.
    case_equality T2 T3.
Admitted.
□

```

Exercise: 2 stars, standard (ext_type_checking_complete) Theorem `type_checking_complete`

: $\forall \textit{Gamma} \textit{t} \textit{T}$,
has_type $\textit{Gamma} \textit{t} \textit{T} \rightarrow$
`type_check` $\textit{Gamma} \textit{t} = \text{Some } \textit{T}$.

Proof.

```

intros Gamma t T Hty.
induction Hty; simpl;
  try (rewrite IHHty);
  try (rewrite IHHty1);
  try (rewrite IHHty2);
  try (rewrite IHHty3);
  try (rewrite (eqb_ty_refl T0));
  try (rewrite (eqb_ty_refl T1));

```

```

    try (rewrite (eqb_ty_refl T2));
    try (rewrite (eqb_ty_refl T3));
    eauto.
    - destruct (Gamma _); [assumption| solve_by_invert].
  Admitted.
□

```

End TYPECHECKEREXTENSIONS.

Above, we showed how to write a typechecking function and prove it sound and complete for the typing relation. Do the same for the operational semantics – i.e., write a *function* `stepf` of type `tm → option tm` and prove that it is sound and complete with respect to `step` from chapter `MoreStlc`.

```

Module STEPFUNCTION.
Import MORESTLC.
Import STLCEXTENDED.

```

Exercise: 2 stars, standard, optional (valuef_defn) Fixpoint valuef (*t* : tm) : bool :=

```

  match t with
  | tm_var _ ⇒ false
  | <{ \ x : T, _ }> ⇒ true
  | <{ _ _ }> ⇒ false
  | tm_const _ ⇒ true
  | <{ succ _ }> | <{ pred _ }> | <{ _ × _ }> | <{ if0 _ then _ else _ }> ⇒ false

  | _ ⇒ false
  end.

```

Definition manual_grade_for_valuef_defn : option (nat×string) := None.

□

```

Definition assert (b : bool) (a : option tm) : option tm :=
  if b then a else None.

```

Exercise: 3 stars, standard, optional (stepf_defn) Fixpoint stepf (*t* : tm) : option tm :=

```

  match t with
  | tm_var x ⇒ None
  | <{ \x1:T1, t2 }> ⇒ None
  | <{ t1 t2 }> ⇒
    match stepf t1, stepf t2, t1 with

```

```

| Some t1', _, _ ⇒ Some <{ t1' t2 }>

| None, Some t2', _ ⇒ assert (valuef t1) (Some <{ t1 t2' }>)

| None, None, <{ \x:T, t11 }> ⇒
  assert (valuef t2) (Some <{ [x:=t2]t11 }>)
| _, _, _ ⇒ None
end

| tm_const _ ⇒ None
| <{ succ t1 }> ⇒
  match stepf t1, t1 with
  | Some t1', _ ⇒ Some <{ succ t1' }>

  | None, tm_const n ⇒ Some (tm_const (S n))
  | None, _ ⇒ None
  end

| <{ pred t1 }> ⇒
  match stepf t1, t1 with
  | Some t1', _ ⇒ Some <{ pred t1' }>

  | None, tm_const n ⇒ Some (tm_const (n - 1))
  | _, _ ⇒ None
  end

| <{ t1 × t2 }> ⇒
  match stepf t1, stepf t2, t1, t2 with
  | Some t1', _, _, _ ⇒ Some <{ t1' × t2 }>

  | None, Some t2', _, _ ⇒
    assert (valuef t1) (Some <{ t1 × t2' }>)
  | None, None, tm_const n1, tm_const n2 ⇒ Some (tm_const (mult n1 n2))
  | _, _, _, _ ⇒ None
  end

| <{ if0 guard then t else f }> ⇒
  match stepf guard, guard with
  | Some guard', _ ⇒ Some <{ if0 guard' then t else f }>

  | None, tm_const 0 ⇒ Some t
  | None, tm_const (S _) ⇒ Some f
  | _, _ ⇒ None
  end

```



```

| <{ case t0 of | nil ⇒ t1 | x21 :: x22 ⇒ t2 }> ⇒
  match stepf t0, t0 with
  | Some t0', _ ⇒ Some <{ case t0' of | nil ⇒ t1 | x21 :: x22 ⇒ t2 }>

  | None, <{ nil T }> ⇒ Some t1
  | None, <{ vh :: vt }> ⇒
    assert (valuef vh) (assert (valuef vt)
      (Some <{ [x22:=vt] ([x21:=vh] t2) }> ))
  | None, _ ⇒ None
end

```

```

| _ ⇒ None
end.

```

Definition manual_grade_for_stepf_defn : option (nat×string) := None.
□

Exercise: 2 stars, standard, optional (sound_valuef) Lemma sound_valuef : ∀ t,
valuef t = true → value t.

Proof.

Admitted.

□

Exercise: 2 stars, standard, optional (complete_valuef) Lemma complete_valuef : ∀ t,
value t → valuef t = true.

Proof.

Admitted.

□

Tactic Notation "auto_stepf" ident(H) :=

```

repeat
  match type of  $H$  with
  | (match ? $u$  with  $\_ \Rightarrow \_ \text{ end} = \_$ )  $\Rightarrow$ 
    let  $e := \text{fresh "e"}$  in
    destruct  $u \text{ eqn:} e$ 
  | (assert ? $u \_ = \_$ )  $\Rightarrow$ 

    let  $e := \text{fresh "e"}$  in
    destruct  $u \text{ eqn:} e$ ;
    simpl in  $H$ ;
    [apply sound_valuef in  $e$  | discriminate]
end;

(discriminate + (inversion  $H$ ; subst; auto)).

```

Exercise: 2 stars, standard, optional (sound_stepf) *Theorem* `sound_stepf` : $\forall t t',$
`stepf  $t = \text{Some } t' \rightarrow t \rightarrow t'$ .`

Proof.

```

intros  $t$ .
induction  $t$ ; simpl; intros  $t' H$ ;
  auto_stepf  $H$ .
Admitted.
□

```

Exercise: 2 stars, standard, optional (value_stepf_nf) *Lemma* `value_stepf_nf` : $\forall t,$
`value  $t \rightarrow \text{stepf } t = \text{None}$ .`

Proof.

```

Admitted.
□

```

Exercise: 2 stars, standard, optional (complete_stepf) *Theorem* `complete_stepf` : $\forall$
 $t t',$

`$t \rightarrow t' \rightarrow \text{stepf } t = \text{Some } t'$ .`

Proof.

```

Admitted.
□

```

End STEPFUNCTION.

Exercise: 5 stars, standard, optional (stlc_impl) Using the `Imp` parser described in the *ImpParser* chapter of *Logical Foundations* as a guide, build a parser for extended STLC programs. Combine it with the typechecking and stepping functions from the above exercises to yield a complete typechecker and interpreter for this language.

```
Module STLCIMPL.  
Import STEPFUNCTION.  
End STLCIMPL.  
□
```

Chapter 15

Library **PLF.Records**

15.1 Records: Adding Records to STLC

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import Strings.String.
From PLF Require Import Maps.
From PLF Require Import Smallstep.
From PLF Require Import Stlc.
```

15.2 Adding Records

We saw in chapter **MoreStlc** how records can be treated as just syntactic sugar for nested uses of products. This is OK for simple examples, but the encoding is informal (in reality, if we actually treated records this way, the translation would be carried out in the parser, which we are eliding here), and anyway it is not very efficient, nor does it lead to nice error messages for programs that the typechecker rejects. So it is also interesting to see how records can be treated as first-class citizens of the language. This chapter shows how.

Recall the informal definitions we gave before:

Syntax:

$t ::= \text{Terms: } | \{i=t, \dots, i=t\} \text{ record } | t.i \text{ projection } | \dots$

$v ::= \text{Values: } | \{i=v, \dots, i=v\} \text{ record value } | \dots$

$T ::= \text{Types: } | \{i:T, \dots, i:T\} \text{ record type } | \dots$

Reduction:

$tn \implies tn'$

(ST_Rcd) $\{i1=v1, \dots, im=vm, in=tn, \dots\} \implies \{i1=v1, \dots, im=vm, in=tn', \dots\}$
 $t1 \implies t1'$

(ST_Proj1) $t1.i \implies t1'.i$

(ST_ProjRcd) $\{..., i=vi, ...\}.i ==> vi$
 Typing:
 $\Gamma \vdash t1 : T1 \dots \Gamma \vdash tn : Tn$

(T_Rcd) $\Gamma \vdash \{i1=t1, ..., in=tn\} : \{i1:T1, ..., in:Tn\}$
 $\Gamma \vdash t0 : \{..., i:Ti, ...\}$

(T_Proj) $\Gamma \vdash t0.i : Ti$

15.3 Formalizing Records

Module `STLCExtendedRecords`.

Syntax and Operational Semantics

The most obvious way to formalize the syntax of record types would be this:

Module `FIRSTTRY`.

Definition `alist (X : Type) := list (string × X)`.

Inductive `ty : Type :=`
 | `Base : string → ty`
 | `Arrow : ty → ty → ty`
 | `TRcd : (alist ty) → ty`.

Unfortunately, we encounter here a limitation in Rocq: this type does not automatically give us the induction principle we expect: the induction hypothesis in the `TRcd` case doesn't give us any information about the `ty` elements of the list, making it useless for the proofs we want to do.

End `FIRSTTRY`.

It is possible to get a better induction principle out of Rocq, but the details of how this is done are not very pretty, and the principle we obtain is not as intuitive to use as the ones Rocq generates automatically for simple `Inductive` definitions.

Fortunately, there is a different way of formalizing records that is, in some ways, even simpler and more natural: instead of using the standard Rocq `list` type, we can essentially incorporate its constructors (“nil” and “cons”) in the syntax of our types.

Inductive `ty : Type :=`
 | `Ty_Base : string → ty`
 | `Ty_Arrow : ty → ty → ty`
 | `Ty_RNil : ty`
 | `Ty_RCons : string → ty → ty → ty`.

Similarly, at the level of terms, we have constructors `trnil`, for the empty record, and `rcons`, which adds a single field to the front of a list of fields.

```

Inductive tm : Type :=
| tm_var : string → tm
| tm_app : tm → tm → tm
| tm_abs : string → ty → tm → tm

| tm_rproj : tm → string → tm
| tm_rnil : tm
| tm_rcons : string → tm → tm → tm.

Notation "x" := x (in custom stlc_ty at level 0, x global) : stlc_scope.
Notation "<{ x }>" := x (x custom stlc_ty).
Notation "( t )" := t (in custom stlc_ty at level 0, t custom stlc_ty) : stlc_scope.
Notation "S -> T" := (Ty_Arrow S T) (in custom stlc_ty at level 99, right associativity)
: stlc_scope.
Notation "$ ( t )" := t (in custom stlc_ty at level 0, t constr) : stlc_scope.
Notation "$ ( x )" := x (in custom stlc_tm at level 0, x constr, only parsing) :
stlc_scope.
Notation "x" := x (in custom stlc_tm at level 0, x constr at level 0) : stlc_scope.
Notation "<{ e }>" := e (e custom stlc_tm at level 200) : stlc_scope.
Notation "( x )" := x (in custom stlc_tm at level 0, x custom stlc_tm) : stlc_scope.
Notation "x y" := (tm_app x y) (in custom stlc_tm at level 10, left associativity)
: stlc_scope.
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc_tm at level 200, x global,
                  t custom stlc_ty,
                  y custom stlc_tm at level 200,
                  left associativity).

Coercion tm_var : string >-> tm.
Arguments tm_var _%_string.

Notation "'Base' x" := (Ty_Base x) (in custom stlc_ty at level 0, x constr at level
0) : stlc_scope.

Notation "l ':' t1 '::' t2" := (Ty_RCons l t1 t2) (in custom stlc_ty at level
0, l global, t1 custom stlc_ty at level 0, right associativity).
Notation "l := e1 '::' e2" := (tm_rcons l e1 e2) (in custom stlc_tm at level 3,
right associativity).
Notation "'nil'" := (Ty_RNil) (in custom stlc_ty).
Notation "'nil'" := (tm_rnil) (in custom stlc_tm).
Notation "o -> l" := (tm_rproj o l) (in custom stlc_tm at level 0).

Some examples... Open Scope string_scope.
Open Scope stlc_scope.
Notation a := "a".

```

```

Notation f := "f".
Notation g := "g".
Notation l := "l".
Notation A := <{{ Base "A" }}>.
Notation B := <{{ Base "B" }}>.
Notation k := "k".
Notation il := "i1".
Notation i2 := "i2".

{ i1:A }

{ i1:A→B, i2:A }

```

Well-Formedness

Check <{{ nil → nil }}>.

One issue with generalizing the abstract syntax for records from lists to the nil/cons presentation is that it introduces the possibility of writing strange types like this...

Definition weird_type := <{{ a : A :: B }}>.

where the “tail” of a record type is not actually a record type!

We’ll structure our typing judgement so that no ill-formed types like `weird_type` are ever assigned to terms. To support this, we define predicates `record_ty` and `record_tm`, which identify record types and terms, and `well_formed_ty` which rules out the ill-formed types.

First, a type is a record type if it is built with just *RNil* and *RCons* at the outermost level.

```

Inductive record_ty : ty → Prop :=
| RTnil :
    record_ty <{{ nil }}>
| RTcons : ∀ i T1 T2,
    record_ty <{{ i : T1 :: T2 }}>.

```

With this, we can define well-formed types.

```

Inductive well_formed_ty : ty → Prop :=
| wfBase : ∀ (i : string),
    well_formed_ty <{{ Base i }}>
| wfArrow : ∀ T1 T2,
    well_formed_ty T1 →
    well_formed_ty T2 →
    well_formed_ty <{{ T1 → T2 }}>
| wfrNil :
    well_formed_ty <{{ nil }}>
| wfrCons : ∀ i T1 T2,
    well_formed_ty T1 →

```

```

well_formed_ty T2 →
record_ty T2 →
well_formed_ty <{{ i : T1 :: T2 }}>.

```

Hint Constructors `record_ty well_formed_ty` : core.

Note that `record_ty` is not recursive – it just checks the outermost constructor. The `well_formed_ty` property, on the other hand, verifies that the whole type is well formed in the sense that the tail of every record (the second argument to *RCons*) is a record.

Of course, we should also be concerned about ill-formed terms, not just types; but type-checking can rule those out without the help of an extra *well_formed_tm* definition because it already examines the structure of terms. All we need is an analog of `record_ty` saying that a term is a record term if it is built with *trnil* and *rcons*. Lemma `wf_ty_example`: `well_formed_ty <{{ i1 : (A → A) :: i2 : (B → B) :: nil }}>`.

Proof.

auto.

Qed.

```

Inductive record_tm : tm → Prop :=
| rtnil :
    record_tm <{ nil }>
| rtcons : ∀ i t1 t2,
    record_tm <{ i := t1 :: t2 }>.

```

Hint Constructors `record_tm` : core.

Substitution

Substitution extends easily.

Reserved Notation "'[x := s]' t" (in custom stlc_tm at level 5, x global, s custom stlc_tm, t custom stlc_tm at next level, right associativity).

```

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
match t with
| tm_var y ⇒
    if String.eqb x y then s else t
| <{\y:T, t1}> ⇒
    if String.eqb x y then t else <{\y:T, [x:=s] t1}>
| <{t1 t2}> ⇒
    <{([x:=s] t1) ([x:=s] t2)}>
| <{ t1 -> i }> ⇒
    <{ ( [x := s] t1 ) -> i }>
| <{ nil }> ⇒
    <{ nil }>
| <{ i := t1 :: tr }> ⇒

```



```

    <{ i := [x := s] t1 :: ( [x := s] tr) }>
end

```

where "'[' x ' := ' s ']' t" := (subst x s t) (in custom stlc_tm).

Reduction

A record is a value if all of its fields are.

```

Inductive value : tm → Prop :=
| v_abs : ∀ x T2 t1,
  value <{ \ x : T2, t1 }>
| v_rnil : value <{ nil }>
| v_rcons : ∀ i v1 vr,
  value v1 →
  value vr →
  value <{ i := v1 :: vr }>.

```

Hint Constructors value : core.

To define reduction, we'll need a utility function for extracting one field from record term:

```

Fixpoint tlookup (i:string) (tr:tm) : option tm :=
  match tr with
  | <{ i' := t :: tr' }> => if String.eqb i i' then Some t else tlookup i tr'
  | _ => None
end.

```

The **step** function uses this term-level lookup function in the projection rule.

Reserved Notation "t '->' t'" (at level 40).

```

Inductive step : tm → tm → Prop :=
| ST_AppAbs : ∀ x T2 t1 v2,
  value v2 →
  <{ (\x:T2, t1) v2 }> -> <{ [x:=v2] t1 }>
| ST_App1 : ∀ t1 t1' t2,
  t1 -> t1' →
  <{ t1 t2 }> -> <{ t1' t2 }>
| ST_App2 : ∀ v1 t2 t2',
  value v1 →
  t2 -> t2' →
  <{ v1 t2 }> -> <{ v1 t2' }>
| ST_Proj1 : ∀ t1 t1' i,
  t1 -> t1' →
  <{ t1 -> i }> -> <{ t1' -> i }>
| ST_ProjRcd : ∀ tr i vi,

```

```

    value tr →
    tlookup i tr = Some vi →
    <{ tr -> i }> -> vi
| ST_Rcd_Head : ∀ i t1 t1' tr2 ,
    t1 -> t1' →
    <{ i := t1 :: tr2 }> -> <{ i := t1' :: tr2 }>
| ST_Rcd_Tail : ∀ i v1 tr2 tr2' ,
    value v1 →
    tr2 -> tr2' →
    <{ i := v1 :: tr2 }> -> <{ i := v1 :: tr2' }>

```

where "*t* '→' *t*'" := (step *t t*').

Notation multistep := (multi step).

Notation "*t1* '→*' *t2*" := (multistep *t1 t2*) (at level 40).

Hint Constructors step : core.

Typing

Next we define the typing rules. These are nearly direct transcriptions of the inference rules shown above: the only significant difference is the use of **well_formed_ty**. In the informal presentation we used a grammar that only allowed well-formed record types, so we didn't have to add a separate check.

One sanity condition that we'd like to maintain is that, whenever **has_type** *Gamma t T* holds, will also be the case that **well_formed_ty** *T*, so that **has_type** never assigns ill-formed types to terms. In fact, we prove this theorem below. However, we don't want to clutter the definition of **has_type** with unnecessary uses of **well_formed_ty**. Instead, we place **well_formed_ty** checks only where needed: where an inductive call to **has_type** won't already be checking the well-formedness of a type. For example, we check **well_formed_ty** *T* in the **T_Var** case, because there is no inductive **has_type** call that would enforce this. Similarly, in the **T_Abs** case, we require a proof of **well_formed_ty** *T11* because the inductive call to **has_type** only guarantees that *T12* is well-formed.

```

Fixpoint Tlookup (i:string) (Tr:ty) : option ty :=
  match Tr with
  | <{{ i' : T :: Tr' }}> =>
    if String.eqb i i' then Some T else Tlookup i Tr'
  | _ => None
end.

```

Definition context := partial_map ty.

Notation "*x* '|->' *v* ';' *m* " := (update *m x v*)

(in custom stlc_tm at level 0, *x* constr at level 0, *v* custom stlc_ty, right associativity) : stlc_scope.

```

Notation "x '|->' v " := (update empty x v)
  (in custom stlc_tm at level 0, x constr at level 0, v custom stlc_ty) : stlc_scope.
Notation "'empty'" := empty (in custom stlc_tm) : stlc_scope.
Reserved Notation "<{ Gamma '|- ' t '\in' T }>"
  (at level 0, Gamma custom stlc_tm at level 200, t custom stlc_tm, T
  custom stlc_ty).
Inductive has_type (Gamma : context) : tm → ty → Prop :=
| T_Var : ∀ x T,
  Gamma x = Some T →
  well_formed_ty T →
  <{ Gamma |- x \in T }>
| T_Abs : ∀ x T11 T12 t12,
  well_formed_ty T11 →
  <{ x |-> T11; Gamma |- t12 \in T12 }> →
  <{ Gamma |- \x : T11, t12 \in T11 → T12 }>
| T_App : ∀ T1 T2 t1 t2,
  <{ Gamma |- t1 \in T1 → T2 }> →
  <{ Gamma |- t2 \in T1 }> →
  <{ Gamma |- ( t1 t2 ) \in T2 }>

| T_Proj : ∀ i t Ti Tr,
  <{ Gamma |- t \in Tr }> →
  Tlookup i Tr = Some Ti →
  <{ Gamma |- ( t -> i ) \in Ti }>
| T_RNil :
  <{ Gamma |- nil \in nil }>
| T_RCons : ∀ i t T tr Tr,
  <{ Gamma |- t \in T }> →
  <{ Gamma |- tr \in Tr }> →
  record_ty Tr →
  record_tm tr →
  <{ Gamma |- ( i := t :: tr ) \in i : T :: Tr }>

where "<{ Gamma '|- ' t '\in' T }>" := (has_type Gamma t T).
Hint Constructors has_type : core.

```

15.3.1 Examples

Exercise: 2 stars, standard (examples) Finish the proofs below. Feel free to use Rocq's automation features in this proof. However, if you are not confident about how the type system works, you may want to carry out the proofs first using the basic features (`apply` instead of `eapply`, in particular) and then perhaps compress it using automation.

Before starting to prove *anything*, make sure you understand what it is saying. :-)

Lemma typing_example_2 :

```
<{ empty |- (\a : ( i1 : (A → A) :: i2 : (B → B) :: nil), a -> i2)
  ( i1 := (\a : A, a) :: i2 := (\a : B, a ) :: nil ) \in B → B }>.
```

Proof.

Admitted.

Example typing_nonexample :

```
¬ ∃ T,
  <{ (a |-> <{{ i2 : (A → A) :: nil }}>) |-
    ( i1 := (\a : B, a) :: a ) \in
      T }>.
```

Proof.

Admitted.

Example typing_nonexample_2 : ∀ y,

```
¬ ∃ T,
  <{ (y |-> A) |-
    (\a : ( i1 : A :: nil ), a -> i1 )
    ( i1 := y :: i2 := y :: nil ) \in T }>.
```

Proof.

Admitted.

□

15.3.2 Properties of Typing

The proofs of progress and preservation for this system are essentially the same as for the pure simply typed lambda-calculus, but we need to add some technical lemmas involving records.

Well-Formedness

Lemma wf_rcd_lookup : ∀ i T Ti,

```
well_formed_ty T →
  Tlookup i T = Some Ti →
  well_formed_ty Ti.
```

Proof with eauto.

```
intros i T.
induction T; intros; try solve_by_invert.
-
  inversion H. subst. unfold Tlookup in H0.
  destruct (String.eqb i s)...
  inversion H0. subst... Qed.
```

Lemma step_preserves_record_tm : ∀ tr tr',

```

record_tm tr →
tr -> tr' →
record_tm tr'.

```

Proof.

```

intros tr tr' Hrt Hstp.
inversion Hrt; subst; inversion Hstp; subst; auto.

```

Qed.

```

Lemma has_type_wf : ∀ Gamma t T,
  <{ Gamma |- t \in T }> → well_formed_ty T.

```

Proof with eauto.

```

intros Gamma t T Htyp.
induction Htyp...
-
  inversion IHHtyp1...
-
  eapply wf_rcd_lookup...

```

Qed.

Field Lookup

Lemma: If $\text{empty} \vdash v : T$ and $\text{Tlookup } i \ T$ returns $\text{Some } Ti$, then $\text{tlookup } i \ v$ returns $\text{Some } ti$ for some term ti such that $\text{empty} \vdash ti \in Ti$.

Proof: By induction on the typing derivation $Htyp$. Since $\text{Tlookup } i \ T = \text{Some } Ti$, T must be a record type, this and the fact that v is a value eliminate most cases by inspection, leaving only the T_RCons case.

If the last step in the typing derivation is by T_RCons , then $t = rcons \ i0 \ t \ tr$ and $T = RCons \ i0 \ T \ Tr$ for some $i0$, t , tr , T and Tr .

This leaves two possibilities to consider - either $i0 = i$ or not.

- If $i = i0$, then since $\text{Tlookup } i \ (RCons \ i0 \ T \ Tr) = \text{Some } Ti$ we have $T = Ti$. It follows that t itself satisfies the theorem.
- On the other hand, suppose $i \neq i0$. Then
 $\text{Tlookup } i \ T = \text{Tlookup } i \ Tr$
and
 $\text{tlookup } i \ t = \text{tlookup } i \ tr$,
so the result follows from the induction hypothesis. $\square$

Here is the formal statement:

```

Lemma lookup_field_in_value : ∀ v T i Ti,
  value v →
  <{ empty |- v \in T }> →

```

```

Tlookup i T = Some Ti →
  ∃ ti, tlookup i v = Some ti ∧ <{ empty |- ti \in Ti }>.
Proof with eauto.
  intros v T i Ti Hval Htyp Hget.
  remember empty as Gamma.
  induction Htyp; subst; try solve_by_invert...
-
  simpl in Hget. simpl. destruct (String.eqb i i0).
  +
    simpl. injection Hget as Hget. subst.
    ∃ t...
  +
    destruct IHHtyp2 as [vi [Hgeti Htypi] ]...
    inversion Hval... Qed.

```

Progress

```

Theorem progress : ∀ t T,
  <{ empty |- t \in T }> →
  value t ∨ ∃ t', t -> t'.

```

```

Proof with eauto.
  intros t T Ht.
  remember empty as Gamma.
  generalize dependent HeqGamma.
  induction Ht; intros HeqGamma; subst.
-
  inversion H.
-
  left...
-
  right.
  destruct IHht1; subst...
  +
    destruct IHht2; subst...
    ×

    inversion H; subst; try solve_by_invert.
    ∃ <{ [x:=t2]t0 }>...
    ×

```

```

      destruct H0 as [t2' Hstp].  $\exists$  <{ t1 t2' }>...
+
      destruct H as [t1' Hstp].  $\exists$  <{ t1' t2 }>...
-
right. destruct IHHt...
+
      destruct (lookup_field_in_value _ _ _ H0 Ht H)
      as [ti [Hlkup _] ].
       $\exists$  ti...
+
      destruct H0 as [t' Hstp].  $\exists$  <{ t' -> i }>...
-
left...
-
destruct IHHt1...
+
      destruct IHHt2; try reflexivity.
       $\times$ 

      left...
       $\times$ 

      right. destruct H2 as [tr' Hstp].
       $\exists$  <{ i := t :: tr' }>...
+
      right. destruct H1 as [t' Hstp].
       $\exists$  <{ i := t' :: tr }>... Qed.

```

15.3.3 Weakening

The weakening lemma is proved as in pure STLC.

Lemma weakening : $\forall$ *Gamma Gamma' t T*,
 includedin *Gamma Gamma'* $\rightarrow$
 <{ *Gamma* |- *t* \in *T* }> $\rightarrow$
 <{ *Gamma'* |- *t* \in *T* }>.

```

Proof.
  intros Gamma Gamma' t T H Ht.
  generalize dependent Gamma'.
  induction Ht; eauto using includedin_update.
Qed.

```

```

Lemma weakening_empty :  $\forall$  Gamma t T,
  <{ empty |- t \in T }>  $\rightarrow$ 
  <{ Gamma |- t \in T }>.

```

```

Proof.
  intros Gamma t T.
  eapply weakening.
  discriminate.
Qed.

```

Preservation

As before, we prove the substitution lemma by induction on the term t . The only new case (compared to the proof in `StlcProp.v`) is the case of `rcons`. For this case, we must do a little extra work to show that substituting into a term doesn't change whether it is a record term.

```

Lemma substitution_preserves_typing :  $\forall$  Gamma x U t v T,
  <{ x |-> U ; Gamma |- t \in T }>  $\rightarrow$ 
  <{ empty |- v \in U }>  $\rightarrow$ 
  <{ Gamma |- [x:=v]t \in T }>.

```

```

Proof.
  intros Gamma x U t v T Ht Hv.
  generalize dependent Gamma. generalize dependent T.
  induction t; intros T Gamma H;

  inversion H; clear H; subst; simpl; eauto.
-
  rename s into y. destruct (eqb_spec x y); subst.
  +
    rewrite update_eq in H1.
    injection H1 as H1; subst.
    apply weakening_empty. assumption.
  +
    apply T_Var. rewrite update_neq in H1; auto. assumption.
-
  rename s into y, t into T.
  destruct (eqb_spec x y); subst; apply T_Abs; try assumption.
  +
    rewrite update_shadow in H5. assumption.

```



```

+
  apply IHt.
  rewrite update_permute; auto.
-
  apply T_RCons; eauto.
  inversion H7; subst; simpl; auto.
Qed.

Theorem preservation :  $\forall t t' T,$ 
  <{ empty |-  $t \setminus \text{in } T$  }>  $\rightarrow$ 
   $t \rightarrow t' \rightarrow$ 
  <{ empty |-  $t' \setminus \text{in } T$  }>.
Proof with eauto.
  intros  $t t' T HT$ . generalize dependent  $t'$ .
  remember empty as Gamma.
  induction HT;
    intros  $t' HE$ ; subst;
    try solve [inversion HE; subst; auto].
-
  inversion HE; subst...
+
  apply substitution_preserves_typing with T1...
  inversion HT1...
-

  inversion HE; subst...
  destruct (lookup_field_in_value _ _ _ H2 HT H)
    as [vi [Hget Htyp] ].
  rewrite H4 in Hget. injection Hget as Hget. subst...
-

  inversion HE; subst...
  apply T_RCons... eapply step_preserves_record_tm...
Qed.

End STLCEXTENDEDRECORDS.

```

Chapter 16

Library `PLF.References`

16.1 References: Typing Mutable References

Up to this point, we have considered a variety of *pure* language features, including functional abstraction, basic types such as numbers and booleans, and structured types such as records and variants. These features form the backbone of most programming languages – including purely functional languages such as Haskell and “mostly functional” languages such as ML, as well as imperative languages such as C and object-oriented languages such as Java, C#, and Scala.

Most practical languages also include various *impure* features that cannot be described in the simple semantic framework we have used so far. In particular, besides just yielding results, computation in these languages may assign to mutable variables (reference cells, arrays, mutable record fields, etc.); perform input and output to files, displays, or network connections; make non-local transfers of control via exceptions, jumps, or continuations; engage in inter-process synchronization and communication; and so on. In the literature on programming languages, such “side effects” of computation are collectively referred to as *computational effects*.

In this chapter, we’ll see how one sort of computational effect – mutable references – can be added to the calculi we have studied. The main extension will be dealing explicitly with a *store* (or *heap*) and *pointers* that name store locations. This extension is fairly straightforward to define; the most interesting part is the refinement we need to make to the statement of the type preservation theorem.

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
```

```
Set Warnings "-deprecated-syntactic-definition".
```

```
From Stdlib Require Import Strings.String.
```

```
From Stdlib Require Import Init.Nat.
```

```
From Stdlib Require Import Arith.
```

```
From Stdlib Require Import PeanoNat.
```

```
From Stdlib Require Import Lia.
```

```
From PLF Require Import Maps.
```

```

From PLF Require Import Smallstep.
From Stdlib Require Import List. Import DATATYPES.
Check length.
Import NAT.

```

16.2 Definitions

Pretty much every programming language provides some form of assignment operation that changes the contents of a previously allocated piece of storage. (Rocq’s internal language Gallina is a rare exception!)

In some languages – notably ML and its relatives – the mechanisms for name binding and those for assignment are kept separate. We can have a variable `x` whose *value* is the number 5, or we can have a variable `y` whose value is a *reference* (or *pointer*) to a mutable cell whose current contents is 5. These are different things, and the difference is visible to the programmer. We can add `x` to another number, but not assign to it. We can use `y` to assign a new value to the cell that it points to (by writing `y:=84`), but we cannot use `y` directly as an argument to an operation like `+`. Instead, we must explicitly *dereference* it, writing `!y` to obtain its current contents.

In most other languages – in particular, in all members of the C family, including Java – *every* variable name refers to a mutable cell, and the operation of dereferencing a variable to obtain its current contents is implicit.

For purposes of formal study, it is useful to keep these mechanisms separate. The development in this chapter will closely follow ML’s model. Applying the lessons learned here to C-like languages is a straightforward matter of collapsing some distinctions and rendering some operations such as dereferencing implicit instead of explicit.

16.3 Syntax

In this chapter, we study adding mutable references to the simply-typed lambda calculus with natural numbers.

Module `STLCREF`.

The basic operations on references are *allocation*, *dereferencing*, and *assignment*.

- To allocate a reference, we use the *ref* operator, providing an initial value for the new cell.

For example, `ref 5` creates a new cell containing the value 5 and reduces to a reference to that cell.

- To read the current value of this cell, we use the dereferencing operator `!`.

For example, `!(ref 5)` reduces to 5.

- To change the value stored in a cell, we use the assignment operator.
If r is a reference, $r := 7$ will store the value 7 in the cell referenced by r .

Types

We start with the simply typed lambda calculus over the natural numbers. Besides the base natural-number type and arrow types, we add two more types to deal with references. First, we need the *unit type*, which we will use as the result type of an assignment operation. We then add *reference types*.

If T is a type, then *Ref* T is the type of references to cells holding values of type T .

$T ::= \text{Nat} \mid \text{Unit} \mid T \rightarrow T \mid \text{Ref } T$

```
Inductive ty : Type :=
| Ty_Nat : ty
| Ty_Unit : ty
| Ty_Arrow : ty → ty → ty
| Ty_Ref : ty → ty.
```

Terms

Besides the usual variables, abstractions, applications, terms related to natural numbers, and *unit*, we need four more sorts of terms to handle mutable references:

$t ::= \dots$ Terms | ref t allocation | $!t$ dereference | $t := t$ assignment | l location

```
Inductive tm : Type :=

| tm_var : string → tm
| tm_app : tm → tm → tm
| tm_abs : string → ty → tm → tm
| tm_const : nat → tm
| tm_succ : tm → tm
| tm_pred : tm → tm
| tm_mult : tm → tm → tm
| tm_if0 : tm → tm → tm → tm

| tm_unit : tm
| tm_ref : tm → tm
| tm_deref : tm → tm
| tm_assign : tm → tm → tm
| tm_loc : nat → tm.
```

Declare Custom Entry stlc_ty.

Declare Custom Entry stlc_tm.

Declare Scope stlc_scope.

```

Notation "x" := x (in custom stlc_ty at level 0, x global) : stlc_scope.
Notation "<{ x }>" := x (x custom stlc_ty).
Notation "( t )" := t (in custom stlc_ty at level 0, t custom stlc_ty) : stlc_scope.
Notation "S -> T" :=
  (Ty_Arrow S T) (in custom stlc_ty at level 99, right associativity) : stlc_scope.
Notation "$ ( t )" := t (in custom stlc_ty at level 0, t constr) : stlc_scope.
Notation "$ ( x )" := x (in custom stlc_tm at level 0, x constr, only parsing) :
stlc_scope.
Notation "x" := x (in custom stlc_tm at level 0, x constr at level 0) : stlc_scope.
Notation "<{ e }>" := e (e custom stlc_tm at level 200) : stlc_scope.
Notation "( x )" := x (in custom stlc_tm at level 0, x custom stlc_tm) : stlc_scope.
Notation "x y" :=
  (tm_app x y) (in custom stlc_tm at level 10, left associativity)
  : stlc_scope.
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc_tm at level 200, x global,
                  t custom stlc_ty,
                  y custom stlc_tm at level 200,
                  left associativity).
Coercion tm_var : string >-> tm.
Arguments tm_var _%-string.
Notation "'Nat'" := Ty_Nat (in custom stlc_ty at level 0).
Notation "'succ' x" := (tm_succ x) (in custom stlc_tm at level 10,
                                   x custom stlc_tm at level 0) : stlc_scope.
Notation "'pred' x" := (tm_pred x) (in custom stlc_tm at level 10,
                                   x custom stlc_tm at level 0) : stlc_scope.
Notation "x * y" := (tm_mult x y) (in custom stlc_tm at level 95,
                                   right associativity) : stlc_scope.
Notation "'if0' x 'then' y 'else' z" :=
  (tm_if0 x y z) (in custom stlc_tm at level 0,
                  x custom stlc_tm at level 0,
                  y custom stlc_tm at level 0,
                  z custom stlc_tm at level 0) : stlc_scope.
Coercion tm_const : nat >-> tm.
Notation "'Unit'" :=
  (Ty_Unit) (in custom stlc_ty at level 0) : stlc_scope.
Notation "'unit'" := tm_unit (in custom stlc_tm at level 0) : stlc_scope.
Notation "'Ref' t" :=
  (Ty_Ref t) (in custom stlc_ty at level 4).
Notation "'loc' x" := (tm_loc x) (in custom stlc_tm at level 2) : stlc_scope.

```

Notation "'ref' x" := (tm_ref x) (in custom stlc_tm at level 2) : stlc_scope.
 Notation "'!' x" := (tm_deref x) (in custom stlc_tm at level 2) : stlc_scope.
 Notation " e1 ':= ' e2 " := (tm_assign e1 e2) (in custom stlc_tm at level 21) :
 stlc_scope.

Open Scope stlc_scope.

Intuitively:

- *ref t* (formally, *ref t*) allocates a new reference cell with the value *t* and reduces to the location of the newly allocated cell;
- *!t* (formally, *deref t*) reduces to the contents of the cell referenced by *t*;
- *t1 := t2* (formally, *assign t1 t2*) assigns *t2* to the cell referenced by *t1*; and
- *l* (formally, *loc l*) is a reference to the cell at location *l*. We'll discuss locations later.

In informal examples, we'll also freely use the extensions of the STLC developed in the *MoreStlc* chapter; however, to keep the proofs small, we won't bother formalizing them again here. (It would be easy to do so, since there are no very interesting interactions between those features and references.)

Typing (Preview)

Informally, the typing rules for allocation, dereferencing, and assignment look roughly like this:

$\Gamma \vdash t1 \text{ \textbackslash in } T1$	
(T_Ref) $\Gamma \vdash \text{ref } t1 \text{ \textbackslash in Ref } T1$	$\Gamma \vdash t1 \text{ \textbackslash in Ref } T1$
(T_Deref) $\Gamma \vdash !t1 \text{ \textbackslash in } T1$	$\Gamma \vdash t1 \text{ \textbackslash in Ref } T2 \quad \Gamma \vdash t2 \text{ \textbackslash in } T2$
(T_Assign) $\Gamma \vdash t1 := t2 : \text{Unit}$	

The rule for locations will require a bit more machinery, and this will motivate some changes to the other rules; we'll come back to this later.

Values and Substitution

Besides abstractions, numbers, and the unit value, we have one new type of value: locations.

A location is a concrete pointer value – think of it like an address in memory. Locations should never be written directly by programmers; they are used to keep track of the intermediate states of programs involving references. In particular, the result of a *ref* expression is a location.

```

Inductive value : tm → Prop :=
| v_abs : ∀ x T2 t1,
  value <{\x:T2, t1}>
| v_nat : ∀ n : nat,
  value <{ n }>
| v_unit :
  value <{ unit }>
| v_loc : ∀ l,
  value <{ loc l }>.

```

Hint Constructors value : core.

Extending substitution to handle the new syntax of terms is straightforward: substituting into a pointer leaves it unchanged. Reserved Notation "'[' x ']:= ' s ']' t" (in custom stlc_tm at level 5, x global, s custom stlc_tm, t custom stlc_tm at next level, right associativity).

```

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  | tm_var y ⇒
    if String.eqb x y then s else t
  | <{\y:T, t1}> ⇒
    if String.eqb x y then t else <{\y:T, [x:=s] t1}>
  | <{t1 t2}> ⇒
    <{([x:=s] t1) ([x:=s] t2)}>

  | tm_const _ ⇒
    t
  | <{succ t1}> ⇒
    <{succ [x := s] t1}>
  | <{pred t1}> ⇒
    <{pred [x := s] t1}>
  | <{t1 × t2}> ⇒
    <{([x := s] t1) × ([x := s] t2)}>
  | <{if0 t1 then t2 else t3}> ⇒
    <{if0 [x := s] t1 then [x := s] t2 else [x := s] t3}>

  | <{ unit }> ⇒
    t

  | <{ ref t1 }> ⇒
    <{ ref ([x:=s] t1) }>
  | <{ !t1 }> ⇒
    <{ !([x:=s] t1) }>

```

```

| <{ t1 := t2 }> ⇒
  <{ ([x:=s] t1) := ([x:=s] t2) }>
| <{ loc _ }> ⇒
  t
end

```

```

where "'[' x ']:= ' s ']' t" := (subst x s t) (in custom stlc_tm) : stlc_scope.

```

16.4 Pragmatics

16.4.1 Side Effects and Sequencing

The fact that we've chosen the result of an assignment expression to be the trivial value *unit* allows a nice abbreviation for *sequencing*. For example, we can write

```

r:=succ(!r); !r
as an abbreviation for
(\x:Unit, !r) (r:=succ(!r)).

```

This has the effect of reducing two expressions in order and returning the value of the second. Restricting the type of the first expression to *Unit* helps the typechecker to catch some silly errors by permitting us to throw away the first value only if it is really guaranteed to be trivial.

Notice that, if the second expression is also an assignment, then the type of the whole sequence will be *Unit*, so we can validly place it to the left of another `;` to build longer sequences of assignments:

`r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r` Formally, we introduce sequencing as a *derived form* `tseq` that expands into an abstraction and an application.

```

Definition x : string := "x".
Definition y : string := "y".
Definition z : string := "z".
Hint Unfold x : core.
Hint Unfold y : core.
Hint Unfold z : core.

```

```

Definition tseq t1 t2 :=
  <{ (\ x : Unit, t2) t1 }>.

```

```

Notation "t1 ; t2" := (tseq t1 t2) (in custom stlc_tm at level 3) : stlc_scope.

```

16.4.2 References and Aliasing

It is important to bear in mind the difference between the *reference* that is bound to some variable `r` and the *cell* in the store that is pointed to by this reference.

If we make a copy of `r`, for example by binding its value to another variable `s`, what gets copied is only the reference, not the contents of the cell itself.

For example, after reducing

```
let r = ref 5 in let s = r in s := 82; (!r)+1
```

the cell referenced by `r` will contain the value 82, while the result of the whole expression will be 83. The references `r` and `s` are said to be *aliases* for the same cell.

The possibility of aliasing can make programs with references quite tricky to reason about. For example, the expression

```
r := 5; r := !s
```

assigns 5 to `r` and then immediately overwrites it with `s`'s current value; this has exactly the same effect as the single assignment

```
r := !s
```

unless we happen to do it in a context where `r` and `s` are aliases for the same cell!

```
let r = ref 0 in let s = r in r := 5; r := !s
```

16.4.3 Shared State

Of course, aliasing is also a large part of what makes references useful. In particular, it allows us to set up “implicit communication channels” – shared state – between different parts of a program. For example, suppose we define a reference cell and two functions that manipulate its contents:

```
let c = ref 0 in let incc = \_:Unit, (c := succ (!c); !c) in let decc = \_:Unit, (c := pred (!c); !c) in ...
```

Note that, since their argument types are *Unit*, the arguments to the abstractions in the definitions of *incc* and *decc* are not providing any useful information to the bodies of these functions (using the wildcard `_` as the name of the bound variable is a reminder of this). Instead, their purpose of these abstractions is to “slow down” the execution of the function bodies. Since function abstractions are values, the two *lets* are executed simply by binding these functions to the names *incc* and *decc*, rather than by actually incrementing or decrementing `c`. Later, each call to one of these functions results in its body being executed once and performing the appropriate mutation on `c`. Such functions are often called *thunks*.

In the context of these declarations, calling *incc* results in changes to `c` that can be observed by calling *decc*. For example, if we replace the `...` with *(incc unit; incc unit; decc unit)*, the result of the whole program will be 1.

16.4.4 Objects

We can go a step further and write a *function* that creates `c`, *incc*, and *decc*, packages *incc* and *decc* together into a record, and returns this record:

```
newcounter = \_:Unit, let c = ref 0 in let incc = \_:Unit, (c := succ (!c); !c) in let decc = \_:Unit, (c := pred (!c); !c) in {i=incc, d=decc}
```

Now, each time we call *newcounter*, we get a new record of functions that share access to the same storage cell `c`. The caller of *newcounter* can't get at this storage cell directly, but

can affect it indirectly by calling the two functions. In other words, we've created a simple form of *object*.

let c1 = newcounter unit in let c2 = newcounter unit in // we've allocated two separate storage cells now! let r1 = c1.i unit in let r2 = c2.i unit in r2 // yields 1, not 2!

Exercise: 1 star, standard, optional (store_draw) Draw (on paper) the contents of the store at the point in execution where the first two *lets* have finished and the third one is about to begin.

16.4.5 References to Compound Types

A reference cell need not contain just a number: the primitives we've defined above allow us to create references to values of any type, including functions. For example, we can use references to functions to give a(n inefficient) implementation of arrays of numbers as follows.

Write *NatArray* for the type *Ref (Nat→Nat)*.

Recall the *equal* function from the *MoreStlc* chapter:

equal = fix (\eq:Nat->Nat->Bool, \m:Nat, \n:Nat, if m=0 then iszero n else if n=0 then false else eq (pred m) (pred n))

To build a new array, we allocate a reference cell and fill it with a function that, when given an index, always returns 0.

newarray = _:Unit, ref (\n:Nat,0)

To look up an element of an array, we apply the function to the desired index.

lookup = \a:NatArray, \n:Nat, (!a) n

The interesting part of the encoding is the *update* function. It takes an array, an index, and a new value to be stored at that index, and it does its job by creating (and storing in the reference) a new function that, when it is asked for the value at this very index, returns the new value that was given to *update*, while on all other indices it passes the lookup to the function that was previously stored in the reference.

update = \a:NatArray, \m:Nat, \v:Nat, let oldf = !a in a := (\n:Nat, if equal m n then v else oldf n);

References to values containing other references can also be very useful, allowing us to define data structures such as mutable lists and trees.

Exercise: 2 stars, standard, especially useful (compact_update) If we defined *update* more compactly like this

update = \a:NatArray, \m:Nat, \v:Nat, a := (\n:Nat, if equal m n then v else (!a) n)
would it behave the same?

Definition *manual_grade_for_compact_update* : *option (nat×string)* := *None*.

□

16.4.6 Null References

There is one final significant difference between our references and C-style mutable variables: in C-like languages, variables holding pointers into the heap may sometimes have the value *NULL*. Dereferencing such a “null pointer” is an error, and results either in a clean exception (Java and C#) or in arbitrary and possibly insecure behavior (C and relatives like C++). Null pointers cause significant trouble in C-like languages: the fact that any pointer might be null means that any dereference operation in the program can potentially fail.

Even in ML-like languages, there are sometimes situations where we may or may not have a valid pointer in our hands. Fortunately, there is no need to extend the basic mechanisms of references to represent such situations: the sum types introduced in the *MoreStlc* chapter already give us what we need.

First, we can use sums to build an analog of the *option* types introduced in the *Lists* chapter of *Logical Foundations*. Define *Option T* to be an abbreviation for *Unit + T*.

Then a “nullable reference to a *T*” is simply an element of the type *Option (Ref T)*.

16.4.7 Garbage Collection

A last issue that we should mention before we move on with formalizing references is storage *de*-allocation. We have not provided any primitives for freeing reference cells when they are no longer needed. Instead, like many modern languages (including OCaml, Haskell, Java, etc.) we rely on the run-time system to perform *garbage collection*, automatically identifying and reusing cells that can no longer be reached by the program.

This is *not* just a question of taste in language design: it is extremely difficult to achieve type safety in the presence of an explicit deallocation operation.

One reason for this is the familiar *dangling reference* problem: we allocate a cell holding a number, save a reference to it in some data structure, use it for a while, then deallocate it and allocate a new cell holding a boolean, possibly reusing the same storage. Now we can have two names for the same storage cell – one with type *Ref Nat* and the other with type *Ref Bool*.

Exercise: 2 stars, standard (type_safety_violation) Show how this can lead to a violation of type safety.

Definition *manual_grade_for_type_safety_violation* : *option (nat × string)* := *None*.

□

16.5 Operational Semantics

16.5.1 Locations

The most subtle aspect of the treatment of references appears when we consider how to formalize their operational behavior. One way to see why is to ask, “What should be the

values of type *Ref T*?” The crucial observation that we need to take into account is that reducing a *ref* operator should *do* something – namely, allocate some storage – and the result of the operation should be a reference to this storage.

What, then, is a reference?

The run-time store in most programming-language implementations is essentially just a big array of bytes. The run-time system keeps track of which parts of this array are currently in use. When we need to allocate a new reference cell, we allocate a large enough segment from the free region of the store (4 bytes for integer cells, 8 bytes for cells storing *Floats*, etc.), record somewhere that it is being used, and return the index (typically, a 32- or 64-bit integer) of the start of the newly allocated region. These indices are references.

For present purposes, there is no need to be quite so concrete. We can think of the store as an array of *values*, rather than an array of bytes, abstracting away from the different sizes of the run-time representations of different values. A reference, then, is simply an index into the store. (If we like, we can even abstract away from the fact that these indices are numbers, but for purposes of formalization in Rocq it is convenient to use numbers.) We use the word *location* instead of *reference* or *pointer* to emphasize this abstract quality.

Treating locations abstractly in this way will prevent us from modeling the *pointer arithmetic* found in low-level languages such as C. This limitation is intentional. While pointer arithmetic is occasionally very useful, especially for implementing low-level services such as garbage collectors, it cannot be tracked by most type systems: knowing that location *n* in the store contains a *float* doesn’t tell us anything useful about the type of location *n*+4. In C, pointer arithmetic is a notorious source of type-safety violations.

16.5.2 Stores

Recall that, in the small-step operational semantics for Imp, the step relation needed to carry along an auxiliary state in addition to the program being executed. In the same way, once we have added reference cells to the STLC, our step relation must carry along a store to keep track of the contents of reference cells.

We could re-use the same functional representation we used for states in Imp, but for carrying out the proofs in this chapter it is actually more convenient to represent a store simply as a *list* of values. (The reason we didn’t use this representation before is that, in Imp, a program could modify any location at any time, so states had to be ready to map *any* variable to a value. However, in the STLC with references, the only way to create a reference cell is with *ref t1*, which puts the value of *t1* in a new reference cell and reduces to the location of the newly created reference cell. When reducing such an expression, we can just add a new reference cell to the end of the list representing the store.)

Definition *store* := *list tm*.

We use *store_lookup n st* to retrieve the value of the reference cell at location *n* in the store *st*. Note that we must give a default value to *nth* in case we try looking up an index which is too large. (In fact, we never actually do this, but *proving* that we don’t will require a bit of work.)

```

Definition store_lookup (n:nat) (st:store) :=
  nth n st <{ unit }>.

```

To update the store, we use the `replace` function, which replaces the contents of a cell at a particular index.

```

Fixpoint replace {A:Type} (n:nat) (x:A) (l:list A) : list A :=
  match l with
  | nil => nil
  | h :: t =>
    match n with
    | 0 => x :: t
    | S n' => h :: replace n' x t
    end
  end.

```

As might be expected, we will also need some technical lemmas about `replace`; they are straightforward to prove.

```

Lemma replace_nil : ∀ A n (x:A),
  replace n x nil = nil.

```

Proof.

```

  destruct n; auto.

```

Qed.

```

Lemma length_replace : ∀ A n x (l:list A),
  length (replace n x l) = length l.

```

Proof with auto.

```

  intros A n x l. generalize dependent n.
  induction l; intros n.
  destruct n...
  destruct n...
  simpl. rewrite IHl...

```

Qed.

```

Lemma lookup_replace_eq : ∀ l t st,
  l < length st →
  store_lookup l (replace l t st) = t.

```

Proof with auto.

```

  intros l t st.
  unfold store_lookup.
  generalize dependent l.
  induction st as [|t' st']; intros l Hlen.
  -
    inversion Hlen.
  -
    destruct l; simpl...

```

```

    apply IHst'. simpl in Hlen. lia.
Qed.

Lemma lookup_replace_neq :  $\forall l1\ l2\ t\ st,$ 
   $l1 \neq l2 \rightarrow$ 
  store_lookup  $l1$  (replace  $l2\ t\ st$ ) = store_lookup  $l1\ st$ .
Proof with auto.
  unfold store_lookup.
  induction l1 as [|l1']; intros l2 t st Hneq.
  -
    destruct st.
    + rewrite replace_nil...
    + destruct l2... contradict Hneq...
  -
    destruct st as [|t2 st2].
    + destruct l2...
    +
      destruct l2...
      simpl; apply IHl1'...
Qed.

```

16.5.3 Reduction

Now we can give the rules for the new constructs:

$\begin{array}{c} \text{(ST\_RefValue)} \text{ ref } v / st \rightarrow \text{loc } st / st, v \\ t1 / st \rightarrow t1' / st' \end{array}$
$\begin{array}{c} \text{(ST\_Ref)} \text{ ref } t1 / st \rightarrow \text{ref } t1' / st' \\ l < st \end{array}$
$\begin{array}{c} \text{(ST\_DerefLoc)} !(loc\ l) / st \rightarrow \text{lookup } l\ st / st \\ t1 / st \rightarrow t1' / st' \end{array}$
$\begin{array}{c} \text{(ST\_Deref)} !t1 / st \rightarrow !t1' / st' \\ l < st \end{array}$
$\begin{array}{c} \text{(ST\_Assign)} \text{loc } l := v / st \rightarrow \text{unit} / \text{replace } l\ v\ st \\ t1 / st \rightarrow t1' / st' \end{array}$
$\begin{array}{c} \text{(ST\_Assign1)} t1 := t2 / st \rightarrow t1' := t2 / st' \\ t2 / st \rightarrow t2' / st' \end{array}$

(ST_Assign2) $v1 := t2 / st \rightarrow v1 := t2' / st'$

Next, we need to extend the operational semantics to take stores into account. Since the result of reducing an expression will in general depend on the contents of the store in which it is reduced, the evaluation rules should take not just a term but also a store as argument. Furthermore, since the reduction of a term can cause side effects on the store, and these may affect the reduction of other terms in the future, the reduction rules need to return a new store. Thus, the shape of the single-step reduction relation needs to change from $t \rightarrow t'$ to $t / st \rightarrow t' / st'$, where st and st' are the starting and ending states of the store.

To carry through this change, we first need to augment all of our existing reduction rules with stores:

value $v2$

(ST_AppAbs) $(\backslash x:T2, t1) v2 / st \rightarrow x := v2 t1 / st$
 $t1 / st \rightarrow t1' / st'$

(ST_App1) $t1 t2 / st \rightarrow t1' t2 / st'$
 value $v1 t2 / st \rightarrow t2' / st'$

(ST_App2) $v1 t2 / st \rightarrow v1 t2' / st'$

Note that the first rule here returns the store unchanged, since function application, in itself, has no side effects. The other two rules simply propagate side effects from premise to conclusion.

Now, the result of reducing a *ref* expression will be a fresh location; this is why we included locations in the syntax of terms and in the set of values. It is crucial to note that making this extension to the syntax of terms does not mean that we intend *programmers* to write terms involving explicit, concrete locations: such terms will arise only as intermediate results during reduction. This may seem odd, but it follows naturally from our design decision to represent the result of every reduction step by a modified *term*. If we had chosen a more “machine-like” model, e.g., with an explicit stack to contain values of bound identifiers, then the idea of adding locations to the set of allowed values might seem more obvious.

In terms of this expanded syntax, we can state reduction rules for the new constructs that manipulate locations and the store. First, to reduce a dereferencing expression $!t1$, we must first reduce $t1$ until it becomes a value:

$t1 / st \rightarrow t1' / st'$

(ST_Deref) $!t1 / st \rightarrow !t1' / st'$

Once $t1$ has finished reducing, we should have an expression of the form $!l$, where l is some location. (A term that attempts to dereference any other sort of value, such as a function or *unit*, is erroneous, as is a term that tries to dereference a location that is larger than the size $|st|$ of the currently allocated store; the reduction rules simply get stuck in this case. The type-safety properties established below assure us that well-typed terms will never misbehave in this way.)

$$l < |st|$$

(ST_DerefLoc) $!(\text{loc } l) / st \rightarrow \text{lookup } l \text{ } st / st$

Next, to reduce an assignment expression $t1 := t2$, we must first reduce $t1$ until it becomes a value (a location), and then reduce $t2$ until it becomes a value (of any sort):

$$t1 / st \rightarrow t1' / st'$$

(ST_Assign1) $t1 := t2 / st \rightarrow t1' := t2 / st'$

$$t2 / st \rightarrow t2' / st'$$

(ST_Assign2) $v1 := t2 / st \rightarrow v1 := t2' / st'$

Once we have finished with $t1$ and $t2$, we have an expression of the form $l := v$, which we execute by updating the store to make location l contain v :

$$l < |st|$$

(ST_Assign) $\text{loc } l := v / st \rightarrow \text{unit} / l := v \text{ } st$

The notation $[l := v]st$ means “the store that maps l to v and maps all other locations to the same thing as st .” Note that the term resulting from this reduction step is just unit ; the interesting result is the updated store.

Finally, to reduce an expression of the form $\text{ref } t1$, we first reduce $t1$ until it becomes a value:

$$t1 / st \rightarrow t1' / st'$$

(ST_Ref) $\text{ref } t1 / st \rightarrow \text{ref } t1' / st'$

Then, to reduce the ref itself, we choose a fresh location at the end of the current store – i.e., location $|st|$ – and yield a new store that extends st with the new value v .

(ST_RefValue) $\text{ref } v / st \rightarrow \text{loc } |st| / st, v$

The value resulting from this step is the newly allocated location itself. (Formally, st, v means $st ++ v :: \text{nil}$ – i.e., to add a new reference cell to the store, we append it to the end.)

Note that these reduction rules do not perform any kind of garbage collection: we simply allow the store to keep growing without bound as reduction proceeds. This does not affect the correctness of the results of reduction (after all, the definition of “garbage” is precisely parts of the store that are no longer reachable and so cannot play any further role in reduction), but it means that a naive implementation of our evaluator might run out of memory where a more sophisticated evaluator would be able to continue by reusing locations whose contents have become garbage.

Here are the rules again, formally:

Reserved Notation “ $t \text{ ' / ' } st \text{ ' } \rightarrow \text{ ' } t' \text{ ' / ' } st' \text{ '}$ ”

(at level 40, st at next level, t' at next level, left associativity).

Inductive step : $\text{tm} \times \text{store} \rightarrow \text{tm} \times \text{store} \rightarrow \text{Prop} :=$

| ST_AppAbs : $\forall x \ T2 \ t1 \ v2 \ st,$


```

    value v2 →
      <{ (\x : T2, t1) v2 }> / st -> <{ [x := v2] t1 }> / st
| ST_App1 : ∀ t1 t1' t2 st st',
  t1 / st -> t1' / st' →
  <{ t1 t2 }> / st -> <{ t1' t2 }> / st'
| ST_App2 : ∀ v1 t2 t2' st st',
  value v1 →
  t2 / st -> t2' / st' →
  <{ v1 t2 }> / st -> <{ v1 t2' }> / st'

| ST_SuccNat : ∀ (n : nat) st,
  <{ succ n }> / st -> tm_const (S n) / st
| ST_Succ : ∀ t1 t1' st st',
  t1 / st -> t1' / st' →
  <{ succ t1 }> / st -> <{ succ t1' }> / st'
| ST_PredNat : ∀ (n : nat) st,
  <{ pred n }> / st -> tm_const (n - 1) / st
| ST_Pred : ∀ t1 t1' st st',
  t1 / st -> t1' / st' →
  <{ pred t1 }> / st -> <{ pred t1' }> / st'
| ST_MultNats : ∀ (n1 n2 : nat) st,
  <{ n1 × n2 }> / st -> tm_const (n1 × n2) / st
| ST_Mult1 : ∀ t1 t2 t1' st st',
  t1 / st -> t1' / st' →
  <{ t1 × t2 }> / st -> <{ t1' × t2 }> / st'
| ST_Mult2 : ∀ v1 t2 t2' st st',
  value v1 →
  t2 / st -> t2' / st' →
  <{ v1 × t2 }> / st -> <{ v1 × t2' }> / st'
| ST_If0 : ∀ t1 t1' t2 t3 st st',
  t1 / st -> t1' / st' →
  <{ if0 t1 then t2 else t3 }> / st -> <{ if0 t1' then t2 else t3 }> / st'
| ST_If0_Zero : ∀ t2 t3 st,
  <{ if0 0 then t2 else t3 }> / st -> t2 / st
| ST_If0_Nonzero : ∀ n t2 t3 st,
  <{ if0 $(S n) then t2 else t3 }> / st -> t3 / st

| ST_RefValue : ∀ v st,
  value v →
  <{ ref v }> / st -> <{ loc $(length st) }> / (st ++ v :: nil)
| ST_Ref : ∀ t1 t1' st st',
  t1 / st -> t1' / st' →

```

```

      <{ ref t1 }> / st -> <{ ref t1' }> / st'
| ST_DerefLoc : ∀ st l,
  l < length st →
  <{ !(loc l) }> / st -> <{ $(store_lookup l st) }> / st
| ST_Deref : ∀ t1 t1' st st',
  t1 / st -> t1' / st' →
  <{ ! t1 }> / st -> <{ ! t1' }> / st'
| ST_Assign : ∀ v l st,
  value v →
  l < length st →
  <{ (loc l) := v }> / st -> <{ unit }> / replace l v st
| ST_Assign1 : ∀ t1 t1' t2 st st',
  t1 / st -> t1' / st' →
  <{ t1 := t2 }> / st -> <{ t1' := t2 }> / st'
| ST_Assign2 : ∀ v1 t2 t2' st st',
  value v1 →
  t2 / st -> t2' / st' →
  <{ v1 := t2 }> / st -> <{ v1 := t2' }> / st'

```

where "t '/' st '->' t' '/' st'" := (step (t,st) (t',st')).

One slightly ugly point should be noted here: In the `ST_RefValue` rule, we extend the state by writing `st ++ v::nil` rather than the more natural `st ++ [v]`. The reason for this is that the notation we've defined for substitution uses square brackets, which clash with the standard library's notation for lists.

Hint Constructors `step` : core.

Definition `multistep` := (multi step).

Notation "t '/' st '->*' t' '/' st'" :=
 (multistep (t,st) (t',st'))
 (at level 40, st at level 39, t' at level 39).

16.6 Typing

The contexts assigning types to free variables are exactly the same as for the STLC: partial maps from identifiers to types.

Definition `context` := partial_map `ty`.

16.6.1 Store typings

Having extended our syntax and reduction rules to accommodate references, our last job is to write down typing rules for the new constructs (and, of course, to check that these rules are sound!).

Naturally, the key question is, “What is the type of a location?”

First of all, notice that this question doesn’t arise when typechecking terms that programmers actually write. Concrete location constants arise only in terms that are the intermediate results of reduction; they are not in the language that programmers write. So we only need to determine the type of a location when we’re in the middle of a reduction sequence, e.g., trying to apply the progress or preservation lemmas. Thus, even though we normally think of typing as a *static* program property, it makes sense for the typing of locations to depend on the *dynamic* progress of the program too.

As a first try, note that when we reduce a term containing concrete locations, the type of the result depends on the contents of the store that we start with. For example, if we reduce the term $!(loc\ 1)$ in the store $[unit, unit]$, the result is $unit$; if we reduce the same term in the store $[unit, \backslash x:Unit, x]$, the result is $\backslash x:Unit, x$. With respect to the former store, the location 1 has type $Unit$, and with respect to the latter it has type $Unit \rightarrow Unit$. This observation leads us immediately to a first attempt at a typing rule for locations:

$\Gamma \vdash \text{lookup } l \text{ st} : T1$

$\Gamma \vdash \text{loc } l : \text{Ref } T1$ That is, to find the type of a location l , we look up the current contents of l in the store and calculate the type $T1$ of the contents. The type of the location is then $\text{Ref } T1$.

Having begun in this way, we need to go a little further to reach a consistent state. In effect, by making the type of a term depend on the store, we have changed the typing relation from a three-place relation (between contexts, terms, and types) to a four-place relation (between contexts, *stores*, terms, and types). Since the store is, intuitively, part of the context in which we calculate the type of a term, let’s write this four-place relation with the store to the left of the turnstile: $\Gamma; st \vdash t : T$. Our rule for typing references now has the form

$\Gamma; st \vdash \text{lookup } l \text{ st} : T1$

$\Gamma; st \vdash \text{loc } l : \text{Ref } T1$

and all the rest of the typing rules in the system are extended similarly with stores. (The other rules do not need to do anything interesting with their stores – just pass them from premise to conclusion.)

However, this rule will not quite do. For one thing, typechecking is rather inefficient, since calculating the type of a location l involves calculating the type of the current contents v of l . If l appears many times in a term t , we will re-calculate the type of v many times in the course of constructing a typing derivation for t . Worse, if v itself contains locations, then we will have to recalculate *their* types each time they appear. Worse yet, the proposed typing rule for locations may not allow us to derive anything at all if the store contains a *cycle*. For example, there is no finite typing derivation for the location 0 with respect to this store:

$\backslash x:\text{Nat}, (!(loc\ 1))\ x, \backslash x:\text{Nat}, (!(loc\ 0))\ x$

Exercise: 3 stars, standard (cyclic_store) Can you find a term whose reduction will create this particular cyclic store?

Theorem cyclic_store:

```

∃ t,
  t / nil ->*
  <{ unit }> / (<{ \x:Nat, (! (loc 1)) x }> :: <{ \x:Nat, (! (loc 0)) x }> :: nil).

```

Proof.

Admitted.

□

These problems arise from the fact that our proposed typing rule for locations requires us to recalculate the type of a location every time we mention it in a term. But this, intuitively, should not be necessary. After all, when a location is first created, we know the type of the initial value that we are storing into it. Suppose we are willing to enforce the invariant that the type of the value contained in a given location *never changes*; that is, although we may later store other values into this location, those other values will always have the same type as the initial one. In other words, we always have in mind a single, definite type for every location in the store, which is fixed when the location is allocated. Then these intended types can be collected together as a *store typing* – a finite function mapping locations to types.

As with the other type systems we’ve seen, this conservative typing restriction on allowed updates means that we will rule out as ill-typed some programs that could reduce perfectly well without getting stuck.

Just as we did for stores, we will represent a store type simply as a list of types: the type at index *i* records the type of the values that we expect to be stored in cell *i*.

Definition store_ty := list ty.

The store_Tlookup function retrieves the type at a particular index.

```

Definition store_Tlookup (n:nat) (ST:store_ty) :=
  nth n ST <{{ Unit }}>.

```

Suppose we are given a store typing *ST* describing the store *st* in which some term *t* will be reduced. Then we can use *ST* to calculate the type of the result of *t* without ever looking directly at *st*. For example, if *ST* is [*Unit*, *Unit* → *Unit*], then we can immediately infer that *!(loc 1)* has type *Unit* → *Unit*. More generally, the typing rule for locations can be reformulated in terms of store typings like this:

$$l < |ST|$$

Gamma; ST |– loc *l* : Ref (store_Tlookup *l* ST) That is, as long as *l* is a valid location, we can compute the type of *l* just by looking it up in *ST*. Typing is again a four-place relation, but it is parameterized on a store *typing* rather than a concrete store. The rest of the typing rules are analogously augmented with store typings.

16.6.2 The Typing Relation

We can now formalize the typing relation for the STLC with references. Here, again, are the rules we're adding to the base STLC (with numbers and *Unit*):

$$l < |\text{ST}|$$

(T_Loc) $\Gamma; \text{ST} \vdash \text{loc } l : \text{Ref } (\text{store\_Tlookup } l \text{ ST})$
 $\Gamma; \text{ST} \vdash t_1 : T_1$

(T_Ref) $\Gamma; \text{ST} \vdash \text{ref } t_1 : \text{Ref } T_1$
 $\Gamma; \text{ST} \vdash t_1 : \text{Ref } T_1$

(T_Deref) $\Gamma; \text{ST} \vdash !t_1 : T_1$
 $\Gamma; \text{ST} \vdash t_1 : \text{Ref } T_2 \quad \Gamma; \text{ST} \vdash t_2 : T_2$

(T_Assign) $\Gamma; \text{ST} \vdash t_1 := t_2 : \text{Unit}$

Notation " $x \mapsto v$; m " := (update m x v)
 (in custom stlc_tm at level 0, x constr at level 0,
 v custom stlc_ty, right associativity) : stlc_scope.

Notation " $x \mapsto v$ " := (update empty x v)
 (in custom stlc_tm at level 0, x constr at level 0, v custom stlc_ty) : stlc_scope.

Notation "'empty'" := empty (in custom stlc_tm) : stlc_scope.

Reserved Notation "<{ Γ '/' $\text{ST} \vdash t$ \in T }>"
 (at level 0, Γ custom stlc_tm at level 200, t custom stlc_tm, T
 custom stlc_ty).

Inductive **has_type** ($\text{ST} : \text{store\_ty}$) : context $\rightarrow$ **tm** $\rightarrow$ **ty** $\rightarrow$ Prop :=

| T_Var : $\forall \Gamma x T_1$,
 $\Gamma x = \text{Some } T_1 \rightarrow$
 $\langle \{ \Gamma / \text{ST} \vdash x \in T_1 \} \rangle$
| T_Abs : $\forall \Gamma x T_1 T_2 t_1$,
 $\langle \{ x \mapsto T_2; \Gamma / \text{ST} \vdash t_1 \in T_1 \} \rangle \rightarrow$
 $\langle \{ \Gamma / \text{ST} \vdash \lambda x:T_2, t_1 \in T_2 \rightarrow T_1 \} \rangle$
| T_App : $\forall T_1 T_2 \Gamma t_1 t_2$,
 $\langle \{ \Gamma / \text{ST} \vdash t_1 \in T_2 \rightarrow T_1 \} \rangle \rightarrow$
 $\langle \{ \Gamma / \text{ST} \vdash t_2 \in T_2 \} \rangle \rightarrow$
 $\langle \{ \Gamma / \text{ST} \vdash t_1 t_2 \in T_1 \} \rangle$
| T_Nat : $\forall \Gamma (n : \text{nat})$,
 $\langle \{ \Gamma / \text{ST} \vdash n \in \text{Nat} \} \rangle$
| T_Succ : $\forall \Gamma t_1$,
 $\langle \{ \Gamma / \text{ST} \vdash t_1 \in \text{Nat} \} \rangle \rightarrow$
 $\langle \{ \Gamma / \text{ST} \vdash \text{succ } t_1 \in \text{Nat} \} \rangle$
| T_Pred : $\forall \Gamma t_1$,

```

    <{ Gamma / ST |- t1 \in Nat }> →
    <{ Gamma / ST |- pred t1 \in Nat }>
| T_Mult : ∀ Gamma t1 t2,
    <{ Gamma / ST |- t1 \in Nat }> →
    <{ Gamma / ST |- t2 \in Nat }> →
    <{ Gamma / ST |- t1 × t2 \in Nat }>
| T_If0 : ∀ Gamma t1 t2 t3 T0,
    <{ Gamma / ST |- t1 \in Nat }> →
    <{ Gamma / ST |- t2 \in T0 }> →
    <{ Gamma / ST |- t3 \in T0 }> →
    <{ Gamma / ST |- if0 t1 then t2 else t3 \in T0 }>
| T_Unit : ∀ Gamma,
    <{ Gamma / ST |- unit \in Unit }>
| T_Loc : ∀ Gamma l,
    l < length ST →
    <{ Gamma / ST |- (loc l) \in Ref $(store_Tlookup l ST) }>
| T_Ref : ∀ Gamma t1 T1,
    <{ Gamma / ST |- t1 \in T1 }> →
    <{ Gamma / ST |- (ref t1) \in Ref T1 }>
| T_Deref : ∀ Gamma t1 T1,
    <{ Gamma / ST |- t1 \in Ref T1 }> →
    <{ Gamma / ST |- (! t1) \in T1 }>
| T_Assign : ∀ Gamma t1 t2 T2,
    <{ Gamma / ST |- t1 \in Ref T2 }> →
    <{ Gamma / ST |- t2 \in T2 }> →
    <{ Gamma / ST |- (t1 := t2) \in Unit }>

```

where " $\langle \{ \textit{Gamma} \text{ '/' } \textit{ST} \text{ ' |- ' } t \text{ ' \in ' } T \} \rangle$ " := (**has_type** *ST* *Gamma* *t* *T*).

Hint Constructors **has_type** : core.

Of course, these typing rules will accurately predict the results of reduction only if the concrete store used during reduction actually conforms to the store typing that we assume for purposes of typechecking. This proviso exactly parallels the situation with free variables in the basic STLC: the substitution lemma promises that, if $\textit{Gamma} \vdash t : T$, then we can replace the free variables in t with values of the types listed in $\textit{Gamma}$ to obtain a closed term of type T , which, by the type preservation theorem will reduce to a final result of type T if it yields any result at all. We will see below how to formalize an analogous intuition for stores and store typings.

However, for purposes of typechecking the terms that programmers actually write, we do not need to do anything tricky to guess what store typing we should use. Concrete locations arise only in terms that are the intermediate results of reduction; they are not in the language that programmers write. Thus, we can simply typecheck the programmer's terms with respect to the *empty* store typing. As reduction proceeds and new locations are

created, we will always be able to see how to extend the store typing by looking at the type of the initial values being placed in newly allocated cells; this intuition is formalized in the statement of the type preservation theorem below.

16.7 Properties

Our final task is to check that standard type safety properties continue to hold for the STLC with references. The progress theorem (“well-typed terms are not stuck”) can be stated and proved almost as for the STLC; we just need to add a few straightforward cases to the proof to deal with the new constructs. The preservation theorem is a bit more interesting, so let’s look at it first.

16.7.1 Well-Typed Stores

Since we have extended both the reduction relation (with initial and final stores) and the typing relation (with a store typing), we need to change the statement of preservation to include these parameters. But clearly we cannot just add stores and store typings without saying anything about how they are related – i.e., this is wrong:

Theorem `preservation_wrong` : $\forall ST\ T\ t\ st\ t'\ st',$
 $\langle \{ \text{empty} / ST \mid -\ t \in T \} \rangle \rightarrow$
 $t / st \rightarrow t' / st' \rightarrow$
 $\langle \{ \text{empty} / ST \mid -\ t' \in T \} \rangle.$

Abort.

If we typecheck with respect to some set of assumptions about the types of the values in the store and then reduce with respect to a store that violates these assumptions, the result will be disaster. We say that a store st is *well typed* with respect a store typing ST if the term at each location l in st has the type at location l in ST . Since only closed terms ever get stored in locations (why?), it suffices to type them in the empty context. The following definition of `store_well_typed` formalizes this.

Definition `store_well_typed` (ST :store_ty) (st :store) :=
 $\text{length } ST = \text{length } st \wedge$
 $(\forall l, l < \text{length } st \rightarrow$
 $\langle \{ \text{empty} / ST \mid -\ \$(\text{store\_lookup } l\ st)$
 $\quad \in \$(\text{store\_Tlookup } l\ ST) \} \rangle).$

Informally, we will write $ST \mid -\ st$ for `store_well_typed` $ST\ st$.

Intuitively, a store st is consistent with a store typing ST if every value in the store has the type predicted by the store typing. The only subtle point is the fact that, when typing the values in the store, we supply the very same store typing to the typing relation. This allows us to type circular stores like the one we saw above.

Exercise: 3 stars, standard (store_not_unique) Can you find a store st , and two different store typings $ST1$ and $ST2$ such that both $ST1 \vdash st$ and $ST2 \vdash st$?

Theorem store_not_unique:

$\exists st, \exists ST1, \exists ST2,$
 $\text{store\_well\_typed } ST1 \ st \wedge$
 $\text{store\_well\_typed } ST2 \ st \wedge$
 $ST1 \neq ST2.$

Proof.

Admitted.

□

We can now state something closer to the desired preservation property:

Theorem preservation_lesswrong : $\forall ST \ T \ t \ st \ t' \ st',$

$\langle \{ \text{empty} / ST \mid - \ t \ \text{in } T \} \rangle \rightarrow$
 $t / st \rightarrow t' / st' \rightarrow$
 $\text{store\_well\_typed } ST \ st \rightarrow$
 $\langle \{ \text{empty} / ST \mid - \ t' \ \text{in } T \} \rangle.$

Abort.

This statement is fine for all of the reduction rules except the allocation rule `ST_RefValue`. The problem is that this rule yields a store with a larger domain than the initial store, which falsifies the conclusion of the above statement: if st' includes a binding for a fresh location l , then l cannot be in the domain of ST , and it will not be the case that t' (which definitely mentions l) is typable under ST .

For example, the term `ref 5` is typeable in the empty context and empty store typing, but it steps to `loc 0`, which is not typeable with an empty store typing.

16.7.2 Extending Store Typings

Evidently, since the store can increase in size during reduction, we need to allow the store *typing* to grow during reduction as well. This motivates the following definition. We say that the store type ST' *extends* ST if ST' is just ST with some new types added to the end.

Inductive extends : store_ty $\rightarrow$ store_ty $\rightarrow$ Prop :=
| extends_nil : $\forall ST',$
 $\text{extends } ST' \ \text{nil}$
| extends_cons : $\forall x \ ST' \ ST,$
 $\text{extends } ST' \ ST \rightarrow$
 $\text{extends } (x :: ST') \ (x :: ST).$

For example, the store typing `[Nat]` with a single location of type `Nat` extends the empty store typing.

Hint Constructors extends : core.

We'll need a few technical lemmas about extended contexts.

First, looking up a type in an extended store typing yields the same result as in the original:

```
Lemma extends_lookup :  $\forall l \ ST \ ST'$ ,
   $l < \text{length } ST \rightarrow$ 
  extends  $ST' \ ST \rightarrow$ 
  store_Tlookup  $l \ ST' = \text{store\_Tlookup } l \ ST$ .
```

Proof with auto.

```
intros l ST.
generalize dependent l.
induction ST as [|a ST2]; intros l ST' Hlen HST'.
- inversion Hlen.
- unfold store_Tlookup in *.
  destruct ST'.
  + inversion HST'.
  +
    inversion HST'; subst.
    destruct l as [|l'].
     $\times$  auto.
     $\times$  simpl. apply IHST2...
      simpl in Hlen; lia.
```

Qed.

Next, if ST' extends ST , the length of ST' is at least that of ST .

```
Lemma length_extends :  $\forall l \ ST \ ST'$ ,
   $l < \text{length } ST \rightarrow$ 
  extends  $ST' \ ST \rightarrow$ 
   $l < \text{length } ST'$ .
```

Proof with eauto.

```
intros. generalize dependent l. induction H0; intros l Hlen.
- inversion Hlen.
- simpl in *.
  destruct l; try lia.
  apply  $\rightarrow \text{Nat.succ\_lt\_mono}$ . apply IHextends. lia.
```

Qed.

Finally, $ST ++ T$ extends ST , and **extends** is reflexive.

```
Lemma extends_app :  $\forall ST \ T$ ,
  extends  $(ST ++ T) \ ST$ .
```

Proof.

```
induction ST; intros T.
auto.
simpl. auto.
```

Qed.

Lemma extends_refl : $\forall ST,$
 extends $ST ST$.

Proof.

induction ST; auto.

Qed.

16.7.3 Preservation, Finally

We can now give the final, correct statement of the type preservation property:

Definition preservation_theorem := $\forall ST\ t\ t'\ T\ st\ st',$
 <{ empty / $ST \mid -\ t \setminus \text{in } T$ }> $\rightarrow$
 store_well_typed $ST\ st \rightarrow$
 $t / st \rightarrow t' / st' \rightarrow$
 $\exists ST',$
 extends $ST' ST \wedge$
 <{ empty / $ST' \mid -\ t' \setminus \text{in } T$ }> $\wedge$
 store_well_typed $ST' st'$.

Note that the preservation theorem merely asserts that there is *some* store typing ST' extending ST (i.e., agreeing with ST on the values of all the old locations) such that the new term t' is well typed with respect to ST' ; it does not tell us exactly what ST' is. It is intuitively clear, of course, that ST' is either ST or else exactly $ST ++ T1::\text{nil}$, where $T1$ is the type of the value $v1$ in the extended store $st ++ v1::\text{nil}$, but stating this explicitly would complicate the statement of the theorem without making it any more useful: the weaker version above is already in the right form (because its conclusion implies its hypothesis) to “turn the crank” repeatedly and conclude that every *sequence* of reduction steps preserves well-typedness. Combining this with the progress property, we obtain the usual guarantee that “well-typed programs never go wrong.”

In order to prove this, we’ll need a few lemmas, as usual.

16.7.4 Substitution Lemma

First, we need an easy extension of the standard substitution lemma, along with the same machinery about context invariance that we used in the proof of the substitution lemma for the STLC.

Lemma weakening : $\forall Gamma\ Gamma'\ ST\ t\ T,$
 includedin $Gamma\ Gamma' \rightarrow$
 <{ $Gamma / ST \mid -\ t \setminus \text{in } T$ }> $\rightarrow$
 <{ $Gamma' / ST \mid -\ t \setminus \text{in } T$ }>.

Proof.

intros Gamma Gamma' ST t T H Ht.
 generalize dependent Gamma'.

```

    induction Ht; eauto using includedin_update.
Qed.

Lemma weakening_empty :  $\forall \text{Gamma } ST \text{ } t \text{ } T,$ 
    <{ empty /  $ST \vdash t \text{ in } T$  }>  $\rightarrow$ 
    <{  $\text{Gamma} / ST \vdash t \text{ in } T$  }>.
Proof.
    intros Gamma ST t T.
    eapply weakening.
    discriminate.
Qed.

Lemma substitution_preserves_typing :  $\forall \text{Gamma } ST \text{ } x \text{ } U \text{ } t \text{ } v \text{ } T,$ 
    <{  $x \mapsto U; \text{Gamma} / ST \vdash t \text{ in } T$  }>  $\rightarrow$ 
    <{ empty /  $ST \vdash v \text{ in } U$  }>  $\rightarrow$ 
    <{  $\text{Gamma} / ST \vdash [x:=v]t \text{ in } T$  }>.
Proof.
    intros Gamma ST x U t v T Ht Hv.
    generalize dependent Gamma. generalize dependent T.
    induction t; intros T Gamma H;

    inversion H; clear H; subst; simpl; eauto.
-
    rename s into y. destruct (String.eqb_spec x y); subst.
    +
        rewrite update_eq in H2.
        injection H2 as H2; subst.
        apply weakening_empty. assumption.
    +
        apply T_Var. rewrite update_neq in H2; auto.
-
    rename s into y.
    destruct (String.eqb_spec x y); subst; apply T_Abs.
    +
        rewrite update_shadow in H5. assumption.
    +
        apply IHt.
        rewrite update_permute; auto.
Qed.

```

16.7.5 Assignment Preserves Store Typing

Next, we must show that replacing the contents of a cell in the store with a new value of appropriate type does not change the overall type of the store. (This is needed for the

ST_Assign rule.)

Lemma assign_pres_store_typing : $\forall ST\ st\ l\ t,$
 $l < \text{length } st \rightarrow$
 $\text{store\_well\_typed } ST\ st \rightarrow$
 $\langle \{ \text{empty} / ST \mid -\ t \ \backslash \text{in } \$(\text{store\_Tlookup } l\ ST) \} \rangle \rightarrow$
 $\text{store\_well\_typed } ST\ (\text{replace } l\ t\ st).$

Proof with auto.

```
intros ST st l t Hlen HST Ht.
inversion HST; subst.
split. rewrite length_replace...
intros l' Hl'.
destruct (l' =? l) eqn: Heql1'.
-
  apply eqb_eq in Heql1'; subst.
  rewrite lookup_replace_eq...
-
  apply eqb_neq in Heql1'.
  rewrite lookup_replace_neq...
  rewrite length_replace in Hl'.
  apply H0...
```

Qed.

16.7.6 Weakening for Stores

Finally, we need a lemma on store typings, stating that, if a store typing is extended with a new location, the extended one still allows us to assign the same types to the same terms as the original.

Lemma store_weakening : $\forall \text{Gamma } ST\ ST'\ t\ T,$
 $\text{extends } ST'\ ST \rightarrow$
 $\langle \{ \text{Gamma} / ST \mid -\ t \ \backslash \text{in } T \} \rangle \rightarrow$
 $\langle \{ \text{Gamma} / ST' \mid -\ t \ \backslash \text{in } T \} \rangle.$

Proof with eauto.

```
intros. induction H0; eauto.
-
  rewrite ← (extends_lookup _ _ ST')...
  apply T_Loc.
  eapply length_extends...
```

Qed.

We can use the `store_weakening` lemma to prove that if a store is well typed with respect to a store typing, then the store extended with a new term t will still be well typed with respect to the store typing extended with t 's type.

Lemma store_well_typed_app : $\forall ST\ st\ t1\ T1,$

```

store_well_typed ST st →
<{ empty / ST |- t1 \in T1 }> →
store_well_typed (ST ++ T1::nil) (st ++ t1::nil).
Proof with auto.
intros.
unfold store_well_typed in *.
destruct H as [Hlen Hmatch].
rewrite app_length, add_comm. simpl.
rewrite app_length, add_comm. simpl.
split...
-
  intros l Hl.
  unfold store_lookup, store_Tlookup.
  apply le_lt_eq_dec in Hl; destruct Hl as [Hlt | Heq].
  +
    apply ←Nat.succ_lt_mono in Hlt.
    rewrite !app_nth1...
    × apply store_weakening with ST. apply extends_app.
      apply Hmatch...
    × rewrite Hlen...
  +
    injection Heq as Heq; subst.
    rewrite app_nth2; try lia.
    rewrite ← Hlen.
    rewrite sub_diag. simpl.
    apply store_weakening with ST...
    { apply extends_app. }
    rewrite app_nth2; [|lia].
    rewrite sub_diag. simpl. assumption.
Qed.

```

16.7.7 Preservation!

Now that we've got everything set up right, the proof of preservation is actually quite straightforward.

Begin with one technical lemma:

Lemma `nth_eq_last` : $\forall A (l:\text{list } A) x d,$
 $\text{nth } (\text{length } l) (l ++ x::\text{nil}) d = x.$

Proof.

```
induction l; intros; [ auto | simpl; rewrite IHl; auto ].
```

Qed.

And here, at last, is the preservation theorem:

Theorem preservation : $\forall ST\ t\ t'\ T\ st\ st',$

```
<{ empty / ST |- t \in T }> →
store_well_typed ST st →
t / st -> t' / st' →
∃ ST',
  extends ST' ST ∧
  <{ empty / ST' |- t' \in T }> ∧
  store_well_typed ST' st'.
```

Proof with eauto using store_weakening, extends_refl.

```
remember empty as Gamma.
intros ST t t' T st st' Ht.
generalize dependent t'.
induction Ht; intros t' HST Hstep;
  subst; try solve_by_invert; inversion Hstep; subst;
  try (eauto using store_weakening, extends_refl).
- ∃ ST.
  inversion Ht1; subst.
  split; try split... eapply substitution_preserves_typing...
-
  eapply IHHT1 in H0...
  destruct H0 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  eapply IHHT2 in H5...
  destruct H5 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  +
    eapply IHHT in H0...
    destruct H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
-
  +
    eapply IHHT in H0...
    destruct H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
-
  eapply IHHT1 in H0...
  destruct H0 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  eapply IHHT2 in H5...
```

```

destruct H5 as [ST' [Hext [Hty Hsty]]].
∃ ST'...
-
+
  eapply IHHT1 in H0...
  destruct H0 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'. split...
-
  ∃ (ST ++ T1::nil).
  inversion HST; subst.
  split.
  { apply extends_app. }
  split.
  { replace <{{ Ref T1 }}>
    with <{{ Ref $(store_Tlookup (length st) (ST ++ T1::nil)) }}>.
    { apply T_Loc.
      rewrite ← H.rewrite app_length, add_comm. simpl. lia. }
    unfold store_Tlookup. rewrite ← H.rewrite nth_eq_last.
    reflexivity. }
  apply store_well_typed_app; assumption.
-
  eapply IHHT in H0...
  destruct H0 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  ∃ ST. split; try split...
  destruct HST as [_ Hsty].
  replace T1 with (store_Tlookup 1 ST).
  apply Hsty...
  inversion Ht; subst...
-
  eapply IHHT in H0...
  destruct H0 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  ∃ ST. split; try split...
  eapply assign_pres_store_typing...
  inversion Ht1; subst...
-
  eapply IHHT1 in H0...
  destruct H0 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...

```

```

-
  eapply IHht2 in H5...
  destruct H5 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
Qed.

```

Exercise: 3 stars, standard, optional (preservation_informal) Write a careful informal proof of the preservation theorem, concentrating on the `T_App`, `T_Deref`, `T_Assign`, and `T_Ref` cases.

Definition `manual_grade_for_preservation_informal` : `option (nat × string)` := `None`.

□

16.7.8 Progress

As we've said, progress for this system is pretty easy to prove; the proof is very similar to the proof of progress for the STLC, with a few new cases for the new syntactic constructs.

Theorem `progress` : $\forall ST\ t\ T\ st,$
 $\langle \{ \text{empty} / ST \mid -\ t \in T \} \rangle \rightarrow$
`store_well_typed` $ST\ st \rightarrow$
 $(\text{value } t \vee \exists t'\ st',\ t / st \rightarrow t' / st').$

Proof with `eauto`.

```

intros ST t T st Ht HST. remember empty as Gamma.
induction Ht; subst; try solve_by_invert...
-
  right. destruct IHht1 as [Ht1p | Ht1p]...
  +
    inversion Ht1p; subst; try solve_by_invert.
    destruct IHht2 as [Ht2p | Ht2p]...
    ×
      destruct Ht2p as [t2' [st' Hstep]].
      ∃ <{ (λ x0 : T0, t0) t2' }>, st'...
  +
    destruct Ht1p as [t1' [st' Hstep]].
    ∃ <{ t1' t2 }>, st'...
-
  right. destruct IHht as [Ht1p | Ht1p]...
  +
    inversion Ht1p; subst; try solve [ inversion Ht ].
    ×
      ∃ <{ $(S n) }>, st...
  +
    destruct Ht1p as [t1' [st' Hstep]].

```



```

    ∃ <{ succ t1' }>, st'...
-
right. destruct IHHt as [Ht1p | Ht1p]...
+
  inversion Ht1p; subst; try solve [inversion Ht ].
  ×
  ∃ <{ $(n - 1) }>, st...
+
  destruct Ht1p as [t1' [st' Hstep]].
  ∃ <{ pred t1' }>, st'...
-
right. destruct IHHt1 as [Ht1p | Ht1p]...
+
  inversion Ht1p; subst; try solve [inversion Ht1].
  destruct IHHt2 as [Ht2p | Ht2p]...
  ×
  inversion Ht2p; subst; try solve [inversion Ht2].
  ∃ <{ $(n × n0) }>, st...
  ×
  destruct Ht2p as [t2' [st' Hstep]].
  ∃ <{ n × t2' }>, st'...
+
  destruct Ht1p as [t1' [st' Hstep]].
  ∃ <{ t1' × t2 }>, st'...
-
right. destruct IHHt1 as [Ht1p | Ht1p]...
+
  inversion Ht1p; subst; try solve [inversion Ht1].
  destruct n.
  × ∃ t2, st...
  × ∃ t3, st...
+
  destruct Ht1p as [t1' [st' Hstep]].
  ∃ <{ if0 t1' then t2 else t3 }>, st'...
-
right. destruct IHHt as [Ht1p | Ht1p]...
+
  destruct Ht1p as [t1' [st' Hstep]].
  ∃ <{ref t1'}>, st'...
-
right. destruct IHHt as [Ht1p | Ht1p]...
+

```

```

    inversion Ht1p; subst; try solve_by_invert.
    eexists. eexists. apply ST_DerefLoc...
    inversion Ht; subst. inversion HST; subst.
    rewrite ← H...
+
    destruct Ht1p as [t1' [st' Hstep]].
    ∃ <{ ! t1' }>, st'...
-
right. destruct IHHt1 as [Ht1p|Ht1p]...
+
    destruct IHHt2 as [Ht2p|Ht2p]...
    ×
        inversion Ht1p; subst; try solve_by_invert.
        eexists. eexists. apply ST_Assign...
        inversion HST; subst. inversion Ht1; subst.
        rewrite H in H4...
    ×
        destruct Ht2p as [t2' [st' Hstep]].
        ∃ <{ t1 := t2' }>, st'...
+
    destruct Ht1p as [t1' [st' Hstep]].
    ∃ <{ t1' := t2 }>, st'...
Qed.

```

16.8 References and Nontermination

An important fact about the STLC (proved in chapter [Norm](#)) is that it is *normalizing* – that is, every well-typed term can be reduced to a value in a finite number of steps.

What about STLC + references? Surprisingly, adding references causes us to lose the normalization property: there exist well-typed terms in the STLC + references which can continue to reduce forever, without ever reaching a normal form!

How can we construct such a term? The main idea is to make a function which calls itself. We first make a function which calls another function stored in a reference cell; the trick is that we then smuggle in a reference to itself!

```
(\r:Ref (Unit -> Unit), r := (\x:Unit,(!r) unit); (!r) unit) (ref (\x:Unit,unit))
```

First, *ref* $(\backslash x:Unit, unit)$ creates a reference to a cell of type $Unit \rightarrow Unit$. We then pass this reference as the argument to a function which binds it to the name *r*, and assigns to it the function $\backslash x:Unit, (!r) unit$ – that is, the function which ignores its argument and calls the function stored in *r* on the argument *unit*; but of course, that function is itself! To start the divergent loop, we execute the function stored in the cell by evaluating $(!r) unit$.

Here is the divergent term in Rocq:

```
Module EXAMPLEVARIABLES.
```

```

Open Scope string_scope.

Definition x := "x".
Definition y := "y".
Definition r := "r".
Definition s := "s".

End EXAMPLEVARIABLES.

Module REFSANDNONTERMINATION.
Import EXAMPLEVARIABLES.

Definition loop_fun :=
  <{ \x : Unit, (!r) unit }>.

Definition loop :=
  <{ (\r : Ref (Unit → Unit),
    (( r := loop_fun );
    ( (! r) unit ) ))
    (ref (\x : Unit, unit)) }> .
  This term is well typed:

Import LISTNOTATIONS.

Lemma loop_typeable : ∃ T, <{ empty / [] |- loop \in T }>.
Proof with eauto.
  eexists. unfold loop. unfold loop_fun.
  eapply T_App...
  eapply T_Abs...
  eapply T_App...
  eapply T_Abs. eapply T_App. eapply T_Deref. eapply T_Var.
  rewrite update_neq; [|intros; discriminate].
  rewrite update_eq. reflexivity. auto.
  eapply T_Assign.
  eapply T_Var. rewrite update_eq. reflexivity.
  eapply T_Abs.
  eapply T_App...
  eapply T_Deref. eapply T_Var. reflexivity.
Qed.

```

To show formally that the term diverges, we first define the **step_closure** of the single-step reduction relation, written $\rightarrow^+$. This is just like the reflexive step closure of single-step reduction (which we're been writing $\rightarrow^*$), except that it is not reflexive: $t \rightarrow^+ t'$ means that t can reach t' by *one or more* steps of reduction.

```

Inductive step_closure {X:Type} (R: relation X) : X → X → Prop :=
| sc_one : ∀ (x y : X),
    R x y → step_closure R x y
| sc_step : ∀ (x y z : X),

```

$$R\ x\ y \rightarrow$$

$$\text{step\_closure } R\ y\ z \rightarrow$$

$$\text{step\_closure } R\ x\ z.$$

Definition multistep1 := (step_closure step).

Notation "t1 '/' st '->+' t2 '/' st'" :=
 (multistep1 (t1,st) (t2,st'))
 (at level 40, st at level 39, t2 at level 39).

Now, we can show that the expression `loop` reduces to the expression `!(loc 0) unit` and the size-one store `[r:=(loc 0)]loop_fun`.

As a convenience, we introduce a slight variant of the `normalize` tactic, called `reduce`, which tries solving the goal with `multi_refl` at each step, instead of waiting until the goal can't be reduced any more. Of course, the whole point is that `loop` doesn't normalize, so the old `normalize` tactic would just go into an infinite loop reducing it forever!

Ltac print_goal := match goal with $\vdash ?x \Rightarrow$ idtac x end.

Ltac reduce :=
 repeat (print_goal; eapply multi_step ;
 [(eauto 10; fail) | compute];
 try solve [apply multi_refl]).

Next, we use `reduce` to show that `loop` steps to `!(loc 0) unit`, starting from the empty store.

Lemma loop_steps_to_loop_fun :

loop / nil ->*
 $\langle \{ \text{!(loc 0)} \} \text{ unit} \rangle / \text{cons } \langle \{ [r := \text{loc 0}] \text{ loop\_fun} \} \rangle \text{ nil}.$

Proof.

unfold loop.

reduce.

Qed.

Finally, we show that the latter expression reduces in two steps to itself!

Lemma loop_fun_step_self :

$\langle \{ \text{!(loc 0)} \} \text{ unit} \rangle / \text{cons } \langle \{ [r := \text{loc 0}] \text{ loop\_fun} \} \rangle \text{ nil} ->+$
 $\langle \{ \text{!(loc 0)} \} \text{ unit} \rangle / \text{cons } \langle \{ [r := \text{loc 0}] \text{ loop\_fun} \} \rangle \text{ nil}.$

Proof with eauto.

unfold loop_fun; simpl.

eapply sc_step. apply ST_App1...

eapply sc_one. compute. apply ST_AppAbs...

Qed.

Exercise: 4 stars, standard (factorial_ref) Use the above ideas to implement a factorial function in STLC with references. (There is no need to prove formally that it really behaves like the factorial. Just uncomment the example below to make sure it gives the correct result when applied to the argument 4.)

Definition factorial : **tm**

. *Admitted*.

Lemma factorial_type : <{ empty / nil |- *factorial* \in (Nat → Nat) }>.

Proof with eauto.

Admitted.

If your definition is correct, you should be able to just uncomment the example below; the proof should be fully automatic using the *reduce* tactic.

□

16.9 Additional Exercises

Exercise: 5 stars, standard, optional (garbage_collector) Challenge problem: modify our formalization to include an account of garbage collection, and prove that it satisfies whatever nice properties you can think to prove about it.

□

End REFSANDNONTERMINATION.

End STLCREF.

Chapter 17

Library `PLF.RecordSub`

17.1 RecordSub: Subtyping with Records

In this chapter, we combine two significant extensions of the pure STLC – records (from chapter `Records`) and subtyping (from chapter `Sub`) – and explore their interactions. Most of the concepts have already been discussed in those chapters, so the presentation here is somewhat terse. We just comment where things are nonstandard.

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import Strings.String.
From PLF Require Import Maps.
From PLF Require Import Smallstep.
Module RECORDSUB.
```

17.2 Core Definitions

Syntax

```
Inductive ty : Type :=
  | Ty_Top : ty
  | Ty_Base : string → ty
  | Ty_Arrow : ty → ty → ty

  | Ty_RNil : ty
  | Ty_RCons : string → ty → ty → ty.
Inductive tm : Type :=
  | tm_var : string → tm
  | tm_app : tm → tm → tm
```

```

| tm_abs : string → ty → tm → tm
| tm_rproj : tm → string → tm

| tm_rnil : tm
| tm_rcons : string → tm → tm → tm.

```

Declare Custom Entry stlc.

Declare Custom Entry stlc_ty.

```

Notation "<{ e }>" := e (e custom stlc at level 99).
Notation "<{{ e }}>" := e (e custom stlc_ty at level 99).
Notation "( x )" := x (in custom stlc, x at level 99).
Notation "( x )" := x (in custom stlc_ty, x at level 99).
Notation "x" := x (in custom stlc at level 0, x constr at level 0).
Notation "x" := x (in custom stlc_ty at level 0, x constr at level 0).
Notation "S->T" := (Ty_Arrow S T) (in custom stlc_ty at level 50, right associativity).
Notation "x y" := (tm_app x y) (in custom stlc at level 1, left associativity).
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc at level 90, x at level 99,
                  t custom stlc_ty at level 99,
                  y custom stlc at level 99,
                  left associativity).

```

Coercion tm_var : string >-> tm.

```

Notation "{ x }" := x (in custom stlc at level 1, x constr).
Notation "'Base' x" := (Ty_Base x) (in custom stlc_ty at level 0).
Notation "l ':' t1 '::' t2" := (Ty_RCons l t1 t2) (in custom stlc_ty at level 3, right
associativity).
Notation "l := e1 '::' e2" := (tm_rcons l e1 e2) (in custom stlc at level 3, right
associativity).
Notation "'nil'" := (Ty_RNil) (in custom stlc_ty).
Notation "'nil'" := (tm_rnil) (in custom stlc).
Notation "o -> l" := (tm_rproj o l) (in custom stlc at level 0).
Notation "'Top'" := (Ty_Top) (in custom stlc_ty at level 0).

```

Well-Formedness

The syntax of terms and types is a bit too loose, in the sense that it admits things like a record type whose final “tail” is *Top* or some arrow type rather than *Nil*. To avoid such cases, it is useful to assume that all the record types and terms that we see will obey some simple well-formedness conditions.

```

Inductive record_ty : ty → Prop :=
| RTnil :
  record_ty <{{ nil }}>

```

```

| RTcons : ∀ i T1 T2,
  record_ty <{{ i : T1 :: T2 }}>.
Inductive record_tm : tm → Prop :=
| rnil :
  record_tm <{ nil }>
| rtcons : ∀ i t1 t2,
  record_tm <{ i := t1 :: t2 }>.
Inductive well_formed_ty : ty → Prop :=
| wfTop :
  well_formed_ty <{{ Top }}>
| wfBase : ∀ (i : string),
  well_formed_ty <{{ Base i }}>
| wfArrow : ∀ T1 T2,
  well_formed_ty T1 →
  well_formed_ty T2 →
  well_formed_ty <{{ T1 → T2 }}>
| wfRNil :
  well_formed_ty <{{ nil }}>
| wfRCons : ∀ i T1 T2,
  well_formed_ty T1 →
  well_formed_ty T2 →
  record_ty T2 →
  well_formed_ty <{{ i : T1 :: T2 }}>.
Hint Constructors record_ty record_tm well_formed_ty : core.

```

Substitution

Substitution and reduction are as before.

Reserved Notation "'[x := s] t'" (in *custom stlc* at level 20, *x* constr).

```

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  | tm_var y ⇒
    if String.eqb x y then s else t
  | <{\y:T, t1}> ⇒
    if String.eqb x y then t else <{\y:T, [x:=s] t1}>
  | <{t1 t2}> ⇒
    <{([x:=s] t1) ([x:=s] t2)}>
  | <{ t1 -> i }> ⇒
    <{ ( [x := s] t1 ) -> i }>
  | <{ nil }> ⇒
    <{ nil }>

```



```

| <{ i := t1 :: tr }> =>
  <{ i := [x := s] t1 :: ([x := s] tr) }>
end

```

where "'[x := s]' t" := (subst x s t) (in *custom stlc*).

Reduction

```

Inductive value : tm → Prop :=
| v_abs : ∀ x T2 t1,
  value <{ \ x : T2, t1 }>
| v_rnil : value <{ nil }>
| v_rcons : ∀ i v1 vr,
  value v1 →
  value vr →
  value <{ i := v1 :: vr }>.

```

Hint Constructors value : core.

```

Fixpoint Tlookup (i:string) (Tr:ty) : option ty :=
  match Tr with
  | <{{ i' : T :: Tr' }}> =>
    if String.eqb i i' then Some T else Tlookup i Tr'
  | _ => None
  end.

```

```

Fixpoint tlookup (i:string) (tr:tm) : option tm :=
  match tr with
  | <{ i' := t :: tr' }> =>
    if String.eqb i i' then Some t else tlookup i tr'
  | _ => None
  end.

```

Reserved Notation "t '→' t'" (at level 40).

```

Inductive step : tm → tm → Prop :=
| ST_AppAbs : ∀ x T2 t1 v2,
  value v2 →
  <{ (\x:T2, t1) v2 }> -> <{ [x:=v2] t1 }>
| ST_App1 : ∀ t1 t1' t2,
  t1 -> t1' →
  <{ t1 t2 }> -> <{ t1' t2 }>
| ST_App2 : ∀ v1 t2 t2',
  value v1 →
  t2 -> t2' →
  <{ v1 t2 }> -> <{ v1 t2' }>

```

```

| ST_Proj1 :  $\forall t1\ t1'\ i,$ 
   $t1 \rightarrow t1' \rightarrow$ 
   $\langle\{ t1 \rightarrow i \}\rangle \rightarrow \langle\{ t1' \rightarrow i \}\rangle$ 
| ST_ProjRcd :  $\forall tr\ i\ vi,$ 
  value  $tr \rightarrow$ 
   $tlookup\ i\ tr = Some\ vi \rightarrow$ 
   $\langle\{ tr \rightarrow i \}\rangle \rightarrow vi$ 
| ST_Rcd_Head :  $\forall i\ t1\ t1'\ tr2,$ 
   $t1 \rightarrow t1' \rightarrow$ 
   $\langle\{ i := t1 :: tr2 \}\rangle \rightarrow \langle\{ i := t1' :: tr2 \}\rangle$ 
| ST_Rcd_Tail :  $\forall i\ v1\ tr2\ tr2',$ 
  value  $v1 \rightarrow$ 
   $tr2 \rightarrow tr2' \rightarrow$ 
   $\langle\{ i := v1 :: tr2 \}\rangle \rightarrow \langle\{ i := v1 :: tr2' \}\rangle$ 

```

where "t '→' t'" := (step t t').

Hint Constructors step : core.

17.3 Subtyping

Now we come to the interesting part, where the features we've added start to interact. We begin by defining the subtyping relation and developing some of its important technical properties.

17.3.1 Definition

The definition of subtyping is essentially just what we sketched in the discussion of record subtyping in chapter Sub, but we need to add well-formedness side conditions to some of the rules. Also, we replace the “n-ary” width, depth, and permutation subtyping rules by binary rules that deal with just the first field.

Reserved Notation "T '<:' U" (at level 40).

Inductive subtype : ty → ty → Prop :=

```

| S_Refl :  $\forall T,$ 
  well_formed_ty T →
   $T <: T$ 
| S_Trans :  $\forall S\ U\ T,$ 
   $S <: U \rightarrow$ 
   $U <: T \rightarrow$ 
   $S <: T$ 
| S_Top :  $\forall S,$ 

```

```

    well_formed_ty S →
    S <: <{{ Top }}>
| S_Arrow : ∀ S1 S2 T1 T2,
    T1 <: S1 →
    S2 <: T2 →
    <{{ S1 → S2 }}> <: <{{ T1 → T2 }}>

| S_RcdWidth : ∀ i T1 T2,
    well_formed_ty <{{ i : T1 :: T2 }}> →
    <{{ i : T1 :: T2 }}> <: <{{ nil }}>
| S_RcdDepth : ∀ i S1 T1 Sr2 Tr2,
    S1 <: T1 →
    Sr2 <: Tr2 →
    record_ty Sr2 →
    record_ty Tr2 →
    <{{ i : S1 :: Sr2 }}> <: <{{ i : T1 :: Tr2 }}>
| S_RcdPerm : ∀ i1 i2 T1 T2 Tr3,
    well_formed_ty <{{ i1 : T1 :: i2 : T2 :: Tr3 }}> →
    i1 ≠ i2 →
    <{{ i1 : T1 :: i2 : T2 :: Tr3 }}>
    <: <{{ i2 : T2 :: i1 : T1 :: Tr3 }}>

```

where "T '<:' U" := (subtype T U).

Hint Constructors subtype : core.

17.3.2 Examples

Module EXAMPLES.

Open Scope *string_scope*.

Notation x := "x".

Notation y := "y".

Notation z := "z".

Notation j := "j".

Notation k := "k".

Notation i := "i".

Notation A := <{{ Base "A" }}>.

Notation B := <{{ Base "B" }}>.

Notation C := <{{ Base "C" }}>.

Definition TRcd_j :=

<{{ j : (B → B) :: nil }}>. Definition TRcd_kj :=

<{{ k : (A → A) :: TRcd_j }}>.

Example subtyping_example_0 :
 $\langle\{\{ C \rightarrow \text{TRcd\_kj} \}\}\rangle <: \langle\{\{ C \rightarrow \text{nil} \}\}\rangle.$

Proof.

 apply S_Arrow.
 apply S_Refl. auto.
 unfold TRcd_kj, TRcd_j. apply S_RcdWidth; auto.

Qed.

The following facts are mostly easy to prove in Rocq. To get full benefit, make sure you also understand how to prove them on paper!

Exercise: 2 stars, standard (subtyping_example_1) Example subtyping_example_1 :
 :

 TRcd_kj <: TRcd_j.

Proof with eauto.

Admitted.

□

Exercise: 1 star, standard (subtyping_example_2) Example subtyping_example_2 :
 :

$\langle\{\{ \text{Top} \rightarrow \text{TRcd\_kj} \}\}\rangle <:$
 $\langle\{\{ (C \rightarrow C) \rightarrow \text{TRcd\_j} \}\}\rangle.$

Proof with eauto.

Admitted.

□

Exercise: 1 star, standard (subtyping_example_3) Example subtyping_example_3 :
 :

$\langle\{\{ \text{nil} \rightarrow (j : A :: \text{nil}) \}\}\rangle <:$
 $\langle\{\{ (k : B :: \text{nil}) \rightarrow \text{nil} \}\}\rangle.$

Proof with eauto.

Admitted.

□

Exercise: 2 stars, standard (subtyping_example_4) Example subtyping_example_4 :
 :

$\langle\{\{ x : A :: y : B :: z : C :: \text{nil} \}\}\rangle <:$
 $\langle\{\{ z : C :: y : B :: x : A :: \text{nil} \}\}\rangle.$

Proof with eauto.

Admitted.

□

End EXAMPLES.

17.3.3 Properties of Subtyping

Well-Formedness

To get started proving things about subtyping, we need a couple of technical lemmas that intuitively (1) allow us to extract the well-formedness assumptions embedded in subtyping derivations and (2) record the fact that fields of well-formed record types are themselves well-formed types.

```
Lemma subtype_wf : ∀ S T,
  subtype S T →
  well_formed_ty T ∧ well_formed_ty S.
Proof with eauto.
  intros S T Hsub.
  induction Hsub;
    intros; try (destruct IHsub1; destruct IHsub2)...
  -
    split... inversion H. subst. inversion H5... Qed.
```

```
Lemma wf_rcd_lookup : ∀ i T Ti,
  well_formed_ty T →
  Tlookup i T = Some Ti →
  well_formed_ty Ti.
Proof with eauto.
  intros i T.
  induction T; intros; try solve_by_invert.
  -
    inversion H. subst. unfold Tlookup in H0.
    destruct (String.eqb i s)... inversion H0; subst... Qed.
```

Field Lookup

The record matching lemmas get a little more complicated in the presence of subtyping, for two reasons. First, record types no longer necessarily describe the exact structure of the corresponding terms. And second, reasoning by induction on typing derivations becomes harder in general, because typing is no longer syntax directed.

```
Lemma rcd_types_match : ∀ S T i Ti,
  subtype S T →
  Tlookup i T = Some Ti →
  ∃ Si, Tlookup i S = Some Si ∧ subtype Si Ti.
Proof with (eauto using wf_rcd_lookup).
  intros S T i Ti Hsub Hget. generalize dependent Ti.
  induction Hsub; intros Ti Hget;
    try solve_by_invert.
  -
```

```

  ∃ Ti...
-
  destruct (IHHsub2 Ti) as [Ui Hui]... destruct Hui.
  destruct (IHHsub1 Ui) as [Si Hsi]... destruct Hsi.
  ∃ Si...
-
  rename i0 into k.
  unfold Tlookup. unfold Tlookup in Hget.
  destruct (String.eqb i k)...
+
  inversion Hget. subst. ∃ S1...
-
  ∃ Ti. split.
+
  unfold Tlookup. unfold Tlookup in Hget.
  destruct (eqb_spec i i1)...
  ×
  destruct (eqb_spec i i2)...
  destruct H0.
  subst...
+
  inversion H. subst. inversion H5. subst... Qed.

```

Exercise: 3 stars, standard (rcd_types_match_informal) Write a careful informal proof of the `rcd_types_match` lemma.

Definition `manual_grade_for_rcd_types_match_informal` : `option (nat×string)` := `None`.

□

Inversion Lemmas

Exercise: 3 stars, standard, optional (sub_inversion_arrow) Lemma `sub_inversion_arrow`

```

: ∀ U V1 V2,
  U <: <{{ V1 → V2 }}> →
  ∃ U1 U2,
    (U = <{{ U1 → U2 }}>) ∧ (V1 <: U1) ∧ (U2 <: V2).

```

Proof with `eauto`.

```

intros U V1 V2 Hs.
remember <{{ V1 → V2 }}> as V.
generalize dependent V2. generalize dependent V1.
Admitted.

```

□

17.4 Typing

Definition `context` := `partial_map ty`.

Reserved Notation "`Gamma` '`|-`' `t` '`\in`' `T`" (at level 40,
`t` custom stlc at level 99, `T` custom stlc_ty
 at level 0).

Inductive `has_type` : `context` $\rightarrow$ `tm` $\rightarrow$ `ty` $\rightarrow$ Prop :=

```

| T_Var :  $\forall$  Gamma (x : string) T,
  Gamma x = Some T  $\rightarrow$ 
  well_formed_ty T  $\rightarrow$ 
  Gamma |- x \in T
| T_Abs :  $\forall$  Gamma x T11 T12 t12,
  well_formed_ty T11  $\rightarrow$ 
  (x  $\mapsto$  T11; Gamma) |- t12 \in T12  $\rightarrow$ 
  Gamma |- ( $\lambda$  x : T11, t12) \in (T11  $\rightarrow$  T12)
| T_App :  $\forall$  T1 T2 Gamma t1 t2,
  Gamma |- t1 \in (T1  $\rightarrow$  T2)  $\rightarrow$ 
  Gamma |- t2 \in T1  $\rightarrow$ 
  Gamma |- t1 t2 \in T2
| T_Proj :  $\forall$  Gamma i t T Ti,
  Gamma |- t \in T  $\rightarrow$ 
  Tlookup i T = Some Ti  $\rightarrow$ 
  Gamma |- t  $\rightarrow$  i \in Ti

| T_Sub :  $\forall$  Gamma t S T,
  Gamma |- t \in S  $\rightarrow$ 
  subtype S T  $\rightarrow$ 
  Gamma |- t \in T

| T_RNil :  $\forall$  Gamma,
  Gamma |- nil \in nil
| T_RCons :  $\forall$  Gamma i t T tr Tr,
  Gamma |- t \in T  $\rightarrow$ 
  Gamma |- tr \in Tr  $\rightarrow$ 
  record_ty Tr  $\rightarrow$ 
  record_tm tr  $\rightarrow$ 
  Gamma |- i := t :: tr \in (i : T :: Tr)

```

where "`Gamma` '`|-`' `t` '`\in`' `T`" := (`has_type` *Gamma* *t* *T*).

Hint Constructors `has_type` : core.

17.4.1 Typing Examples

Module EXAMPLES2.

Import EXAMPLES.

Exercise: 1 star, standard (typing_example_0) Definition trcd_kj :=
<{ k := ($\lambda z : A, z$) :: j := ($\lambda z : B, z$) :: nil }>.

Example typing_example_0 :

empty |- trcd_kj \in TRcd_kj.

Proof.

Admitted.

□

Exercise: 2 stars, standard (typing_example_1) Example typing_example_1 :
empty |- ($\lambda x : \text{TRcd\_j}, x \rightarrow j$) trcd_kj \in ($B \rightarrow B$).

Proof with eauto.

Admitted.

□

Exercise: 2 stars, standard, optional (typing_example_2) Example typing_example_2
:

empty |- ($\lambda z : (C \rightarrow C) \rightarrow \text{TRcd\_j}, (z (\lambda x : C, x)) \rightarrow j$)
($\lambda z : (C \rightarrow C), \text{trcd\_kj}$) \in ($B \rightarrow B$).

Proof with eauto.

Admitted.

□

End EXAMPLES2.

17.4.2 Properties of Typing

Well-Formedness

Lemma has_type_wf : $\forall$ Gamma t T,
has_type Gamma t T $\rightarrow$ well_formed_ty T.

Proof with eauto.

intros Gamma t T Htyp.

induction Htyp...

-

inversion IHHtyp1...

-

eapply wf_red_lookup...

-


```

    apply subtype_wf in H.
    destruct H...
Qed.

Lemma step_preserves_record_tm :  $\forall$  tr tr',
  record_tm tr  $\rightarrow$ 
  tr  $\rightarrow$  tr'  $\rightarrow$ 
  record_tm tr'.
Proof.
  intros tr tr' Hrt Hstp.
  inversion Hrt; subst; inversion Hstp; subst; eauto.
Qed.

```

Field Lookup

```

Lemma lookup_field_in_value :  $\forall$  v T i Ti,
  value v  $\rightarrow$ 
  empty  $\vdash$  v  $\backslash$ in T  $\rightarrow$ 
  Tlookup i T = Some Ti  $\rightarrow$ 
   $\exists$  vi, tlookup i v = Some vi  $\wedge$  empty  $\vdash$  vi  $\backslash$ in Ti.
Proof with eauto.
  remember empty as Gamma.
  intros t T i Ti Hval Htyp. generalize dependent Ti.
  induction Htyp; intros; subst; try solve_by_invert.
-
  apply (rcd_types_match S) in H0...
  destruct H0 as [Si [HgetSi Hsub]].
  eapply IHHtyp in HgetSi...
  destruct HgetSi as [vi [Hget Htyvi]]...
-
  simpl in H0. simpl. simpl in H1.
  destruct (String.eqb i i0).
+
  injection H1 as H1. subst.  $\exists$  t...
+
  eapply IHHtyp2 in H1...
  inversion Hval... Qed.

```

Progress

Exercise: 3 stars, standard (canonical_forms_of_arrow_types) Lemma canonical_forms_of_arrow_

```

:  $\forall$  Gamma s T1 T2,
  Gamma  $\vdash$  s  $\backslash$ in (T1  $\rightarrow$  T2)  $\rightarrow$ 
  value s  $\rightarrow$ 

```

```

    ∃ x S1 s2,
      s = <{ \ x : S1, s2 }>.
Proof with eauto.
  Admitted.
□

Theorem progress : ∀ t T,
  empty |- t \in T →
  value t ∨ ∃ t', t -> t'.
Proof with eauto.
  intros t T Ht.
  remember empty as Gamma.
  revert HeqGamma.
  induction Ht;
    intros HeqGamma; subst...
-
  inversion H.
-
  right.
  destruct IHHt1; subst...
+
  destruct IHHt2; subst...
×
  destruct (canonical_forms_of_arrow_types empty t1 T1 T2)
    as [x [S1 [t12 Heqt1]]]...
  subst. ∃ <{ [x:=t2] t12 }>...
×
  destruct H0 as [t2' Hstp]. ∃ <{ t1 t2' }> ...
+
  destruct H as [t1' Hstp]. ∃ <{ t1' t2 }>...
-
  right. destruct IHHt...
+
  destruct (lookup_field_in_value t T i Ti)
    as [t' [Hget Ht']]...
+
  destruct H0 as [t' Hstp]. ∃ <{ t' -> i }>...
-
  destruct IHHt1...
+
  destruct IHHt2...
×
  right. destruct H2 as [tr' Hstp].

```

$\exists \langle \{ i := t :: tr' \} \rangle \dots$
 +
`right. destruct H1 as [t' Hstp].`
 $\exists \langle \{ i := t' :: tr \} \rangle \dots$ Qed.

Theorem : For any term t and type T , if $\text{empty} \vdash t : T$ then t is a value or $t \rightarrow t'$ for some term t' .

Proof: Let t and T be given such that $\text{empty} \vdash t : T$. We proceed by induction on the given typing derivation.

- The cases where the last step in the typing derivation is T_Abs or T_RNil are immediate because abstractions and $\{\}$ are always values. The case for T_Var is vacuous because variables cannot be typed in the empty context.
- If the last step in the typing derivation is by T_App , then there are terms $t1\ t2$ and types $T1\ T2$ such that $t = t1\ t2$, $T = T2$, $\text{empty} \vdash t1 : T1 \rightarrow T2$ and $\text{empty} \vdash t2 : T1$.

The induction hypotheses for these typing derivations yield that $t1$ is a value or steps, and that $t2$ is a value or steps.

- Suppose $t1 \rightarrow t1'$ for some term $t1'$. Then $t1\ t2 \rightarrow t1'\ t2$ by ST_App1 .
- Otherwise $t1$ is a value.
 - * Suppose $t2 \rightarrow t2'$ for some term $t2'$. Then $t1\ t2 \rightarrow t1\ t2'$ by rule ST_App2 because $t1$ is a value.
 - * Otherwise, $t2$ is a value. By Lemma *canonical_forms_for_arrow_types*, $t1 = \backslash x:S1, s2$ for some x , $S1$, and $s2$. But then $(\backslash x:S1, s2)\ t2 \rightarrow [x:=t2]s2$ by ST_AppAbs , since $t2$ is a value.

- If the last step of the derivation is by T_Proj , then there are a term tr , a type Tr , and a label i such that $t = tr.i$, $\text{empty} \vdash tr : Tr$, and $Tlookup\ i\ Tr = \text{Some } T$.

By the IH, either tr is a value or it steps. If $tr \rightarrow tr'$ for some term tr' , then $tr.i \rightarrow tr'.i$ by rule ST_Proj1 .

If tr is a value, then Lemma *lookup_field_in_value* yields that there is a term ti such that $tlookup\ i\ tr = \text{Some } ti$. It follows that $tr.i \rightarrow ti$ by rule $ST\_ProjRcd$.

- If the final step of the derivation is by T_Sub , then there is a type S such that $S <: T$ and $\text{empty} \vdash t : S$. The desired result is exactly the induction hypothesis for the typing subderivation.
- If the final step of the derivation is by T_RCons , then there exist some terms $t1\ tr$, types $T1\ Tr$ and a label t such that $t = \{i=t1, tr\}$, $T = \{i:T1, Tr\}$, `record_ty` tr , `record_tm` Tr , $\text{empty} \vdash t1 : T1$ and $\text{empty} \vdash tr : Tr$.

The induction hypotheses for these typing derivations yield that $t1$ is a value or steps, and that tr is a value or steps. We consider each case:

- Suppose $t1 \rightarrow t1'$ for some term $t1'$. Then $\{i=t1, tr\} \rightarrow \{i=t1', tr\}$ by rule `ST_Rcd_Head`.
- Otherwise $t1$ is a value.
 - * Suppose $tr \rightarrow tr'$ for some term tr' . Then $\{i=t1, tr\} \rightarrow \{i=t1, tr'\}$ by rule `ST_Rcd_Tail`, since $t1$ is a value.
 - * Otherwise, tr is also a value. So, $\{i=t1, tr\}$ is a value by `v_rcons`.

Inversion Lemma

Lemma `typing_inversion_abs` : $\forall \text{Gamma } x \text{ } S1 \text{ } t2 \text{ } T,$
 $\text{Gamma} \vdash \backslash x : S1, t2 \backslash \text{in } T \rightarrow$
 $(\exists S2, \langle \{ S1 \rightarrow S2 \} \rangle <: T$
 $\wedge (x \mapsto S1; \text{Gamma}) \vdash t2 \backslash \text{in } S2).$

Proof with `eauto`.

```
intros Gamma x S1 t2 T H.
remember <{ \ x : S1, t2 }> as t.
induction H;
  inversion Heqt; subst; intros; try solve_by_invert.
-
  assert (Hwf := has_type_wf _ _ _ H0).
   $\exists T12...$ 
-
  destruct IHhas_type as [S2 [Hsub Hty]]...
  Qed.
```

Lemma `abs_arrow` : $\forall x \text{ } S1 \text{ } s2 \text{ } T1 \text{ } T2,$
 $\text{empty} \vdash \backslash x : S1, s2 \backslash \text{in } (T1 \rightarrow T2) \rightarrow$
 $T1 <: S1$
 $\wedge (x \mapsto S1) \vdash s2 \backslash \text{in } T2.$

Proof with `eauto`.

```
intros x S1 s2 T1 T2 Hty.
apply typing_inversion_abs in Hty.
destruct Hty as [S2 [Hsub Hty]].
apply sub_inversion_arrow in Hsub.
destruct Hsub as [U1 [U2 [Heq [Hsub1 Hsub2]]]].
inversion Heq; subst... Qed.
```

17.4.3 Weakening

The weakening lemma is proved as in pure STLC.

Lemma `weakening` : $\forall \text{Gamma } \text{Gamma}' \text{ } t \text{ } T,$

```

includedin Gamma Gamma' →
Gamma |- t \in T →
Gamma' |- t \in T.

```

Proof.

```

intros Gamma Gamma' t T H Ht.
generalize dependent Gamma'.
induction Ht; eauto using includedin_update.

```

Qed.

```

Lemma weakening_empty : ∀ Gamma t T,
  empty |- t \in T →
  Gamma |- t \in T.

```

Proof.

```

intros Gamma t T.
eapply weakening.
discriminate.

```

Qed.

Preservation

```

Lemma substitution_preserves_typing : ∀ Gamma x U t v T,
  (x |-> U ; Gamma) |- t \in T →
  empty |- v \in U →
  Gamma |- [x:=v]t \in T.

```

Proof.

Proof.

```

intros Gamma x U t v T Ht Hv.
remember (x |-> U ; Gamma) as Gamma'.
generalize dependent Gamma.
induction Ht; intros Gamma' G; simpl; eauto.
-
  rename x0 into y.
  destruct (eqb_spec x y) as [Hxy|Hxy]; subst.
  +
    rewrite update_eq in H.
    injection H as H. subst.
    apply weakening_empty. assumption.
  +
    apply T_Var; [|assumption].
    rewrite update_neq in H; assumption.
-
  rename x0 into y. subst.
  destruct (eqb_spec x y) as [Hxy|Hxy]; apply T_Abs; try assumption.

```

```

+
  subst. rewrite update_shadow in Ht. assumption.
+
  subst. apply IHHt.
  rewrite update_permute; auto.
-
  apply T_RCons; eauto.
  inversion H0; subst; simpl; auto.
Qed.

```

Theorem preservation : $\forall t \ t' \ T,$
 $\text{empty} \vdash t \text{ in } T \rightarrow$
 $t \rightarrow t' \rightarrow$
 $\text{empty} \vdash t' \text{ in } T.$

Proof with eauto.

```

intros t t' T HT. generalize dependent t'.
remember empty as Gamma.
induction HT;
  intros t' HE; subst;
  try solve [inversion HE; subst; eauto].
-
  inversion HE; subst...
+
  destruct (abs_arrow _ _ _ _ HT1) as [HA1 HA2].
  apply substitution_preserves_typing with T0...
-
  inversion HE; subst...
  destruct (lookup_field_in_value _ _ _ H2 HT H)
    as [vi [Hget Hty]].
  rewrite H4 in Hget. inversion Hget. subst...
-
  inversion HE; subst...
  eauto using step_preserves_record_tm. Qed.

```

Theorem: If t, t' are terms and T is a type such that $\text{empty} \vdash t : T$ and $t \rightarrow t'$, then $\text{empty} \vdash t' : T$.

Proof: Let t and T be given such that $\text{empty} \vdash t : T$. We go by induction on the structure of this typing derivation, leaving t' general. Cases T_Abs and T_RNil are vacuous because abstractions and $\{\}$ don't step. Case T_Var is vacuous as well, since the context is empty.

- If the final step of the derivation is by T_App , then there are terms $t1 \ t2$ and types $T1 \ T2$ such that $t = t1 \ t2$, $T = T2$, $\text{empty} \vdash t1 : T1 \rightarrow T2$ and $\text{empty} \vdash t2 : T1$.
 By inspection of the definition of the step relation, there are three ways $t1 \ t2$ can step.

Cases **ST_App1** and **ST_App2** follow immediately by the induction hypotheses for the typing subderivations and a use of **T_App**.

Suppose instead $t1 \ t2$ steps by **ST_AppAbs**. Then $t1 = \backslash x:S, t12$ for some type S and term $t12$, and $t' = [x:=t2]t12$.

By Lemma **abs_arrow**, we have $T1 <: S$ and $x:S1 \vdash s2 : T2$. It then follows by lemma **substitution_preserves_typing** that $\text{empty} \vdash [x:=t2] t12 : T2$ as desired.

- If the final step of the derivation is by **T_Proj**, then there is a term tr , type Tr and label i such that $t = tr.i$, $\text{empty} \vdash tr : Tr$, and $\text{Tlookup } i \ Tr = \text{Some } T$.

The IH for the typing derivation gives us that, for any term tr' , if $tr \rightarrow tr'$ then $\text{empty} \vdash tr' : Tr$. Inspection of the definition of the step relation reveals that there are two ways a projection can step. Case **ST_Proj1** follows immediately by the IH.

Instead suppose $tr \rightarrow i$ steps by **ST_ProjRcd**. Then tr is a value and there is some term vi such that $\text{tlookup } i \ tr = \text{Some } vi$ and $t' = vi$. But by lemma **lookup_field_in_value**, $\text{empty} \vdash vi : Ti$ as desired.

- If the final step of the derivation is by **T_Sub**, then there is a type S such that $S <: T$ and $\text{empty} \vdash t : S$. The result is immediate by the induction hypothesis for the typing subderivation and an application of **T_Sub**.
- If the final step of the derivation is by **T_RCons**, then there exist some terms $t1 \ tr$, types $T1 \ Tr$ and a label t such that $t = i:=t1 :: tr$, $T = i:T1 :: Tr$, **record_ty** tr , **record_tm** Tr , $\text{empty} \vdash t1 : T1$ and $\text{empty} \vdash tr : Tr$.

By the definition of the step relation, t must have stepped by **ST_Rcd_Head** or **ST_Rcd_Tail**.

In the first case, the result follows by the IH for $t1$'s typing derivation and **T_RCons**.

In the second case, the result follows by the IH for tr 's typing derivation, **T_RCons**, and a use of the **step_preserves_record_tm** lemma.

End **RECORDSUB**.

Chapter 18

Library **PLF.Norm**

18.1 Norm: Normalization of STLC

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import List.
From Stdlib Require Import Strings.String.
From PLF Require Import Maps.
From PLF Require Import Smallstep.
Hint Constructors multi : core.
```

This optional chapter is based on chapter 12 of *Types and Programming Languages* (Pierce). It may be useful to look at the two together, as that chapter includes explanations and informal proofs that are not repeated here.

In this chapter, we consider another fundamental theoretical property of the simply typed lambda-calculus: the fact that the evaluation of a well-typed program is guaranteed to halt in a finite number of steps—i.e., every well-typed term is *normalizable*.

Unlike the type-safety properties we have considered so far, the normalization property does not extend to full-blown programming languages, because these languages nearly always extend the simply typed lambda-calculus with constructs, such as general recursion (see the **MoreStlc** chapter) or recursive types, that can be used to write nonterminating programs. However, the issue of normalization reappears at the level of *types* when we consider the metatheory of polymorphic versions of the lambda calculus such as System F-omega: in this system, the language of types effectively contains a copy of the simply typed lambda-calculus, and the termination of the typechecking algorithm will hinge on the fact that a “normalization” operation on type expressions is guaranteed to terminate.

Another reason for studying normalization proofs is that they are some of the most beautiful—and mind-blowing—mathematics to be found in the type theory literature, often (as here) involving the fundamental proof technique of *logical relations*.

The calculus we shall consider here is the simply typed lambda-calculus over a single base type **bool** and with pairs. We’ll give most details of the development for the basic lambda-

calculus terms treating **bool** as an uninterpreted base type, and leave the extension to the boolean operators and pairs to the reader. Even for the base calculus, normalization is not entirely trivial to prove, since each reduction of a term can duplicate redexes in subterms.

Exercise: 2 stars, standard (norm_fail) Where do we fail if we attempt to prove normalization by a straightforward induction on the size of a well-typed term?

Definition `manual_grade_for_norm_fail` : `option (nat × string)` := `None`.

□

Exercise: 5 stars, standard, especially useful (norm) The best ways to understand an intricate proof like this is are (1) to help fill it in and (2) to extend it. We've left out some parts of the following development, including some proofs of lemmas and the all the cases involving products and conditionals. Fill them in.

Definition `manual_grade_for_norm` : `option (nat × string)` := `None`.

□

18.2 Language

We begin by repeating the relevant language definition, which is similar to those in the `MoreStlc` chapter, plus supporting results including type preservation and step determinism. (We won't need progress.) You may just wish to skip down to the Normalization section...

Syntax and Operational Semantics

Inductive `ty` : `Type` :=

```
| Ty_Bool : ty
| Ty_Arrow : ty → ty → ty
| Ty_Prod : ty → ty → ty
```

.

Inductive `tm` : `Type` :=

```
| tm_var : string → tm
| tm_app : tm → tm → tm
| tm_abs : string → ty → tm → tm
```

```
| tm_true : tm
| tm_false : tm
| tm_if : tm → tm → tm → tm
```

```
| tm_pair : tm → tm → tm
```

```

| tm_fst : tm → tm
| tm_snd : tm → tm.

```

Declare Custom Entry stlc.

```

Notation "<{ e }>" := e (e custom stlc at level 99).
Notation "( x )" := x (in custom stlc, x at level 99).
Notation "x" := x (in custom stlc at level 0, x constr at level 0).
Notation "S-> T" := (Ty_Arrow S T) (in custom stlc at level 50, right associativity).
Notation "x y" := (tm_app x y) (in custom stlc at level 1, left associativity).
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc at level 90, x at level 99,
                  t custom stlc at level 99,
                  y custom stlc at level 99,
                  left associativity).

```

Coercion tm_var : string >-> tm.

```

Notation "{ x }" := x (in custom stlc at level 1, x constr).

```

```

Notation "'Bool'" := Ty_Bool (in custom stlc at level 0).

```

```

Notation "'if' x 'then' y 'else' z" :=
  (tm_if x y z) (in custom stlc at level 89,
                  x custom stlc at level 99,
                  y custom stlc at level 99,
                  z custom stlc at level 99,
                  left associativity).

```

```

Notation "'true'" := true (at level 1).

```

```

Notation "'true'" := tm_true (in custom stlc at level 0).

```

```

Notation "'false'" := false (at level 1).

```

```

Notation "'false'" := tm_false (in custom stlc at level 0).

```

```

Notation "X * Y" :=
  (Ty_Prod X Y) (in custom stlc at level 2, X custom stlc, Y custom stlc at level 0).

```

```

Notation "( x ', y )" := (tm_pair x y) (in custom stlc at level 0,
                                         x custom stlc at level 99,
                                         y custom stlc at level 99).

```

```

Notation "t '.fst'" := (tm_fst t) (in custom stlc at level 1).

```

```

Notation "t '.snd'" := (tm_snd t) (in custom stlc at level 1).

```

Substitution

```

Reserved Notation "'[ x ' := s ]' t" (in custom stlc at level 20, x constr).

```

```

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  | tm_var y =>
    if String.eqb x y then s else t

```

```

| <{ \ y : T, t1 }> =>
  if String.eqb x y then t else <{ \y:T, [x:=s] t1 }>
| <{t1 t2}> =>
  <{ ([x:=s] t1) ([x:=s] t2) }>
| <{true}> =>
  <{true}>
| <{false}> =>
  <{false}>
| <{if t1 then t2 else t3}> =>
  <{if ([x:=s] t1) then ([x:=s] t2) else ([x:=s] t3)}>
| <{(t1, t2)}> =>
  <{ ([x:=s] t1), ([x:=s] t2) }>
| <{t0.fst}> =>
  <{ ([x:=s] t0).fst }>
| <{t0.snd}> =>
  <{ ([x:=s] t0).snd }>
end

where "'[ x := s ]' t" := (subst x s t) (in custom stlc).

```

Reduction

Inductive **value** : **tm** → Prop :=

```

| v_abs : ∀ x T2 t1,
  value <{ \x:T2, t1 }>
| v_true :
  value <{true}>
| v_false :
  value <{false}>
| v_pair : ∀ v1 v2,
  value v1 →
  value v2 →
  value <{(v1, v2)}>.

```

Hint Constructors **value** : *core*.

Reserved Notation "t '→' t'" (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

```

| ST_AppAbs : ∀ x T2 t1 v2,
  value v2 →
  <{ (\x:T2, t1) v2 }> -> <{ [x:=v2] t1 }>
| ST_App1 : ∀ t1 t1' t2,
  t1 -> t1' →
  <{t1 t2}> -> <{t1' t2}>

```

```

| ST_App2 : ∀ v1 t2 t2',
  value v1 →
  t2 -> t2' →
  <{v1 t2}> -> <{v1 t2'}>
| ST_IfTrue : ∀ t1 t2,
  <{if true then t1 else t2}> -> t1
| ST_IfFalse : ∀ t1 t2,
  <{if false then t1 else t2}> -> t2
| ST_If : ∀ t1 t1' t2 t3,
  t1 -> t1' →
  <{if t1 then t2 else t3}> -> <{if t1' then t2 else t3}>
| ST_Pair1 : ∀ t1 t1' t2,
  t1 -> t1' →
  <{ (t1, t2) }> -> <{ (t1' , t2) }>
| ST_Pair2 : ∀ v1 t2 t2',
  value v1 →
  t2 -> t2' →
  <{ (v1 , t2) }> -> <{ (v1 , t2') }>
| ST_Fst1 : ∀ t0 t0',
  t0 -> t0' →
  <{ t0.fst }> -> <{ t0'.fst }>
| ST_FstPair : ∀ v1 v2,
  value v1 →
  value v2 →
  <{ (v1, v2).fst }> -> v1
| ST_Snd1 : ∀ t0 t0',
  t0 -> t0' →
  <{ t0.snd }> -> <{ t0'.snd }>
| ST_SndPair : ∀ v1 v2,
  value v1 →
  value v2 →
  <{ (v1, v2).snd }> -> v2

```

where "t '→' t'" := (step t t').

Hint Constructors step : core.

Notation multistep := (multi step).

Notation "t1 '→'* t2" := (multistep t1 t2) (at level 40).

Notation step_normal_form := (normal_form step).

Lemma value__normal : ∀ t, value t → step_normal_form t.

Proof with eauto.

intros t H; induction H; intros [t' ST]; inversion ST...

Qed.

Typing

Definition context := partial_map ty.

Reserved Notation "Gamma '|-' t '\in' T" (at level 40,
t custom stlc, T custom stlc at level 0).

Inductive has_type : context → tm → ty → Prop :=

```
| T_Var : ∀ Gamma x T1,  
  Gamma x = Some T1 →  
  Gamma |- x \in T1  
| T_Abs : ∀ Gamma x T1 T2 t1,  
  (x |-> T2 ; Gamma) |- t1 \in T1 →  
  Gamma |- \x:T2, t1 \in (T2 → T1)  
| T_App : ∀ T1 T2 Gamma t1 t2,  
  Gamma |- t1 \in (T2 → T1) →  
  Gamma |- t2 \in T2 →  
  Gamma |- t1 t2 \in T1  
| T_True : ∀ Gamma,  
  Gamma |- true \in Bool  
| T_False : ∀ Gamma,  
  Gamma |- false \in Bool  
| T_If : ∀ t1 t2 t3 T1 Gamma,  
  Gamma |- t1 \in Bool →  
  Gamma |- t2 \in T1 →  
  Gamma |- t3 \in T1 →  
  Gamma |- if t1 then t2 else t3 \in T1  
| T_Pair : ∀ Gamma t1 t2 T1 T2,  
  Gamma |- t1 \in T1 →  
  Gamma |- t2 \in T2 →  
  Gamma |- (t1, t2) \in (T1 × T2)  
| T_Fst : ∀ Gamma t0 T1 T2,  
  Gamma |- t0 \in (T1 × T2) →  
  Gamma |- t0.fst \in T1  
| T_Snd : ∀ Gamma t0 T1 T2,  
  Gamma |- t0 \in (T1 × T2) →  
  Gamma |- t0.snd \in T2
```

where "Gamma '|-' t '\in' T" := (has_type Gamma t T).

Hint Constructors has_type : core.

Hint Extern 2 (**has_type** - (app - -) -) $\Rightarrow$ eapply T_App; auto : *core*.
 Hint Extern 2 (- = -) $\Rightarrow$ compute; reflexivity : *core*.

18.2.1 Weakening

The weakening lemma is proved as in pure STLC.

Lemma weakening : $\forall$ *Gamma Gamma' t T*,
 includedin *Gamma Gamma'* $\rightarrow$
Gamma $\vdash t \text{ \textit{in} } T \rightarrow$
Gamma' $\vdash t \text{ \textit{in} } T$.

Proof.

intros *Gamma Gamma' t T H Ht*.
 generalize dependent *Gamma'*.
 induction *Ht*; eauto using includedin_update.

Qed.

Lemma weakening_empty : $\forall$ *Gamma t T*,
 empty $\vdash t \text{ \textit{in} } T \rightarrow$
Gamma $\vdash t \text{ \textit{in} } T$.

Proof.

intros *Gamma t T*.
 eapply weakening.
 discriminate.

Qed.

Substitution

Lemma substitution_preserves_typing : $\forall$ *Gamma x U t v T*,
 (*x* $\mapsto U$; *Gamma*) $\vdash t \text{ \textit{in} } T \rightarrow$
 empty $\vdash v \text{ \textit{in} } U \rightarrow$
Gamma $\vdash [x:=v]t \text{ \textit{in} } T$.

Proof with eauto.

intros *Gamma x U t v T Ht Hv*.
 generalize dependent *Gamma*. generalize dependent *T*.
 induction *t*; intros *T Gamma H*;

inversion *H*; clear *H*; subst; simpl; eauto.

-

rename *s* into *y*. destruct (eqb_spec *x y*); subst.

+

rewrite update_eq in *H2*.
 injection *H2* as *H2*; subst.
 apply weakening_empty. assumption.

```

+
  apply T_Var. rewrite update_neq in H2; auto.
-
  rename s into y, t into S.
  destruct (eqb_spec x y); subst; apply T_Abs.
+
  rewrite update_shadow in H5. assumption.
+
  apply IHt.
  rewrite update_permute; auto.
Qed.

```

Preservation

```

Theorem preservation :  $\forall t t' T,$ 
  empty  $\vdash t \backslash \text{in } T \rightarrow$ 
   $t \rightarrow t' \rightarrow$ 
  empty  $\vdash t' \backslash \text{in } T$ .
Proof with eauto.
intros t t' T HT. generalize dependent t'.
remember empty as Gamma.
induction HT;
  intros t' HE; subst; inversion HE; subst...
-
  inversion HE; subst...
+
  apply substitution_preserves_typing with T2...
  inversion HT1...
-
  inversion HT...
-
  inversion HT...
Qed.

```

Context Invariance

```

Inductive appears_free_in : string  $\rightarrow$  tm  $\rightarrow$  Prop :=
| afi_var :  $\forall (x : \text{string}),$ 
  appears_free_in x <{ x }>
| afi_app1 :  $\forall x t1 t2,$ 
  appears_free_in x t1  $\rightarrow$  appears_free_in x <{ t1 t2 }>
| afi_app2 :  $\forall x t1 t2,$ 
  appears_free_in x t2  $\rightarrow$  appears_free_in x <{ t1 t2 }>

```

```

| afi_abs : ∀ x y T11 t12,
  y ≠ x →
  appears_free_in x t12 →
  appears_free_in x <{ \y : T11, t12 }>

| afi_test0 : ∀ x t0 t1 t2,
  appears_free_in x t0 →
  appears_free_in x <{ if t0 then t1 else t2 }>
| afi_test1 : ∀ x t0 t1 t2,
  appears_free_in x t1 →
  appears_free_in x <{ if t0 then t1 else t2 }>
| afi_test2 : ∀ x t0 t1 t2,
  appears_free_in x t2 →
  appears_free_in x <{ if t0 then t1 else t2 }>

| afi_pair1 : ∀ x t1 t2,
  appears_free_in x t1 →
  appears_free_in x <{ (t1, t2) }>
| afi_pair2 : ∀ x t1 t2,
  appears_free_in x t2 →
  appears_free_in x <{ (t1, t2) }>
| afi_fst : ∀ x t,
  appears_free_in x t →
  appears_free_in x <{ t.fst }>
| afi_snd : ∀ x t,
  appears_free_in x t →
  appears_free_in x <{ t.snd }>

```

Hint Constructors appears_free_in : core.

Definition closed (t:tm) :=
 ∀ x, ¬ appears_free_in x t.

Lemma context_invariance : ∀ Gamma Gamma' t S,
 Gamma |- t \in S →
 (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
 Gamma' |- t \in S.

Proof.

```

intros.
generalize dependent Gamma'.
induction H; intros; eauto 12.
-
  apply T_Var. rewrite ← H0; auto.
-

```



```

    apply T_Abs.
    apply IHhas_type. intros x1 Hafi.
    destruct (eqb_spec x x1); subst.
    + rewrite update_eq.
      rewrite update_eq.
      reflexivity.
    + rewrite update_neq; [| assumption].
      rewrite update_neq; [| assumption].
      auto.
Qed.

Theorem false_eqb_string :  $\forall x y : \text{string},$ 
   $x \neq y \rightarrow \text{String.eqb } x y = \text{false}.$ 
Proof.
  intros x y. rewrite String.eqb_neq.
  intros H. apply H. Qed.

Lemma free_in_context :  $\forall x t T \text{ Gamma},$ 
  appears_free_in x t  $\rightarrow$ 
  Gamma |- t \in T  $\rightarrow$ 
   $\exists T', \text{Gamma } x = \text{Some } T'.$ 
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  induction Htyp; inversion Hafi; subst...
  -
    destruct IHHtyp as [T' Hctx]...  $\exists T'.$ 
    unfold update, t_update in Hctx.
    rewrite false_eqb_string in Hctx...
Qed.

Corollary typable_empty_closed :  $\forall t T,$ 
  empty |- t \in T  $\rightarrow$ 
  closed t.
Proof.
  intros. unfold closed. intros x H1.
  destruct (free_in_context _ _ _ H1 H) as [T' C].
  discriminate C. Qed.

```

Determinism

To prove determinism, we introduce a helpful tactic. It identifies cases in which a value takes a step and solves them by using `value_normal`.

```

Ltac solve_by_value_nf :=
  match goal with | H : value ?v, H' : ?v -> ?v' ⊢ _ ⇒
    exfalse; apply value_normal in H; eauto

```

```

end.
Lemma step_deterministic :
  deterministic step.
Proof with eauto.
  unfold deterministic.
  intros  $t$   $t'$   $t''$   $E1$   $E2$ .
  generalize dependent  $t''$ .
  induction  $E1$ ; intros  $t''$   $E2$ ; inversion  $E2$ ; subst; clear  $E2$ ;
  try solve_by_invert; try f_equal; try solve_by_value_nf; eauto.
- inversion  $E1$ ; subst; solve_by_value_nf.
- inversion  $H2$ ; subst; solve_by_value_nf.
- inversion  $E1$ ; subst; solve_by_value_nf.
- inversion  $H2$ ; subst; solve_by_value_nf.
Qed.

```

18.3 Normalization

Now for the actual normalization proof.

Our goal is to prove that every well-typed term reduces to a normal form. In fact, it turns out to be convenient to prove something slightly stronger, namely that every well-typed term reduces to a *value*. This follows from the weaker property anyway via Progress (why?) but otherwise we don't need Progress, and we didn't bother re-proving it above.

Here's the key definition:

Definition $\text{halts} (t:\mathbf{tm}) : \text{Prop} := \exists t', t \rightarrow^* t' \wedge \text{value } t'$.

A trivial fact:

Lemma $\text{value\_halts} : \forall v, \text{value } v \rightarrow \text{halts } v$.

Proof.

```

  intros  $v$   $H$ . unfold halts.
   $\exists v$ . split.
- apply multi_refl.
- assumption.

```

Qed.

The key issue in the normalization proof (as in many proofs by induction) is finding a strong enough induction hypothesis. To this end, we begin by defining, for each type $\mathbf{T}$, a set R_T of closed terms of type $\mathbf{T}$. We will specify these sets using a relation $\mathbf{R}$ and write $\mathbf{R} \ \mathbf{T} \ t$ when t is in R_T . (The sets R_T are sometimes called *saturated sets* or *reducibility candidates*.)

Here is the definition of $\mathbf{R}$ for the base language:

- $\mathbf{R} \ \mathbf{bool} \ t$ iff t is a closed term of type **bool** and t halts in a value

- $R (T1 \rightarrow T2) t$ iff t is a closed term of type $T1 \rightarrow T2$ and t halts in a value *and* for any term s such that $R T1 s$, we have $R T2 (t s)$.

This definition gives us the strengthened induction hypothesis that we need. Our primary goal is to show that all *programs* —i.e., all closed terms of base type—halt. But closed terms of base type can contain subterms of functional type, so we need to know something about these as well. Moreover, it is not enough to know that these subterms halt, because the application of a normalized function to a normalized argument involves a substitution, which may enable more reduction steps. So we need a stronger condition for terms of functional type: not only should they halt themselves, but, when applied to halting arguments, they should yield halting results.

The form of R is characteristic of the *logical relations* proof technique. (Since we are just dealing with unary relations here, we could perhaps more properly say *logical properties*.) If we want to prove some property P of all closed terms of type A , we proceed by proving, by induction on types, that all terms of type A *possess* property P , all terms of type $A \rightarrow A$ *preserve* property P , all terms of type $(A \rightarrow A) \rightarrow (A \rightarrow A)$ *preserve the property of preserving* property P , and so on. We do this by defining a family of properties, indexed by types. For the base type A , the property is just P . For functional types, it says that the function should map values satisfying the property at the input type to values satisfying the property at the output type.

When we come to formalize the definition of R in Rocq, we hit a problem. The most obvious formulation would be as a parameterized Inductive proposition like this:

```
Inductive R : ty -> tm -> Prop := | R_bool : forall b t, empty |- t \in Bool -> halts t
-> R Bool t | R_arrow : forall T1 T2 t, empty |- t \in (Arrow T1 T2) -> halts t -> (forall
s, R T1 s -> R T2 (app t s)) -> R (Arrow T1 T2) t.
```

Unfortunately, Rocq rejects this definition because it violates the *strict positivity requirement* for inductive definitions, which says that the type being defined must not occur to the left of an arrow in the type of a constructor argument. Here, it is the third argument to R_arrow , namely $(\forall s, R T1 s \rightarrow R TS (\text{app } t s))$, and specifically the $R T1 s$ part, that violates this rule. (The outermost arrows separating the constructor arguments don't count when applying this rule; otherwise we could never have genuinely inductive properties at all!) The reason for the rule is that types defined with non-positive recursion can be used to build non-terminating functions, which as we know would be a disaster for Rocq's logical soundness. Even though the relation we want in this case might be perfectly innocent, Rocq still rejects it because it fails the positivity test.

Fortunately, it turns out that we *can* define R using a **Fixpoint**:

```
Fixpoint R (T:ty) (t:tm) : Prop :=
  empty |- t \in T ^ halts t ^
  (match T with
  | <{ Bool }> => True
  | <{ T1 -> T2 }> => (forall s, R T1 s -> R T2 <{t s}> )
```

```
| <{ T1 × T2 }> ⇒ False
end).
```

As immediate consequences of this definition, we have that every element of every set R_T halts in a value and is closed with type T :

Lemma `R_halts` : $\forall \{T\} \{t\}, R\ T\ t \rightarrow \text{halts } t$.

Proof.

```
intros.
```

```
destruct T; unfold R in H; destruct H as [- [H _]]; assumption.
```

Qed.

Lemma `R_typable_empty` : $\forall \{T\} \{t\}, R\ T\ t \rightarrow \text{empty} \mid -\ t \setminus \text{in } T$.

Proof.

```
intros.
```

```
destruct T; unfold R in H; destruct H as [H _]; assumption.
```

Qed.

Now we proceed to show the main result, which is that every well-typed term of type T is an element of R_T . Together with `R_halts`, that will show that every well-typed term halts in a value.

18.3.1 Membership in R_T Is Invariant Under Reduction

We start with a preliminary lemma that shows a kind of strong preservation property, namely that membership in R_T is *invariant* under reduction. We will need this property in both directions, i.e., both to show that a term in R_T stays in R_T when it takes a forward step, and to show that any term that ends up in R_T after a step must have been in R_T to begin with.

First of all, an easy preliminary lemma. Note that in the forward direction the proof depends on the fact that our language is deterministic. This lemma might still be true for nondeterministic languages, but the proof would be harder!

Lemma `step_preserves_halting` :

```
  ∀ t t', (t -> t') → (halts t ↔ halts t').
```

Proof.

```
  intros t t' ST. unfold halts.
```

```
  split.
```

```
-
```

```
  intros [t'' [STM V]].
```

```
  destruct STM.
```

```
    + exfalso; apply value__normal in V; eauto.
```

```
    + rewrite (step_deterministic _ _ ST H). ∃ z. split; assumption.
```

```
-
```

```
  intros [t'0 [STM V]].
```

```
  ∃ t'0. split; eauto.
```

Qed.

Now the main lemma, which comes in two parts, one for each direction. Each proceeds by induction on the structure of the type T . In fact, this is where we make fundamental use of the structure of types.

One requirement for staying in R_T is to stay in type T . In the forward direction, we get this from ordinary type Preservation.

Lemma `step_preserves_R` : $\forall T\ t\ t', (t \rightarrow t') \rightarrow R\ T\ t \rightarrow R\ T\ t'$.

Proof.

```
induction T; intros t t' E Rt; unfold R; fold R; unfold R in Rt; fold R in Rt;
  destruct Rt as [typable_empty_t [halts_t RRt]].
split. eapply preservation; eauto.
split. apply (step_preserves_halting _ _ E); eauto.
auto.
split. eapply preservation; eauto.
split. apply (step_preserves_halting _ _ E); eauto.
intros.
eapply IHT2.
apply ST_App1. apply E.
apply RRt; auto.
Admitted.
```

The generalization to multiple steps is trivial:

Lemma `multistep_preserves_R` : $\forall T\ t\ t', (t \rightarrow^* t') \rightarrow R\ T\ t \rightarrow R\ T\ t'$.

Proof.

```
intros T t t' STM; induction STM; intros.
assumption.
apply IHSTM. eapply step_preserves_R. apply H. assumption.
```

Qed.

In the reverse direction, we must add the fact that t has type T before stepping as an additional hypothesis.

Lemma `step_preserves_R'` : $\forall T\ t\ t', \text{empty} \vdash t \text{ \textit{in} } T \rightarrow (t \rightarrow t') \rightarrow R\ T\ t' \rightarrow R\ T\ t$.

Proof.

Admitted.

Lemma `multistep_preserves_R'` : $\forall T\ t\ t', \text{empty} \vdash t \text{ \textit{in} } T \rightarrow (t \rightarrow^* t') \rightarrow R\ T\ t' \rightarrow R\ T\ t$.

Proof.

```
intros T t t' HT STM.
induction STM; intros.
assumption.
eapply step_preserves_R'. assumption. apply H. apply IHSTM.
```

eapply preservation; eauto. auto.
Qed.

18.3.2 Closed Instances of Terms of Type t Belong to R_T

Now we proceed to show that every term of type T belongs to R_T . Here, the induction will be on typing derivations (it would be surprising to see a proof about well-typed terms that did not somewhere involve induction on typing derivations!). The only technical difficulty here is in dealing with the abstraction case. Since we are arguing by induction, the demonstration that a term $\text{abs } x \ T1 \ t2$ belongs to $R\_ (T1 \rightarrow T2)$ should involve applying the induction hypothesis to show that $t2$ belongs to $R\_ (T2)$. But $R\_ (T2)$ is defined to be a set of *closed* terms, while $t2$ may contain x free, so this does not make sense.

This problem is resolved by using a standard trick to suitably generalize the induction hypothesis: instead of proving a statement involving a closed term, we generalize it to cover all closed *instances* of an open term t . Informally, the statement of the lemma will look like this:

If $x1:T1, \dots, xn:Tn \vdash t : T$ and $v1, \dots, vn$ are values such that $R \ T1 \ v1, R \ T2 \ v2, \dots, R \ Tn \ vn$, then $R \ T \ ([x1:=v1][x2:=v2] \dots [xn:=vn]t)$.

The proof will proceed by induction on the typing derivation $x1:T1, \dots, xn:Tn \vdash t : T$; the most interesting case will be the one for abstraction.

Multisubstitutions, Multi-Extensions, and Instantiations

However, before we can proceed to formalize the statement and proof of the lemma, we'll need to build some (rather tedious) machinery to deal with the fact that we are performing *multiple* substitutions on term t and *multiple* extensions of the typing context. In particular, we must be precise about the order in which the substitutions occur and how they act on each other. Often these details are simply elided in informal paper proofs, but of course Rocq won't let us do that. Since here we are substituting closed terms, we don't need to worry about how one substitution might affect the term put in place by another. But we still do need to worry about the *order* of substitutions, because it is quite possible for the same identifier to appear multiple times among the $x1, \dots, xn$ with different associated vi and Ti .

To make everything precise, we will assume that environments are extended from left to right, and multiple substitutions are performed from right to left. To see that this is consistent, suppose we have an environment written as $\dots, y:\text{bool}, \dots, y:\text{nat}, \dots$ and a corresponding term substitution written as $\dots[y:= (\text{tbool true})] \dots [y:= (\text{const } 3)] \dots t$. Since environments are extended from left to right, the binding $y:\text{nat}$ hides the binding $y:\text{bool}$; since substitutions are performed right to left, we do the substitution $y:= (\text{const } 3)$ first, so that the substitution $y:= (\text{tbool true})$ has no effect. Substitution thus correctly preserves the type of the term.

With these points in mind, the following definitions should make sense.

A *multisubstitution* is the result of applying a list of substitutions, which we call an *environment*.

Definition env := list (string × tm).

Fixpoint msubst (ss:env) (t:tm) : tm :=
 match ss with
 | nil ⇒ t
 | ((x,s)::ss') ⇒ msubst ss' <{ [x:=s] t }>
 end.

We need similar machinery to talk about repeated extension of a typing context using a list of (identifier, type) pairs, which we call a *type assignment*.

Definition tass := list (string × ty).

Fixpoint mupdate (Gamma : context) (xts : tass) :=
 match xts with
 | nil ⇒ Gamma
 | ((x,v)::xts') ⇒ update (mupdate Gamma xts') x v
 end.

We will need some simple operations that work uniformly on environments and type assignments

Fixpoint lookup {X:Set} (k : string) (l : list (string × X))
 : option X :=
 match l with
 | nil ⇒ None
 | (j,x) :: l' ⇒
 if String.eqb j k then Some x else lookup k l'
 end.

Fixpoint drop {X:Set} (n:string) (nxs:list (string × X))
 : list (string × X) :=
 match nxes with
 | nil ⇒ nil
 | ((n',x)::nxs') ⇒
 if String.eqb n' n then drop n nxes'
 else (n',x)::(drop n nxes')
 end.

An *instantiation* combines a type assignment and a value environment with the same domains, where corresponding elements are in R.

Inductive instantiation : tass → env → Prop :=
 | V_nil :
 instantiation nil nil
 | V_cons : ∀ x T v c e,
 value v → R T v →
 instantiation c e →
 instantiation ((x,T)::c) ((x,v)::e).

We now proceed to prove various properties of these definitions.

More Substitution Facts

First we need some additional lemmas on (ordinary) substitution.

Lemma `vacuous_substitution` : $\forall t\ x,$
 $\neg \text{appears\_free\_in } x\ t \rightarrow$
 $\forall t', \langle \{ [x:=t']\} t \rangle = t.$

Proof with `eauto`.

Admitted.

Lemma `subst_closed`: $\forall t,$
 $\text{closed } t \rightarrow$
 $\forall x\ t', \langle \{ [x:=t']\} t \rangle = t.$

Proof.

`intros. apply vacuous_substitution. apply H. Qed.`

Lemma `subst_not_afi` : $\forall t\ x\ v,$
 $\text{closed } v \rightarrow \neg \text{appears\_free\_in } x\ \langle \{ [x:=v]\} t \rangle.$

Proof with `eauto`. `unfold closed, not.`

`induction t; intros x v P A; simpl in A.`

```
-
  destruct (eqb_spec x s)...
  inversion A; subst. auto.
-
  inversion A; subst...
-
  destruct (eqb_spec x s)...
  + inversion A; subst...
  + inversion A; subst...
-
  inversion A.
-
  inversion A.
-
  inversion A; subst...
-
  inversion A; subst...
-
  inversion A; subst...
-
  inversion A; subst...
```

`Qed.`

Lemma `duplicate_subst` : $\forall t'\ x\ t\ v,$


```

  closed  $v \rightarrow \langle \{ [x:=t] ([x:=v] t') \} \rangle = \langle \{ [x:=v] t' \} \rangle$ .
Proof.
  intros. eapply vacuous_substitution. apply subst_not_af. assumption.
Qed.

Lemma swap_subst :  $\forall t\ x\ x1\ v\ v1$ ,
   $x \neq x1 \rightarrow$ 
  closed  $v \rightarrow$  closed  $v1 \rightarrow$ 
   $\langle \{ [x1:=v1] ([x:=v] t) \} \rangle = \langle \{ [x:=v] ([x1:=v1] t) \} \rangle$ .
Proof with eauto.
  induction  $t$ ; intros; simpl.
  -
    destruct (eqb_spec  $x\ s$ ); destruct (eqb_spec  $x1\ s$ ).
    + subst. ex falso...
    + subst. simpl. rewrite String.eqb_refl. apply subst_closed...
    + subst. simpl. rewrite String.eqb_refl. rewrite subst_closed...
    + simpl. rewrite false_eqb_string... rewrite false_eqb_string...
    Admitted.

```

Properties of Multi-Substitutions

Lemma msubst_closed: $\forall t$, closed $t \rightarrow \forall ss$, msubst $ss\ t = t$.

```

Proof.
  induction  $ss$ .
  reflexivity.
  destruct  $a$ . simpl. rewrite subst_closed; assumption.
Qed.

```

Closed environments are those that contain only closed terms.

```

Fixpoint closed_env ( $env:env$ ) :=
  match  $env$  with
  | nil  $\Rightarrow$  True
  |  $(x, t)::env' \Rightarrow$  closed  $t \wedge$  closed_env  $env'$ 
  end.

```

Next come a series of lemmas characterizing how msubst of closed terms distributes over subst and over each term form

Lemma subst_msubst: $\forall env\ x\ v\ t$, closed $v \rightarrow$ closed_env $env \rightarrow$
 msubst $env\ \langle \{ [x:=v] t \} \rangle = \langle \{ [x:=v] \{ msubst (drop\ x\ env)\ t \} \} \rangle$.

```

Proof.
  induction  $env0$ ; intros; auto.
  destruct  $a$ . simpl.
  inversion  $H0$ .
  destruct (eqb_spec  $s\ x$ ).

```

```
- subst. rewrite duplicate_subst; auto.
- simpl. rewrite swap_subst; eauto.
```

Qed.

```
Lemma msubst_var:  $\forall ss\ x, \text{closed\_env } ss \rightarrow$ 
  msubst  $ss$  (tm_var  $x$ ) =
  match lookup  $x\ ss$  with
  | Some  $t \Rightarrow t$ 
  | None  $\Rightarrow$  tm_var  $x$ 
end.
```

Proof.

```
induction  $ss$ ; intros.
  reflexivity.
  destruct  $a$ .
  simpl. destruct (String.eqb  $s\ x$ ).
  apply msubst_closed. inversion  $H$ ; auto.
  apply IH $ss$ . inversion  $H$ ; auto.
```

Qed.

```
Lemma msubst_abs:  $\forall ss\ x\ T\ t,$ 
  msubst  $ss$   $\langle \{ \backslash x : T, t \} \rangle = \langle \{ \backslash x : T, \{\text{msubst } (drop\ x\ ss)\ t\} \} \rangle$ .
```

Proof.

```
induction  $ss$ ; intros.
  reflexivity.
  destruct  $a$ .
  simpl. destruct (String.eqb  $s\ x$ ); simpl; auto.
```

Qed.

```
Lemma msubst_app :  $\forall ss\ t1\ t2,$ 
  msubst  $ss$   $\langle \{ t1\ t2 \} \rangle = \langle \{ \{\text{msubst } ss\ t1\} (\{\text{msubst } ss\ t2\}) \} \rangle$ .
```

Proof.

```
induction  $ss$ ; intros.
  reflexivity.
  destruct  $a$ .
  simpl. rewrite  $\leftarrow$  IH $ss$ . auto.
```

Qed.

You'll need similar functions for the other term constructors.

Properties of Multi-Extensions

We need to connect the behavior of type assignments with that of their corresponding contexts.

```
Lemma mupdate_lookup :  $\forall (c : \text{tass}) (x:\text{string}),$ 
  lookup  $x\ c = (\text{mupdate empty } c)\ x$ .
```

```

Proof.
  induction c; intros.
  auto.
  destruct a. unfold lookup, mupdate, update, t_update. destruct (String.eqb s x);
auto.
Qed.

Lemma mupdate_drop :  $\forall (c: \text{tass}) \text{Gamma } x \ x',$ 
  mupdate Gamma (drop x c) x'
  = if String.eqb x x' then Gamma x' else mupdate Gamma c x'.
Proof.
  induction c; intros.
  - destruct (eqb_spec x x'); auto.
  - destruct a. simpl.
    destruct (eqb_spec s x).
    + subst. rewrite IHc.
      unfold update, t_update. destruct (eqb_spec x x'); auto.
    + simpl. unfold update, t_update. destruct (eqb_spec s x'); auto.
      subst. rewrite false_eqb_string; congruence.
Qed.

```

Properties of Instantiations

These are straightforward.

```

Lemma instantiation_domains_match:  $\forall \{c\} \{e\},$ 
  instantiation c e  $\rightarrow$ 
   $\forall \{x\} \{T\},$ 
  lookup x c = Some T  $\rightarrow \exists t, \text{lookup } x \ e = \text{Some } t.$ 

```

```

Proof.
  intros c e V. induction V; intros x0 T0 C.
  solve_by_invert.
  simpl in *.
  destruct (String.eqb x x0); eauto.

```

Qed.

```

Lemma instantiation_env_closed :  $\forall c \ e,$ 
  instantiation c e  $\rightarrow \text{closed\_env } e.$ 

```

```

Proof.
  intros c e V; induction V; intros.
  econstructor.
  unfold closed_env. fold closed_env.
  split; [[assumption]].
  eapply typable_empty__closed. eapply R_typable_empty. eauto.

```

Qed.

Lemma instantiation_R : $\forall c e,$
instantiation $c e \rightarrow$
 $\forall x t T,$
 lookup $x c = \text{Some } T \rightarrow$
 lookup $x e = \text{Some } t \rightarrow R T t.$

Proof.

intros $c e V$. induction V ; intros $x' t' T' G E$.
 solve_by_invert.
 unfold lookup in *. destruct (String.eqb $x x'$).
 inversion G ; inversion E ; subst. auto.
 eauto.

Qed.

Lemma instantiation_drop : $\forall c env,$
instantiation $c env \rightarrow$
 $\forall x, \text{instantiation } (\text{drop } x c) (\text{drop } x env).$

Proof.

intros $c e V$. induction V .
 intros. simpl. constructor.
 intros. unfold drop.
 destruct (String.eqb $x x0$); auto. constructor; eauto.

Qed.

Congruence Lemmas on Multistep

We'll need just a few of these; add them as the demand arises.

Lemma multistep_App2 : $\forall v t t',$
value $v \rightarrow (t \rightarrow^* t') \rightarrow \langle \{ v t \} \rangle \rightarrow^* \langle \{ v t' \} \rangle.$

Proof.

intros $v t t' V STM$. induction STM .
 apply multi_refl.
 eapply multi_step.
 apply ST_App2; eauto. auto.

Qed.

The R Lemma

We can finally put everything together.

The key lemma about preservation of typing under substitution can be lifted to multi-substitutions:

Lemma msubst_preserves_typing : $\forall c e,$
instantiation $c e \rightarrow$
 $\forall \text{Gamma } t S, (\text{mupdate } \text{Gamma } c) \vdash t \text{ in } S \rightarrow$

$\Gamma \vdash \{ (\text{msubst } e \ t) \} \text{ in } S.$

Proof.

```

intros  $c \ e \ H$ . induction  $H$ ; intros.
simpl in  $H$ . simpl. auto.
simpl in  $H2$ . simpl.
apply  $IH_{\text{instantiation}}$ .
eapply substitution_preserves_typing; eauto.
apply (R_typable_empty  $H0$ ).

```

Qed.

And at long last, the main lemma.

Lemma msubst_R : $\forall \ c \ env \ t \ T,$
 $(\text{mupdate empty } c) \vdash t \text{ in } T \rightarrow$
instantiation $c \ env \rightarrow$
 $R \ T \ (\text{msubst } env \ t).$

Proof.

```

intros  $c \ env0 \ t \ T \ HT \ V$ .
generalize dependent  $env0$ .
remember  $(\text{mupdate empty } c)$  as  $\Gamma$ .
assert  $(\forall \ x, \ \Gamma \ x = \text{lookup } x \ c)$ .
  intros. rewrite  $H_{\text{eq}}\Gamma$ . rewrite mupdate_lookup. auto.
clear  $H_{\text{eq}}\Gamma$ .
generalize dependent  $c$ .
induction  $HT$ ; intros.
-
  rewrite  $H0$  in  $H$ . destruct (instantiation_domains_match  $V \ H$ ) as  $[t \ P]$ .
  eapply instantiation_R; eauto.
  rewrite msubst_var. rewrite  $P$ . auto. eapply instantiation_env_closed; eauto.
-
  rewrite msubst_abs.
  assert  $(WT : \text{empty} \vdash \backslash x : T2, \{ \text{msubst } (\text{drop } x \ env0) \ t1 \} \text{ in } (T2 \rightarrow T1))$ .
  { eapply T_Abs. eapply msubst_preserves_typing.
    { eapply instantiation_drop; eauto. }
    eapply context_invariance.
    { apply  $HT$ . }
  }
  intros.
  unfold update, t_update. rewrite mupdate_drop. destruct (eqb_spec  $x \ x0$ ).
  + auto.
  + rewrite  $H$ .
    clear -  $c \ n$ . induction  $c$ .
    simpl. rewrite false_eqb_string; auto.
    simpl. destruct  $a$ . unfold update, t_update.

```

```

    destruct (String.eqb s x0); auto. }
  unfold R. fold R. split.
  auto.
  split. apply value_halts. apply v_abs.
  intros.
  destruct (R_halts H0) as [v [P Q]].
  pose proof (multistep_preserves_R _ _ _ P H0).
  apply multistep_preserves_R' with (msubst ((x,v)::env0) t1).
  eapply T_App. eauto.
  apply R_typable_empty; auto.
  eapply multi_trans. eapply multistep_App2; eauto.
  eapply multi_R.
  simpl. rewrite subst_msubst.
  eapply ST_AppAbs; eauto.
  eapply typable_empty_closed.
  apply (R_typable_empty H1).
  eapply instantiation_env_closed; eauto.
  eapply (IHHT ((x, T2)::c)).
    intros. unfold update, t_update, lookup. destruct (String.eqb x x0); auto.
  constructor; auto.
-
  rewrite msubst_app.
  destruct (IHHT1 c H env0 V) as [_ [_ P1]].
  pose proof (IHHT2 c H env0 V) as P2. fold R in P1. auto.
Admitted.

```

Normalization Theorem

And the final theorem:

Theorem normalization : $\forall t T, \text{empty} \vdash t \text{ in } T \rightarrow \text{halts } t$.

Proof.

```

  intros.
  replace t with (msubst nil t) by reflexivity.
  apply (@R_halts T).
  apply (msubst_R nil); eauto.
  eapply V_nil.
Qed.

```

Chapter 19

Library PLF.PE

19.1 PE: Partial Evaluation

The `Equiv` chapter introduced constant folding as an example of a program transformation and proved that it preserves the meaning of programs. Constant folding operates on manifest constants such as `ANum` expressions. For example, it simplifies the command `Y := 3 + 1` to the command `Y := 4`. However, it does not propagate known constants along data flow. For example, it does not simplify the sequence

`X := 3; Y := X + 1`

to

`X := 3; Y := 4`

because it forgets that `X` is 3 by the time it gets to `Y`.

We might naturally want to enhance constant folding so that it propagates known constants and uses them to simplify programs. Doing so constitutes a rudimentary form of *partial evaluation*. As we will see, partial evaluation is so called because it is like running a program, except only part of the program can be evaluated because only part of the input to the program is known. For example, we can only simplify the program

`X := 3; Y := (X + 1) - Y`

to

`X := 3; Y := 4 - Y`

without knowing the initial value of `Y`.

```
From PLF Require Import Maps.
From Stdlib Require Import Bool.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat.
From Stdlib Require Import PeanoNat. Import NAT.
From Stdlib Require Import Lia.
From Stdlib Require Import FunctionalExtensionality.
From Stdlib Require Import List.
Import LISTNOTATIONS.
```

```

Set Warnings "-parsing".
From PLF Require Import Smallstep.
From PLF Require Import Imp.

```

19.2 Generalizing Constant Folding

The starting point of partial evaluation is to represent our partial knowledge about the state. For example, between the two assignments above, the partial evaluator may know only that *X* is 3 and nothing about any other variable.

19.2.1 Partial States

Conceptually speaking, we can think of such partial states as the type `string → option nat` (as opposed to the type `string → nat` of concrete, full states). However, in addition to looking up and updating the values of individual variables in a partial state, we may also want to compare two partial states to see if and where they differ, to handle conditional control flow. It is not possible to compare two arbitrary functions in this way, so we represent partial states in a more concrete format: as a list of `string × nat` pairs.

Definition `pe_state := list (string × nat)`.

The idea is that a variable (of type `string`) appears in the list if and only if we know its current `nat` value. The `pe_lookup` function thus interprets this concrete representation. (If the same variable appears multiple times in the list, the first occurrence wins, but we will define our partial evaluator to never construct such a `pe_state`.)

```

Fixpoint pe_lookup (pe_st : pe_state) (V:string) : option nat :=
  match pe_st with
  | [] ⇒ None
  | (V',n')::pe_st ⇒ if String.eqb V V' then Some n'
                     else pe_lookup pe_st V
  end.

```

For example, `empty_pe_state` represents complete ignorance about every variable – the function that maps every identifier to `None`.

Definition `empty_pe_state : pe_state := []`.

More generally, if the `list` representing a `pe_state` does not contain some identifier, then that `pe_state` must map that identifier to `None`. Before we prove this fact, we first define a useful tactic for reasoning with `string` equality. The tactic

`compare V V'`

means to reason by cases over `String.eqb V V'`. In the case where `V = V'`, the tactic substitutes `V` for `V'` throughout.

Tactic Notation "compare" *ident(i) ident(j)* :=
 let *H* := fresh "Heq" *i j* in


```
destruct (String.eqb_spec i j) as [H|H];
[ subst j | ].
```

Theorem `pe_domain`: $\forall pe\_st\ V\ n,$
`pe_lookup` $pe\_st\ V = \text{Some } n \rightarrow$
 $\text{In } V\ (\text{map } (@fst\ \_)\ pe\_st).$

Proof. `intros pe_st V n H. induction pe_st as [| [V' n'] pe_st].`
`- inversion H.`
`- simpl in H. simpl. compare V V'; auto. Qed.`

In what follows, we will make heavy use of the `In` property from the standard library, also defined in *Logic.v*:

Print *In*.

Besides the various lemmas about `In` that we've already come across, the following one (taken from the standard library) will also be useful:

Check `filter_In`.

If a type `A` has an operator `eqb` for testing equality of its elements, we can compute a boolean `inb eqb a l` for testing whether `In a l` holds or not.

```
Fixpoint inb {A : Type} (eqb : A → A → bool) (a : A) (l : list A) :=
  match l with
  | [] ⇒ false
  | a'::l' ⇒ eqb a a' || inb eqb a l'
  end.
```

It is easy to relate `inb` to `In` with the `reflect` property:

Lemma `inbP` : $\forall A : \text{Type}, \forall eqb : A \rightarrow A \rightarrow \text{bool},$
 $(\forall a1\ a2, \text{reflect } (a1 = a2) (eqb\ a1\ a2)) \rightarrow$
 $\forall a\ l, \text{reflect } (\text{In } a\ l) (\text{inb } eqb\ a\ l).$

Proof.

```
intros A eqb beqP a l.
induction l as [| a' l' IH].
- constructor. intros [].
- simpl. destruct (beqP a a').
  + subst. constructor. left. reflexivity.
  + simpl. destruct IH; constructor.
    × right. trivial.
    × intros [H1 | H2]; congruence.
```

Qed.

19.2.2 Arithmetic Expressions

Partial evaluation of `aexp` is straightforward – it is basically the same as constant folding, `fold_constants_aexp`, except that sometimes the partial state tells us the current value of a

variable and we can replace it by a constant expression.

```

Fixpoint pe_aexp (pe_st : pe_state) (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | Ald i ⇒ match pe_lookup pe_st i with
              | Some n ⇒ ANum n
              | None ⇒ Ald i
            end
  | <{ a1 + a2 }> ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
      | (a1', a2') ⇒ <{ a1' + a2' }>
      end
  | <{ a1 - a2 }> ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
      | (a1', a2') ⇒ <{ a1' - a2' }>
      end
  | <{ a1 × a2 }> ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 × n2)
      | (a1', a2') ⇒ <{ a1' × a2' }>
      end
  end.

```

This partial evaluator folds constants but does not apply the associativity of addition.

Example test_pe_aexp1:

```

pe_aexp [(X,3)] <{X + 1 + Y}>
= <{4 + Y}>.

```

Proof. reflexivity. Qed.

Example test_pe_aexp2:

```

pe_aexp [(Y,3)] <{X + 1 + Y}>
= <{X + 1 + 3}>.

```

Proof. reflexivity. Qed.

Now, in what sense is `pe_aexp` correct? It is reasonable to define the correctness of `pe_aexp` as follows: whenever a full state `st:state` is *consistent* with a partial state `pe_st:pe_state` (in other words, every variable to which `pe_st` assigns a value is assigned the same value by `st`), evaluating `a` and evaluating `pe_aexp pe_st a` in `st` yields the same result. This statement is indeed true.

Definition pe_consistent (st:state) (pe_st:pe_state) :=

```

  ∀ V n, Some n = pe_lookup pe_st V → st V = n.

```

Theorem pe_aexp_correct_weak: ∀ st pe_st, pe_consistent st pe_st →

```

  ∀ a, aeval st a = aeval st (pe_aexp pe_st a).
Proof. unfold pe_consistent. intros st pe_st H a.
  induction a; simpl;
  try reflexivity;
  try (destruct (pe_aexp pe_st a1);
    destruct (pe_aexp pe_st a2);
    rewrite IHa1; rewrite IHa2; reflexivity).
-
  remember (pe_lookup pe_st x) as l. destruct l.
  + rewrite H with (n:=n) by apply Heql. reflexivity.
  + reflexivity.
Qed.

```

However, we will soon want our partial evaluator to remove assignments. For example, it will simplify

$X := 3; Y := X - Y; X := 4$

to just

$Y := 3 - Y; X := 4$

by delaying the assignment to X until the end. To accomplish this simplification, we need the result of partial evaluating

$\text{pe\_aexp } (X, 3) (X - Y)$

to be equal to $3 - Y$ and *not* the original expression $X - Y$. After all, it would be incorrect, not just inefficient, to transform

$X := 3; Y := X - Y; X := 4$

to

$Y := X - Y; X := 4$

even though the output expressions $3 - Y$ and $X - Y$ both satisfy the correctness criterion that we just proved. Indeed, if we were to just define $\text{pe\_aexp } pe\_st a = a$ then the theorem `pe_aexp_correct_weak` would already trivially hold.

Instead, we want to prove that the `pe_aexp` is correct in a stronger sense: evaluating the expression produced by partial evaluation ($\text{aeval } st (\text{pe\_aexp } pe\_st a)$) must not depend on those parts of the full state st that are already specified in the partial state pe_st . To be more precise, let us define a function *pe_override*, which updates st with the contents of pe_st . In other words, *pe_override* carries out the assignments listed in pe_st on top of st .

```

Fixpoint pe_update (st:state) (pe_st:pe_state) : state :=
  match pe_st with
  | [] ⇒ st
  | (V, n)::pe_st ⇒ t_update (pe_update st pe_st) V n
  end.

```

Example `test_pe_update`:

```

pe_update (Y !-> 1) [(X, 3); (Z, 2)]
= (X !-> 3 ; Z !-> 2 ; Y !-> 1).

```

Proof. `reflexivity. Qed.`

Although `pe_update` operates on a concrete `list` representing a `pe_state`, its behavior is defined entirely by the `pe_lookup` interpretation of the `pe_state`.

Theorem `false_eqb_string` : $\forall x\ y : \text{string},$
 $x \neq y \rightarrow \text{String.eqb } x\ y = \text{false}.$

Proof.

```
intros x y. rewrite String.eqb_neq.
intros H. apply H. Qed.
```

Theorem `pe_update_correct`: $\forall st\ pe\_st\ V0,$
`pe_update` $st\ pe\_st\ V0 =$
`match` `pe_lookup` $pe\_st\ V0$ `with`
`| Some` $n \Rightarrow n$
`| None` $\Rightarrow st\ V0$
`end.`

Proof. `intros. induction` pe_st `as` $[[V\ n]\ pe\_st].$ `reflexivity.`

`simpl in *.` `unfold` `t_update.`

`compare` $V0\ V;$ `auto.` `rewrite` `String.eqb_refl;` `auto.` `rewrite` `false_eqb_string;` `auto.`

`Qed.`

We can relate `pe_consistent` to `pe_update` in two ways. First, overriding a state with a partial state always gives a state that is consistent with the partial state. Second, if a state is already consistent with a partial state, then overriding the state with the partial state gives the same state.

Theorem `pe_update_consistent`: $\forall st\ pe\_st,$
`pe_consistent` (`pe_update` $st\ pe\_st$) $pe\_st.$

Proof. `intros` $st\ pe\_st\ V\ n\ H.$ `rewrite` `pe_update_correct.`

`destruct` (`pe_lookup` $pe\_st\ V$); `inversion` $H.$ `reflexivity.` `Qed.`

Theorem `pe_consistent_update`: $\forall st\ pe\_st,$
`pe_consistent` $st\ pe\_st \rightarrow \forall V, st\ V = \text{pe\_update } st\ pe\_st\ V.$

Proof. `intros` $st\ pe\_st\ H\ V.$ `rewrite` `pe_update_correct.`

`remember` (`pe_lookup` $pe\_st\ V$) `as` $l.$ `destruct` $l;$ `auto.` `Qed.`

Now we can state and prove that `pe_aexp` is correct in the stronger sense that will help us define the rest of the partial evaluator.

Intuitively, running a program using partial evaluation is a two-stage process. In the first, *static* stage, we partially evaluate the given program with respect to some partial state to get a *residual* program. In the second, *dynamic* stage, we evaluate the residual program with respect to the rest of the state. This dynamic state provides values for those variables that are unknown in the static (partial) state. Thus, the residual program should be equivalent to *prepending* the assignments listed in the partial state to the original program.

Theorem `pe_aexp_correct`: $\forall (pe\_st:pe\_state)\ (a:aexp)\ (st:state),$
`aeval` (`pe_update` $st\ pe\_st$) $a = \text{aeval } st\ (\text{pe\_aexp } pe\_st\ a).$

Proof.

`intros` $pe\_st\ a\ st.$

```

induction a; simpl;
  try reflexivity;
  try (destruct (pe_aexp pe_st a1);
        destruct (pe_aexp pe_st a2);
        rewrite IHa1; rewrite IHa2; reflexivity).
rewrite pe_update_correct. destruct (pe_lookup pe_st x); reflexivity.
Qed.

```

19.2.3 Boolean Expressions

The partial evaluation of boolean expressions is similar. (In fact, it is entirely analogous to the constant folding of boolean expressions, because our language has no boolean variables.)

```

Fixpoint pe_bexp (pe_st : pe_state) (b : bexp) : bexp :=
  match b with
  | <{ true }> => <{ true }>
  | <{ false }> => <{ false }>
  | <{ a1 = a2 }> =>
    match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
    | (ANum n1, ANum n2) => if n1 =? n2 then <{ true }> else <{ false }>
    | (a1', a2') => <{ a1' = a2' }>
    end
  | <{ a1 ≠ a2 }> =>
    match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
    | (ANum n1, ANum n2) => if negb (n1 =? n2) then <{ true }> else <{ false }>
    | (a1', a2') => <{ a1' ≠ a2' }>
    end
  | <{ a1 ≤ a2 }> =>
    match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
    | (ANum n1, ANum n2) => if n1 <=? n2 then <{ true }> else <{ false }>
    | (a1', a2') => <{ a1' ≤ a2' }>
    end
  | <{ a1 > a2 }> =>
    match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
    | (ANum n1, ANum n2) => if n1 <=? n2 then <{ false }> else <{ true }>
    | (a1', a2') => <{ a1' > a2' }>
    end
  | <{ ¬ b1 }> =>
    match (pe_bexp pe_st b1) with
    | <{ true }> => <{ false }>
    | <{ false }> => <{ true }>
    | b1' => <{ ¬ b1' }>
    end
  end

```

```

| <{ b1 && b2 }> =>
  match (pe_bexp pe_st b1, pe_bexp pe_st b2) with
  | (<{ true }>, <{ true }>) => <{ true }>
  | (<{ true }>, <{ false }>) => <{ false }>
  | (<{ false }>, <{ true }>) => <{ false }>
  | (<{ false }>, <{ false }>) => <{ false }>
  | (b1', b2') => <{ b1' && b2' }>
  end
end.

```

Example test_pe_bexp1:

```

pe_bexp [(X,3)] <{¬(X ≤ 3)}>
= <{ false }>.

```

Proof. reflexivity. Qed.

Example test_pe_bexp2: $\forall b:\text{bexp}$,

```

b = <{ ¬(X ≤ (X + 1)) }> →
pe_bexp [] b = b.

```

Proof. intros b H . rewrite $\rightarrow H$. reflexivity. Qed.

The correctness of `pe_bexp` is analogous to the correctness of `pe_aexp` above.

Theorem `pe_bexp_correct`: $\forall (pe\_st:\text{pe\_state}) (b:\text{bexp}) (st:\text{state})$,

```

beval (pe_update st pe_st) b = beval st (pe_bexp pe_st b).

```

Proof.

```

intros pe_st b st.
induction b; simpl;
try reflexivity;
try (remember (pe_aexp pe_st a1) as a1';
    remember (pe_aexp pe_st a2) as a2';
    assert (H1: aeval (pe_update st pe_st) a1 = aeval st a1');
    assert (H2: aeval (pe_update st pe_st) a2 = aeval st a2');
    try (subst; apply pe_aexp_correct);
    destruct a1'; destruct a2'; rewrite H1; rewrite H2;
    simpl; try destruct (n =? n0);
    try destruct (n <=? n0); reflexivity);
try (destruct (pe_bexp pe_st b); rewrite IHb; reflexivity);
try (destruct (pe_bexp pe_st b1);
    destruct (pe_bexp pe_st b2);
    rewrite IHb1; rewrite IHb2; reflexivity).

```

Qed.

19.3 Partial Evaluation of Commands, Without Loops

What about the partial evaluation of commands? The analogy between partial evaluation and full evaluation continues: Just as full evaluation of a command turns an initial state into a final state, partial evaluation of a command turns an initial partial state into a final partial state. The difference is that, because the state is partial, some parts of the command may not be executable at the static stage. Therefore, just as `pe_aexp` returns a residual `aexp` and `pe_bexp` returns a residual `bexp` above, partially evaluating a command yields a residual command.

Another way in which our partial evaluator is similar to a full evaluator is that it does not terminate on all commands. It is not hard to build a partial evaluator that terminates on all commands; what is hard is building a partial evaluator that terminates on all commands yet automatically performs desired optimizations such as unrolling loops. Often a partial evaluator can be coaxed into terminating more often and performing more optimizations by writing the source program differently so that the separation between static and dynamic information becomes more apparent. Such coaxing is the art of *binding-time improvement*. The binding time of a variable tells when its value is known – either “static”, or “dynamic.”

Anyway, for now we will just live with the fact that our partial evaluator is not a total function from the source command and the initial partial state to the residual command and the final partial state. To model this non-termination, just as with the full evaluation of commands, we use an inductively defined relation. We write

$$c1 / st ==> c1' / st'$$

to mean that partially evaluating the source command `c1` in the initial partial state `st` yields the residual command `c1'` and the final partial state `st'`. For example, we want something like

$$\square / (X := 3 ; Y := Z * (X + X)) ==> (Y := Z * 6) / (X, 3)$$

to hold. The assignment to `X` appears in the final partial state, not the residual command.

(Writing something like `st = [ c1 ] => c1' / st'` would be closer to the notation used in `Imp`; perhaps this should be changed!)

19.3.1 Assignment

Let's start by considering how to partially evaluate an assignment. The two assignments in the source program above needs to be treated differently. The first assignment `X := 3`, is *static*: its right-hand-side is a constant (more generally, simplifies to a constant), so we should update our partial state at `X` to 3 and produce no residual code. (Actually, we produce a residual `skip`.) The second assignment `Y := Z * (X + X)` is *dynamic*: its right-hand-side does not simplify to a constant, so we should leave it in the residual code and remove `Y`, if present, from our partial state. To implement these two cases, we define the functions `pe_add` and `pe_remove`. Like `pe_update` above, these functions operate on a concrete `list` representing a `pe_state`, but the theorems `pe_add_correct` and `pe_remove_correct` specify their behavior by the `pe_lookup` interpretation of the `pe_state`.

`Fixpoint pe_remove (pe_st:pe_state) (V:string) : pe_state :=`

```

match pe_st with
| [] ⇒ []
| (V', n') :: pe_st ⇒ if String.eqb V V' then pe_remove pe_st V
                        else (V', n') :: pe_remove pe_st V
end.

Theorem pe_remove_correct: ∀ pe_st V V0,
  pe_lookup (pe_remove pe_st V) V0
= if String.eqb V V0 then None else pe_lookup pe_st V0.
Proof. intros pe_st V V0. induction pe_st as [| [V' n'] pe_st].
- destruct (String.eqb V V0); reflexivity.
- simpl. compare V V'.
  + rewrite IHpe_st.
    destruct (String.eqb_spec V V0). reflexivity.
    rewrite false_eqb_string; auto.
  + simpl. compare V0 V'.
    × rewrite false_eqb_string; auto.
    × rewrite IHpe_st. reflexivity.
Qed.

Definition pe_add (pe_st:pe_state) (V:string) (n:nat) : pe_state :=
  (V, n) :: pe_remove pe_st V.

```

```

Theorem pe_add_correct: ∀ pe_st V n V0,
  pe_lookup (pe_add pe_st V n) V0
= if String.eqb V V0 then Some n else pe_lookup pe_st V0.
Proof. intros pe_st V n V0. unfold pe_add. simpl.
  compare V V0.
- rewrite String.eqb_refl; auto.
- rewrite pe_remove_correct.
  repeat rewrite false_eqb_string; auto.
Qed.

```

We will use the two theorems below to show that our partial evaluator correctly deals with dynamic assignments and static assignments, respectively.

```

Theorem pe_update_update_remove: ∀ st pe_st V n,
  t_update (pe_update st pe_st) V n =
  pe_update (t_update st V n) (pe_remove pe_st V).
Proof. intros st pe_st V n. apply functional_extensionality.
  intros V0. unfold t_update. rewrite !pe_update_correct.
  rewrite pe_remove_correct. destruct (String.eqb V V0); reflexivity.
Qed.

Theorem pe_update_update_add: ∀ st pe_st V n,
  t_update (pe_update st pe_st) V n =
  pe_update st (pe_add pe_st V n).

```



```

Proof. intros st pe_st V n. apply functional_extensionality. intros V0.
  unfold t_update. rewrite !pe_update_correct. rewrite pe_add_correct.
  destruct (String.eqb V V0); reflexivity. Qed.

```

19.3.2 Conditional

Trickier than assignments to partially evaluate is the conditional, `if b1 then c1 else c2 end`. If *b1* simplifies to `BTrue` or `BFalse` then it's easy: we know which branch will be taken, so just take that branch. If *b1* does not simplify to a constant, then we need to take both branches, and the final partial state may differ between the two branches!

The following program illustrates the difficulty:

```

X := 3; if Y <= 4 then Y := 4; if X <= Y then Y := 999 else skip end else skip end

```

Suppose the initial partial state is empty. We don't know statically how *Y* compares to 4, so we must partially evaluate both branches of the (outer) conditional. On the `then` branch, we know that *Y* is set to 4 and can even use that knowledge to simplify the code somewhat. On the `else` branch, we still don't know the exact value of *Y* at the end. What should the final partial state and residual program be?

One way to handle such a dynamic conditional is to take the intersection of the final partial states of the two branches. In this example, we take the intersection of (*Y*,4),(*X*,3) and (*X*,3), so the overall final partial state is (*X*,3). To compensate for forgetting that *Y* is 4, we need to add an assignment *Y* := 4 to the end of the `then` branch. So, the residual program will be something like

```

skip; if Y <= 4 then skip; skip; Y := 4 else skip end

```

Programming this case in Rocq calls for several auxiliary functions: we need to compute the intersection of two `pe_states` and turn their difference into sequences of assignments.

First, we show how to compute whether two `pe_states` disagree at a given variable. In the theorem `pe_disagree_domain`, we prove that two `pe_states` can only disagree at variables that appear in at least one of them.

```

Definition pe_disagree_at (pe_st1 pe_st2 : pe_state) (V:string) : bool :=
  match pe_lookup pe_st1 V, pe_lookup pe_st2 V with
  | Some x, Some y => negb (x =? y)
  | None, None => false
  | -, - => true
  end.

```

```

Theorem pe_disagree_domain: ∀ (pe_st1 pe_st2 : pe_state) (V:string),
  true = pe_disagree_at pe_st1 pe_st2 V →
  In V (map (@fst -) pe_st1 ++ map (@fst -) pe_st2).

```

```

Proof. unfold pe_disagree_at. intros pe_st1 pe_st2 V H.
  apply in_app_iff.
  remember (pe_lookup pe_st1 V) as lookup1.
  destruct lookup1 as [n1]. left. apply pe_domain with n1. auto.
  remember (pe_lookup pe_st2 V) as lookup2.

```

```
destruct lookup2 as [n2]]. right. apply pe_domain with n2. auto.
inversion H. Qed.
```

We define the `pe_compare` function to list the variables where two given `pe_states` disagree. This list is exact, according to the theorem `pe_compare_correct`: a variable appears on the list if and only if the two given `pe_states` disagree at that variable. Furthermore, we use the `pe_unique` function to eliminate duplicates from the list.

```
Fixpoint pe_unique (l : list string) : list string :=
  match l with
  | [] => []
  | x::l =>
    x :: filter (fun y => if String.eqb x y then false else true) (pe_unique l)
  end.
```

Theorem `pe_unique_correct`: $\forall l x,$
 $\text{In } x l \leftrightarrow \text{In } x (\text{pe\_unique } l).$

Proof. intros $l x$. induction l as [| h t]. reflexivity.
 simpl in *. split.

```
-
  intros. inversion H; clear H.
  left. assumption.
  destruct (String.eqb_spec h x).
  left. assumption.
  right. apply filter_In. split.
  apply IHt. assumption.
  rewrite false_eqb_string; auto.

-
  intros. inversion H; clear H.
  left. assumption.
  apply filter_In in H0. inversion H0. right. apply IHt. assumption.

Qed.
```

Definition `pe_compare` ($pe\_st1\ pe\_st2 : pe\_state$) : `list string` :=
`pe_unique (filter (pe_disagree_at pe_st1 pe_st2)`
`(map (@fst -) pe_st1 ++ map (@fst -) pe_st2)).`

Theorem `pe_compare_correct`: $\forall pe\_st1\ pe\_st2\ V,$
 $\text{pe\_lookup } pe\_st1\ V = \text{pe\_lookup } pe\_st2\ V \leftrightarrow$
 $\neg \text{In } V (\text{pe\_compare } pe\_st1\ pe\_st2).$

Proof. intros $pe\_st1\ pe\_st2\ V$.
 unfold `pe_compare`. rewrite $\leftarrow$ `pe_unique_correct`. rewrite `filter_In`.
 split; intros Heq .

```
-
  intro. destruct H. unfold pe_disagree_at in H0. rewrite Heq in H0.
  destruct (pe_lookup pe_st2 V).
```

```

rewrite → eqb_refl in H0. inversion H0.
inversion H0.
-
assert (Hagree: pe_disagree_at pe_st1 pe_st2 V = false).
{
  remember (pe_disagree_at pe_st1 pe_st2 V) as disagree.
  destruct disagree; [| reflexivity].
  apply pe_disagree_domain in Heqdisagree.
  exfalso. apply Heq. split. assumption. reflexivity. }
unfold pe_disagree_at in Hagree.
destruct (pe_lookup pe_st1 V) as [n1];
destruct (pe_lookup pe_st2 V) as [n2];
  try reflexivity; try solve_by_invert.
rewrite negb_false_iff in Hagree.
apply eqb_eq in Hagree. subst. reflexivity. Qed.

```

The intersection of two partial states is the result of removing from one of them all the variables where the two disagree. We define the function `pe_removes`, in terms of `pe_remove` above, to perform such a removal of a whole list of variables at once.

The theorem `pe_compare_removes` testifies that the `pe_lookup` interpretation of the result of this intersection operation is the same no matter which of the two partial states we remove the variables from. Because `pe_update` only depends on the `pe_lookup` interpretation of partial states, `pe_update` also does not care which of the two partial states we remove the variables from; that theorem `pe_compare_update` is used in the correctness proof shortly.

```

Fixpoint pe_removes (pe_st:pe_state) (ids : list string) : pe_state :=
  match ids with
  | [] ⇒ pe_st
  | V::ids ⇒ pe_remove (pe_removes pe_st ids) V
  end.

```

```

Theorem pe_removes_correct: ∀ pe_st ids V,
  pe_lookup (pe_removes pe_st ids) V =
  if inb String.eqb V ids then None else pe_lookup pe_st V.

```

```

Proof. intros pe_st ids V. induction ids as [| V' ids]. reflexivity.
  simpl. rewrite pe_remove_correct. rewrite IHids.
  compare V' V.
  - rewrite String.eqb_refl. reflexivity.
  - rewrite false_eqb_string; try congruence. reflexivity.
Qed.

```

```

Theorem pe_compare_removes: ∀ pe_st1 pe_st2 V,
  pe_lookup (pe_removes pe_st1 (pe_compare pe_st1 pe_st2)) V =
  pe_lookup (pe_removes pe_st2 (pe_compare pe_st1 pe_st2)) V.
Proof.

```

```

intros pe_st1 pe_st2 V. rewrite !pe_removes_correct.
destruct (inbP _ _ String.eqb_spec V (pe_compare pe_st1 pe_st2)).
- reflexivity.
- apply pe_compare_correct. auto. Qed.

```

Theorem *pe_compare_update*: $\forall$ *pe_st1 pe_st2 st*,
 $\text{pe\_update } st (\text{pe\_removes } pe\_st1 (\text{pe\_compare } pe\_st1 pe\_st2)) =$
 $\text{pe\_update } st (\text{pe\_removes } pe\_st2 (\text{pe\_compare } pe\_st1 pe\_st2)).$

Proof. intros. apply functional_extensionality. intros *V*.
 rewrite !*pe_update_correct*. rewrite *pe_compare_removes*. reflexivity.
 Qed.

Finally, we define an *assign* function to turn the difference between two partial states into a sequence of assignment commands. More precisely, *assign pe_st ids* generates an assignment command for each variable listed in *ids*.

```

Fixpoint assign (pe_st : pe_state) (ids : list string) : com :=
  match ids with
  | []  $\Rightarrow$  <{ skip }>
  | V::ids  $\Rightarrow$  match pe_lookup pe_st V with
    | Some n  $\Rightarrow$  <{ assign pe_st ids; V := n }>
    | None  $\Rightarrow$  assign pe_st ids
  end
end.

```

The command generated by *assign* always terminates, because it is just a sequence of assignments. The (total) function *assigned* below computes the effect of the command on the (dynamic state). The theorem *assign_removes* then confirms that the generated assignments perfectly compensate for removing the variables from the partial state.

```

Definition assigned (pe_st:pe_state) (ids : list string) (st:state) : state :=
  fun V  $\Rightarrow$  if inb String.eqb V ids then
    match pe_lookup pe_st V with
    | Some n  $\Rightarrow$  n
    | None  $\Rightarrow$  st V
    end
  else st V.

```

Theorem *assign_removes*: $\forall$ *pe_st ids st*,
 $\text{pe\_update } st pe\_st =$
 $\text{pe\_update } (\text{assigned } pe\_st ids st) (\text{pe\_removes } pe\_st ids).$

Proof. intros *pe_st ids st*. apply functional_extensionality. intros *V*.
 rewrite !*pe_update_correct*. rewrite *pe_removes_correct*. unfold *assigned*.
 destruct (inbP _ _ String.eqb_spec *V ids*); destruct (pe_lookup *pe_st V*); reflexivity.
 Qed.

Lemma *ceval_extensionality*: $\forall$ *c st st1 st2*,
 $st = [c] \Rightarrow st1 \rightarrow (\forall V, st1 V = st2 V) \rightarrow st = [c] \Rightarrow st2.$

```

Proof. intros c st st1 st2 H Heq.
  apply functional_extensionality in Heq. rewrite ← Heq. apply H. Qed.

Theorem eval_assign: ∀ pe_st ids st,
  st = [ assign pe_st ids ] => assigned pe_st ids st.
Proof. intros pe_st ids st. induction ids as [| V ids]; simpl.
- eapply ceval_extensionality. apply E_Skip. reflexivity.
-
  remember (pe_lookup pe_st V) as lookup. destruct lookup.
+ eapply E_Seq. apply IHids. unfold assigned. simpl.
  eapply ceval_extensionality. apply E_Asgn. simpl. reflexivity.
  intros V0. unfold t_update. compare V V0.
  × rewrite ← Heqlookup. rewrite String.eqb_refl. reflexivity.
  × rewrite false_eqb_string; simpl; congruence.
+ eapply ceval_extensionality. apply IHids.
  unfold assigned. intros V0. simpl. compare V V0.
  × rewrite ← Heqlookup.
  rewrite String.eqb_refl.
  destruct (inbP _ _ String.eqb_spec V ids); reflexivity.
  × rewrite false_eqb_string; simpl; congruence.

Qed.

```

19.3.3 The Partial Evaluation Relation

At long last, we can define a partial evaluator for commands without loops, as an inductive relation! The inequality conditions in `PE_AsgnDynamic` and `PE_If` are just to keep the partial evaluator deterministic; they are not required for correctness.

Reserved Notation "`c1' /' st' ==> c1' /' st'`"
 (at level 40, `st` at level 39, `c1'` at level 39).

```

Inductive pe_com : com → pe_state → com → pe_state → Prop :=
| PE_Skip : ∀ pe_st,
  <{skip}> / pe_st ==> <{skip}> / pe_st
| PE_AsgnStatic : ∀ pe_st a1 (n1 : nat) l,
  pe_aexp pe_st a1 = <{ n1 }> →
  <{ l := a1 }> / pe_st ==> <{skip}> / pe_add pe_st l n1
| PE_AsgnDynamic : ∀ pe_st a1 a1' l,
  pe_aexp pe_st a1 = a1' →
  (∀ n : nat, a1' ≠ <{ n }>) →
  <{ l := a1 }> / pe_st ==> <{ l := a1' }> / pe_remove pe_st l
| PE_Seq : ∀ pe_st pe_st' pe_st'' c1 c2 c1' c2',
  c1 / pe_st ==> c1' / pe_st' →
  c2 / pe_st' ==> c2' / pe_st'' →
  <{ c1 ; c2 }> / pe_st ==> <{ c1' ; c2' }> / pe_st''

```

```

| PE_IfTrue :  $\forall$  pe_st pe_st' b1 c1 c2 c1',
  pe_bexp pe_st b1 = <{ true }>  $\rightarrow$ 
  c1 / pe_st ==> c1' / pe_st'  $\rightarrow$ 
  <{if b1 then c1 else c2 end}> / pe_st ==> c1' / pe_st'
| PE_IfFalse :  $\forall$  pe_st pe_st' b1 c1 c2 c2',
  pe_bexp pe_st b1 = <{ false }>  $\rightarrow$ 
  c2 / pe_st ==> c2' / pe_st'  $\rightarrow$ 
  <{if b1 then c1 else c2 end}> / pe_st ==> c2' / pe_st'
| PE_If :  $\forall$  pe_st pe_st1 pe_st2 b1 c1 c2 c1' c2',
  pe_bexp pe_st b1  $\neq$  <{ true }>  $\rightarrow$ 
  pe_bexp pe_st b1  $\neq$  <{ false }>  $\rightarrow$ 
  c1 / pe_st ==> c1' / pe_st1  $\rightarrow$ 
  c2 / pe_st ==> c2' / pe_st2  $\rightarrow$ 
  <{if b1 then c1 else c2 end}> / pe_st
  ==> <{if pe_bexp pe_st b1
    then c1' ; assign pe_st1 (pe_compare pe_st1 pe_st2)
    else c2' ; assign pe_st2 (pe_compare pe_st1 pe_st2) end}>
  / pe_removes pe_st1 (pe_compare pe_st1 pe_st2)

```

where "*c1* '/' *st* '==>' *c1'* '/' *st'*" := (**pe_com** *c1 st c1' st'*).

Local Hint Constructors **pe_com** : *core*.

Local Hint Constructors **ceval** : *core*.

19.3.4 Examples

Below are some examples of using the partial evaluator. To make the **pe_com** relation actually usable for automatic partial evaluation, we would need to define more automation tactics in Rocq. That is not hard to do, but it is not needed here.

Example *pe_example1*:

```

<{X := 3 ; Y := Z  $\times$  (X + X)}>
/ [] ==> <{skip; Y := Z  $\times$  6}> / [(X,3)].

```

Proof. eapply PE_Seq. eapply PE_AsgnStatic. reflexivity.

eapply PE_AsgnDynamic. reflexivity. intros *n H*. inversion *H*. Qed.

Example *pe_example2*:

```

<{X := 3 ; if X  $\leq$  4 then X := 4 else skip end}>
/ [] ==> <{skip; skip}> / [(X,4)].

```

Proof. eapply PE_Seq. eapply PE_AsgnStatic. reflexivity.

eapply PE_IfTrue. reflexivity.

eapply PE_AsgnStatic. reflexivity. Qed.

Example *pe_example3*:

```

<{X := 3;
  if Y  $\leq$  4 then

```

```

    Y := 4;
    if X = Y then Y := 999 else skip end
  else skip end}> / []
==> <{skip;
  if Y ≤ 4 then
    (skip; skip); (skip; Y := 4)
  else skip; skip end}>
/ [(X,3)].

```

Proof. *erewrite* f_equal2 with (f := fun c st ⇒ _ / _ ==> c / st).
 eapply PE_Seq. eapply PE_AsgnStatic. reflexivity.
 eapply PE_If; intuition eauto; try *solve_by_invert*.
 econstructor. eapply PE_AsgnStatic. reflexivity.
 eapply PE_IfFalse. reflexivity. econstructor.
 reflexivity. reflexivity. Qed.

19.3.5 Correctness of Partial Evaluation

Finally let's prove that this partial evaluator is correct!

Reserved Notation "c' '/' pe_st' '/' st' ==> st'"
 (at level 40, *pe_st'* at level 39, *st'* at level 39).

Inductive **pe_ceval**

```

(c':com) (pe_st':pe_state) (st:state) (st'':state) : Prop :=
| pe_ceval_intro : ∀ st',
  st = [ c' ] => st' →
  pe_update st' pe_st' = st'' →
  c' / pe_st' / st ==> st''
where "c' '/' pe_st' '/' st' ==> st'" := (pe_ceval c' pe_st' st st'').

```

Local Hint Constructors **pe_ceval** : *core*.

Theorem **pe_com_complete**:

```

∀ c pe_st pe_st' c', c / pe_st ==> c' / pe_st' →
∀ st st'',
(pe_update st pe_st = [ c ] => st'') →
(c' / pe_st' / st ==> st'').

```

Proof. intros c pe_st pe_st' c' Hpe.

induction Hpe; intros st st'' Heval;

try (inversion Heval; subst;

```

  try (rewrite → pe_bexp_correct, → H in *; solve_by_invert);
  []);

```

eauto.

- econstructor. econstructor.

rewrite → pe_aexp_correct. rewrite ← pe_update_update_add.

rewrite → H. reflexivity.

```

- econstructor. econstructor. reflexivity.
  rewrite → pe_aexp_correct. rewrite ← pe_update_update_remove.
  reflexivity.
-
  edestruct IHHpe1. eassumption. subst.
  edestruct IHHpe2. eassumption.
  eauto.
- inversion Heval; subst.
  + edestruct IHHpe1. eassumption.
    econstructor. apply E_IfTrue. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite ← assign_removes. eassumption.
  + edestruct IHHpe2. eassumption.
    econstructor. apply E_IfFalse. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite → pe_compare_update.
    rewrite ← assign_removes. eassumption.

```

Qed.

Theorem pe_com_sound:

```

  ∀ c pe_st pe_st' c', c / pe_st ==> c' / pe_st' →
  ∀ st st'',
  (c' / pe_st' / st ==> st'') →
  (pe_update st pe_st =[ c ]=> st'').

```

Proof. intros c pe_st pe_st' c' Hpe.

```

induction Hpe;
  intros st st'' [st' Heval Heq];
  try (inversion Heval; []); subst; auto.
- rewrite ← pe_update_update_add. apply E_Asgn.
  rewrite → pe_aexp_correct. rewrite → H. reflexivity.
- rewrite ← pe_update_update_remove. apply E_Asgn.
  rewrite ← pe_aexp_correct. reflexivity.
- eapply E_Seq; eauto.
- apply E_IfTrue.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.
- apply E_IfFalse.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.
-
  inversion Heval; subst; inversion H7;
  (eapply ceval_deterministic in H8; [] apply eval_assign); subst.
  +
    apply E_IfTrue. rewrite → pe_bexp_correct. assumption.
    rewrite ← assign_removes. eauto.

```



```

+
  rewrite → pe_compare_update.
  apply E_lfFalse. rewrite → pe_bexp_correct. assumption.
  rewrite ← assign_removes. eauto.
Qed.

```

The main theorem. Thanks to David Menendez for this formulation!

Corollary `pe_com_correct`:

```

∀ c pe_st pe_st' c', c / pe_st ==> c' / pe_st' →
  ∀ st st'',
    (pe_update st pe_st =[ c ]=> st'') ↔
    (c' / pe_st' / st ==> st'').

```

Proof. `intros c pe_st pe_st' c' H st st''`. `split`.

```

- apply pe_com_complete. apply H.
- apply pe_com_sound. apply H.

```

`Qed`.

19.4 Partial Evaluation of Loops

It may seem straightforward at first glance to extend the partial evaluation relation `pe_com` above to loops. Indeed, many loops are easy to deal with. Considered this repeated-squaring loop, for example:

```
while 1 <= X do Y := Y * Y; X := X - 1 end
```

If we know neither `X` nor `Y` statically, then the entire loop is dynamic and the residual command should be the same. If we know `X` but not `Y`, then the loop can be unrolled all the way and the residual command should be, for example,

```
Y := Y * Y; Y := Y * Y; Y := Y * Y
```

if `X` is initially 3 (and finally 0). In general, a loop is easy to partially evaluate if the final partial state of the loop body is equal to the initial state, or if its guard condition is static.

But there are other loops for which it is hard to express the residual program we want in Imp. For example, take this program for checking whether `Y` is even or odd:

```
X := 0; while 1 <= Y do Y := Y - 1; X := 1 - X end
```

The value of `X` alternates between 0 and 1 during the loop. Ideally, we would like to unroll this loop, not all the way but *two-fold*, into something like

```
while 1 <= Y do Y := Y - 1; if 1 <= Y then Y := Y - 1 else X := 1; EXIT end end; X := 0
```

Unfortunately, there is no *EXIT* command in Imp. Without extending the range of control structures available in our language, the best we can do is to repeat loop-guard tests or add flag variables. Neither option is terribly attractive.

Still, as a digression, below is an attempt at performing partial evaluation on Imp commands. We add one more command argument `c''` to the `pe_com` relation, which keeps track of a loop to roll up.

Module LOOP.

Reserved Notation " $c1 \text{ '/' } st \text{ '==>' } c1' \text{ '/' } st' \text{ '/' } c''$ "

(at level 40, st at level 39, $c1'$ at level 39, st' at level 39).

Inductive **pe_com** : **com** → pe_state → **com** → pe_state → **com** → Prop :=

| PE_Skip : $\forall pe\_st,$
 $\langle \{ \text{skip} \} \rangle / pe\_st ==> \langle \{ \text{skip} \} \rangle / pe\_st / \langle \{ \text{skip} \} \rangle$
| PE_AsgnStatic : $\forall pe\_st \ a1 \ (n1 : \mathbf{nat}) \ l,$
 $pe\_aexp \ pe\_st \ a1 = \langle \{ n1 \} \rangle \rightarrow$
 $\langle \{ l := a1 \} \rangle / pe\_st ==> \langle \{ \text{skip} \} \rangle / pe\_add \ pe\_st \ l \ n1 / \langle \{ \text{skip} \} \rangle$
| PE_AsgnDynamic : $\forall pe\_st \ a1 \ a1' \ l,$
 $pe\_aexp \ pe\_st \ a1 = a1' \rightarrow$
 $(\forall n : \mathbf{nat}, a1' \neq \langle \{ n \} \rangle) \rightarrow$
 $\langle \{ l := a1 \} \rangle / pe\_st ==> \langle \{ l := a1' \} \rangle / pe\_remove \ pe\_st \ l / \langle \{ \text{skip} \} \rangle$
| PE_Seq : $\forall pe\_st \ pe\_st' \ pe\_st'' \ c1 \ c2 \ c1' \ c2' \ c'',$
 $c1 / pe\_st ==> c1' / pe\_st' / \langle \{ \text{skip} \} \rangle \rightarrow$
 $c2 / pe\_st' ==> c2' / pe\_st'' / c'' \rightarrow$
 $\langle \{ c1 ; c2 \} \rangle / pe\_st ==> \langle \{ c1' ; c2' \} \rangle / pe\_st'' / c''$
| PE_IfTrue : $\forall pe\_st \ pe\_st' \ b1 \ c1 \ c2 \ c1' \ c'',$
 $pe\_bexp \ pe\_st \ b1 = \langle \{ \text{true} \} \rangle \rightarrow$
 $c1 / pe\_st ==> c1' / pe\_st' / c'' \rightarrow$
 $\langle \{ \text{if } b1 \text{ then } c1 \text{ else } c2 \text{ end} \} \rangle / pe\_st ==> c1' / pe\_st' / c''$
| PE_IfFalse : $\forall pe\_st \ pe\_st' \ b1 \ c1 \ c2 \ c2' \ c'',$
 $pe\_bexp \ pe\_st \ b1 = \langle \{ \text{false} \} \rangle \rightarrow$
 $c2 / pe\_st ==> c2' / pe\_st' / c'' \rightarrow$
 $\langle \{ \text{if } b1 \text{ then } c1 \text{ else } c2 \text{ end} \} \rangle / pe\_st ==> c2' / pe\_st' / c''$
| PE_If : $\forall pe\_st \ pe\_st1 \ pe\_st2 \ b1 \ c1 \ c2 \ c1' \ c2' \ c'',$
 $pe\_bexp \ pe\_st \ b1 \neq \langle \{ \text{true} \} \rangle \rightarrow$
 $pe\_bexp \ pe\_st \ b1 \neq \langle \{ \text{false} \} \rangle \rightarrow$
 $c1 / pe\_st ==> c1' / pe\_st1 / c'' \rightarrow$
 $c2 / pe\_st ==> c2' / pe\_st2 / c'' \rightarrow$
 $\langle \{ \text{if } b1 \text{ then } c1 \text{ else } c2 \text{ end} \} \rangle / pe\_st$
 $==> \langle \{ \text{if } pe\_bexp \ pe\_st \ b1$
 $\quad \text{then } c1' ; \text{assign } pe\_st1 \ (pe\_compare \ pe\_st1 \ pe\_st2)$
 $\quad \text{else } c2' ; \text{assign } pe\_st2 \ (pe\_compare \ pe\_st1 \ pe\_st2) \text{ end} \} \rangle$
 $\quad / pe\_removes \ pe\_st1 \ (pe\_compare \ pe\_st1 \ pe\_st2)$
 $\quad / c''$
| PE_WhileFalse : $\forall pe\_st \ b1 \ c1,$
 $pe\_bexp \ pe\_st \ b1 = \langle \{ \text{false} \} \rangle \rightarrow$
 $\langle \{ \text{while } b1 \text{ do } c1 \text{ end} \} \rangle / pe\_st ==> \langle \{ \text{skip} \} \rangle / pe\_st / \langle \{ \text{skip} \} \rangle$
| PE_WhileTrue : $\forall pe\_st \ pe\_st' \ pe\_st'' \ b1 \ c1 \ c1' \ c2' \ c2'',$
 $pe\_bexp \ pe\_st \ b1 = \langle \{ \text{true} \} \rangle \rightarrow$
 $c1 / pe\_st ==> c1' / pe\_st' / \langle \{ \text{skip} \} \rangle \rightarrow$

```

    <{while b1 do c1 end}> / pe_st' ==> c2' / pe_st'' / c2'' →
    pe_compare pe_st pe_st'' ≠ [] →
    <{while b1 do c1 end}> / pe_st ==> <{c1'; c2'}> / pe_st'' / c2''
| PE_While : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2' c2'',
    pe_bexp pe_st b1 ≠ <{ false }> →
    pe_bexp pe_st b1 ≠ <{ true }> →
    c1 / pe_st ==> c1' / pe_st' / <{skip}> →
    <{while b1 do c1 end}> / pe_st' ==> c2' / pe_st'' / c2'' →
    pe_compare pe_st pe_st'' ≠ [] →
    (c2'' = <{skip}> ∨ c2'' = <{while b1 do c1 end}>) →
    <{while b1 do c1 end}> / pe_st
    ==> <{if pe_bexp pe_st b1
      then c1'; c2'; assign pe_st'' (pe_compare pe_st pe_st'')
      else assign pe_st (pe_compare pe_st pe_st'') end}>
      / pe_removes pe_st (pe_compare pe_st pe_st'')
      / c2''
| PE_WhileFixedEnd : ∀ pe_st b1 c1,
    pe_bexp pe_st b1 ≠ <{ false }> →
    <{while b1 do c1 end}> / pe_st ==> <{skip}> / pe_st / <{while b1 do c1 end}>
| PE_WhileFixedLoop : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2',
    pe_bexp pe_st b1 = <{ true }> →
    c1 / pe_st ==> c1' / pe_st' / <{skip}> →
    <{while b1 do c1 end}> / pe_st'
    ==> c2' / pe_st'' / <{while b1 do c1 end}> →
    pe_compare pe_st pe_st'' = [] →
    <{while b1 do c1 end}> / pe_st
    ==> <{while true do skip end}> / pe_st / <{skip}>

| PE_WhileFixed : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2',
    pe_bexp pe_st b1 ≠ <{ false }> →
    pe_bexp pe_st b1 ≠ <{ true }> →
    c1 / pe_st ==> c1' / pe_st' / <{skip}> →
    <{while b1 do c1 end}> / pe_st'
    ==> c2' / pe_st'' / <{while b1 do c1 end}> →
    pe_compare pe_st pe_st'' = [] →
    <{while b1 do c1 end}> / pe_st
    ==> <{while pe_bexp pe_st b1 do c1'; c2' end}> / pe_st / <{skip}>

```

where "c1 '/' st '==>' c1' '/' st' '/' c'" := (pe_com c1 st c1' st' c').

Local Hint Constructors **pe_com** : core.

19.4.1 Examples

```
Ltac step i :=
  (eapply i; intuition eauto; try solve_by_invert);
  repeat (try eapply PE_Seq;
    try (eapply PE_AsgnStatic; simpl; reflexivity);
    try (eapply PE_AsgnDynamic;
      [ simpl; reflexivity
        | intuition eauto; solve_by_invert ])).
```

```
Definition square_loop: com :=
  <{while 1 ≤ X do
    Y := Y × Y;
    X := X - 1
  end}>.
```

Example pe_loop_example1:

```
square_loop / []
==> <{while 1 ≤ X do
  (Y := Y × Y;
   X := X - 1); skip
end}> / [] / <{skip}>.
```

Proof. *erewrite* f_equal2 with (*f* := fun *c st* => _ / _ ==> *c* / *st* / <{skip}>).
step PE_WhileFixed. *step* PE_WhileFixedEnd. reflexivity.
 reflexivity. reflexivity. Qed.

Example pe_loop_example2:

```
<{X := 3; square_loop}> / []
==> <{skip;
  (Y := Y × Y; skip);
  (Y := Y × Y; skip);
  (Y := Y × Y; skip);
  skip}> / [(X,0)] / <{skip}>.
```

Proof. *erewrite* f_equal2 with (*f* := fun *c st* => _ / _ ==> *c* / *st* / <{skip}>).
 eapply PE_Seq. eapply PE_AsgnStatic. reflexivity.
step PE_WhileTrue.
step PE_WhileTrue.
step PE_WhileTrue.
step PE_WhileFalse.
 inversion *H*. inversion *H*. inversion *H*.
 reflexivity. reflexivity. Qed.

Example pe_loop_example3:

```
<{Z := 3; subtract_slowly}> / []
==> <{skip;
  if X ≠ 0 then
```

```

(skip; X := X - 1);
if X ≠ 0 then
  (skip; X := X - 1);
  if X ≠ 0 then
    (skip; X := X - 1);
    while X ≠ 0 do
      (skip; X := X - 1); skip
    end;
    skip; Z := 0
  else skip; Z := 1 end; skip
else skip; Z := 2 end; skip
else skip; Z := 3 end}> / [] / <{skip}>.
Proof. erewrite f_equal2 with (f := fun c st => _ / _ ==> c / st / <{skip}>).
  eapply PE_Seq. eapply PE_AsgnStatic. reflexivity.
  step PE_While.
  step PE_While.
  step PE_While.
  step PE_WhileFixed.
  step PE_WhileFixedEnd.
  reflexivity. inversion H. inversion H. inversion H.
  reflexivity. reflexivity. Qed.

```

Example pe_loop_example4:

```

<{X := 0;
  while X ≤ 2 do
    X := 1 - X
  end}> / [] ==> <{skip; while true do skip end}> / [(X,0)] / <{skip}>.
Proof. erewrite f_equal2 with (f := fun c st => _ / _ ==> c / st / <{skip}>).
  eapply PE_Seq. eapply PE_AsgnStatic. reflexivity.
  step PE_WhileFixedLoop.
  step PE_WhileTrue.
  step PE_WhileFixedEnd.
  inversion H. reflexivity. reflexivity. reflexivity. Qed.

```

19.4.2 Correctness

Because this partial evaluator can unroll a loop n -fold where n is a (finite) integer greater than one, in order to show it correct we need to perform induction not structurally on dynamic evaluation but on the number of times dynamic evaluation enters a loop body.

Reserved Notation "c1 '/' st '==>' st' '#' n"
 (at level 40, *st* at level 39, *st'* at level 39).

Inductive `ceval_count` : `com` → state → state → `nat` → Prop :=
 | E_Skip : ∀ *st*,

```

    <{skip}> / st ==> st # 0
| E'Asgn : ∀ st a1 n l,
    aeval st a1 = n →
    <{l := a1}> / st ==> (t_update st l n) # 0
| E'Seq : ∀ c1 c2 st st' st'' n1 n2,
    c1 / st ==> st' # n1 →
    c2 / st' ==> st'' # n2 →
    <{c1 ; c2}> / st ==> st'' # (n1 + n2)
| E'IfTrue : ∀ st st' b1 c1 c2 n,
    beval st b1 = true →
    c1 / st ==> st' # n →
    <{if b1 then c1 else c2 end}> / st ==> st' # n
| E'IfFalse : ∀ st st' b1 c1 c2 n,
    beval st b1 = false →
    c2 / st ==> st' # n →
    <{if b1 then c1 else c2 end}> / st ==> st' # n
| E'WhileFalse : ∀ b1 st c1,
    beval st b1 = false →
    <{while b1 do c1 end}> / st ==> st # 0
| E'WhileTrue : ∀ st st' st'' b1 c1 n1 n2,
    beval st b1 = true →
    c1 / st ==> st' # n1 →
    <{while b1 do c1 end}> / st' ==> st'' # n2 →
    <{while b1 do c1 end}> / st ==> st'' # S (n1 + n2)

```

where "c1 '/' st '==>' st' # n" := (ceval_count c1 st st' n).

Local Hint Constructors **ceval_count** : core.

Theorem ceval_count_complete: ∀ c st st',
 st = [c] => st' → ∃ n, c / st ==> st' # n.

Proof. intros c st st' Heval.

```

induction Heval;
  try inversion IHHeval1;
  try inversion IHHeval2;
  try inversion IHHeval;
  eauto. Qed.

```

Theorem ceval_count_sound: ∀ c st st' n,
 c / st ==> st' # n → st = [c] => st'.

Proof. intros c st st' n Heval. induction Heval; eauto. Qed.

Theorem pe_compare_nil_lookup: ∀ pe_st1 pe_st2,
 pe_compare pe_st1 pe_st2 = [] →
 ∀ V, pe_lookup pe_st1 V = pe_lookup pe_st2 V.

```

Proof. intros pe_st1 pe_st2 H V.
  apply (pe_compare_correct pe_st1 pe_st2 V).
  rewrite H. intro. inversion H0. Qed.

Theorem pe_compare_nil_update:  $\forall$  pe_st1 pe_st2,
  pe_compare pe_st1 pe_st2 = []  $\rightarrow$ 
   $\forall$  st, pe_update st pe_st1 = pe_update st pe_st2.
Proof. intros pe_st1 pe_st2 H st.
  apply functional_extensionality. intros V.
  rewrite !pe_update_correct.
  apply pe_compare_nil_lookup with (V:=V) in H.
  rewrite H. reflexivity. Qed.

Reserved Notation "c' '/' pe_st' '/' c'' '/' st '==>' st'' '#' n"
  (at level 40, pe_st' at level 39, c'' at level 39,
   st at level 39, st'' at level 39).

Inductive pe_ceval_count (c':com) (pe_st':pe_state) (c'':com)
  (st:state) (st'':state) (n:nat) : Prop :=
| pe_ceval_count_intro :  $\forall$  st' n',
  st = [ c' ] => st'  $\rightarrow$ 
  c'' / pe_update st' pe_st' ==> st'' # n'  $\rightarrow$ 
  n'  $\leq$  n  $\rightarrow$ 
  c' / pe_st' / c'' / st ==> st'' # n
where "c' '/' pe_st' '/' c'' '/' st '==>' st'' '#' n" :=
  (pe_ceval_count c' pe_st' c'' st st'' n).

Local Hint Constructors pe_ceval_count : core.

Lemma pe_ceval_count_le:  $\forall$  c' pe_st' c'' st st'' n n',
  n'  $\leq$  n  $\rightarrow$ 
  c' / pe_st' / c'' / st ==> st'' # n'  $\rightarrow$ 
  c' / pe_st' / c'' / st ==> st'' # n.
Proof. intros c' pe_st' c'' st st'' n n' Hle H. inversion H.
  econstructor; try eassumption. lia. Qed.

Theorem pe_com_complete:
   $\forall$  c pe_st pe_st' c' c'', c / pe_st ==> c' / pe_st' / c''  $\rightarrow$ 
   $\forall$  st st'' n,
  (c / pe_update st pe_st ==> st'' # n)  $\rightarrow$ 
  (c' / pe_st' / c'' / st ==> st'' # n).
Proof. intros c pe_st pe_st' c' c'' Hpe.
  induction Hpe; intros st st'' n Heval;
  try (inversion Heval; subst;
    try (rewrite  $\rightarrow$  pe_bexp_correct,  $\rightarrow$  H in *; solve_by_invert);
    []);
  eauto.

```

```

- econstructor. econstructor.
  rewrite → pe_aexp_correct. rewrite ← pe_update_update_add.
  rewrite → H. apply E'Skip. auto.
- econstructor. econstructor. reflexivity.
  rewrite → pe_aexp_correct. rewrite ← pe_update_update_remove.
  apply E'Skip. auto.
-
  edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
  inversion Hskip. subst.
  edestruct IHHpe2. eassumption.
  econstructor; eauto. lia.
- inversion Heval; subst.
  + edestruct IHHpe1. eassumption.
    econstructor. apply E_lfTrue. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite ← assign_removes. eassumption. eassumption.
  + edestruct IHHpe2. eassumption.
    econstructor. apply E_lfFalse. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite → pe_compare_update.
    rewrite ← assign_removes. eassumption. eassumption.
-
  edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
  inversion Hskip. subst.
  edestruct IHHpe2. eassumption.
  econstructor; eauto. lia.
- inversion Heval; subst.
  + econstructor. apply E_lfFalse.
    rewrite ← pe_bexp_correct. assumption.
    apply eval_assign.
    rewrite ← assign_removes. inversion H2; subst; auto.
    auto.
  +
    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    econstructor. apply E_lfTrue.
    rewrite ← pe_bexp_correct. assumption.
    repeat eapply E_Seq; eauto. apply eval_assign.
    rewrite → pe_compare_update, ← assign_removes. eassumption.
    lia.
- exfalso.

```



```

generalize dependent (S (n1 + n2)). intros n.
clear - H H0 IHHpe1 IHHpe2. generalize dependent st.
induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
+ rewrite pe_bexp_correct, H in H7. inversion H7.
+
  edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
  inversion Hskip. subst.
  edestruct IHHpe2. eassumption.
  rewrite ← (pe_compare_nil_update _ _ H0) in H7.
  apply H1 in H7; [| lia]. inversion H7.
- generalize dependent st.
  induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
  + rewrite pe_bexp_correct in H8. eauto.
  + rewrite pe_bexp_correct in H5.
    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    rewrite ← (pe_compare_nil_update _ _ H1) in H8.
    apply H2 in H8; [| lia]. inversion H8.
    econstructor; [ eapply E_WhileTrue; eauto | eassumption | lia].
Qed.

```

Theorem pe_com_sound:

$$\begin{aligned}
& \forall c \text{ pe\_st pe\_st}' c' c'', c / \text{pe\_st} \implies c' / \text{pe\_st}' / c'' \rightarrow \\
& \forall st \text{ st}'' n, \\
& (c' / \text{pe\_st}' / c'' / st \implies st'' \# n) \rightarrow \\
& (\text{pe\_update } st \text{ pe\_st} = [c] \implies st'').
\end{aligned}$$

Proof. intros c pe_st pe_st' c' c'' Hpe.

```

induction Hpe;
  intros st st'' n [st' n' Heval Heval' Hle];
  try (inversion Heval; [|; subst]);
  try (inversion Heval'; [|; subst]); eauto.
- rewrite ← pe_update_update_add. apply E_Asgn.
  rewrite → pe_aexp_correct. rewrite → H. reflexivity.
- rewrite ← pe_update_update_remove. apply E_Asgn.
  rewrite ← pe_aexp_correct. reflexivity.
- eapply E_Seq; eauto.
- apply E_IfTrue.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
  eapply IHHpe. eauto.
- apply E_IfFalse.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
  eapply IHHpe. eauto.

```

```

- inversion Heval; subst; inversion H7; subst; clear H7.
+
  eapply ceval_deterministic in H8; [| apply eval_assign]. subst.
  rewrite ← assign_removes in Heval'.
  apply E.IfTrue. rewrite → pe_bexp_correct. assumption.
  eapply IHHpe1. eauto.
+
  eapply ceval_deterministic in H8; [| apply eval_assign]. subst.
  rewrite → pe_compare_update in Heval'.
  rewrite ← assign_removes in Heval'.
  apply E.IfFalse. rewrite → pe_bexp_correct. assumption.
  eapply IHHpe2. eauto.
- apply E.WhileFalse.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
- eapply E.WhileTrue.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
  eapply IHHpe1. eauto. eapply IHHpe2. eauto.
- inversion Heval; subst.
+
  inversion H9. subst. clear H9.
  inversion H10. subst. clear H10.
  eapply ceval_deterministic in H11; [| apply eval_assign]. subst.
  rewrite → pe_compare_update in Heval'.
  rewrite ← assign_removes in Heval'.
  eapply E.WhileTrue. rewrite → pe_bexp_correct. assumption.
  eapply IHHpe1. eauto.
  eapply IHHpe2. eauto.
+ apply ceval_count_sound in Heval'.
  eapply ceval_deterministic in H9; [| apply eval_assign]. subst.
  rewrite ← assign_removes in Heval'.
  inversion H2; subst.
  × inversion Heval'. subst. apply E.WhileFalse.
    rewrite → pe_bexp_correct. assumption.
  × assumption.
- eapply ceval_count_sound. apply Heval'.
-
  apply loop_never_stops in Heval. inversion Heval.
-
  clear - H1 IHHpe1 IHHpe2 Heval.
  remember <{while pe_bexp pe_st b1 do c1'; c2' end}> as c'.
  induction Heval;
    inversion Heqc'; subst; clear Heqc'.

```

```

+ apply E_WhileFalse.
  rewrite pe_bexp_correct. assumption.
+
  assert (IHHeval2' := IHHeval2 (refl_equal _)).
  apply ceval_count_complete in IHHeval2'. inversion IHHeval2'.
  clear IHHeval1 IHHeval2 IHHeval2'.
  inversion Heval1. subst.
  eapply E_WhileTrue. rewrite pe_bexp_correct. assumption. eauto.
  eapply IHHeval2. econstructor. eassumption.
  rewrite ← (pe_compare_nil_update _ _ H1). eassumption. apply le_n.
Qed.

Corollary pe_com_correct:
  ∀ c pe_st pe_st' c', c / pe_st ==> c' / pe_st' / <{skip}> →
  ∀ st st'',
  (pe_update st pe_st = [ c ] => st'') ↔
  (∃ st', st = [ c' ] => st' ∧ pe_update st' pe_st' = st'').
Proof. intros c pe_st pe_st' c' H st st''. split.
- intros Heval.
  apply ceval_count_complete in Heval. inversion Heval as [n Heval'].
  apply pe_com_complete with (st:=st) (st'':=st'') (n:=n) in H.
  inversion H as [? ? ? Hskip ?]. inversion Hskip. subst. eauto.
  assumption.
- intros [st' [Heval Heq] ]. subst st''.
  eapply pe_com_sound in H. apply H.
  econstructor. apply Heval. apply E'Skip. apply le_n.
Qed.

End LOOP.

```

19.5 Partial Evaluation of Flowchart Programs

Instead of partially evaluating *while* loops directly, the standard approach to partially evaluating imperative programs is to convert them into *flowcharts*. In other words, it turns out that adding labels and jumps to our language makes it much easier to partially evaluate. The result of partially evaluating a flowchart is a residual flowchart. If we are lucky, the jumps in the residual flowchart can be converted back to *while* loops, but that is not possible in general; we do not pursue it here.

19.5.1 Basic blocks

A flowchart is made of *basic blocks*, which we represent with the inductive type **block**. A basic block is a sequence of assignments (the constructor **Assign**), concluding with a conditional

jump (the constructor **If**) or an unconditional jump (the constructor **Goto**). The destinations of the jumps are specified by *labels*, which can be of any type. Therefore, we parameterize the **block** type by the type of labels.

```
Inductive block (Label:Type) : Type :=
| Goto : Label → block Label
| If : bexp → Label → Label → block Label
| Assign : string → aexp → block Label → block Label.
```

Arguments Goto {Label} ..

Arguments If {Label} - - ..

Arguments Assign {Label} - - ..

We use the “even or odd” program, expressed above in Imp, as our running example. Converting this program into a flowchart turns out to require 4 labels, so we define the following type.

```
Inductive parity_label : Type :=
| entry : parity_label
| loop : parity_label
| body : parity_label
| done : parity_label.
```

The following **block** is the basic block found at the **body** label of the example program.

```
Definition parity_body : block parity_label :=
  Assign Y <{Y - 1}>
  (Assign X <{1 - X}>
   (Goto loop)).
```

To evaluate a basic block, given an initial state, is to compute the final state and the label to jump to next. Because basic blocks do not *contain* loops or other control structures, evaluation of basic blocks is a total function – we don’t need to worry about non-termination.

```
Fixpoint keval {L:Type} (st:state) (k : block L) : state × L :=
  match k with
  | Goto l ⇒ (st, l)
  | If b l1 l2 ⇒ (st, if beval st b then l1 else l2)
  | Assign i a k ⇒ keval (t_update st i (aeval st a)) k
  end.
```

Example keval_example:

```
keval empty_st parity_body
= ((X !-> 1 ; Y !-> 0), loop).
```

Proof. reflexivity. Qed.

19.5.2 Flowchart programs

A flowchart program is simply a lookup function that maps labels to basic blocks. Actually, some labels are *halting states* and do not map to any basic block. So, more precisely, a flowchart **program** whose labels are of type L is a function from L to **option** (**block** L).

Definition `program` (L :Type) : Type := $L \rightarrow$ **option** (**block** L).

Definition `parity` : `program parity_label` := `fun l =>`
`match l with`
`| entry => Some (Assign X 0 (Goto loop))`
`| loop => Some (If <{1 ≤ Y}> body done)`
`| body => Some parity_body`
`| done => None`
`end.`

Unlike a basic block, a program may not terminate, so we model the evaluation of programs by an inductive relation **peval** rather than a recursive function.

Inductive **peval** { L :Type} (p : `program` L)
: `state` $\rightarrow L \rightarrow$ `state` $\rightarrow L \rightarrow$ Prop :=
| E_None: $\forall st\ l,$
 $p\ l = \text{None} \rightarrow$
peval $p\ st\ l\ st\ l$
| E_Some: $\forall st\ l\ k\ st'\ l'\ st''\ l'',$
 $p\ l = \text{Some } k \rightarrow$
`keval` $st\ k = (st', l') \rightarrow$
peval $p\ st'\ l'\ st''\ l'' \rightarrow$
peval $p\ st\ l\ st''\ l''$.

Example `parity_eval`: **peval** `parity` `empty_st` `entry` `empty_st` `done`.

Proof. `erewrite f_equal` with ($f :=$ `fun st => peval _ _ _ st _`).

`eapply E_Some. reflexivity. reflexivity.`

`eapply E_Some. reflexivity. reflexivity.`

`apply E_None. reflexivity.`

`apply functional_extensionality. intros i. rewrite t_update_same; auto.`

Qed.

19.5.3 Partial Evaluation of Basic Blocks and Flowchart Programs

Partial evaluation changes the label type in a systematic way: if the label type used to be L , it becomes `pe_state` $\times L$. So the same label in the original program may be unfolded, or blown up, into multiple labels by being paired with different partial states. For example, the label `loop` in the `parity` program will become two labels: (`[(X,0)]`, `loop`) and (`[(X,1)]`, `loop`). This change of label type is reflected in the types of `pe_block` and `pe_program` defined presently.

```

Fixpoint pe_block {L:Type} (pe_st:pe_state) (k : block L)
: block (pe_state × L) :=
match k with
| Goto l ⇒ Goto (pe_st, l)
| If b l1 l2 ⇒
  match pe_bexp pe_st b with
  | BTrue ⇒ Goto (pe_st, l1)
  | BFalse ⇒ Goto (pe_st, l2)
  | b' ⇒ If b' (pe_st, l1) (pe_st, l2)
  end
| Assign i a k ⇒
  match pe_aexp pe_st a with
  | ANum n ⇒ pe_block (pe_add pe_st i n) k
  | a' ⇒ Assign i a' (pe_block (pe_remove pe_st i) k)
  end
end.

```

Example pe_block_example:

```

pe_block [(X,0)] parity_body
= Assign Y <{Y - 1}> (Goto ([(X,1)], loop)).

```

Proof. reflexivity. Qed.

Theorem pe_block_correct: $\forall (L:\text{Type}) \text{ st } pe\_st \text{ k } st' \text{ pe\_st}' (l':L),$
 $\text{keval } st \text{ (pe\_block } pe\_st \text{ k)} = (st', (pe\_st', l')) \rightarrow$
 $\text{keval (pe\_update } st \text{ pe\_st)} \text{ k} = (\text{pe\_update } st' \text{ pe\_st}', l').$

Proof. intros. generalize dependent pe_st. generalize dependent st.

```

induction k as [l | b l1 l2 | i a k];
  intros st pe_st H.
- inversion H; reflexivity.
-
  replace (keval st (pe_block pe_st (If b l1 l2)))
    with (keval st (If (pe_bexp pe_st b) (pe_st, l1) (pe_st, l2)))
    in H by (simpl; destruct (pe_bexp pe_st b); reflexivity).
  simpl in *. rewrite pe_bexp_correct.
  destruct (beval st (pe_bexp pe_st b)); inversion H; reflexivity.
-
  simpl in *. rewrite pe_aexp_correct.
  destruct (pe_aexp pe_st a); simpl;
    try solve [rewrite pe_update_update_add; apply IHk; apply H];
    solve [rewrite pe_update_update_remove; apply IHk; apply H].

```

Qed.

Definition pe_program {L:Type} (p : program L)
: program (pe_state × L) :=
fun pe_l ⇒ match pe_l with | (pe_st, l) ⇒

```

      option_map (pe_block pe_st) (p l)
    end.

Inductive pe_peval {L:Type} (p : program L)
  (st:state) (pe_st:pe_state) (l:L) (st'o:state) (l':L) : Prop :=
| pe_peval_intro : ∀ st' pe_st',
  peval (pe_program p) st (pe_st, l) st' (pe_st', l') →
  pe_update st' pe_st' = st'o →
  pe_peval p st pe_st l st'o l'.

Theorem pe_program_correct:
  ∀ (L:Type) (p : program L) st pe_st l st'o l',
  peval p (pe_update st pe_st) l st'o l' ↔
  pe_peval p st pe_st l st'o l'.

Proof. intros.
  split.
- intros Heval.
  remember (pe_update st pe_st) as sto.
  generalize dependent pe_st. generalize dependent st.
  induction Heval as
  [ sto l Hlookup | sto l k st'o l' st''o l'' Hlookup Hkeval Heval ];
  intros st pe_st Heqsto; subst sto.
+ eapply pe_peval_intro. apply E_None.
  simpl. rewrite Hlookup. reflexivity. reflexivity.
+
  remember (keval st (pe_block pe_st k)) as x.
  destruct x as [st' [pe_st' l'_]].
  symmetry in Heqx. erewrite pe_block_correct in Hkeval by apply Heqx.
  inversion Hkeval. subst st'o l'_. clear Hkeval.
  edestruct IHHeval. reflexivity. subst st''o. clear IHHeval.
  eapply pe_peval_intro; [| reflexivity]. eapply E_Some; eauto.
  simpl. rewrite Hlookup. reflexivity.
- intros [st' pe_st' Heval Heqst'o].
  remember (pe_st, l) as pe_st_l.
  remember (pe_st', l') as pe_st'_l'.
  generalize dependent pe_st. generalize dependent l.
  induction Heval as
  [ st [pe_st_l _] Hlookup
  | st [pe_st_l _] pe_k st' [pe_st'_l'_] st'' [pe_st'' l'']
    Hlookup Hkeval Heval ];
  intros l pe_st Heqpe_st_l;
  inversion Heqpe_st_l; inversion Heqpe_st'_l'; repeat subst.
+ apply E_None. simpl in Hlookup.
  destruct (p l'); [ solve [ inversion Hlookup ] | reflexivity ].

```

```

+
  simpl in Hlookup. remember (p l) as k.
  destruct k as [k]; inversion Hlookup; subst.
  eapply E_Some; eauto. apply pe_block_correct. apply Hkeval.
Qed.

```


Chapter 20

Library **PLF.Postscript**

20.1 Postscript

Congratulations: We’ve made it to the end!

20.2 Looking Back

We’ve covered a lot of ground. Here’s a quick review of the whole trajectory we’ve followed, starting at the beginning of *Logical Foundations...*

- *Functional programming*:
 - “declarative” programming style (recursion over immutable data structures, rather than looping over mutable arrays or pointer structures)
 - higher-order functions
 - polymorphism
- *Logic*, the mathematical basis for software engineering:
 - logic calculus
 - _____ ~ _____
 - software engineering mechanical/civil engineering
 - inductively defined sets and relations
 - inductive proofs
 - proof objects
- *Rocq*, an industrial-strength proof assistant

- functional core language
- core tactics
- automation
- *Foundations of programming languages*
 - notations and definitional techniques for precisely specifying
 - * abstract syntax
 - * operational semantics
 - big-step style
 - small-step style
 - * type systems
 - program equivalence
 - Hoare logic
 - fundamental metatheory of type systems
 - * progress and preservation
 - theory of subtyping

20.3 Looking Around

Large-scale applications of these core topics can be found everywhere, both in ongoing research projects and in real-world software systems. Here are a few recent examples involving formal, machine-checked verification of real-world software and hardware systems, to give a sense of what is being done today...

CompCert

CompCert is a fully verified optimizing compiler for almost all of the ISO C90 / ANSI C language, generating code for x86, ARM, and PowerPC processors. The whole of CompCert is written in Gallina and extracted to an efficient OCaml program using Rocq’s extraction facilities.

“The CompCert project investigates the formal verification of realistic compilers usable for critical embedded software. Such verified compilers come with a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program. By ruling out the possibility of compiler-introduced bugs, verified compilers strengthen the guarantees that can be obtained by applying formal methods to source programs.”

In 2011, CompCert was included in a landmark study on fuzz-testing a large number of real-world C compilers using the CSmith tool. The CSmith authors wrote:

- The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

{<http://compcert.inria.fr>}

seL4

seL4 is a fully verified microkernel, considered to be the world’s first OS kernel with an end-to-end proof of implementation correctness and security enforcement. It is implemented in C and ARM assembly and specified and verified using Isabelle. The code is available as open source.

“seL4 has been comprehensively formally verified: a rigorous process to prove mathematically that its executable code, as it runs on hardware, correctly implements the behaviour allowed by the specification, and no others. Furthermore, we have proved that the specification has the desired safety and security properties (integrity and confidentiality)... The verification was achieved at a cost that is significantly less than that of traditional high-assurance development approaches, while giving guarantees traditional approaches cannot provide.”

{<https://sel4.systems>}.

CertiKOS

CertiKOS is a clean-slate, fully verified hypervisor, written in CompCert C and verified in Rocq.

“The CertiKOS project aims to develop a novel and practical programming infrastructure for constructing large-scale certified system software. By combining recent advances in programming languages, operating systems, and formal methods, we hope to attack the following research questions: (A) what OS kernel structure can offer the best support for extensibility, security, and resilience? (B) which semantic models and program logics can best capture these abstractions? (C) what are the right programming languages and environments for developing such certified kernels? and (D) how to build automation facilities to make certified software development really scale?”

{<http://flint.cs.yale.edu/certikos/>}

Ironclad

Ironclad Apps is a collection of fully verified web applications, including a “notary” for securely signing statements, a password hasher, a multi-user trusted counter, and a differentially-private database.

The system is coded in the verification-oriented programming language Dafny and verified using Boogie, a verification tool based on Hoare logic.

“An Ironclad App lets a user securely transmit her data to a remote machine with the guarantee that every instruction executed on that machine adheres to a formal abstract specification of the app’s behavior. This does more than eliminate implementation vulnerabilities such as buffer overflows, parsing errors, or data leaks; it tells the user exactly how the app will behave at all times. We provide these guarantees via complete, low-level software verification. We then use cryptography and secure hardware to enable secure channels from the verified software to remote users.”

{<https://github.com/Microsoft/Ironclad/tree/master/ironclad-apps>}

Verdi

Verdi is a framework for implementing and formally verifying distributed systems.

“Verdi supports several different fault models ranging from idealistic to realistic. Verdi’s verified system transformers (VSTs) encapsulate common fault tolerance techniques. Developers can verify an application in an idealized fault model, and then apply a VST to obtain an application that is guaranteed to have analogous properties in a more adversarial environment. Verdi is developed using the Rocq proof assistant, and systems are extracted to OCaml for execution. Verdi systems, including a fault-tolerant key-value store, achieve comparable performance to unverified counterparts.”

{<http://verdi.uwplse.org>}

DeepSpec

The Science of Deep Specification is an NSF “Expedition” project (running from 2016 to 2020) that focuses on the specification and verification of full functional correctness of both software and hardware. It also sponsors workshops and summer schools.

- Website: {<https://deepspec.org/>}
- Overview presentations:
 - {<https://deepspec.org/page/About/>}
 - {<https://www.youtube.com/watch?v=IPNdsnRWBkk>}

REMS

REMS is a european project on Rigorous Engineering of Mainstream Systems. It has produced detailed formal specifications of a wide range of critical real-world interfaces, protocols, and APIs, including the C language, the ELF linker format, the ARM, Power, MIPS, CHERI, and RISC-V instruction sets, the weak memory models of ARM and Power processors, and POSIX filesystems.

“The project is focussed on lightweight rigorous methods: precise specification (post hoc and during design) and testing against specifications, with full verification only in some cases. The project emphasises building useful (and reusable) semantics and tools. We are building accurate full-scale mathematical models of some of the key computational abstractions (processor architectures, programming languages, concurrent OS interfaces, and network protocols), studying how this can be done, and investigating how such models can be used for new verification research and in new systems and programming language research. Supporting all this, we are also working on new specification tools and their foundations.”

{<https://www.cl.cam.ac.uk/~pes20/rems/>}

Others

There’s much more. Other projects worth checking out include:

- Vellvm (formal specification and verification of LLVM optimization passes)
- Zach Tatlock’s formally certified browser
- Tobias Nipkow’s formalization of most of Java
- The CakeML verified ML compiler
- Greg Morrisett’s formal specification of the x86 instruction set and the RockSalt Software Fault Isolation tool (a better, faster, more secure version of Google’s Native Client)
- Ur/Web, a programming language for verified web applications embedded in Rocq
- the Princeton Verified Software Toolchain

20.4 Looking Forward

Some good places to learn more...

- This book includes several optional chapters covering topics that you may find useful. Take a look at the table of contents and the chapter dependency diagram to find them.
- More on Hoare logic and program verification
 - The Formal Semantics of Programming Languages: An Introduction, by Glynn Winskel *Winskel* 1993 (in Bib.v).
 - Many practical verification tools, e.g. Microsoft’s Boogie system, Java Extended Static Checking, etc.

- More on the foundations of programming languages:
 - Concrete Semantics with Isabelle/HOL, by Tobias Nipkow and Gerwin Klein *Nipkow* 2014 (in Bib.v)
 - Types and Programming Languages, by Benjamin C. Pierce *Pierce* 2002 (in Bib.v).
 - Practical Foundations for Programming Languages, by Robert Harper *Harper* 2016 (in Bib.v).
 - Foundations for Programming Languages, by John C. Mitchell *Mitchell* 1996 (in Bib.v).
- Iron Lambda (<http://iron.ouroborus.net/>) is a collection of Rocq formalisations for functional languages of increasing complexity. It fills part of the gap between the end of the Software Foundations course and what appears in current research papers. The collection has at least Progress and Preservation theorems for a number of variants of STLC and the polymorphic lambda-calculus (System F).
- Finally, here are some of the main conferences on programming languages and formal verification:
 - Principles of Programming Languages (POPL)
 - Programming Language Design and Implementation (PLDI)
 - International Conference on Functional Programming (ICFP)
 - Computer Aided Verification (CAV)
 - Interactive Theorem Proving (ITP)
 - Certified Programs and Proofs (CPP)
 - SPLASH/OOPSLA conferences
 - Principles in Practice workshop (PiP)
 - CoqPL workshop

Date

Chapter 21

Library **PLF.Bib**

21.1 Bib: Bibliography

21.2 Resources cited in this volume

Aydemir 2008 Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering Formal Metatheory. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California, pages 3-15. ACM, January 2008. {<https://www.cis.upenn.edu/~bcpierce/papers/binders.pdf>}

Harper 2015 Practical Foundations for Programming Languages, by Robert Harper. Cambridge University Press. Second edition, 2016. {<https://tinyurl.com/z82xwta>}

Mitchell 1996 Foundations for Programming Languages, by John C. Mitchell. MIT Press, 1996. {<https://tinyurl.com/zkosavw>}

Nipkow 2014 Concrete Semantics with Isabelle/HOL, by Tobias Nipkow and Gerwin Klein. Springer 2014. {<http://www.concrete-semantics.org>}

Pierce 2002 Types and Programming Languages, by Benjamin C. Pierce. MIT Press, 2002. {<https://tinyurl.com/gtnudmu>}

Pugh 1991 Pugh, William. “The Omega test: a fast and practical integer programming algorithm for dependence analysis.” Proceedings of the 1991 ACM/IEEE conference on Supercomputing. ACM, 1991. {<https://dl.acm.org/citation.cfm?id=125848>}

Winskel 1993 The Formal Semantics of Programming Languages: An Introduction, by Glynn Winskel. MIT Press, 1993. {<https://tinyurl.com/j2k6ev7>}

Date

Chapter 22

Library `PLF.LibTactics`

22.1 `LibTactics`: A Collection of Handy General-Purpose Tactics

This file contains a set of tactics that extends the set of builtin tactics provided with the standard distribution of Rocq. 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, Rocq tactics will be improved as time goes by, and this file should ultimately be useless. In the meanwhile, serious Rocq users will probably find it very useful.

The present file contains the implementation and the detailed documentation of those tactics. The SF reader need not read this file; instead, he/she is encouraged to read the chapter named `UseTactics.v`, which is gentle introduction to the most useful tactics from the `LibTactic` library.

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: tactics 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 sub-goals, 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 *rapplly*

```
Set Implicit Arguments.
```

```
From Stdlib Require Import List.
```

```
Declare Scope ltac_scope.
```

22.2 Fixing Stdlib

```
#[global]
Remove Hints Bool.trans_eq_bool : core.
```

22.3 Tools for Programming with Ltac

22.3.1 Identity Continuation

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

22.3.2 Untyped Arguments for Tactics

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

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

22.3.3 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.
```

22.3.4 Wildcard Arguments for Tactics

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

```
Inductive ltac_Wild : Set :=  
  | ltac_wild : ltac_Wild.
```

```
Notation "'--'" := ltac_wild : ltac_scope.
```

`ltac_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 ltac_Wilds : Set :=  
  | ltac_wilds : ltac_Wilds.
```

```
Notation "'---'" := ltac_wilds : ltac_scope.
```

```
Open Scope ltac_scope.
```

22.3.5 Position Markers

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

```
Inductive ltac_Mark : Type :=  
  | ltac_mark : ltac_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*. It fails if *Mark* does not appear in the context.

```
Ltac gen_until_mark :=  
  match goal with H: ?T ⊢ _ =>  
  match T with  
  | ltac_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
  | ltac_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.

```

22.3.6 List of Arguments for Tactics

A datatype of type **list Boxer** is used to manipulate list of Rocq 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,
 v4 at level 0, v5 at level 0, v6 at level 0)
: ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7" :=
((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
 ::(boxer v6)::(boxer v7)::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)
: ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8" :=
((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
 ::(boxer v6)::(boxer v7)::(boxer v8)::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)
: ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9" :=
((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
 ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::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)
: ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10" :=
((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
 ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)::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)
: ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11" :=
((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
 ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
 ::(boxer v11)::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)
: ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12" :=
((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
 ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
 ::(boxer v11)::(boxer v12)::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)
: 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 => constr:(E)
  | _ => constr:((boxer E)::nil)
end.

```

22.3.7 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) => 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:Boxer) (T:Boxer) (A: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 : ∀ (A:Boxer) (D:Boxer) (T: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*:Boxer);
let *H* := fresh "TEMP" in
assert (*H* : ltac_database (boxer D) (boxer T) *A*);
[subst *A*; auto
| subst *A*; match type of *H* with ltac_database _ _ (boxer ?*L*) ⇒
generalize *L* end; clear *H*].

22.3.8 On-the-Fly Removal of Hypotheses

In a list of arguments » *H1 H2 .. HN* passed to a tactic such as *lets* or *applies* 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 `rm (A:Type) (X:A) := X.`

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* ⊢ _ ⇒ try clear *H* end.

rm_inside E calls *rm_term Ei* for any subterm of the form `rm Ei` found in *E*

Ltac *rm_inside E* :=
let go *E* := *rm_inside E* in
match *E* with
| `rm ?X` ⇒ *rm_term X*
| `?X1 ?X2` ⇒
go *X1*; go *X2*
| `?X1 ?X2 ?X3` ⇒
go *X1*; go *X2*; go *X3*
| `?X1 ?X2 ?X3 ?X4` ⇒
go *X1*; go *X2*; go *X3*; go *X4*
| `?X1 ?X2 ?X3 ?X4 ?X5` ⇒
go *X1*; go *X2*; go *X3*; go *X4*; go *X5*
| `?X1 ?X2 ?X3 ?X4 ?X5 ?X6` ⇒
go *X1*; go *X2*; go *X3*; go *X4*; go *X5*; go *X6*

```

| ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ⇒
  go X1; go X2; go X3; go X4; go X5; go X6; go X7
| ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ⇒
  go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8
| ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ?X9 ⇒
  go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8; go X9
| ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ?X9 ?X10 ⇒
  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.

```

22.3.9 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.

```

From Stdlib Require Import BinNums 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 Rocq 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 Rocq 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.

22.3.10 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  $\Rightarrow$  idtac
  | S 0  $\Rightarrow$  idtac
  | S ?N'  $\Rightarrow$  apply dup_lemma; [ | dup_tactic N' ]
  end.
```

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

```
Tactic Notation "dup" :=
  dup 2.
```

22.3.11 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 ].
```

22.3.12 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  $\vdash ?G \Rightarrow$  check_noevar G end.
```

22.3.13 Helper Function for Introducing Evars

`with_evar` T (`fun` $M \Rightarrow tac$) creates a new evvar 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.
```

22.3.14 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: _  $\vdash$  _  $\Rightarrow$  constr:(H) end.
```

22.3.15 More 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 _  $\vdash$  _  $\Rightarrow$  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.
```

22.3.16 Deconstructing Terms

`get_head` E is a tactic that returns the head constant of the term E , ie, when applied to a term of the form $P\ x1\ \dots\ xN$ 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  $\Rightarrow$  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.
```

22.3.17 Action at Occurence and Action Not at Occurence

ltac_action_at K of E do Tac isolates the *K*-th occurrence 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.
```

22.3.18 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`.

`#[global]`

Hint Unfold `eq' : core`.

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

22.4 Check as a tactic instead of a top-level command, with possibility to turn off

`check E` is like `Check E`, but works as a tactic. It can be turned off using `Ltac check_enabled ::= constr:(false)`.

`Ltac check_enabled := constr:(true)`.

Tactic Notation "`check`" `uconstr(E)` :=
 lazymatch `check_enabled` with
 | `false` ⇒ `idtac`
 | `true` ⇒
 first [`generalize E`; `match goal with` `⊢ ?T → _` ⇒ `idtac E`; `idtac ":" T`
`end`; `intros _`
 | `idtac E`; `fail 1 "does not typecheck"`]
`end`.

22.5 Common Tactics for Simplifying Goals Like `intuition`

`Ltac jauto_set_hyps :=`
 repeat match goal with `H: ?T ⊢ _` ⇒
 match `T` with
 | `_ ∧ _` ⇒ `destruct H`
 | `iff _ _` ⇒ `destruct H`
 | `∃ a, _` ⇒ `destruct H`
 | `_` ⇒ `generalize H`; `clear H`
 `end`
 `end`.

`Ltac jauto_set_goal :=`
 repeat match goal with
 | `⊢ ∃ a, _` ⇒ `esplit`

```

| ⊢ _ ∧ _ ⇒ split
| ⊢ iff _ _ ⇒ split
end.

Ltac jauto_set :=
  intros; jauto_set_hyps;
  intros; jauto_set_goal;
  unfold not in *.

```

22.6 Backward and Forward Chaining

22.6.1 Application

```

Ltac old_refine f :=
  refine f.

```

rappl 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 "rappl" 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 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
  ].

```

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

```

Ltac nrapply 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 _ _ _ _ _ _ _ _ _ _ _ _) ].

```

The tactics *applies*_{*N*} *T*, where *N* is a natural number, provides a more efficient way of using *applies* *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.`

`applies_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 "applies_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'.`

`applies_to H1,...,HN E` applies E to several hypotheses

`Tactic Notation "applies_to" hyp(H1) "," hyp(H2) constr(E) :=`
`applies_to H1 E; applies_to H2 E.`

`Tactic Notation "applies_to" hyp(H1) "," hyp(H2) "," hyp(H3) constr(E) :=`
`applies_to H1 E; applies_to H2 E; applies_to H3 E.`

`Tactic Notation "applies_to" hyp(H1) "," hyp(H2) "," hyp(H3) "," hyp(H4) constr(E)`
`:=`
`applies_to H1 E; applies_to H2 E; applies_to H3 E; applies_to H4 E.`

`constructors` calls `constructor` or `econstructor`.

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

22.6.2 Assertions

`asserts H: T` is another syntax for `assert (H : T)`, which also works with introduction patterns.

`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 \[H1 \[H2 \[.. HN\]\]\]: 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.

```

22.6.3 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 insterted. Underscore arguments $--$ are interpret 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 $tactic\ (\gg\ E1\ ..\ EN)$ 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 $tactic\ (\gg\ E1\ ..\ EN)$ 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_all 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_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_evar t A cont' | fail 3 ]
            end
          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
      end
  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 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 -> ?Q => first [ app_assert t U cont' | fail 3 ]
            | ∀ _ : U, _ => first
              [ app_typeclass t cont'
                | app_evar t U cont'
                | fail 3 ]
            | ?P -> ?Q => first [ app_assert t P cont | fail 3 ]
            | ∀ _ : ?A, _ => first
              [ app_typeclass t cont
                | 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-evan t Type cont
                      | fail 3 ]
      | _ ⇒ first [ app-evan t Type cont
                   | fail 3 ]
    end
  | ∀ ..?A, _ ⇒
    let V := type of v in
    match type of V with
    | Prop ⇒ first [ app-typeclass t cont
                    | app-evan t A cont
                    | fail 3 ]
    | _ ⇒ first [ cont' (t v)
                  | app-typeclass t cont
                  | app-evan t A cont
                  | fail 3 ]
    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 : ($\gg E0 E1 \dots EN$) 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 : $E0 E1 \dots EN$ 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: ( $\gg E0 A1$ ).
```

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

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

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

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

```
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 *E0*). *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: (» E0 A1 A2 A3 A4 A5).

Tactic Notation "forwards" simple_intropattern(I1) simple_intropattern(I2)
  ":" constr(E) :=
  forwards [I1 I2]: E.

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

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

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

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

```


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

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

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

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*).

forwards_nounfold *l*: *E* is like *forwards* *l*: *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(*Ei*) :=
 let *args* := *list_boxer_of* *Ei* in
 let *args* := (eval simpl in (*args* ++ ((boxer ___)::nil))) in
build_app *args* ltac:(fun *R* ⇒ *lets_base* *I* *R*).

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

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

applys (» *E0 E1 .. 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 .. 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 "applies" constr(E0) constr(A1) :=
  applies (» E0 A1).
Tactic Notation "applies" constr(E0) constr(A1) constr(A2) :=
  applies (» E0 A1 A2).
Tactic Notation "applies" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applies (» E0 A1 A2 A3).
Tactic Notation "applies" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  applies (» E0 A1 A2 A3 A4).
Tactic Notation "applies" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  applies (» E0 A1 A2 A3 A4 A5).

  fapplies (» E0 E1 .. EN) instantiates lemma E0 on the arguments Ei and on the argument
  --- meaning that all evargs should be explicitly instantiated, and apply the resulting term to
  the current goal. fapplies E0 E1 E2 .. EN is also available.

Ltac fapplies_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 "fapplies" constr(E0) :=
  match type of E0 with
  | list Boxer => fapplies_build E0
  | _ => fapplies_build (» E0)
  end.

Tactic Notation "fapplies" constr(E0) constr(A1) :=
  fapplies (» E0 A1).
Tactic Notation "fapplies" constr(E0) constr(A1) constr(A2) :=
  fapplies (» E0 A1 A2).
Tactic Notation "fapplies" constr(E0) constr(A1) constr(A2) constr(A3) :=
  fapplies (» E0 A1 A2 A3).
Tactic Notation "fapplies" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  fapplies (» E0 A1 A2 A3 A4).
Tactic Notation "fapplies" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  fapplies (» 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 ltac:(fun  $R \Rightarrow$  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  $H$  ( $\gg A1 A2$ ).

Tactic Notation "specializes" hyp( $H$ ) constr( $A1$ ) constr( $A2$ ) constr( $A3$ ) :=
  specializes  $H$  ( $\gg A1 A2 A3$ ).

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

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

  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 – Note: this could be
generalized.

Ltac specializes_var_base  $H$  :=
  match type of  $H$  with
  | ? $P \rightarrow ?Q \Rightarrow$  fail 1
  |  $\forall \_ :-, \_ \Rightarrow$  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  $H$ .

```

22.6.4 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 | applies H ].
```

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

```
Tactic Notation "sapply" constr(H) :=
  first [ apply H | eapply H | rapply H | applies H
         | hnf; apply H | hnf; eapply H | hnf; applies H
         | fapply H ].
```

22.6.5 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).
```

22.6.6 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* ∧ *B* by showing first *A* and then *A* → *B*, through the command *logic* (*foral* *A* *B*, *A* → (*A* → *B*) → *A* ∧ *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).

22.6.7 Application Modulo Equalities

The tactic *equates* replaces a goal of the form $P \ x \ y \ z$ with a goal of the form $P \ x \ ?a \ z$ 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* *il* ... *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* : $\forall (x1 : A1), \text{Type}$).

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 *equates_0* : $\forall (P \ Q : \text{Prop})$,

$P \rightarrow P = Q \rightarrow Q$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_1* :

$\forall (P : A0 \rightarrow \text{Prop}) \ x1 \ y1$,

$P \ y1 \rightarrow x1 = y1 \rightarrow P \ x1$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_2* :

$\forall \ y1 \ (P : A0 \rightarrow \forall (x1 : A1), \text{Prop}) \ x1 \ x2$,

$P \ y1 \ x2 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_3* :

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

$P \ y1 \ x2 \ x3 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2 \ x3$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_4* :

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

$P \ y1 \ x2 \ x3 \ x4 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2 \ x3 \ x4$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_5* :

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

$P \ y1 \ x2 \ x3 \ x4 \ x5 \rightarrow x1 = y1 \rightarrow P \ x1 \ x2 \ x3 \ x4 \ x5$.

Proof using. intros. subst. auto. Qed.

Lemma equates_6 :

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

Proof using. intros. subst. auto. Qed.

End equatesLemma.

Ltac equates_lemma n :=

match number_to_nat n with
| 0 $\Rightarrow$ constr:(equates_0)
| 1 $\Rightarrow$ constr:(equates_1)
| 2 $\Rightarrow$ constr:(equates_2)
| 3 $\Rightarrow$ constr:(equates_3)
| 4 $\Rightarrow$ constr:(equates_4)
| 5 $\Rightarrow$ constr:(equates_5)
| 6 $\Rightarrow$ constr:(equates_6)
| _ $\Rightarrow$ 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 $\Rightarrow$ constr:(E)
| _ $\Rightarrow$ constr:((boxer E)::nil)
end in
let rec go pos :=
match pos with
| nil $\Rightarrow$ cont tt
| (boxer ?n)::?pos' $\Rightarrow$ equates_one n; [go pos' |]
end in
go all_pos.

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

Tactic Notation "equates" constr(n1) constr(n2) :=
equates ($\gg$ n1 n2).

Tactic Notation "equates" constr(n1) constr(n2) constr(n3) :=
equates ($\gg$ n1 n2 n3).

Tactic Notation "equates" constr(n1) constr(n2) constr(n3) constr(n4) :=
equates ($\gg$ n1 n2 n3 n4).

applies_eq H i1 .. iK is the same as *equates i1 .. iK* followed by *applies H* on the first subgoal.

DEPRECATED: use *applies_eq H* instead.

Tactic Notation "applies_eq" constr(*H*) constr(*E*) :=

equates_several E ltac:(fun _ => *sapply H*).

Tactic Notation "applies_eq" constr(*H*) constr(*n1*) constr(*n2*) :=

applies_eq H (> *n1 n2*).

Tactic Notation "applies_eq" constr(*H*) constr(*n1*) constr(*n2*) constr(*n3*) :=

applies_eq H (> *n1 n2 n3*).

Tactic Notation "applies_eq" constr(*H*) constr(*n1*) constr(*n2*) constr(*n3*) constr(*n4*)

:=

applies_eq H (> *n1 n2 n3 n4*).

applies_eq H helps proving a goal of the form *P x1 .. xN* from an hypothesis *H* that concludes *P y1 .. yN*, where the arguments *xi* and *yi* 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 *applies K*, and attempts *reflexivity* on the side equalities.

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

P = *Q* $\rightarrow$

Q $\rightarrow$

P.

Proof using. intros. subst. auto. Qed.

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

P = *Q* $\rightarrow$

P T = *Q T*.

Proof using. intros. subst. auto. Qed.

Lemma *applies_eq_step* : $\forall A B (P Q:A \rightarrow B) x y$,

P = *Q* $\rightarrow$

x = *y* $\rightarrow$

P x = *Q y*.

Proof using. intros. subst. auto. Qed.

Ltac *applies_eq_loop tt* :=

match goal with

| $\vdash ?P ?x \Rightarrow$

first [eapply *applies_eq_step*; [*applies_eq_loop tt* |]

| eapply *applies_eq_step_dep*; *applies_eq_loop tt*]

| _ $\Rightarrow$ *reflexivity*

end.

Ltac *applies_eq_core H* :=

eapply *applies_eq_init*;

[*applies_eq_loop tt* | *applies H*];

```

    try reflexivity.
Tactic Notation "applies_eq" constr(H) :=
    applies_eq_core H.

```

22.6.8 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" :=
    exfalse.

```

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 $C\ x1 \dots xN = D\ y1 \dots yM$, 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* ($\gg E1 \dots EN$), which tries to apply *applies* ($\gg E1 \dots EN$) and if it does not work then tries *forwards H*: ($\gg E1 \dots EN$) followed with *false*

```

Ltac false_then E cont :=
    false_goal; first
    [ applies E
    | 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 ( $\gg E\ E1$ ).

```

```

Tactic Notation "false" constr(E) constr(E1) constr(E2) :=
    false ( $\gg E\ E1\ E2$ ).

```

```

Tactic Notation "false" constr(E) constr(E1) constr(E2) constr(E3) :=
    false ( $\gg E\ E1\ E2\ E3$ ).

```


Tactic Notation "false" constr(*E*) constr(*E1*) constr(*E2*) constr(*E3*) constr(*E4*) :=
false ($\gg$ *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*:_ $\vdash$ _ $\Rightarrow$
 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* $\neq$?*x* $\vdash$ _ $\Rightarrow$
false_goal; apply *H*; reflexivity end.

22.7 Introduction and Generalization

22.7.1 Introduction

introv is used to name only non-dependent hypothesis.

- If *introv* 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 *introv* 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 *introv* does nothing at all.

```

Ltac introv_rec :=
  match goal with
  |  $\vdash ?P \rightarrow ?Q$   $\Rightarrow$  idtac
  |  $\vdash \forall \_, \_ \Rightarrow$  intro; introv_rec
  |  $\vdash \_ \Rightarrow$  idtac
  end.

Ltac introv_noarg :=
  match goal with
  |  $\vdash ?P \rightarrow ?Q$   $\Rightarrow$  idtac
  |  $\vdash \forall \_, \_ \Rightarrow$  introv_rec
  |  $\vdash ?G \Rightarrow$  hnf;
    match goal with
    |  $\vdash ?P \rightarrow ?Q$   $\Rightarrow$  idtac
    |  $\vdash \forall \_, \_ \Rightarrow$  introv_rec
    end
  |  $\vdash \_ \Rightarrow$  idtac
  end.

Ltac introv_noarg_not_optimized :=
  intro; match goal with  $H:\vdash \_ \Rightarrow$  revert  $H$  end; introv_rec.

Ltac introv_arg  $H$  :=
  hnf; match goal with
  |  $\vdash ?P \rightarrow ?Q$   $\Rightarrow$  intros  $H$ 
  |  $\vdash \forall \_, \_ \Rightarrow$  intro; introv_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: _ $\vdash$ _ $\Rightarrow$ hnf in H end.

22.7.2 Introduction using $\Rightarrow$ and $=\Rightarrow$

Ltac *ltac_intros_post* := *idtac.*

Tactic Notation " $=>$ " :=
intros.

Tactic Notation " $=>$ " *simple_intropattern(I1) :=*

```

intros I1.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2) :=
  intros I1 I2.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) :=
  intros I1 I2 I3.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) :=
  intros I1 I2 I3 I4.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5) :=
  intros I1 I2 I3 I4 I5.
Tactic Notation "=>" 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 "=>" 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.

Ltac intro_nondeps_aux_special_intro G :=
  fail.

Ltac intro_nondeps_aux is_already_hnf :=
  match goal with
  | ⊢ (?P → ?Q) ⇒ idtac
  | ⊢ ?G → _ ⇒ intro_nondeps_aux_special_intro G;
    intros; intro_nondeps_aux true
  | ⊢ (∀ _,_) ⇒ 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 "=>" :=
  intro_nondeps tt.

Tactic Notation "=>" simple_intropattern(I1) :=
  =>; intros I1.

Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2) :=
  =>; intros I1 I2.

Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) :=
  =>; intros I1 I2 I3.

Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) :=
  =>; intros I1 I2 I3 I4.

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

Tactic Notation "=>" 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 "=>" 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.

```

22.7.3 Generalization

gen X1 .. XN is a shorthand for calling **generalize dependent** successively on variables *XN...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.

22.7.4 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 ← X: E in |-.

```

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) → 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
  | ⊢ context[let _ := ?E in _] ⇒ tac E; cbv zeta
  | H: context[let _ := ?E in _] ⊢ _ ⇒ tac E; cbv zeta in H
  end.

```

```

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

```

```

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

```

22.8 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 *).

```

erewrites *E* is similar to *erewrite* 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.

```

Tactic Notation "erewrites" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ erewrite M ).
Tactic Notation "erewrites" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ erewrite M in H).
Tactic Notation "erewrites" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ erewrite M in *).
Tactic Notation "erewrites" "<-" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ erewrite ← M ).
Tactic Notation "erewrites" "<-" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ erewrite ← M in H).
Tactic Notation "erewrites" "<-" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ erewrite ← 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 *T1 = T2* 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" $\text{constr}(E)$ "at" $\text{constr}(K)$:=
 match type of E with $?T1 = ?T2 \Rightarrow$
 $\text{ltac\_action\_at } K \text{ of } T1 \text{ do } (\text{rewrites } E) \text{ end.}$

Tactic Notation "rewrites" " $<-$ " $\text{constr}(E)$ "at" $\text{constr}(K)$:=
 match type of E with $?T1 = ?T2 \Rightarrow$
 $\text{ltac\_action\_at } K \text{ of } T2 \text{ do } (\text{rewrites } \leftarrow E) \text{ end.}$

Tactic Notation "rewrites" $\text{constr}(E)$ "at" $\text{constr}(K)$ "in" $\text{hyp}(H)$:=
 match type of E with $?T1 = ?T2 \Rightarrow$
 $\text{ltac\_action\_at } K \text{ of } T1 \text{ in } H \text{ do } (\text{rewrites } E \text{ in } H) \text{ end.}$

Tactic Notation "rewrites" " $<-$ " $\text{constr}(E)$ "at" $\text{constr}(K)$ "in" $\text{hyp}(H)$:=
 match type of E with $?T1 = ?T2 \Rightarrow$
 $\text{ltac\_action\_at } K \text{ of } T2 \text{ in } H \text{ do } (\text{rewrites } \leftarrow E \text{ in } H) \text{ end.}$

22.8.1 Replace

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

Tactic Notation "replaces" $\text{constr}(E)$ "with" $\text{constr}(F)$:=
 let $T := \text{fresh "TEMP"}$ in assert ($T: E = F$); [| $\text{replace } E \text{ with } F$; clear T].

Tactic Notation "replaces" $\text{constr}(E)$ "with" $\text{constr}(F)$ "in" $\text{hyp}(H)$:=
 let $T := \text{fresh "TEMP"}$ in assert ($T: E = F$); [| $\text{replace } E \text{ with } F \text{ in } H$; clear T].

$\text{replaces } E \text{ at } K \text{ with } F$ replaces the K -th occurrence of E with F in the current goal. Syntax $\text{replaces } E \text{ at } K \text{ with } F \text{ in } H$ is also available.

Tactic Notation "replaces" $\text{constr}(E)$ "at" $\text{constr}(K)$ "with" $\text{constr}(F)$:=
 let $T := \text{fresh "TEMP"}$ in assert ($T: E = F$); [| $\text{rewrites } T \text{ at } K$; clear T].

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

22.8.2 Change

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" $\text{constr}(E1)$ "with" $\text{constr}(E2)$ "in" $\text{hyp}(H)$:=
 $\text{asserts\_rewrite } (E1 = E2) \text{ in } H$; [reflexivity |].

Tactic Notation "changes" $\text{constr}(E1)$ "with" $\text{constr}(E2)$:=
 $\text{asserts\_rewrite } (E1 = E2)$; [reflexivity |].

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

22.8.3 Renaming

renames *X1* to *Y1*, ..., *XN* to *YN* is a shorthand for a sequence of renaming operations *rename* *Xi* into *Yi*.

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*.

22.8.4 Unfolding

unfolds unfolds the head definition in the goal, i.e. if the goal has form *P x1 ... xN* 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
 | $\forall \_,\_ \Rightarrow$ *intros*; *apply_to_head_of* *E* *cont*
 | $?A = ?B \Rightarrow$ *first* [*go* *A* | *go* *B*]
 | $?A \Rightarrow$ *go* *A*

```

end.

Ltac unfolds_base :=
  match goal with ⊢ ?G ⇒
    apply_to_head_of G ltac:(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 x1 ... xN
  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,...,HN 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.

  unfolds P1,...,PN is a shortcut for unfold P1,...,PN 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 *.

folds P1,...,*PN* is a shortcut for fold *P1* in *; ..; fold *PN* 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*.

22.8.5 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 $F := (\text{eval simpl in } E) \text{ in change } F \text{ with } E.$

$\text{unsimpl } E \text{ in } H$ is similar to $\text{unsimpl } E$ but it applies inside a particular hypothesis H .

Tactic Notation "unsimpl" constr(E) "in" $\text{hyp}(H) :=$

let $F := (\text{eval simpl in } E) \text{ in change } F \text{ with } E \text{ in } H.$

$\text{unsimpl } E \text{ in } *$ applies $\text{unsimpl } E$ everywhere possible. $\text{unsimpls } E$ is a synonymous.

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

let $F := (\text{eval simpl in } E) \text{ in change } F \text{ with } E \text{ in } *.$

Tactic Notation "unsimpls" constr(E) :=

$\text{unsimpl } E \text{ in } *.$

$\text{nosimpl } t$ protects the Rocq term t against some forms of simplification. See Gonthier's work for details on this trick.

Notation "'nosimpl' t" := (match tt with tt $\Rightarrow$ t end)
(at level 10).

22.8.6 Reduction

Tactic Notation "hnfs" := hnf in *.

22.8.7 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 $H: ?x = ?y \vdash \_ \Rightarrow$
first [$\text{subst } x$ | $\text{subst } y$] 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 $\text{substs\_below limit} :=$

match goal with $H: ?T \vdash \_ \Rightarrow$
match T with
| $\text{limit} \Rightarrow \text{idtac}$
| $?x = ?y \Rightarrow$
first [$\text{subst } x; \text{substs\_below limit}$
| $\text{subst } y; \text{substs\_below limit}$
| $\text{generalizes } H; \text{substs\_below limit}; \text{intro}$]
end end.

$\text{substs below body } E$ 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 $\text{substs below } H$ 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
 | (,_,_,_,_) = (,_,_,_,_) ⇒ injection *H*; clear *H*; do 4 *intro_subst_hyp*
 | (,_,_,_) = (,_,_,_) ⇒ injection *H*; clear *H*; do 4 *intro_subst_hyp*
 | (,_,_) = (,_,_) ⇒ injection *H*; clear *H*; do 3 *intro_subst_hyp*
 | (,_) = (,_) ⇒ injection *H*; clear *H*; do 2 *intro_subst_hyp*
 | ?*x* = ?*x* ⇒ clear *H*
 | ?*x* = ?*y* ⇒ 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* := _ ⊢ _ ⇒ 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.

22.8.8 Tactics to Work with Proof Irrelevance

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

22.8.9 Proving Equalities

The tactic *fequal* enhances Rocq's tactic **f_equal**, which does not simplify equalities between tuples, nor between dependent pairs of the form *exist* _ _ or *existT* _ -. For support of dependent pairs, the file *LibEqual* must be imported.

Subgoals solvable by **reflexivity** are automatically discharged. See also the the variant *fequals*, which discharges more subgoals.

Note: only **args_eq_2** is actually useful for the implementation of *fequal*, if we rely on Rocq's **f_equal** tactic for other arities. We provide these lemmas to show the pattern of lemmas to exploit for implementing *fequal* independently of **f_equal**.

Section FuncEq.

Variables (*A1 A2 A3 A4 A5 A6 A7 B* : Type).

```
Lemma args_eq_1 : ∀ (f:A1→B) x1 y1,
  x1 = y1 →
  f x1 = f y1.
```

Proof using. intros. subst. auto. 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. auto. 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. auto. 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. auto. Qed.

```
Lemma args_eq_5 : ∀ (f:A1→A2→A3→A4→A5→B) x1 y1 x2 y2 x3 y3 x4 y4 x5 y5,
```

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow x5 = y5 \rightarrow$
 $f\ x1\ x2\ x3\ x4\ x5 = f\ y1\ y2\ y3\ y4\ y5.$

Proof using. intros. subst. auto. Qed.

Lemma args_eq_6 : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow A6 \rightarrow B)\ x1\ y1\ x2\ y2\ x3\ y3\ x4\ y4\ x5\ y5\ x6\ y6,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow x5 = y5 \rightarrow x6 = y6 \rightarrow$
 $f\ x1\ x2\ x3\ x4\ x5\ x6 = f\ y1\ y2\ y3\ y4\ y5\ y6.$

Proof using. intros. subst. auto. Qed.

Lemma args_eq_7 : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow A6 \rightarrow A7 \rightarrow B)\ x1\ y1\ x2\ y2\ x3\ y3\ x4\ y4\ x5\ y5\ x6\ y6\ x7\ y7,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow x5 = y5 \rightarrow x6 = y6 \rightarrow x7 = y7 \rightarrow$
 $f\ x1\ x2\ x3\ x4\ x5\ x6\ x7 = f\ y1\ y2\ y3\ y4\ y5\ y6\ y7.$

Proof using. intros. subst. auto. Qed.

End FuncEq.

Ltac fequal_post :=
 try solve [reflexivity].

fequal_support_for_exist, implemented in *LibEqual*, is meant to simplify goals of the form $exist\ \_ = exist\ \_$ and $existT\ \_ = existT\ \_$, by exploiting proof irrelevance.

Ltac fequal_support_for_exist tt :=
 fail.

For a n-ary tuple, *fequal*, unlike *f_equal* enforces a recursive call on the (n-1)-ary tuple associated with the right component.

Ltac fequal_base :=
 match goal with
 | $\vdash (\_,\_,\_) = (\_,\_,\_) \Rightarrow$ apply args_eq_2; [*fequal_base* |]
 | $\vdash \_ \Rightarrow$ first
 [*fequal_support_for_exist* tt
 | apply args_eq_1
 | apply args_eq_2
 | apply args_eq_3
 | apply args_eq_4
 | apply args_eq_5
 | apply args_eq_6
 | apply args_eq_7
 | f_equal]
 end.

Tactic Notation "fequal" :=
fequal_base; *fequal_post*.

fequals is the same as *fequal* except that it tries and solve all trivial subgoals, using *reflexivity* and *congruence* (as well as the proof-irrelevance principle). *fequals* applies to

goals of the form $f\ x1 \dots xN = f\ y1 \dots yN$ and produces some subgoals of the form $xi = yi$).

`Ltac fequals_post :=`

`try solve [ reflexivity | congruence | apply proof_irrelevance ].`

`Tactic Notation "fequals" :=`

`fequal; fequals_post.`

fequals_rec calls *fequals* recursively. It is equivalent to `repeat (progress fequals)`.

`Tactic Notation "fequals_rec" :=`

`repeat (progress fequals).`

22.9 Inversion

22.9.1 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 *introv* is used to name only hypotheses.

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

`invert keep H; introv I1.`

`Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I1)`

`simple_intropattern(I2) :=`

`invert keep H; introv I1 I2.`

`Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I1)`

`simple_intropattern(I2) simple_intropattern(I3) :=`

`invert keep H; introv 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).

```

22.9.2 Inversion with Substitution

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

```

Axiom inj_pair2 :
  ∀ (U : Type) (P : U → Type) (p : U) (x y : P p),
  existT P p x = existT P p y → 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*. Note: maybe 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 => invert H).
Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1) :=
  lets_inverts_base E ltac:(fun H => invert H as I1).
Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1)
  simple_intropattern(I2) :=
  lets_inverts_base E ltac:(fun H => invert 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 => invert 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 => invert H as I1 I2 I3 I4).
```

22.9.3 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.

```

22.9.4 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) :=
  inversion H; 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) :=
  inversion H; 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) :=
  injection H; 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) :=
  injection H; clear H; intros; subst.

```

22.9.5 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) :=
  let X := fresh "TEMP" in
  set (X := E) in *; def_to_eq_sym X H E;
  destruct X.
```

```
Tactic Notation "cases" constr(E) :=
  let H := fresh "Eq" in cases E 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 :=
  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
  | ⊢ context [if ?B then _ else _] ⇒ cases_if_on B as Eq
  | K: context [if ?B then _ else _] ⊢ _ ⇒ 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 _] ⇒
    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  $Eq$   $Eq$ .

  cases' is provided for compatibility with remember.
  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
  |  $\{-\}+\{-\}$   $\Rightarrow$  destruct  $E$  as [ $Eq|Eq$ ]; try subst_hyp  $Eq$ 
  | _  $\Rightarrow$  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.

```

22.10 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*.

From *Stdlib.Program* Require Import 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 (fun a ⇒ *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 $\text{wf } R$ 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 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
        | applies well_founded_ind E ];
  clearX tt;
  first [ induction_wf_process_wf_hyp tt
        | intros X; cont tt ].

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

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)

```
From Stdlib Require Import Compare_dec.
From Stdlib Require Import Lia.
```

```
Lemma induct_height_max2 :  $\forall n1\ n2 : \text{nat},$ 
   $\exists n, n1 < n \wedge n2 < n.$ 
```

Proof using.

```
intros. destruct (lt_dec n1 n2).
 $\exists (S\ n2).$  lia.
 $\exists (S\ n1).$  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  
  |  $\_ \Rightarrow \exists (S\ x)$ ; eauto  
end.
```

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

22.11 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.
```

22.12 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.
```



```

| -  $\wedge$  -  $\wedge$  -  $\wedge$  -  $\wedge$  -  $\wedge$  -  $\Rightarrow$  constr:(6)
| -  $\wedge$  -  $\wedge$  -  $\wedge$  -  $\wedge$  -  $\Rightarrow$  constr:(5)
| -  $\wedge$  -  $\wedge$  -  $\wedge$  -  $\Rightarrow$  constr:(4)
| -  $\wedge$  -  $\wedge$  -  $\Rightarrow$  constr:(3)
| -  $\wedge$  -  $\Rightarrow$  constr:(2)
| -  $\rightarrow$  ? $T'$   $\Rightarrow$  get_term_conjunction_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_conjunction_arity  $T'$ 
      end

```

end.

Ltac *get_goal_conjunction_arity* :=
match **goal** **with** $\vdash$? $T \Rightarrow$ *get_term_conjunction_arity* T **end.**

splits applies to a goal of the form $(T1 \wedge \dots \wedge TN)$ and destruct it into N subgoals $T1$.. TN . 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 .

N-ary Conjunctions Deconstruction
 Underlying implementation of *destructs*.

Ltac *destructs_conjunction_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** [? [? [? [? ?]]]]
 | 6 $\Rightarrow$ **destruct** T **as** [? [? [? [? [? ?]]]]]
 | 7 $\Rightarrow$ **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.

```

Proving Goals that are *N*-ary Disjunctions
 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)
  | _ ∨ _ ∨ _ => constr:(3)
  | _ ∨ _ => constr:(2)
  | _ → ?T' => get_term_disjunction_arity T'
  | _ => let P := get_head T in
    let T' := eval unfold P in T in
    match T' with
    | T => fail 1
    | _ => get_term_disjunction_arity T'
    end
  end

```

end.

```
Ltac get_goal_disjunction_arity :=
  match goal with ⊢ ?T ⇒ get_term_disjunction_arity T end.
```

branch N applies to a goal of the form $P1 \vee \dots \vee PK \vee \dots \vee PN$ and leaves the goal PK . It is 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 $P1 \vee \dots \vee PK \vee \dots \vee PN$ and leaves the goal PK .

```
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.
```

N-ary Disjunction Deconstruction
Underlying implementation of *branches*.

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

branches T allows destructing a term T which is a N -ary disjunction. It is equivalent to `destruct T as [  $H1$  | .. |  $HN$  ]`, 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*: _ $\vee$ _ $\vdash$ _ $\Rightarrow$ *branches h* end.

N-ary Existentials

$\exists T1 \dots TN$ is a shorthand for $\exists T1; \dots; \exists TN$. 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. $\exists T1 .. TN \text{ --}$ is equivalent to $\exists T1 .. TN \text{ -- -- --}$ with as many *--* as possible.

Tactic Notation "exists_original" constr(*T1*) :=
 $\exists T1$.

Tactic Notation "exists" constr(*T1*) :=
 match *T1* with
 | ltac_wild $\Rightarrow$ *esplit*
 | ltac_wilds $\Rightarrow$ repeat *esplit*
 | _ $\Rightarrow \exists T1$
 end.

Tactic Notation "exists" constr(*T1*) constr(*T2*) :=
 $\exists T1; \exists T2$.

Tactic Notation "exists" constr(*T1*) constr(*T2*) constr(*T3*) :=
 $\exists T1; \exists T2; \exists T3$.

Tactic Notation "exists" constr(*T1*) constr(*T2*) constr(*T3*) constr(*T4*) :=
 $\exists T1; \exists T2; \exists T3; \exists T4$.

Tactic Notation "exists" constr(*T1*) constr(*T2*) constr(*T3*) constr(*T4*)
 constr(*T5*) :=
 $\exists T1; \exists T2; \exists T3; \exists T4; \exists T5$.

Tactic Notation "exists" constr(*T1*) constr(*T2*) constr(*T3*) constr(*T4*)
 constr(*T5*) constr(*T6*) :=
 $\exists T1; \exists T2; \exists T3; \exists T4; \exists T5; \exists T6$.

For compatibility with Rocq syntax, $\exists T1, \dots, TN$ is also provided.

Tactic Notation "exists" constr(*T1*) ", " constr(*T2*) :=
 $\exists T1 T2$.

Tactic Notation "exists" constr(*T1*) ", " constr(*T2*) ", " constr(*T3*) :=
 $\exists T1 T2 T3$.

Tactic Notation "exists" constr(*T1*) ", " constr(*T2*) ", " constr(*T3*) ", " constr(*T4*) :=
 $\exists T1 T2 T3 T4$.

Tactic Notation "exists" constr(*T1*) ", " constr(*T2*) ", " constr(*T3*) ", " constr(*T4*) ", "
 constr(*T5*) :=
 $\exists T1 T2 T3 T4 T5$.

Tactic Notation "exists" constr(*T1*) ", " constr(*T2*) ", " constr(*T3*) ", " constr(*T4*) ", "
 constr(*T5*) ", " constr(*T6*) :=
 $\exists T1 T2 T3 T4 T5 T6$.

The tactic $\exists$, without arguments, repeats *esplit* as many times as there are visible existentials at the head of the goal. In 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_positive 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.
```

Existentials and Conjunctions in Hypotheses

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
  |  $\_ \wedge \_ \Rightarrow$ 
      let h1 := fresh "TEMP" in
      let h2 := fresh "TEMP" in
      destruct H as [ h1 h2 ];
      unpack_hypothesis h1;
```

```

      unpack_hypothesis h2
|  $\exists$  (varname: _), _  $\Rightarrow$ 

      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.

```

22.15 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 ].

```

22.16 Tactics to Invoke Automation

22.16.1 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.

```

22.16.2 *hint* to Add Hints Local to a Lemma

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

Tactic Notation "hint" constr(*E*) :=

let *H* := fresh "Hint" in *lets 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).

22.16.3 *jauto*, a New Automation Tactic

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].

22.16.4 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 := try solve [*jauto*].

$\text{autos}\neg$ 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 $\text{autos}\neg$

Tactic Notation "autos" :=

auto_tilde.

Tactic Notation "autos" "~" :=

auto_tilde.

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

lets: E1; auto_tilde.

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

lets: E1; autos $\neg$ E2.

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

lets: E1; autos $\neg$ E2 E3.

Tactic Notation "autos" "~" constr(*E1*) constr(*E2*) constr(*E3*) constr(*E4*) :=

lets: E1; autos $\neg$ E2 E3 E4.

Tactic Notation "autos" "~" constr(*E1*) constr(*E2*) constr(*E3*) constr(*E4*)

constr(*E5*):=

lets: E1; autos $\neg$ E2 E3 E4 E5.

Tactic Notation "autos" :=

auto_tilde.

Tactic Notation "autos" "~" :=

auto_tilde.

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

lets: E1; auto_tilde.

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

lets: E1; autos $\neg$ E2.

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

lets: E1; autos $\neg$ E2 E3.

Tactic Notation "autos" "~" constr(*E1*) "," constr(*E2*) "," constr(*E3*) "," constr(*E4*)

:=

lets: E1; autos $\neg$ E2 E3 E4.

Tactic Notation "autos" "~" constr(*E1*) "," constr(*E2*) "," constr(*E3*) "," constr(*E4*)

","

constr(*E5*):=

lets: E1; autos $\neg$ E2 E3 E4 E5.

$\text{autos}\times$ 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; autos× E2.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) :=
  lets: E1; autos× E2 E3.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) constr(E4) :=
  lets: E1; autos× E2 E3 E4.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) constr(E4)
  constr(E5):=
  lets: E1; autos× E2 E3 E4 E5.

Tactic Notation "autos" "*" :=
  auto_star.
Tactic Notation "autos" "*" constr(E1) :=
  lets: E1; auto_star.
Tactic Notation "autos" "*" constr(E1) "," constr(E2) :=
  lets: E1; autos× E2.
Tactic Notation "autos" "*" constr(E1) "," constr(E2) "," constr(E3) :=
  lets: E1; autos× E2 E3.
Tactic Notation "autos" "*" constr(E1) "," constr(E2) "," constr(E3) "," constr(E4)
:=
  lets: E1; autos× E2 E3 E4.
Tactic Notation "autos" "*" constr(E1) "," constr(E2) "," constr(E3) "," constr(E4)
  ","
  constr(E5):=
  lets: E1; autos× E2 E3 E4 E5.

  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¬ and auto_false× 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  $\Rightarrow$  auto).
Tactic Notation "auto_false" "~" :=
  auto_false_base ltac:(fun tt  $\Rightarrow$  auto_tilde).
Tactic Notation "auto_false" "*" :=
  auto_false_base ltac:(fun tt  $\Rightarrow$  auto_star).

Tactic Notation "dauto" :=
  dintuition eauto.

```

22.16.5 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 "applies_eq" "~" constr(H) :=
  applies_eq H; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(E) :=
  applies_eq H E; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(n1) constr(n2) :=
  applies_eq H n1 n2; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) :=
  applies_eq H n1 n2 n3; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) constr(n4)
:=
  applies_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  $\neg$  ( $\gg$  E0 E1).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) :=
    false  $\neg$  ( $\gg$  E0 E1 E2).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) constr(E3) :=
    false  $\neg$  ( $\gg$  E0 E1 E2 E3).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) constr(E3) constr(E4)
:=
    false  $\neg$  ( $\gg$  E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "~" :=
    try solve [ false  $\neg$  ].
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  $\neg$  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" "~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
  forwards I: E0 A1 A2; auto_tilde.
Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  forwards I: E0 A1 A2 A3; auto_tilde.
Tactic Notation "forwards" "~" 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" "~" *simple_intropattern(I)* ":" 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" "~" ":" constr(*E*) :=
forwards: E; auto_tilde.

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

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

Tactic Notation "forwards" "~" ":" constr(*E0*)
 constr(*A1*) constr(*A2*) constr(*A3*) :=
forwards: E0 A1 A2 A3; auto_tilde.

Tactic Notation "forwards" "~" ":" constr(*E0*)
 constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*) :=
forwards: E0 A1 A2 A3 A4; auto_tilde.

Tactic Notation "forwards" "~" ":" constr(*E0*)
 constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*) constr(*A5*) :=
forwards: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "applies" "~" constr(*H*) :=
supply H; auto_tilde. Tactic Notation "applies" "~" constr(*E0*) constr(*A1*) :=
applies E0 A1; auto_tilde.

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

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

Tactic Notation "applies" "~" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) :=
applies E0 A1 A2 A3; auto_tilde.

Tactic Notation "applies" "~" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*)
 :=
applies E0 A1 A2 A3 A4; auto_tilde.

Tactic Notation "applies" "~" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*)
 constr(*A5*) :=
applies E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "specializes" "~" *hyp(H)* :=
specializes H; auto_tilde.

Tactic Notation "specializes" "~" *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" "~" constr(E) :=
    fapply E; auto_tilde.
Tactic Notation "sapply" "~" constr(E) :=
    sapply E; auto_tilde.
Tactic Notation "logic" "~" constr(E) :=
    logic_base E ltac:(fun _ => auto_tilde).
Tactic Notation "intros_all" "~" :=
    intros_all; auto_tilde.
Tactic Notation "unfolds" "~" :=
    unfolds; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) :=
    unfolds F1; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) "," constr(F2) :=
    unfolds F1, F2; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) "," constr(F2) "," constr(F3) :=
    unfolds F1, F2, F3; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) "," constr(F2) "," constr(F3) ","
constr(F4) :=
    unfolds F1, F2, F3, F4; auto_tilde.
Tactic Notation "simple" "~" :=
    simpl; auto_tilde.
Tactic Notation "simple" "~" "in" hyp(H) :=
    simpl in H; auto_tilde.
Tactic Notation "simpls" "~" :=
    simpls; auto_tilde.
Tactic Notation "hnfs" "~" :=
    hnfs; auto_tilde.
Tactic Notation "hnfs" "~" "in" hyp(H) :=
    hnf in H; auto_tilde.
Tactic Notation "subst" "~" :=
    subst; 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" "~" constr(E) "in" hyp(H) :=
    rewrite E in H; auto_tilde.
Tactic Notation "rewrite" "~" "<-" constr(E) "in" hyp(H) :=
    rewrite ← E in H; auto_tilde.

Tactic Notation "rewrites" "~" constr(E) :=
    rewrites E; auto_tilde.
Tactic Notation "rewrites" "~" constr(E) "in" hyp(H) :=
    rewrites E in H; auto_tilde.
Tactic Notation "rewrites" "~" constr(E) "in" "*" :=
    rewrites E in *; auto_tilde.
Tactic Notation "rewrites" "~" "<-" constr(E) :=
    rewrites ← E; auto_tilde.
Tactic Notation "rewrites" "~" "<-" constr(E) "in" hyp(H) :=
    rewrites ← E in H; auto_tilde.
Tactic Notation "rewrites" "~" "<-" constr(E) "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" "~" constr(E) "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" "~" constr(E) "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 "erewrites" "~" constr(*E*) :=
 erewrites *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" "~" constr(*E*) "as" ident(*H*) :=
 cases *E* as *H*; *auto_tilde*.
 Tactic Notation "cases" "~" constr(*E*) :=
 cases *E*; *auto_tilde*.
 Tactic Notation "case_if" "~" :=


```

    case_if; auto_tilde.
Tactic Notation "case_ifs" "~" :=
    case_ifs; auto_tilde.
Tactic Notation "case_if" "~" "in" hyp(H) :=
    case_if in H; auto_tilde.
Tactic Notation "cases_if" "~" :=
    cases_if; auto_tilde.
Tactic Notation "cases_if" "~" "in" hyp(H) :=
    cases_if in H; auto_tilde.
Tactic Notation "destruct_if" "~" :=
    destruct_if; auto_tilde.
Tactic Notation "destruct_if" "~" "in" hyp(H) :=
    destruct_if in H; auto_tilde.
Tactic Notation "cases'" "~" constr(E) "as" ident(H) :=
    cases' E as H; auto_tilde.
Tactic Notation "cases'" "~" constr(E) :=
    cases' E; auto_tilde.
Tactic Notation "cases_if'" "~" "as" ident(H) :=
    cases_if' as H; auto_tilde.
Tactic Notation "cases_if'" "~" :=
    cases_if'; auto_tilde.
Tactic Notation "decides_equality" "~" :=
    decides_equality; auto_tilde.
Tactic Notation "iff" "~" :=
    iff; auto_tilde.
Tactic Notation "iff" "~" simple_intropattern(I) :=
    iff I; auto_tilde.
Tactic Notation "splits" "~" :=
    splits; auto_tilde.
Tactic Notation "splits" "~" constr(N) :=
    splits N; auto_tilde.
Tactic Notation "destructs" "~" 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.

```

22.16.6 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 *Rocq.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 "applies" "*" constr(H) :=
  supply H; auto_star. Tactic Notation "applies" "*" constr(E0) constr(A1) :=
  applies E0 A1; auto_star.
Tactic Notation "applies" "*" constr(E0) constr(A1) :=
  applies E0 A1; auto_star.
Tactic Notation "applies" "*" constr(E0) constr(A1) constr(A2) :=
  applies E0 A1 A2; auto_star.
Tactic Notation "applies" "*" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applies E0 A1 A2 A3; auto_star.
Tactic Notation "applies" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  applies E0 A1 A2 A3 A4; auto_star.
Tactic Notation "applies" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
  constr(A5) :=
    applies 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 "subst" "*" :=
  subst; 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  $\leftarrow$  E; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" hyp(H):=
  rewrites  $\leftarrow$  E in H; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" "*":=
  rewrites  $\leftarrow$  E in *; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) :=
  rewrite_all E; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) :=
  rewrite_all  $\leftarrow$  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  $\leftarrow$  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  $\leftarrow$  E in *; auto_star.
Tactic Notation "asserts_rewrite" "*" constr(E) :=
  asserts_rewrite E; auto_star.
Tactic Notation "asserts_rewrite" "*" "<-" constr(E) :=
  asserts_rewrite  $\leftarrow$  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  $\leftarrow$  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  $\leftarrow$  E in *; auto_tilde.
Tactic Notation "cuts_rewrite" "*" constr(E) :=
  cuts_rewrite E; auto_star.
Tactic Notation "cuts_rewrite" "*" "<-" constr(E) :=
  cuts_rewrite  $\leftarrow$  E; auto_star.
Tactic Notation "cuts_rewrite" "*" constr(E) "in" hyp(H) :=
  cuts_rewrite E in H; auto_star.

```


Tactic Notation "cuts_rewrite" "*" "<-" *constr*(*E*) "in" *hyp*(*H*) :=
 cuts_rewrite $\leftarrow$ *E* in *H*; *auto_star*.

Tactic Notation "erewrite" "*" *constr*(*E*) :=
 erewrite *E*; *auto_star*.

Tactic Notation "erewrites" "*" *constr*(*E*) :=
 erewrites *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*) :=

$\text{destruct\_if in } H; \text{auto\_star}.$
Tactic Notation "cases'" "*" $\text{constr}(E)$ "as" $\text{ident}(H)$:=
 $\text{cases' } E \text{ as } H; \text{auto\_star}.$
Tactic Notation "cases'" "*" $\text{constr}(E)$:=
 $\text{cases' } E; \text{auto\_star}.$
Tactic Notation "cases_if'" "*" "as" $\text{ident}(H)$:=
 $\text{cases\_if' as } H; \text{auto\_star}.$
Tactic Notation "cases_if'" "*" :=
 $\text{cases\_if' }; \text{auto\_star}.$
Tactic Notation "decides_equality" "*" :=
 $\text{decides\_equality}; \text{auto\_star}.$
Tactic Notation "iff" "*" :=
 $\text{iff}; \text{auto\_star}.$
Tactic Notation "iff" "*" $\text{simple\_intropattern}(I)$:=
 $\text{iff } I; \text{auto\_star}.$
Tactic Notation "splits" "*" :=
 $\text{splits}; \text{auto\_star}.$
Tactic Notation "splits" "*" $\text{constr}(N)$:=
 $\text{splits } N; \text{auto\_star}.$
Tactic Notation "destructs" "*" $\text{constr}(T)$:=
 $\text{destructs } T; \text{auto\_star}.$
Tactic Notation "destructs" "*" $\text{constr}(N)$ $\text{constr}(T)$:=
 $\text{destructs } N \ T; \text{auto\_star}.$
Tactic Notation "branch" "*" $\text{constr}(N)$:=
 $\text{branch } N; \text{auto\_star}.$
Tactic Notation "branch" "*" $\text{constr}(K)$ "of" $\text{constr}(N)$:=
 $\text{branch } K \text{ of } N; \text{auto\_star}.$
Tactic Notation "branches" "*" $\text{constr}(T)$:=
 $\text{branches } T; \text{auto\_star}.$
Tactic Notation "branches" "*" $\text{constr}(N)$ $\text{constr}(T)$:=
 $\text{branches } N \ T; \text{auto\_star}.$
Tactic Notation "exists" "*" :=
 $\exists; \text{auto\_star}.$
Tactic Notation "exists" "*" $\text{constr}(T1)$:=
 $\exists \ T1; \text{auto\_star}.$
Tactic Notation "exists" "*" $\text{constr}(T1)$ $\text{constr}(T2)$:=
 $\exists \ T1 \ T2; \text{auto\_star}.$
Tactic Notation "exists" "*" $\text{constr}(T1)$ $\text{constr}(T2)$ $\text{constr}(T3)$:=
 $\exists \ T1 \ T2 \ T3; \text{auto\_star}.$
Tactic Notation "exists" "*" $\text{constr}(T1)$ $\text{constr}(T2)$ $\text{constr}(T3)$ $\text{constr}(T4)$:=
 $\exists \ T1 \ T2 \ T3 \ T4; \text{auto\_star}.$

```

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

```

22.17 Tactics to Sort Out the Proof Context

22.17.1 Hiding Hypotheses

Definition `ltac_something` (*P*:Type) (*e*:*P*) := *e*.

Notation "'Something'" :=
 (@ltac_something - -).

Lemma `ltac_something_eq` : ∀ (*e*:Type),
e = (@ltac_something - *e*).

Proof using. `auto`. `Qed`.

Lemma `ltac_something_hide` : ∀ (*e*:Type),
e → (@ltac_something - *e*).

Proof using. `auto`. `Qed`.

Lemma `ltac_something_show` : ∀ (*e*:Type),
 (@ltac_something - *e*) → *e*.

Proof using. `auto`. `Qed`.

hide_def *x* and *show_def* *x* can be used to hide/show the body of the definition *x*.

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.

```

22.17.2 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.
```

22.17.3 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 H1 H2 .. HN 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 :
```

```
  ∀ x y:nat, y < 2 → x = x → x ≥ 2 → x < 3 → 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.

```

22.18 Tactics for Development Purposes

22.18.1 Skipping Subgoals

SF DOES NOT NEED an alternative implementation of the *skip* tactic.

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 :=
  exfalse; 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 $\vdash ?G \Rightarrow$ *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 $K \Rightarrow$..) 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.

```

22.19 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 compability definitions through the top-level command: `Import LIBTACTICSCOMPATIBILITY`.

```

Module LIBTACTICSCOMPATIBILITY.
  Tactic Notation "apply" "*" constr(H) :=
    supply H; auto_star.
  Tactic Notation "subst" "*" :=
    subst; auto_star.
End LIBTACTICSCOMPATIBILITY.

Open Scope nat_scope.

```

Chapter 23

Library `PLF.UseTactics`

23.1 UseTactics: Tactic Library for Rocq: A Gentle Introduction

Rocq comes with a set of builtin tactics, such as `reflexivity`, `intros`, `inversion` and so on. While it is possible to conduct proofs using only those tactics, you can significantly increase your productivity by working with a set of more powerful tactics. This chapter describes a number of such useful tactics, which, for various reasons, are not yet available by default in Rocq. These tactics are defined in the *LibTactics.v* file.

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
```

```
From Stdlib Require Import Arith.
```

```
From PLF Require Maps.
```

```
From PLF Require Stlc.
```

```
From PLF Require Types.
```

```
From PLF Require Smallstep.
```

```
From PLF Require LibTactics.
```

```
From PLF Require Equiv.
```

```
From PLF Require References.
```

```
From PLF Require Hoare.
```

```
From PLF Require Sub.
```

```
Import LIBTACTICS.
```

Remark: `SSReflect` is another package providing powerful tactics. The library “LibTactics” differs from “SSReflect” in two respects:

- “SSReflect” was primarily developed for proving mathematical theorems, whereas “LibTactics” was primarily developed for proving theorems on programming languages. In particular, “LibTactics” provides a number of useful tactics that have no counterpart in the “SSReflect” package.

- “SSReflect” entirely rethinks the presentation of tactics, whereas “LibTactics” mostly stick to the traditional presentation of Rocq tactics, simply providing a number of additional tactics. For this reason, “LibTactics” is probably easier to get started with than “SSReflect”.

This chapter is a tutorial focusing on the most useful features from the “LibTactics” library. It does not aim at presenting all the features of “LibTactics”. The detailed specification of tactics can be found in the source file *LibTactics.v*. Further documentation as well as demos can be found at {<https://www.chargueraud.org/softs/tlc/>}.

In this tutorial, tactics are presented using examples taken from the core chapters of the “Software Foundations” course. To illustrate the various ways in which a given tactic can be used, we use a tactic that duplicates a given goal. More precisely, *dup* produces two copies of the current goal, and *dup n* produces *n* copies of it.

23.2 Tactics for Naming and Performing Inversion

This section presents the following tactics:

- *intros*, for naming hypotheses more efficiently,
- *inverts*, for improving the *inversion* tactic.

23.2.1 The Tactic *intros*

Module INTROEXAMPLES.

Import MAPS.

Import IMP.

Import EQUIV.

Import STLC.

The tactic *intros* allows to automatically introduce the variables of a theorem and explicitly name the hypotheses involved. In the example shown next, the variables *c*, *st*, *st1* and *st2* involved in the statement of determinism need not be named explicitly, because their names were already given in the statement of the lemma. On the contrary, it is useful to provide names for the two hypotheses, which we name *E1* and *E2*, respectively.

Theorem ceval_deterministic: $\forall c\ st\ st1\ st2,$

$st = [c] \Rightarrow st1 \rightarrow$

$st = [c] \Rightarrow st2 \rightarrow$

$st1 = st2.$

Proof.

intros E1 E2. Abort.

When there is no hypothesis to be named, one can call *intros* without any argument.

Theorem `dist_exists_or` : $\forall (X : \text{Type}) (P Q : X \rightarrow \text{Prop}),$
 $(\exists x, P x \vee Q x) \leftrightarrow (\exists x, P x) \vee (\exists x, Q x).$

Proof.

`intros. Abort.`

The tactic *intros* also applies to statements in which $\forall$ and $\rightarrow$ are interleaved.

Theorem `ceval_deterministic'` : $\forall c\ st\ st1,$
 $(st = [c] => st1) \rightarrow$
 $\forall st2,$
 $(st = [c] => st2) \rightarrow$
 $st1 = st2.$

Proof.

`intros E1 E2. Abort.`

Like the arguments of `intros`, the arguments of *intros* can be structured patterns.

Theorem `exists_impl` : $\forall X (P : X \rightarrow \text{Prop}) (Q : \text{Prop}) (R : \text{Prop}),$
 $(\forall x, P x \rightarrow Q) \rightarrow$
 $((\exists x, P x) \rightarrow Q).$

Proof.

`intros [x H2]. eauto.`

`Qed.`

Remark: the tactic *intros* works even when definitions need to be unfolded in order to reveal hypotheses.

End `INTROVEXAMPLES`.

23.2.2 The Tactic *inverts*

Module `INVERTSEXAMPLES`.

Import `MAPS`.

Import `IMP`.

Import `EQUIV`.

The `inversion` tactic of Rocq is not very satisfying for three reasons. First, it produces a bunch of equalities which one typically wants to substitute away, using `subst`. Second, it introduces meaningless names for hypotheses. Third, a call to `inversion H` does not remove *H* from the context, even though in most cases an hypothesis is no longer needed after being inverted. The tactic *inverts* address all of these three issues. It is intended to be used in place of the tactic `inversion`.

The following example illustrates how the tactic *inverts* *H* behaves mostly like `inversion H` except that it performs some substitutions in order to eliminate the trivial equalities that are being produced by `inversion`.

Theorem `skip_left` : $\forall c,$
 $\text{cequiv} \langle \{\text{skip}; c\} \rangle c.$

Proof.

```
  introv. split; intros H.
  dup.   - inversion H. subst. inversion H2. subst. assumption.
        - inverts H. inverts H2. assumption.
```

Abort.

A slightly more interesting example appears next.

Theorem `ceval_deterministic`: $\forall c\ st\ st1\ st2,$

```
  st = [ c ] => st1 →
  st = [ c ] => st2 →
  st1 = st2.
```

Proof.

```
  introv E1 E2. generalize dependent st2.
  induction E1; intros st2 E2.
  admit. admit.   dup.   - inversion E2. subst. admit.
                        - inverts E2. admit.
```

Abort.

The tactic *inverts* *H* *as.* is like *inverts* *H* except that the variables and hypotheses being produced are placed in the goal rather than in the context. This strategy allows naming those new variables and hypotheses explicitly, using either *intros* or *introv*.

Theorem `ceval_deterministic'`: $\forall c\ st\ st1\ st2,$

```
  st = [ c ] => st1 →
  st = [ c ] => st2 →
  st1 = st2.
```

Proof.

```
  introv E1 E2. generalize dependent st2.
  induction E1; intros st2 E2;
    inverts E2 as.
  - reflexivity.
  -
```

```
    subst n.
    reflexivity.
  -
```

```
    intros st3 Red1 Red2.
    assert (st' = st3) as EQ1.
    { apply IHE1_1; assumption. }
    subst st3.
    apply IHE1_2. assumption.
  -
```

```

    intros.
    apply IHE1. assumption.
-
    intros.
    rewrite H in H5. inversion H5.
Abort.

```

In the particular case where a call to `inversion` produces a single subgoal, one can use the syntax `inverts H as H1 H2 H3` for calling `inverts` and naming the new hypotheses *H1*, *H2* and *H3*. In other words, the tactic `inverts H as H1 H2 H3` is equivalent to `inverts H as; introv H1 H2 H3`. An example follows.

```

Theorem skip_left': ∀ c,
  cequiv <{ skip ; c }> c.
Proof.
  introv. split; intros H.
  inverts H as U V.   inverts U. assumption.
Abort.

```

End INVERTSEXAMPLES.

A more involved example appears next. In particular, this example shows that the name of the hypothesis being inverted can be reused.

```

Module INVERTSEXAMPLES1.
Import TYPES.
Import STLC.
Import STLC.
Import MAPS.

```

Example typing_nonexample_1 :

```

  ¬ ∃ T,
    <{ empty |-
      \x:Bool,
        \y:Bool,
          (x y) \in
            T }>.

```

```

Proof.
  dup 3.
  - intros Hc. destruct Hc as [T Hc].
    inversion Hc; subst; clear Hc.
    inversion H4; subst; clear H4.
    inversion H5; subst; clear H5 H4.
    inversion H2; subst; clear H2.
    discriminate H1.
  - intros C. destruct C.

```

```

inverts H as H1.
inverts H1 as H2.
inverts H2 as H3 H4.
inverts H3 as H5.
inverts H5.

- intros C. destruct C.
inverts H as H.
inverts H as H.
inverts H as H1 H2.
inverts H1 as H1.
inverts H1.
Qed.
End INVERTSEXAMPLES1.

```

Note: in the rare cases where one needs to perform an inversion on an hypothesis H without clearing H from the context, one can use the tactic *inverts keep H*, where the keyword *keep* indicates that the hypothesis should be kept in the context.

23.3 Tactics for N-ary Connectives

Because Rocq encodes conjunctions and disjunctions using binary constructors $\wedge$ and $\vee$, working with a conjunction or a disjunction of N facts can sometimes be quite cumbersome. For this reason, “LibTactics” provides tactics offering direct support for n-ary conjunctions and disjunctions. It also provides direct support for n-ary existentials.

This section presents the following tactics:

- *splits* for decomposing n-ary conjunctions,
- *branch* for decomposing n-ary disjunctions

```

Module NARYEXAMPLES.
  Import REFERENCES.
  Import SMALLSTEP.
  Import STLCREF.

```

23.3.1 The Tactic *splits*

The tactic *splits* applies to a goal made of a conjunction of n propositions and it produces n subgoals. For example, it decomposes the goal $G1 \wedge G2 \wedge G3$ into the three subgoals $G1$, $G2$ and $G3$.

```

Lemma demo_splits :  $\forall n m,$ 
   $n > 0 \wedge n < m \wedge m < n+10 \wedge m \neq 3.$ 

```


Proof.

```
intros. splits.
```

Abort.

Note that *splits* is less aggressive than *repeat split*, which might unfold predicates unexpectedly.

23.3.2 The Tactic *branch*

The tactic *branch* *k* can be used to prove a *n*-ary disjunction. For example, if the goal takes the form $G1 \vee G2 \vee G3$, the tactic *branch* 2 leaves only *G2* as subgoal. The following example illustrates the behavior of the *branch* tactic.

Lemma *demo_branch* : $\forall n\ m,$

$n < m \vee n = m \vee m < n.$

Proof.

```
intros.
```

```
destruct (lt_eq_lt_dec n m) as [ [H1|H2] |H3].
```

```
- branch 1. apply H1.
```

```
- branch 2. apply H2.
```

```
- branch 3. apply H3.
```

Qed.

End NARYEXAMPLES.

23.4 Tactics for Working with Equality

One of the major weaknesses of Rocq compared with other interactive proof assistants is its relatively poor support for reasoning with equalities. The tactics described next aims at simplifying pieces of proof scripts manipulating equalities.

This section presents the following tactics:

- *asserts_rewrite* for introducing an equality to rewrite with,
- *fequals* for improving the *f_equal* tactic,
- *applies_eq* for proving $P\ x\ y$ using an hypothesis $P\ x\ z$, automatically producing an equality $y = z$ as subgoal.

Module EQUALITYEXAMPLES.

23.4.1 The Tactic *asserts_rewrite*

The tactic *asserts_rewrite* is an enhanced version of *replace*. A call to *asserts_rewrite* ($E1 = E2$) first produces the goal $E1 = E2$, then replaces *E1* with *E2* in the goal.

Theorem `mult_0_plus` : $\forall n\ m : \mathbf{nat},$
 $(0 + n) \times m = n \times m.$

Proof.

```

dup.
intros n m.
assert (H: 0 + n = n). reflexivity. rewrite → H.
reflexivity.

intros n m.
asserts_rewrite (0 + n = n).
  reflexivity.    reflexivity. Qed.

```

Unlike `replace`, the tactic `asserts_rewrite` can be provided a lemma statement as argument. For example, one can write `asserts_rewrite` $(\forall a\ b, a*(S\ b) = a \times b + a)$. This formulation is useful when `a` and `b` are big terms, since there is no need to repeat their statements.

Theorem `mult_0_plus''` : $\forall u\ v\ w\ x\ y\ z : \mathbf{nat},$
 $(u + v) \times (S\ (w \times x + y)) = z.$

Proof.

```

intros. asserts_rewrite ( $\forall a\ b, a*(S\ b) = a \times b + a$ ).

```

Abort.

The syntax `asserts_rewrite` $(E1 = E2)$ `in` `H` allows rewriting in an hypothesis `H` rather than in the goal. The tactic `cuts_rewrite` is similar to `asserts_rewrite` except that it the two subgoals produced are swapped.

23.4.2 The Tactic *fequals*

The tactic *fequals* is similar to `f_equal` except that it directly discharges all the trivial subgoals produced. Moreover, the tactic *fequals* features an enhanced treatment of equalities between tuples.

Lemma `demo_fequals` : $\forall (a\ b\ c\ d\ e : \mathbf{nat})\ (f : \mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat}),$
 $a = 1 \rightarrow$
 $b = e \rightarrow$
 $e = 2 \rightarrow$
 $f\ a\ b\ c\ d = f\ 1\ 2\ c\ 4.$

Proof.

```

intros. fequals.

```

Abort.

23.4.3 The Tactic *applies_eq*

The tactic *applies_eq* is a variant of `eapply` that introduces equalities for subterms that do not unify. For example, assume the goal is the proposition $P\ x\ y$ and assume we have the

assumption H asserting that $P\ x\ z$ holds. We know that we can prove y to be equal to z . So, we could call the tactic `assert_rewrite` ($y = z$) and change the goal to $P\ x\ z$, but this would require copy-pasting the values of y and z . With the tactic `applies_eq`, we can call `applies_eq H`, which proves the goal and leaves only the subgoal $y = z$.

Axiom `big_expression_using` : $\text{nat} \rightarrow \text{nat}$.

Lemma `demo_applies_eq_1` : $\forall (P : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop})\ x\ y\ z,$
 $P\ x\ (\text{big\_expression\_using}\ z) \rightarrow$
 $P\ x\ (\text{big\_expression\_using}\ y).$

Proof.

introv H. dup.

assert (Eq: `big_expression_using` $y = \text{big\_expression\_using}\ z$).

admit. rewrite Eq. apply H.

`applies_eq` H.

admit. Abort.

When we have a mismatch on two arguments, the tactic `applies_eq` produces two equalities. Consider the following example.

Lemma `demo_applies_eq_2` : $\forall (P : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop})\ x1\ x2\ y1\ y2,$
 $P\ (\text{big\_expression\_using}\ x2)\ (\text{big\_expression\_using}\ y2) \rightarrow$
 $P\ (\text{big\_expression\_using}\ x1)\ (\text{big\_expression\_using}\ y1).$

Proof.

introv H. `applies_eq` H.

Abort.

End EQUALITYEXAMPLES.

23.5 Some Convenient Shorthands

This section of the tutorial introduces a few tactics that help make proof scripts shorter and more readable:

- `unfolds` (without argument) for unfolding the head definition,
- `false` for replacing the goal with `False`,
- `gen` as a shorthand for `generalize dependent`,
- `admits` for naming an admitted fact,
- `admit_rewrite` for rewriting using an admitted equality,
- `admit_goal` to set up a proof by induction by skipping the justification that some order decreases,
- `sort` for re-ordering the proof context by moving all propositions at the bottom.

23.5.1 The Tactic *unfolds*

Module UNFOLDSEXAMPLE.

Import HOARE.

The tactic *unfolds* (without any argument) unfolds the head constant of the goal. This tactic saves the need to name the constant explicitly.

Lemma bexp_eval_true : $\forall b\ st,$
beval *st* *b* = true $\rightarrow$
(bassertion *b*) *st*.

Proof.

intros b st Hbe. dup.

unfold bassertion. assumption.

unfolds. assumption.

Qed.

Remark: contrary to the tactic *hnf*, which may unfold several constants, *unfolds* performs only a single step of unfolding.

Remark: the tactic *unfolds* in *H* can be used to unfold the head definition of the hypothesis *H*.

End UNFOLDSEXAMPLE.

23.5.2 The Tactics *false* and *tryfalse*

The tactic *false* can be used to replace any goal with **False**. In short, it is a shorthand for *ex falso*. Moreover, *false* proves the goal if it contains an absurd assumption, such as **False** or $0 = S\ n$, or if it contains contradictory assumptions, such as $x = \text{true}$ and $x = \text{false}$.

Lemma demo_false : $\forall n,$
 $S\ n = 1 \rightarrow$
 $n = 0.$

Proof.

intros. destruct n. reflexivity. false.

Qed.

The tactic *false* can be given an argument: *false* *H* replace the goals with **False** and then applies *H*.

Lemma demo_false_arg :
($\forall n, n < 0 \rightarrow \text{False}$) $\rightarrow$
 $3 < 0 \rightarrow$
 $4 < 0.$

Proof.

intros H L. false H. apply L.

Qed.

The tactic *tryfalse* is a shorthand for `try solve [false]`: it tries to find a contradiction in the goal. The tactic *tryfalse* is generally called after a case analysis.

```
Lemma demo_tryfalse :  $\forall n,$ 
  S  $n = 1 \rightarrow$ 
   $n = 0$ .
Proof.
  intros. destruct n; tryfalse. reflexivity.
Qed.
```

23.5.3 The Tactic *gen*

The tactic *gen* is a shorthand for `generalize dependent` that accepts several arguments at once. An invocation of this tactic takes the form *gen* $x\ y\ z$.

```
Module GENEXAMPLE.
  Import STLC.
  Import STLC.
  Import MAPS.
Lemma substitution_preserves_typing :  $\forall \text{Gamma } x\ U\ t\ v\ T,$ 
   $\langle \{ x \mapsto U ; \text{Gamma} \mid - t \in T \} \rangle \rightarrow$ 
   $\langle \{ \text{empty} \mid - v \in U \} \rangle \rightarrow$ 
   $\langle \{ \text{Gamma} \mid - [x:=v]t \in T \} \rangle$ .
Proof.
  dup.
  - intros Gamma x U t v T Ht Hv.
    generalize dependent Gamma. generalize dependent T.
    induction t; intros T Gamma H;
      inversion H; clear H; subst; simpl; eauto.
    admit. admit.
  - introv Ht Hv. gen Gamma T.
    induction t; intros S Gamma H;
      inversion H; clear H; subst; simpl; eauto.
    admit. admit.
Abort.
End GENEXAMPLE.
```

23.5.4 The Tactics *admits*, *admit_rewrite* and *admit_goal*

Temporarily admitting a given subgoal is very useful when constructing proofs. Several tactics are provided as useful wrappers around the builtin *admit* tactic.

```
Module SKIPEXAMPLE.
```

The tactic *admits* $H : P$ adds the hypothesis $H : P$ to the context, without checking whether the proposition P is true. It is useful for exploiting a fact and postponing its proof. Note: *admits* $H : P$ is simply a shorthand for *assert* $(H : P)$. *admit*.

Theorem *demo_admits* : **True**.

Proof.

```
admits H: (∀ n m : nat, (0 + n) × m = n × m).
```

Abort.

The tactic *admit_rewrite* $(E1 = E2)$ replaces $E1$ with $E2$ in the goal, without checking that $E1$ is actually equal to $E2$.

Theorem *mult_plus_0* : $\forall n m : \mathbf{nat}$,

```
(n + 0) × m = n × m.
```

Proof.

```
dup 3.
```

```
intros n m.
```

```
assert (H: n + 0 = n). admit. rewrite → H. clear H.
```

```
reflexivity.
```

```
intros n m.
```

```
admit_rewrite (n + 0 = n).
```

```
reflexivity.
```

```
intros n m.
```

```
admit_rewrite (∀ a, a + 0 = a).
```

```
reflexivity.
```

Admitted.

The tactic *admit_goal* adds the current goal as hypothesis. This cheat is useful to set up the structure of a proof by induction without having to worry about the induction hypothesis being applied only to smaller arguments. Using *skip_goal*, one can construct a proof in two steps: first, check that the main arguments go through without waisting time on fixing the details of the induction hypotheses; then, focus on fixing the invocations of the induction hypothesis.

Import **IMP**.

Theorem *ceval_deterministic*: $\forall c \ st \ st1 \ st2$,

```
st = [ c ] => st1 →
```

```
st = [ c ] => st2 →
```

```
st1 = st2.
```

Proof.

```
admit_goal.
```

```
intros E1 E2. gen st2.
```

```
induction E1; introv E2; inverts E2 as.
```

```
- reflexivity.
```

```
-
```

```

    subst n.
    reflexivity.
  -
    intros st3 Red1 Red2.
    assert (st' = st3) as EQ1.
    {

        eapply IH. eapply E1_1. eapply Red1. }
    subst st3.
    eapply IH. eapply E1_2. eapply Red2.
  Abort.
End SKIPEXAMPLE.

```

23.5.5 The Tactic *sort*

Module SORTEXAMPLES.

Import IMP.

The tactic *sort* reorganizes the proof context by placing all the variables at the top and all the hypotheses at the bottom, thereby making the proof context more readable.

Theorem ceval_deterministic: $\forall c\ st\ st1\ st2,$

$st = [c] \Rightarrow st1 \rightarrow$

$st = [c] \Rightarrow st2 \rightarrow$

$st1 = st2.$

Proof.

intros c st st1 st2 E1 E2.

generalize dependent st2.

induction E1; intros st2 E2; inverts E2.

admit. admit. sort. Abort.

End SORTEXAMPLES.

23.6 Tactics for Advanced Lemma Instantiation

This last section describes a mechanism for instantiating a lemma by providing some of its arguments and leaving others implicit. Variables whose instantiation is not provided are turned into existential variables, and facts whose instantiation is not provided are turned into subgoals.

Remark: this instantiation mechanism goes far beyond the abilities of the “Implicit Arguments” mechanism. The point of the instantiation mechanism described in this section is that you will no longer need to spend time figuring out how many underscore symbols you need to write.

In this section, we'll use a useful feature of Rocq for decomposing conjunctions and existentials. In short, a tactic like `intros` or `destruct` can be provided with a pattern $(H1 \ \& \ H2 \ \& \ H3 \ \& \ H4 \ \& \ H5)$, which is a shorthand for $[H1 \ [H2 \ [H3 \ [H4 \ H5]]]]$. For example, `destruct (H _ _ _ Htypt) as [T [Hctx Hsub]]`. can be rewritten in the form `destruct (H _ _ _ Htypt) as (T & Hctx & Hsub)`.

23.6.1 Working of *lets*

When we have a lemma (or an assumption) that we want to exploit, we often need to explicitly provide arguments to this lemma, writing something like: `destruct (typing_inversion_var _ _ _ Htypt) as (T & Hctx & Hsub)`. The need to write several times the “underscore” symbol is tedious. Not only we need to figure out how many of them to write down, but it also makes the proof scripts look pretty ugly. With the tactic *lets*, one can simply write: `lets (T & Hctx & Hsub): typing_inversion_var Htypt`.

In short, this tactic *lets* allows to specialize a lemma on a bunch of variables and hypotheses. The syntax is `lets !: E0 E1 .. EN`, for building an hypothesis named `!` by applying the fact `E0` to the arguments `E1` to `EN`. Not all the arguments need to be provided, however the arguments that are provided need to be provided in the correct order. The tactic relies on a first-match algorithm based on types in order to figure out how the to instantiate the lemma with the arguments provided.

Module EXAMPLESLETS.

Import MAPS.

Import SUB.STLCSUB.

Import STRING.

Axiom `typing_inversion_var` : $\forall \text{Gamma} \ (x:\text{string}) \ T,$
 $\langle \{ \text{Gamma} \mid - \ x \ \backslash \text{in} \ T \} \rangle \rightarrow$
 $\exists S,$
 $\text{Gamma} \ x = \text{Some } S \wedge S <: T.$

First, assume we have an assumption `H` with the type of the form `has_type G (var x) T`. We can obtain the conclusion of the lemma `typing_inversion_var` by invoking the tactics `lets K: typing_inversion_var H`, as shown next.

Lemma `demo_lets_1` : $\forall (G:\text{context}) \ (x:\text{string}) \ (T:\text{ty}),$
 $\langle \{ G \mid - \ x \ \backslash \text{in} \ T \} \rangle \rightarrow$
`True.`

Proof.

`intros G x T H. dup.`

`lets K: typing_inversion_var H.`

`destruct K as (S & Eq & Sub).`

`admit.`

`lets (S & Eq & Sub): typing_inversion_var H.`

admit.

Abort.

Assume now that we know the values of G , x and T and we want to obtain S , and have **has_type** G (*var* x) T be produced as a subgoal. To indicate that we want all the remaining arguments of *typing_inversion_var* to be produced as subgoals, we use a triple-underscore symbol *---*. (We'll later introduce a shorthand tactic called *forwards* to avoid writing triple underscores.)

Lemma demo_lets_2 : $\forall (G:\text{context}) (x:\text{string}) (T:\text{ty}), \text{True}$.

Proof.

intros G x T .

lets (S & Eq & Sub): *typing_inversion_var* G x T *---*.

Abort.

Usually, there is only one context G and one type T that are going to be suitable for proving **has_type** G (*tm_var* x) T , so we don't really need to bother giving G and T explicitly. It suffices to call *lets* (S & Eq & Sub): *typing_inversion_var* x . The variables G and T are then instantiated using existential variables.

Lemma demo_lets_3 : $\forall (x:\text{string}), \text{True}$.

Proof.

intros x .

lets (S & Eq & Sub): *typing_inversion_var* x *---*.

Abort.

We may go even further by not giving any argument to instantiate *typing_inversion_var*. In this case, three unification variables are introduced.

Lemma demo_lets_4 : **True**.

Proof.

lets (S & Eq & Sub): *typing_inversion_var* *---*.

Abort.

Note: if we provide *lets* with only the name of the lemma as argument, it simply adds this lemma in the proof context, without trying to instantiate any of its arguments.

Lemma demo_lets_5 : **True**.

Proof.

lets H : *typing_inversion_var*.

Abort.

A last useful feature of *lets* is the double-underscore symbol, which allows skipping an argument when several arguments have the same type. In the following example, our assumption quantifies over two variables n and m , both of type **nat**. We would like m to be instantiated as the value 3, but without specifying a value for n . This can be achieved by writing *lets* K : H *--* 3.

Lemma demo_lets_underscore :

$(\forall n\ m, n \leq m \rightarrow n < m+1) \rightarrow$

True.

Proof.

`intros H.`

`lets K: H 3.      clear K.`

`lets K: H -- 3.      clear K.`

Abort.

Note: one can write `lets: E0 E1 E2` in place of `lets H: E0 E1 E2`. In this case, the name *H* is chosen arbitrarily.

Note: the tactics *lets* accepts up to five arguments. Another syntax is available for providing more than five arguments. It consists in using a list introduced with the special symbol `»`, for example `lets H: (» E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 10)`.

End EXAMPLESLETS.

23.6.2 Working of *applys*, *forwards* and *specializes*

The tactics *applys*, *forwards* and *specializes* are shorthand that may be used in place of *lets* to perform specific tasks.

- *forwards* is a shorthand for instantiating all the arguments

of a lemma. More precisely, `forwards H: E0 E1 E2 E3` is the same as `lets H: E0 E1 E2 E3 ---`, where the triple-underscore has the same meaning as explained earlier on.

- *applys* allows building a lemma using the advanced instantiation

mode of *lets*, and then apply that lemma right away. So, `applys E0 E1 E2 E3` is the same as `lets H: E0 E1 E2 E3` followed with `eapply H` and then `clear H`.

- *specializes* is a shorthand for instantiating in-place

an assumption from the context with particular arguments. More precisely, `specializes H E0 E1` is the same as `lets H': H E0 E1` followed with `clear H` and `rename H' into H`.

Examples of use of *applys* appear further on. Several examples of use of *forwards* can be found in the tutorial chapter [UseAuto](#).

23.7 Summary

In this chapter we have presented a number of tactics that help make proof script more concise and more robust on change.

- *introv* and *inverts* improve naming and inversions.

- `false` and `tryfalse` help discarding absurd goals.
- `unfolds` automatically calls `unfold` on the head definition.
- `gen` helps setting up goals for induction.
- `splits` and `branch`, to deal with n-ary constructs.
- `asserts_rewrite`, `cuts_rewrite`, and `fequals` help working with equalities.
- `lets`, `forwards`, `specializes` and `applies` provide means of very conveniently instantiating lemmas.
- `applies_eq` can save the need to perform manual rewriting steps before being able to apply a lemma.
- `admits`, `admit_rewrite` and `admit_goal` give the flexibility to choose which subgoals to try and discharge first.

Making use of these tactics can boost one's productivity in Rocq proofs.

If you are interested in using `LibTactics.v` in your own developments, make sure you get the latest version from: `{https://www.chargueraud.org/softs/tlc/}`.

Chapter 24

Library `PLF.UseAuto`

24.1 UseAuto: Theory and Practice of Automation in Rocq Proofs

In a machine-checked proof, every single detail has to be justified. This can result in huge proof scripts. Fortunately, Rocq comes with a proof-search mechanism and with several decision procedures that enable the system to automatically synthesize simple pieces of proof. Automation is very powerful when set up appropriately. The purpose of this chapter is to explain the basics of how automation works in Rocq.

The chapter is organized in two parts. The first part focuses on a general mechanism called “proof search.” In short, proof search consists in naively trying to apply lemmas and assumptions in all possible ways. The second part describes “decision procedures”, which are tactics that are very good at solving proof obligations that fall in some particular fragments of the logic of Rocq.

Many of the examples used in this chapter consist of small lemmas that have been made up to illustrate particular aspects of automation. These examples are completely independent from the rest of the Software Foundations course. This chapter also contains some bigger examples which are used to explain how to use automation in realistic proofs. These examples are taken from other chapters of the course (mostly from STLC), and the proofs that we present make use of the tactics from the library `LibTactics.v`, which is presented in the chapter `UseTactics`.

```
From Stdlib Require Import Arith.
Set Warnings "-parsing".
From PLF Require Import Maps.
From PLF Require Import Smallstep.
From PLF Require Import LibTactics.
From PLF Require Stlc.
From PLF Require Imp.
From Stdlib Require Import List.
```

24.2 Basic Features of Proof Search

The idea of proof search is to replace a sequence of tactics applying lemmas and assumptions with a call to a single tactic, for example `auto`. This form of proof automation saves a lot of effort. It typically leads to much shorter proof scripts, and to scripts that are typically more robust to change. If one makes a little change to a definition, a proof that exploits automation probably won't need to be modified at all. Of course, using too much automation is a bad idea. When a proof script no longer records the main arguments of a proof, it becomes difficult to fix it when it gets broken after a change in a definition. Overall, a reasonable use of automation is generally a big win, as it saves a lot of time both in building proof scripts and in subsequently maintaining those proof scripts.

24.2.1 Strength of Proof Search

We are going to study four proof-search tactics: `auto`, `eauto`, `iauto` and `jauto`. The tactics `auto` and `eauto` are builtin in Rocq. The tactic `iauto` is a shorthand for the builtin tactic `try solve [intuition eauto]`. The tactic `jauto` is defined in the library `LibTactics`, and simply performs some preprocessing of the goal before calling `eauto`. The goal of this chapter is to explain the general principles of proof search and to give rule of thumbs for guessing which of the four tactics mentioned above is best suited for solving a given goal.

Proof search is a compromise between efficiency and expressiveness, that is, a tradeoff between how complex goals the tactic can solve and how much time the tactic requires for terminating. The tactic `auto` builds proofs only by using the basic tactics `reflexivity`, `assumption`, and `apply`. The tactic `eauto` can also exploit `eapply`. The tactic `jauto` extends `eauto` by being able to open conjunctions and existentials that occur in the context. The tactic `iauto` is able to deal with conjunctions, disjunctions, and negation in a quite clever way; however it is not able to open existentials from the context. Also, `iauto` usually becomes very slow when the goal involves several disjunctions.

Note that proof search tactics never perform any rewriting step (tactics `rewrite`, `subst`), nor any case analysis on an arbitrary data structure or property (tactics `destruct` and `inversion`), nor any proof by induction (tactic `induction`). So, proof search is really intended to automate the final steps from the various branches of a proof. It is not able to discover the overall structure of a proof.

24.2.2 Basics

The tactic `auto` is able to solve a goal that can be proved using a sequence of `intros`, `apply`, `assumption`, and `reflexivity`. Two examples follow. The first one shows the ability for `auto` to call `reflexivity` at any time. In fact, calling `reflexivity` is always the first thing that `auto` tries to do.

Lemma `solving_by_reflexivity` :

`2 + 3 = 5.`

Proof. `auto. Qed.`

The second example illustrates a proof where a sequence of two calls to `apply` are needed. The goal is to prove that if $Q\ n$ implies $P\ n$ for any n and if $Q\ n$ holds for any n , then $P\ 2$ holds.

```
Lemma solving_by_apply : ∀ (P Q : nat→Prop),
  (∀ n, Q n → P n) →
  (∀ n, Q n) →
  P 2.
```

Proof. auto. Qed.

If we are interested to see which proof `auto` came up with, one possibility is to look at the generated proof-term, using the command:

```
Print solving_by_apply.
```

The proof term is:

```
fun (P Q : nat → Prop) (H : ∀ n : nat, Q n → P n) (H0 : ∀ n : nat, Q n) ⇒ H 2 (H0
2)
```

This essentially means that `auto` applied the hypothesis H (the first one), and then applied the hypothesis $H0$ (the second one).

A more practical approach is to invoke the tactic `info_auto`, which reveals the proof script that `auto` used to complete the proof.

The tactic `auto` can invoke `apply` but not `eapply`. So, `auto` cannot exploit lemmas whose instantiation cannot be directly deduced from the proof goal. To exploit such lemmas, one needs to invoke the tactic `eauto`, which is able to call `eapply`.

In the following example, the first hypothesis asserts that $P\ n$ is true when $Q\ m$ is true for some m , and the goal is to prove that $Q\ 1$ implies $P\ 2$. This implication follows directly from the hypothesis by instantiating m as the value 1. The following proof script shows that `eauto` successfully solves the goal, whereas `auto` is not able to do so.

```
Lemma solving_by_eapply : ∀ (P Q : nat→Prop),
  (∀ n m, Q m → P n) →
  Q 1 →
  P 2.
```

Proof. auto. eauto. Qed.

24.2.3 Conjunctions

So far, we've seen that `eauto` is stronger than `auto` in the sense that it can deal with `eapply`. In the same way, we are going to see how `jauto` and `iauto` are stronger than `auto` and `eauto` in the sense that they provide better support for conjunctions.

The tactics `auto` and `eauto` can prove a goal of the form $F \wedge F'$, where F and F' are two propositions, as soon as both F and F' can be proved in the current context. An example follows.

```
Lemma solving_conj_goal : ∀ (P : nat→Prop) (F : Prop),
  (∀ n, P n) →
```

$F \rightarrow$
 $F \wedge P$ 2.

Proof. auto. Qed.

However, when an assumption is a conjunction, `auto` and `eauto` are not able to exploit this conjunction. It can be quite surprising at first that `eauto` can prove very complex goals but that it fails to prove that $F \wedge F'$ implies F . The tactics `iauto` and `jauto` are able to decompose conjunctions from the context. Here is an example.

Lemma solving_conj_hyp : $\forall (F F' : \text{Prop}),$
 $F \wedge F' \rightarrow$
 F .

Proof. auto. eauto. jauto. Qed.

The tactic `jauto` is implemented by first calling a pre-processing tactic called `jauto_set`, and then calling `eauto`. So, to understand how `jauto` works, one can directly call the tactic `jauto_set`.

Lemma solving_conj_hyp' : $\forall (F F' : \text{Prop}),$
 $F \wedge F' \rightarrow$
 F .

Proof. intros. jauto_set. eauto. Qed.

Next is a more involved goal that can be solved by `iauto` and `jauto`.

Lemma solving_conj_more : $\forall (P Q R : \text{nat} \rightarrow \text{Prop}) (F : \text{Prop}),$
 $(F \wedge (\forall n m, (Q m \wedge R n) \rightarrow P n)) \rightarrow$
 $(F \rightarrow R)$ 2) $\rightarrow$
 Q 1 $\rightarrow$
 P 2 $\wedge F$.

Proof. jauto. Qed.

The strategy of `iauto` and `jauto` is to run a global analysis of the top-level conjunctions, and then call `eauto`. For this reason, those tactics are not good at dealing with conjunctions that occur as the conclusion of some universally quantified hypothesis. The following example illustrates a general weakness of Rocq proof search mechanisms.

Lemma solving_conj_hyp_forall : $\forall (P Q : \text{nat} \rightarrow \text{Prop}),$
 $(\forall n, P n \wedge Q n) \rightarrow$
 P 2.

Proof.

auto. eauto. iauto. jauto.
intros. destruct (H 2). auto.

Qed.

This situation is slightly disappointing, since automation is able to prove the following goal, which is very similar. The only difference is that the universal quantification has been distributed over the conjunction.

Lemma solved_by_jauto : $\forall (P Q : \text{nat} \rightarrow \text{Prop}) (F : \text{Prop}),$

```

    (∀ n, P n) ∧ (∀ n, Q n) →
    P 2.
Proof. jauto. Qed.

```

24.2.4 Disjunctions

The tactics `auto` and `eauto` can handle disjunctions that occur in the goal.

```

Lemma solving_disj_goal : ∀ (F F' : Prop),
  F →
  F ∨ F'.
Proof. auto. Qed.

```

However, only `iauto` is able to automate reasoning on the disjunctions that appear in the context. For example, `iauto` can prove that $F \vee F'$ entails $F' \vee F$.

```

Lemma solving_disj_hyp : ∀ (F F' : Prop),
  F ∨ F' →
  F' ∨ F.
Proof. auto. eauto. jauto. iauto. Qed.

```

More generally, `iauto` can deal with complex combinations of conjunctions, disjunctions, and negations. Here is an example.

```

Lemma solving_tauto : ∀ (F1 F2 F3 : Prop),
  ((¬F1 ∧ F3) ∨ (F2 ∧ ¬F3)) →
  (F2 → F1) →
  (F2 → F3) →
  ¬F2.
Proof. iauto. Qed.

```

However, the ability of `iauto` to automatically perform a case analysis on disjunctions comes with a downside: `iauto` may be very slow. If the context involves several hypotheses with disjunctions, `iauto` typically generates an exponential number of subgoals on which `eauto` is called. One major advantage of `jauto` compared with `iauto` is that it never spends time performing this kind of case analyses.

24.2.5 Existentials

The tactics `eauto`, `iauto`, and `jauto` can prove goals whose conclusion is an existential. For example, if the goal is $\exists x, f x$, the tactic `eauto` introduces an existential variable, say `?25`, in place of `x`. The remaining goal is `f ?25`, and `eauto` tries to solve this goal, allowing itself to instantiate `?25` with any appropriate value. For example, if an assumption `f 2` is available, then the variable `?25` gets instantiated with `2` and the goal is solved, as shown below.

```

Lemma solving_exists_goal : ∀ (f : nat → Prop),
  f 2 →
  ∃ x, f x.

```


Proof.

```
auto. eauto. Qed.
```

A major strength of *jauto* over the other proof search tactics is that it is able to exploit the existentially-quantified hypotheses, i.e., those of the form $\exists x, P$.

```
Lemma solving_exists_hyp :  $\forall (f\ g : \text{nat} \rightarrow \text{Prop}),$   
  ( $\forall x, f\ x \rightarrow g\ x$ )  $\rightarrow$   
  ( $\exists a, f\ a$ )  $\rightarrow$   
  ( $\exists a, g\ a$ ).
```

Proof.

```
auto. eauto. iauto. jauto. Qed.
```

24.2.6 Negation

The tactics *auto* and *eauto* suffer from some limitations with respect to the manipulation of negations, mostly related to the fact that negation, written $\neg P$, is defined as $P \rightarrow \text{False}$ but that the unfolding of this definition is not performed automatically. Consider the following example.

```
Lemma negation_study_1 :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$   
   $P\ 0 \rightarrow$   
  ( $\forall x, \neg P\ x$ )  $\rightarrow$   
  False.
```

Proof.

```
intros P H0 HX.  
eauto. unfold not in *. eauto.
```

Qed.

For this reason, the tactics *iauto* and *jauto* systematically invoke `unfold not in *` as part of their pre-processing. So, they are able to solve the previous goal right away.

```
Lemma negation_study_2 :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$   
   $P\ 0 \rightarrow$   
  ( $\forall x, \neg P\ x$ )  $\rightarrow$   
  False.
```

Proof. *jauto*. Qed.

We will come back later on to the behavior of proof search with respect to the unfolding of definitions.

24.2.7 Equalities

Rocq's proof-search feature is not good at exploiting equalities. It can do very basic operations, like exploiting reflexivity and symmetry, but that's about it. Here is a simple example that *auto* can solve, by first calling *symmetry* and then applying the hypothesis.

```
Lemma equality_by_auto :  $\forall (f\ g : \text{nat} \rightarrow \text{Prop}),$ 
```

```
( $\forall x, f\ x = g\ x$ )  $\rightarrow$ 
   $g\ 2 = f\ 2$ .
```

Proof. auto. Qed.

To automate more advanced reasoning on equalities, one should rather try to use the tactic `congruence`, which is presented at the end of this chapter in the “Decision Procedures” section.

24.3 How Proof Search Works

24.3.1 Search Depth

The tactic `auto` works as follows. It first tries to call `reflexivity` and `assumption`. If one of these calls solves the goal, the job is done. Otherwise `auto` tries to apply the most recently introduced assumption that can be applied to the goal without producing an error. This application produces subgoals. There are two possible cases. If the subgoals produced can be solved by a recursive call to `auto`, then the job is done. Otherwise, if this application produces at least one subgoal that `auto` cannot solve, then `auto` starts over by trying to apply the second most recently introduced assumption. It continues in a similar fashion until it finds a proof or until no assumption remains to be tried.

It is very important to have a clear idea of the backtracking process involved in the execution of the `auto` tactic; otherwise its behavior can be quite puzzling. For example, `auto` is not able to solve the following triviality.

Lemma `search_depth_0` :

```
True  $\wedge$  True  $\wedge$  True  $\wedge$  True  $\wedge$  True  $\wedge$  True.
```

Proof.

```
auto.
```

Abort.

The reason `auto` fails to solve the goal is because there are too many conjunctions. If there had been only five of them, `auto` would have successfully solved the proof, but six is too many. The tactic `auto` limits the number of lemmas and hypotheses that can be applied in a proof, so as to ensure that the proof search eventually terminates. By default, the maximal number of steps is five. One can specify a different bound, writing for example `auto 6` to search for a proof involving at most six steps. For example, `auto 6` would solve the previous lemma. (Similarly, one can invoke `eauto 6` or `intuition eauto 6`.) The argument `n` of `auto n` is called the “search depth.” The tactic `auto` is simply defined as a shorthand for `auto 5`.

The behavior of `auto n` can be summarized as follows. It first tries to solve the goal using `reflexivity` and `assumption`. If this fails, it tries to apply a hypothesis (or a lemma that has been registered in the hint database), and this application produces a number of subgoals. The tactic `auto (n-1)` is then called on each of those subgoals. If all the subgoals are solved, the job is completed, otherwise `auto n` tries to apply a different hypothesis.

During the process, `auto n` calls `auto (n-1)`, which in turn might call `auto (n-2)`, and so on. The tactic `auto 0` only tries `reflexivity` and `assumption`, and does not try to apply any lemma. Overall, this means that when the maximal number of steps allowed has been exceeded, the `auto` tactic stops searching and backtracks to try and investigate other paths.

The following lemma admits a unique proof that involves exactly three steps. So, `auto n` proves this goal iff `n` is greater than three.

```
Lemma search_depth_1 : ∀ (P : nat→Prop),
  P 0 →
  (P 0 → P 1) →
  (P 1 → P 2) →
  (P 2).
```

Proof.

```
auto 0. auto 1. auto 2. auto 3. Qed.
```

We can generalize the example by introducing an assumption asserting that `P k` is derivable from `P (k-1)` for all `k`, and keep the assumption `P 0`. The tactic `auto`, which is the same as `auto 5`, is able to derive `P k` for all values of `k` less than 5. For example, it can prove `P 4`.

```
Lemma search_depth_3 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k - 1) → P k) →
  (P 4).
```

Proof. `auto`. Qed.

However, to prove `P 5`, one needs to call at least `auto 6`.

```
Lemma search_depth_4 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k - 1) → P k) →
  (P 5).
```

Proof. `auto`. `auto 6`. Qed.

Because `auto` looks for proofs at a limited depth, there are cases where `auto` can prove a goal `F` and can prove a goal `F'` but cannot prove `F ∧ F'`. In the following example, `auto` can prove `P 4` but it is not able to prove `P 4 ∧ P 4`, because the splitting of the conjunction consumes one proof step. To prove the conjunction, one needs to increase the search depth, using at least `auto 6`.

```
Lemma search_depth_5 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k - 1) → P k) →
  (P 4 ∧ P 4).
```

Proof. `auto`. `auto 6`. Qed.

24.3.2 Backtracking

In the previous section, we have considered proofs where at each step there was a unique assumption that `auto` could apply. In general, `auto` can have several choices at every step. The strategy of `auto` consists of trying all of the possibilities (using a depth-first search exploration).

To illustrate how automation works, we are going to extend the previous example with an additional assumption asserting that $P\ k$ is also derivable from $P\ (k + 1)$. Adding this hypothesis offers a new possibility that `auto` could consider at every step.

There exists a special commands, named `info_auto` and `info_eauto`, that one can use to see what tactics `auto` uses to solve a goal. There also exists more verbose commands, named `debug auto` and `debug eauto`, that one can use for tracing all the steps that considered during a proof-search.

```
Lemma working_of_auto_1 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k - 1) → P k) →
  (∀ k, P (k + 1) → P k) →
  (P 2).
```

```
Proof. intros P H1 H2 H3. auto. Qed.
```

The output message produced by `debug eauto` is as follows.

```
1 depth=5 1.1 depth=4 simple apply H2 1.1.1 depth=3 simple apply H2 1.1.1.1 depth=3
exact H1
```

The depth indicates the value of `n` with which `eauto n` is called. The tactics shown in the message indicate that the first thing that `eauto` has tried to do is to apply `H2`. The effect of applying `H2` is to replace the goal $P\ 2$ with the goal $P\ 1$. Then, again, `H2` has been applied, changing the goal $P\ 1$ into $P\ 0$. At that point, the goal was exactly the hypothesis `H1`.

It seems that `eauto` was quite lucky there, as it never even tried to use the hypothesis `H3` at any time. The reason is that `auto` always tried to use the `H2` first. So, let's permute the hypotheses `H2` and `H3` and see what happens.

```
Lemma working_of_auto_2 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k + 1) → P k) →
  (∀ k, P (k - 1) → P k) →
  (P 2).
```

```
Proof. intros P H1 H3 H2. auto. Qed.
```

This time, the output message suggests that the proof search investigates many possibilities. If we print the proof term:

```
Print working_of_auto_2.
```

we observe that the proof term refers to `H3`. Thus the proof is not the simplest one, since only `H2` and `H1` are needed.

It turns out that the proof goes through the proof obligation $P\ 3$, even though it is not required to do so. The following tree drawing describes all the goals that `eauto` has been going through.

|5||4||3||2||1||0| – below, tabulation indicates the depth

$P\ 2$

• $> P\ 3$

– $> P\ 4$

* $> P\ 5$

· $> P\ 6$

· $> P\ 7$

· $> P\ 5$

· $> P\ 4$

· $> P\ 5$

· $> P\ 3$

* $\rightarrow P\ 3$

· $> P\ 4$

· $> P\ 5$

· $> P\ 3$

· $> P\ 2$

· $> P\ 3$

· $> P\ 1$

– $> P\ 2$

* $> P\ 3$

· $> P\ 4$

· $> P\ 5$

· $> P\ 3$

· $> P\ 2$

· $> P\ 3$

· $> P\ 1$

* $> P\ 1$

· $> P\ 2$

```

· > P 3
· > P 1
· > P 0
· > !! Done !!

```

The first few lines read as follows. To prove $P\ 2$, `eauto` 5 has first tried to apply $H3$, producing the subgoal $P\ 3$. To solve it, `eauto` 4 has tried again to apply $H3$, producing the goal $P\ 4$. Similarly, the search goes through $P\ 5$, $P\ 6$ and $P\ 7$. When reaching $P\ 7$, the tactic `eauto` 0 is called but as it is not allowed to try and apply any lemma, it fails. So, we come back to the goal $P\ 6$, and try this time to apply hypothesis $H2$, producing the subgoal $P\ 5$. Here again, `eauto` 0 fails to solve this goal.

The process goes on and on, until backtracking to $P\ 3$ and trying to apply $H3$ three times in a row, going through $P\ 2$ and $P\ 1$ and $P\ 0$. This search tree explains why `eauto` came up with a proof term starting with an application of $H3$.

24.3.3 Adding Hints

By default, `auto` (and `eauto`) only tries to apply the hypotheses that appear in the proof context. There are three possibilities for telling `auto` to exploit a lemma that have been proved previously: either add the lemma as a on-the-fly hint with `auto using lemma`; or add the lemma as an assumption just before calling `auto`; or add the lemma as a hint, so that it can be used by every calls to `auto`.

The first and second possibilities are useful to have `auto` exploit a lemma that only serves at this particular point. Note: to add a lemma as an hypothesis, one can type `generalize mylemma`; `intro`, or simply `lets: mylemma`, if using *LibTactics.v*.

The third possibility is useful for lemmas that need to be exploited accross several lemmas. The syntax for adding a lemma as a hint is `Hint Resolve mylemma`. For example:

```
Lemma nat_le_refl : ∀ (x:nat), x ≤ x.
```

```
Proof. apply le_n. Qed.
```

```
Local Hint Resolve nat_le_refl : core.
```

The standard library adds a few lemmas in the hint database. For example, the `eq_sym` lemma, which characterizes symmetry of equality, is registered in module *Rocq.Init.Logic* via the line `Hint Immediate eq_sym not_eq_sym: core..` There `Hint Immediate` is used instead of `Hint Resolve` to limit the search depth after symmetry is exploited. In practice, `auto` is able to prove a result such as the following one.

```
Lemma solving_by_symmetry : ∀ n,
```

```
  n = 5 →
```

```
  5 = n.
```

```
Proof. info_auto. Qed.
```

A convenient shorthand for adding all the constructors of an inductive datatype as hints is the command `Hint Constructors mydatatype`.

Warning: some lemmas, such as transitivity results, should not be added as hints as they would very badly affect the performance of proof search. The description of this problem and the presentation of a general work-around for transitivity lemmas appear further on.

24.3.4 Integration of Automation in Tactics

The library “LibTactics” introduces a convenient feature for invoking automation after calling a tactic. In short, it suffices to add the symbol star ($\times$) to the name of a tactic. For example, `apply $\times$  H` is equivalent to `apply H; auto_star`, where `auto_star` is a tactic that can be defined as needed.

The definition of `auto_star`, which determines the meaning of the star symbol, can be modified whenever needed. Simply write:

```
Ltac auto_star ::= a_new_definition.
```

Observe the use of `::=` instead of `:=`, which indicates that the tactic is being rebound to a new definition. So, the default definition is as follows.

```
Ltac auto_star ::= try solve [ jauto ].
```

Nearly all standard Rocq tactics and all the tactics from “LibTactics” can be called with a star symbol. For example, one can invoke `subst $\times$` , `destruct $\times$  H`, `inverts $\times$  H`, `lets $\times$  l: H x`, `specializes $\times$  H x`, and so on... There are two notable exceptions. The tactic `auto $\times$` is just another name for the tactic `auto_star`. And the tactic `apply $\times$  H` calls `eapply H` (or the more powerful `applies H` if needed), and then calls `auto_star`. Note that there is no `eapply $\times$  H` tactic, use `apply $\times$  H` instead.

In large developments, it can be convenient to use two degrees of automation. Typically, one would use a fast tactic, like `auto`, and a slower but more powerful tactic, like `jauto`. To allow for a smooth coexistence of the two form of automation, `LibTactics.v` also defines a “tilde” version of tactics, like `apply $\neg$  H`, `destruct $\neg$  H`, `subst $\neg$` , `auto $\neg$` and so on. The meaning of the tilde symbol is described by the `auto_tilde` tactic, whose default implementation is `auto`.

```
Ltac auto_tilde ::= auto.
```

In the examples that follow, only `auto_star` is needed.

An alternative, possibly more efficient version of `auto_star` is the following:

```
Ltac auto_star ::= try solve eassumption | auto | jauto .
```

With the above definition, `auto_star` first tries to solve the goal using the assumptions; if it fails, it tries using `auto`, and if this still fails, then it calls `jauto`. Even though `jauto` is strictly stronger than `eassumption` and `auto`, it makes sense to call these tactics first, because, when they succeed, they save a lot of time, and when they fail to prove the goal, they fail very quickly..

24.4 Example Proofs using Automation

Let's see how to use proof search in practice on the main theorems of the “Software Foundations” course, proving in particular results such as determinism, preservation and progress.

24.4.1 Determinism

Module `DETERMINISTICIMP`.

Import `IMP`.

Recall the original proof of the determinism lemma for the IMP language, shown below.

Theorem `ceval_deterministic`: $\forall c\ st\ st1\ st2,$
 $st = [c] \Rightarrow st1 \rightarrow$
 $st = [c] \Rightarrow st2 \rightarrow$
 $st1 = st2.$

Proof.

```
intros c st st1 st2 E1 E2.
generalize dependent st2.
induction E1; intros st2 E2; inversion E2; subst.
- reflexivity.
- reflexivity.
-
  assert (st' = st'0) as EQ1.
  { apply IHE1_1; assumption. }
  subst st'0.
  apply IHE1_2. assumption.
-
  apply IHE1. assumption.
-
  rewrite H in H5. inversion H5.
-
  rewrite H in H5. inversion H5.
-
  apply IHE1. assumption.
-
  reflexivity.
-
  rewrite H in H2. inversion H2.
-
  rewrite H in H4. inversion H4.
-
  assert (st' = st'0) as EQ1.
  { apply IHE1_1; assumption. }
```



```

      subst st'0.
      apply IHE1_2. assumption.
Qed.

```

Exercise: rewrite this proof using `auto` whenever possible. (The solution uses `auto` 9 times.)

```

Theorem ceval_deterministic':  $\forall c\ st\ st1\ st2,$ 
   $st = [c] \Rightarrow st1 \rightarrow$ 
   $st = [c] \Rightarrow st2 \rightarrow$ 
   $st1 = st2.$ 

```

Proof.

```

  admit.

```

Admitted.

In fact, using automation is not just a matter of calling `auto` in place of one or two other tactics. Using automation is about rethinking the organization of sequences of tactics so as to minimize the effort involved in writing and maintaining the proof. This process is eased by the use of the tactics from *LibTactics.v*. So, before trying to optimize the way automation is used, let's first rewrite the proof of determinism:

- use `introv H` instead of `intros x H`,
- use `gen x` instead of `generalize dependent x`,
- use `inverts H` instead of `inversion H`; `subst`,
- use `tryfalse` to handle contradictions, and get rid of the cases where `beval st b1 = true` and `beval st b1 = false` both appear in the context.

```

Theorem ceval_deterministic'':  $\forall c\ st\ st1\ st2,$ 
   $st = [c] \Rightarrow st1 \rightarrow$ 
   $st = [c] \Rightarrow st2 \rightarrow$ 
   $st1 = st2.$ 

```

Proof.

```

  introv E1 E2. gen st2.
  induction E1; intros; inverts E2; tryfalse.
- auto.
- auto.
- assert (st' = st'0). auto. subst. auto.
- auto.
- auto.
- auto.
- assert (st' = st'0). auto. subst. auto.

```

Qed.

To obtain a nice clean proof script, we have to remove the calls `assert (st' = st'0)`. Such a tactic call is not nice because it refers to some variables whose name has been automatically generated. This kind of tactics tend to be very brittle. The tactic `assert (st' = st'0)` is used to assert the conclusion that we want to derive from the induction hypothesis. So, rather than stating this conclusion explicitly, we are going to ask Rocq to instantiate the induction hypothesis, using automation to figure out how to instantiate it. The tactic *forwards*, described in *LibTactics.v* precisely helps with instantiating a fact. So, let's see how it works out on our example.

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
  st = [ c ] => st1 →
  st = [ c ] => st2 →
  st1 = st2.
```

Proof.

```
  introv E1 E2. gen st2.
  induction E1; intros; inverts E2; tryfalse.
  - auto.
  - auto.
  - dup 4.

  + assert (st' = st'0). apply IHE1_1. apply H1.
    skip.

  + forwards: IHE1_1. apply H1.
    skip.

  + forwards: IHE1_1. eauto.
    skip.

  + forwards*: IHE1_1.
    skip.
```

Abort.

To polish the proof script, it remains to factorize the calls to `auto`, using the star symbol. The proof of determinism can then be rewritten in just 4 lines, including no more than 10 tactics.

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
  st = [ c ] => st1 →
  st = [ c ] => st2 →
  st1 = st2.
```

Proof.

```
  introv E1 E2. gen st2.
  induction E1; intros; inverts× E2; tryfalse.
  - forwards*: IHE1_1. subst×.
  - forwards*: IHE1_1. subst×.
```

Qed.

End DETERMINISTICIMP.

24.4.2 Preservation for STLC

To investigate how to automate the proof of the lemma `preservation`, let us first import the definitions required to state that lemma.

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From PLF Require StlcProp.
Module PRESERVATIONPROGRESSSTLC.
Import STLC.
Import STLCPROP.
Import STLC.
Import STLCPROP.
```

Consider the proof of perservation of STLC, shown below. This proof already uses `eauto` through the triple-dot mechanism.

```
Theorem preservation :  $\forall t\ t'\ T,$ 
  <{ empty |-  $t \backslash \text{in } T$  }>  $\rightarrow$ 
   $t \rightarrow t' \rightarrow$ 
  <{ empty |-  $t' \backslash \text{in } T$  }>.
Proof with eauto.
  intros t t' T HT. generalize dependent t'.
  remember empty as Gamma.
  induction HT;
    intros t' HE; subst;
    try solve [inversion HE; subst; auto].
-
  inversion HE; subst...
+
  apply substitution_preserves_typing with T2...
  inversion HT1...
Qed.
```

Exercise: rewrite this proof using tactics from `LibTactics` and calling automation using the star symbol rather than the triple-dot notation. More precisely, make use of the tactics *inverts* $\times$ and *applies* $\times$ to call `auto` $\times$ after a call to *inverts* or to *applies*.

```
Theorem preservation' :  $\forall t\ t'\ T,$ 
  <{ empty |-  $t \backslash \text{in } T$  }>  $\rightarrow$ 
   $t \rightarrow t' \rightarrow$ 
  <{ empty |-  $t' \backslash \text{in } T$  }>.
Proof.
  admit.
Admitted.
```

24.4.3 Progress for STLC

Consider the proof of the progress theorem.

Theorem progress : $\forall t\ T,$
 $\langle \{ \text{empty} \mid -\ t \ \text{in } T \} \rangle \rightarrow$
value $t \vee \exists t',\ t \rightarrow t'.$

Proof with eauto.

```

intros t T Ht.
remember empty as Gamma.
induction Ht; subst Gamma; auto.
-
  discriminate H.
-
  right. destruct IHHT1...
  +
    destruct IHHT2...
    ×
      eapply canonical_forms_fun in Ht1; [|assumption|.
      destruct Ht1 as [x [t0 H1]]. subst.
       $\exists (\langle \{ [x:=t2]t0 \} \rangle)...$ 
    ×
      destruct H0 as [t2' Hstp].  $\exists (\langle \{ t1\ t2' \} \rangle)...$ 
  +
    destruct H as [t1' Hstp].  $\exists (\langle \{ t1'\ t2 \} \rangle)...$ 
-
  right. destruct IHHT1...
  +
    destruct (canonical_forms_bool t1); subst; eauto.
  +
    destruct H as [t1' Hstp].  $\exists \langle \{ \text{if } t1' \text{ then } t2 \text{ else } t3 \} \rangle...$ 

```

Qed.

Exercise: optimize the above proof. Hint: make use of `destruct×` and `inverts×`. The solution fits on 10 short lines.

Theorem progress' : $\forall t\ T,$
 $\langle \{ \text{empty} \mid -\ t \ \text{in } T \} \rangle \rightarrow$
value $t \vee \exists t',\ t \rightarrow t'.$

Proof.

admit.

Admitted.

End PRESERVATIONPROGRESSSTLC.

24.4.4 BigStep and SmallStep

```
From PLF Require Import Smallstep.
From Stdlib Require Import Program.
Module SEMANTICS.
```

Consider the proof relating a small-step reduction judgment to a big-step reduction judgment.

```
Theorem multistep__eval : ∀ t v ,
  normal_form_of step t v → ∃ n , v = C n ∧ t ==> n.
Proof.
  intros t v Hnorm.
  unfold normal_form_of in Hnorm.
  inversion Hnorm as [Hs Hnf] ; clear Hnorm.
  rewrite nf_same_as_value in Hnf. inversion Hnf. clear Hnf.
  ∃ n. split. reflexivity.
  induction Hs ; subst.
-
  apply E_C.
-
  eapply step__eval. eassumption. apply IHHS. reflexivity.
Qed.
```

Our goal is to optimize the above proof. It is generally easier to isolate inductions into separate lemmas. So, we are going to first prove an intermediate result that consists of the judgment over which the induction is being performed.

Exercise: prove the following result, using tactics *introv*, *induction* and *subst*, and *apply*×. The solution fits on 3 short lines.

```
Theorem multistep_eval_ind : ∀ t v ,
  t ->* v → ∀ n , C n = v → t ==> n.
Proof.
  admit.
Admitted.
```

Exercise: using the lemma above, simplify the proof of the result *multistep__eval*. You should use the tactics *introv*, *inverts*, *split*× and *apply*×. The solution fits on 2 lines.

```
Theorem multistep__eval' : ∀ t v ,
  normal_form_of step t v → ∃ n , v = C n ∧ t ==> n.
Proof.
  admit.
Admitted.
```

If we try to combine the two proofs into a single one, we will likely fail, because of a limitation of the `induction` tactic. Indeed, this tactic loses information when applied to a property whose arguments are not reduced to variables, such as $t \rightarrow^* (C\ n)$. You will thus need to use the more powerful tactic called `dependent induction`. (This tactic is available only after importing the `Program` library, as we did above.)

Exercise: prove the lemma `multistep__eval` without invoking the lemma `multistep_eval_ind`, that is, by inlining the proof by induction involved in `multistep_eval_ind`, using the tactic `dependent induction` instead of `induction`. The solution fits on 6 lines.

```
Theorem multistep__eval'' : ∀ t v,
  normal_form_of step t v → ∃ n, v = C n ∧ t ==> n.
```

Proof.

```
  admit.
```

Admitted.

End SEMANTICS.

24.4.5 Preservation for STLCRef

```
From Stdlib Require Import Lia.
```

```
From PLF Require Import References.
```

```
From Stdlib Require Import Program.
```

```
Module PRESERVATIONPROGRESSREFERENCES.
```

```
  Import STLCREF.
```

```
  Hint Resolve store_weakening extends_refl : core.
```

The proof of preservation for `STLCREF` can be found in chapter `References`. The optimized proof script is more than twice shorter. The following material explains how to build the optimized proof script. The resulting optimized proof script for the preservation theorem appears afterwards.

```
Theorem preservation : ∀ ST t t' T st st',
  <{ empty / ST |- t \in T }> →
  store_well_typed ST st →
  t / st -> t' / st' →
  ∃ ST',
    extends ST' ST ∧
    <{ empty / ST' |- t' \in T }> ∧
    store_well_typed ST' st'.
```

Proof.

```
  remember empty as Gamma.
```

```
  introv Ht. gen t'.
```

```
  induction Ht; introv HST Hstep;
```

```

    subst Gamma; inverts Hstep; eauto.
-

 $\exists$  ST. inverts Ht1. splits $\times$ . applies $\times$  substitution_preserves_typing.
-

forwards: IHHt1. eauto. eauto. eauto.
jauto_set_hyps; intros.
jauto_set_goal; intros.
eauto. eauto. eauto.
-

forwards*: IHHt2.
- forwards*: IHHt.
- forwards*: IHHt.
- forwards*: IHHt1.
- forwards*: IHHt2.
- forwards*: IHHt1.
-
+

 $\exists$  (ST ++ T1::nil). inverts keep HST. splits.
  apply extends_app.
  applys_eq T_Loc.
    unfold store_Tlookup. rewrite  $\leftarrow$  H. rewrite $\times$  app_nth2.
    rewrite Nat.sub_diag. simpl. reflexivity.
    rewrite length_app. simpl. lia.
  apply $\times$  store_well_typed_app.
- forwards*: IHHt.
-
+

 $\exists$  ST. splits $\times$ .
lets [_ Hsty]: HST.
applys_eq $\times$  Hsty.
inverts $\times$  Ht.

```

- forwards*: IHHT.

-

+

∃ ST. splits×. applys× assign_pres_store_typing. inverts× Ht1.

- forwards*: IHHT1.

- forwards*: IHHT2.

Qed.

Let's come back to the proof case that was hard to optimize. The difficulty comes from the statement of `nth_eq_last`, which takes the form `nth (length l) (l ++ x::nil) d = x`. This lemma is hard to exploit because its first argument, `length l`, mentions a list `l` that has to be exactly the same as the `l` occurring in `l ++ x::nil`. In practice, the first argument is often a natural number `n` that is provably equal to `length l` yet that is not syntactically equal to `length l`. There is a simple fix for making `nth_eq_last` easy to apply: introduce the intermediate variable `n` explicitly, so that the goal becomes `nth n (l ++ x::nil) d = x`, with a premise asserting `n = length l`.

Lemma `nth_eq_last'` : $\forall (A : \text{Type}) (l : \text{list } A) (x \ d : A) (n : \text{nat}),$

$n = \text{length } l \rightarrow \text{nth } n (l ++ x::\text{nil}) \ d = x.$

Proof. intros. subst. apply `nth_eq_last`. Qed.

The proof case for `ref` from the preservation theorem then becomes much easier to prove, because `rewrite nth_eq_last'` now succeeds.

Lemma `preservation_ref` : $\forall (st : \text{store}) (ST : \text{store\_ty}) \ T1,$

$\text{length } ST = \text{length } st \rightarrow$

$\langle \{ \text{Ref } T1 \} \rangle = \langle \{ \text{Ref } \$(\text{store\_Tlookup } (\text{length } st) (ST ++ T1::\text{nil})) \} \rangle.$

Proof.

intros. dup.

unfold `store_Tlookup`. rewrite× `nth_eq_last'`.

fequal. symmetry. apply× `nth_eq_last'`.

Qed.

The optimized proof of preservation is summarized next.

Theorem `preservation'` : $\forall \ ST \ t \ t' \ T \ st \ st',$

$\langle \{ \text{empty} / ST \mid - \ t \ \backslash \text{in } T \} \rangle \rightarrow$

$\text{store\_well\_typed } ST \ st \rightarrow$

$t / st \rightarrow t' / st' \rightarrow$

$\exists \ ST',$

$\text{extends } ST' \ ST \wedge$

$\langle \{ \text{empty} / ST' \mid - \ t' \ \backslash \text{in } T \} \rangle \wedge$

$\text{store\_well\_typed } ST' \ st'.$

Proof.


```

remember empty as Gamma.
introv Ht. gen t'.
induction Ht; introv HST Hstep; subst Gamma; inverts Hstep; eauto.
-  $\exists$  ST. inverts Ht1. splits $\times$ . applys $\times$  substitution_preserves_typing.
- forwards*: IHht1.
- forwards*: IHht2.
- forwards*: IHht.
- forwards*: IHht.
- forwards*: IHht1.
- forwards*: IHht2.
- forwards*: IHht1.
-  $\exists$  (ST ++ T1::nil). inverts keep HST. splits.
  apply extends_app.
  applys_eq T_Loc.
    unfold store_Tlookup. rewrite $\times$  nth_eq_last'.
    rewrite length_app. simpl. lia.
  apply $\times$  store_well_typed_app.
- forwards*: IHht.
-  $\exists$  ST. splits $\times$ . lets [_ Hsty]: HST.
  applys_eq $\times$  Hsty. inverts $\times$  Ht.
- forwards*: IHht.
-  $\exists$  ST. splits $\times$ . applys $\times$  assign_pres_store_typing. inverts $\times$  Ht1.
- forwards*: IHht1.
- forwards*: IHht2.
Qed.

```

24.4.6 Progress for STLCRef

The proof of progress for [STLCREF](#) can be found in chapter [References](#). The optimized proof script is, here again, about half the length.

Theorem progress : $\forall ST\ t\ T\ st,$
 $\langle \{ \text{empty} / ST \mid -\ t \in T \} \rangle \rightarrow$
 store_well_typed $ST\ st \rightarrow$
 (**value** $t \vee \exists t'\ st', t / st \rightarrow t' / st'$).

Proof.

```

introv Ht HST. remember empty as Gamma.
induction Ht; subst Gamma; tryfalse; try solve [left*].
- right. destruct $\times$  IHht1 as [K|].
  inverts K; inverts Ht1.
  destruct $\times$  IHht2.
- right. destruct $\times$  IHht as [K|].
  inverts K; try solve [inverts Ht]. eauto.

```

```

- right. destruct× IHht as [K|].
  inverts K; try solve [inverts Ht]. eauto.
- right. destruct× IHht1 as [K|].
  inverts K; try solve [inverts Ht1].
  destruct× IHht2 as [M|].
  inverts M; try solve [inverts Ht2]. eauto.
- right. destruct× IHht1 as [K|].
  inverts K; try solve [inverts Ht1]. destruct× n.
- right. destruct× IHht.
- right. destruct× IHht as [K|].
  inverts K; inverts Ht as M.
  inverts HST as N. rewrite× N in M.
- right. destruct× IHht1 as [K|].
  destruct× IHht2.
  inverts K; inverts Ht1 as M.
  inverts HST as N. rewrite× N in M.

```

Qed.

End PRESERVATIONPROGRESSREFERENCES.

24.4.7 Subtyping

From PLF Require Sub.

Module SUBTYPINGINVERSION.

Import SUB.STLCSUB.

Consider the inversion lemma for typing judgment of abstractions in a type system with subtyping. Lemma `abs_arrow` : $\forall x \ S1 \ s2 \ T1 \ T2$,

```

<{ empty |- \x:S1, s2 \in T1→T2 }> →
  T1 <: S1
  ∧ <{ x |-> S1 |- s2 \in T2 }>.

```

Proof with `eauto`.

```

intros x S1 s2 T1 T2 Hty.
apply typing_inversion_abs in Hty.
destruct Hty as [S2 [Hsub Hty1]].
apply sub_inversion_arrow in Hsub.
destruct Hsub as [U1 [U2 [Heq [Hsub1 Hsub2]]]].
injection Heq as Heq; subst...

```

Qed.

Exercise: optimize the proof script, using *introv*, *lets* and *inverts*×. In particular, you will find it useful to replace the pattern `apply K in H. destruct H as |` with `lets | : K H`. The solution fits on 3 lines.

Lemma `abs_arrow'` : $\forall x \ S1 \ s2 \ T1 \ T2$,

```

<{ empty |- \x:S1, s2 \in T1→T2 }> →
  T1 <: S1
  ∧ <{ x |-> S1 |- s2 \in T2 }>.
Proof.
  admit.
Admitted.
End SUBTYPINGINVERSION.

```

24.5 Advanced Topics in Proof Search

24.5.1 Stating Lemmas in the Right Way

Due to its depth-first strategy, `eauto` can get exponentially slower as the depth search increases, even when a short proof exists. In general, to make proof search run reasonably fast, one should avoid using a depth search greater than 5 or 6. Moreover, one should try to minimize the number of applicable lemmas, and usually put first the hypotheses whose proof usefully instantiates the existential variables.

In fact, the ability for `eauto` to solve certain goals actually depends on the order in which the hypotheses are stated. This point is illustrated through the following example, in which P is a property of natural numbers. This property is such that $P\ n$ holds for any n as soon as $P\ m$ holds for at least one m different from zero. The goal is to prove that $P\ 2$ implies $P\ 1$. When the hypothesis about P is stated in the form $\forall\ n\ m, P\ m \rightarrow m \neq 0 \rightarrow P\ n$, then `eauto` works. However, with $\forall\ n\ m, m \neq 0 \rightarrow P\ m \rightarrow P\ n$, the tactic `eauto` fails.

```

Lemma order_matters_1 : ∀ (P : nat→Prop),
  (∀ n m, P m → m ≠ 0 → P n) →
  P 5 →
  P 1.

```

```

Proof.
  eauto. Qed.

```

```

Lemma order_matters_2 : ∀ (P : nat→Prop),
  (∀ n m, m ≠ 0 → P m → P n) →
  P 5 →
  P 1.

```

```

Proof.
  eauto.
  intros P H K.
  eapply H.
  eauto.
Abort.

```

It is very important to understand that the hypothesis $\forall\ n\ m, P\ m \rightarrow m \neq 0 \rightarrow P\ n$ is `eauto`-friendly, whereas $\forall\ n\ m, m \neq 0 \rightarrow P\ m \rightarrow P\ n$ really isn't. Guessing a value of m

for which $P\ m$ holds and then checking that $m \neq 0$ holds works well because there are few values of m for which $P\ m$ holds. So, it is likely that `eauto` comes up with the right one. On the other hand, guessing a value of m for which $m \neq 0$ and then checking that $P\ m$ holds does not work well, because there are many values of m that satisfy $m \neq 0$ but not $P\ m$.

24.5.2 Unfolding of Definitions During Proof-Search

The use of intermediate definitions is generally encouraged in a formal development as it usually leads to more concise and more readable statements. Yet, definitions can make it a little harder to automate proofs. The problem is that it is not obvious for a proof search mechanism to know when definitions need to be unfolded. Note that a naive strategy that consists in unfolding all definitions before calling proof search does not scale up to large proofs, so we avoid it. This section introduces a few techniques for avoiding to manually unfold definitions before calling proof search.

To illustrate the treatment of definitions, let P be an abstract property on natural numbers, and let `myFact` be a definition denoting the proposition $P\ x$ holds for any x less than or equal to 3.

Axiom $P : \text{nat} \rightarrow \text{Prop}$.

Definition `myFact` := $\forall x, x \leq 3 \rightarrow P\ x$.

Proving that `myFact` under the assumption that $P\ x$ holds for any x should be trivial. Yet, `auto` fails to prove it unless we unfold the definition of `myFact` explicitly.

Lemma `demo_hint_unfold_goal_1` :

($\forall x, P\ x$) $\rightarrow$
`myFact`.

Proof.

`auto. unfold myFact. auto. Qed.`

To automate the unfolding of definitions that appear as proof obligation, one can use the command `Hint Unfold myFact` to tell Rocq that it should always try to unfold `myFact` when `myFact` appears in the goal.

Hint Unfold `myFact` : *core*.

This time, automation is able to see through the definition of `myFact`.

Lemma `demo_hint_unfold_goal_2` :

($\forall x, P\ x$) $\rightarrow$
`myFact`.

Proof. `auto. Qed.`

However, the `Hint Unfold` mechanism only works for unfolding definitions that appear in the goal. In general, proof search does not unfold definitions from the context. For example, assume we want to prove that $P\ 3$ holds under the assumption that `True` $\rightarrow$ `myFact`.

Lemma `demo_hint_unfold_context_1` :

(`True` $\rightarrow$ `myFact`) $\rightarrow$

```

P 3.
Proof.
  intros.
  auto.   unfold myFact in *. auto. Qed.

```

There is actually one exception to the previous rule: a constant occurring in an hypothesis is automatically unfolded if the hypothesis can be directly applied to the current goal. For example, `auto` can prove `myFact → P 3`, as illustrated below.

```

Lemma demo_hint_unfold_context_2 :
  myFact →
  P 3.
Proof. auto. Qed.

```

24.5.3 Automation for Proving Absurd Goals

In this section, we'll see that lemmas concluding on a negation are generally not useful as hints, and that lemmas whose conclusion is `False` can be useful hints but having too many of them makes proof search inefficient. We'll also see a practical work-around to the efficiency issue.

Consider the following lemma, which asserts that a number less than or equal to 3 is not greater than 3.

```

Parameter le_not_gt : ∀ x,
  (x ≤ 3) →
  ¬ (x > 3).

```

Equivalently, one could state that a number greater than three is not less than or equal to 3.

```

Parameter gt_not_le : ∀ x,
  (x > 3) →
  ¬ (x ≤ 3).

```

In fact, both statements are equivalent to a third one stating that $x \leq 3$ and $x > 3$ are contradictory, in the sense that they imply `False`.

```

Parameter le_gt_false : ∀ x,
  (x ≤ 3) →
  (x > 3) →
  False.

```

The following investigation aim at figuring out which of the three statments is the most convenient with respect to proof automation. The following material is enclosed inside a `Section`, so as to restrict the scope of the hints that we are adding. In other words, after the end of the section, the hints added within the section will no longer be active.

```

Section DemoAbsurd1.

```

Let's try to add the first lemma, *le_not_gt*, as hint, and see whether we can prove that the proposition $\exists x, x \leq 3 \wedge x > 3$ is absurd.

Hint Resolve *le_not_gt* : core.

Lemma demo_auto_absurd_1 :

$(\exists x, x \leq 3 \wedge x > 3) \rightarrow$

False.

Proof.

intros. *jauto_set*. eauto. eapply *le_not_gt*. eauto. eauto.

Qed.

The lemma *gt_not_le* is symmetric to *le_not_gt*, so it will not be any better. The third lemma, *le_gt_false*, is a more useful hint, because it concludes on **False**, so proof search will try to apply it when the current goal is **False**.

Hint Resolve *le_gt_false* : core.

Lemma demo_auto_absurd_2 :

$(\exists x, x \leq 3 \wedge x > 3) \rightarrow$

False.

Proof.

dup.

intros. *jauto_set*. eauto.

jauto.

Qed.

In summary, a lemma of the form $H1 \rightarrow H2 \rightarrow \mathbf{False}$ is a much more effective hint than $H1 \rightarrow \neg H2$, even though the two statements are equivalent up to the definition of the negation symbol $\neg$.

That said, one should be careful with adding lemmas whose conclusion is **False** as hint. The reason is that whenever reaching the goal **False**, the proof search mechanism will potentially try to apply all the hints whose conclusion is **False** before applying the appropriate one.

End DemoAbsurd1.

Adding lemmas whose conclusion is **False** as hint can be, locally, a very effective solution. However, this approach does not scale up for global hints. For most practical applications, it is reasonable to give the name of the lemmas to be exploited for deriving a contradiction. The tactic *false H*, provided by *LibTactics* serves that purpose: *false H* replaces the goal with **False** and calls *eapply H*. Its behavior is described next. Observe that any of the three statements *le_not_gt*, *gt_not_le* or *le_gt_false* can be used.

Lemma demo_false : $\forall x,$

$(x \leq 3) \rightarrow$

$(x > 3) \rightarrow$

$4 = 5.$

Proof.

```
intros. dup 4.
- false. eapply le_gt_false.
  + auto.      + skip.
- false. eapply le_gt_false.
  + eauto.     + eauto.
- false le_gt_false. eauto. eauto.
- false le_not_gt. eauto. eauto.
```

Abort.

In the above example, `false le_gt_false`; `eauto` proves the goal, but `false le_gt_false`; `auto` does not, because `auto` does not correctly instantiate the existential variable. Note that `false× le_gt_false` would not work either, because the star symbol tries to call `auto` first. So, there are two possibilities for completing the proof: either call `false le_gt_false`; `eauto`, or call `false× (le_gt_false 3)`.

24.5.4 Automation for Transitivity Lemmas

Some lemmas should never be added as hints, because they would very badly slow down proof search. The typical example is that of transitivity results. This section describes the problem and presents a general workaround.

Consider a subtyping relation, written `subtype S T`, that relates two object `S` and `T` of type `typ`. Assume that this relation has been proved reflexive and transitive. The corresponding lemmas are named `subtype_refl` and `subtype_trans`.

Parameter `typ` : Type.

Parameter `subtype` : `typ` → `typ` → Prop.

Parameter `subtype_refl` : ∀ `T`,
 `subtype T T`.

Parameter `subtype_trans` : ∀ `S T U`,
 `subtype S T` → `subtype T U` → `subtype S U`.

Adding reflexivity as hint is generally a good idea, so let's add reflexivity of subtyping as hint.

Hint Resolve `subtype_refl` : core.

Adding transitivity as hint is generally a bad idea. To understand why, let's add it as hint and see what happens. Because we cannot remove hints once we've added them, we are going to open a "Section," so as to restrict the scope of the transitivity hint to that section.

Section HintsTransitivity.

Hint Resolve `subtype_trans` : core.

Now, consider the goal $\forall S\ T, \text{subtype } S\ T$, which clearly has no hope of being solved. Let's call `eauto` on this goal.

```
Lemma transitivity_bad_hint_1 :  $\forall S\ T,$   
   $\text{subtype } S\ T$ .
```

Proof.

```
  intros. eauto. Abort.
```

Note that after closing the section, the hint `subtype_trans` is no longer active.

End HintsTransitivity.

In the previous example, the proof search has spent a lot of time trying to apply transitivity and reflexivity in every possible way. Its process can be summarized as follows. The first goal is $\text{subtype } S\ T$. Since reflexivity does not apply, `eauto` invokes transitivity, which produces two subgoals, $\text{subtype } S\ ?X$ and $\text{subtype } ?X\ T$. Solving the first subgoal, $\text{subtype } S\ ?X$, is straightforward, it suffices to apply reflexivity. This unifies $?X$ with S . So, the second subgoal, $\text{subtype } ?X\ T$, becomes $\text{subtype } S\ T$, which is exactly what we started from...

The problem with the transitivity lemma is that it is applicable to any goal concluding on a subtyping relation. Because of this, `eauto` keeps trying to apply it even though it most often doesn't help to solve the goal. So, one should never add a transitivity lemma as a hint for proof search.

There is a general workaround for having automation to exploit transitivity lemmas without giving up on efficiency. This workaround relies on a powerful mechanism called “external hint.” This mechanism allows to manually describe the condition under which a particular lemma should be tried out during proof search.

For the case of transitivity of subtyping, we are going to tell Rocq to try and apply the transitivity lemma on a goal of the form $\text{subtype } S\ U$ only when the proof context already contains an assumption either of the form $\text{subtype } S\ T$ or of the form $\text{subtype } T\ U$. In other words, we only apply the transitivity lemma when there is some evidence that this application might help. To set up this “external hint,” one has to write the following.

```
Hint Extern 1 ( $\text{subtype } ?S\ ?U$ )  $\Rightarrow$   
  match goal with  
  |  $H : \text{subtype } S\ ?T \vdash \_ \Rightarrow$  apply (@subtype_trans  $S\ T\ U$ )  
  |  $H : \text{subtype } ?T\ U \vdash \_ \Rightarrow$  apply (@subtype_trans  $S\ T\ U$ )  
  end : core.
```

This hint declaration can be understood as follows.

- “Hint Extern” introduces the hint.
- The number “1” corresponds to a priority for proof search. It doesn't matter so much what priority is used in practice.
- The pattern $\text{subtype } ?S\ ?U$ describes the kind of goal on which the pattern should apply. The question marks are used to indicate that the variables $?S$ and $?U$ should be bound to some value in the rest of the hint description.

- The construction `match goal with ... end` tries to recognize patterns in the goal, or in the proof context, or both.
- The first pattern is `H: subtype S ?T ⊢ _`. It indicates that the context should contain an hypothesis `H` of type `subtype S ?T`, where `S` has to be the same as in the goal, and where `?T` can have any value.
- The symbol `⊢ _` at the end of `H: subtype S ?T ⊢ _` indicates that we do not impose further condition on how the proof obligation has to look like.
- The branch `⇒ apply (@subtype_trans S T U)` that follows indicates that if the goal has the form `subtype S U` and if there exists an hypothesis of the form `subtype S T`, then we should try and apply transitivity lemma instantiated on the arguments `S`, `T` and `U`. (Note: the symbol `@` in front of `subtype_trans` is only actually needed when the “Implicit Arguments” feature is activated.)
- The other branch, which corresponds to an hypothesis of the form `H: subtype ?T U` is symmetrical.

Note: the same external hint can be reused for any other transitive relation, simply by renaming `subtype` into the name of that relation.

Let us see an example illustrating how the hint works.

```
Lemma transitivity_workaround_1 : ∀ T1 T2 T3 T4,
  subtype T1 T2 →
  subtype T2 T3 →
  subtype T3 T4 →
  subtype T1 T4.
```

Proof.

```
intros. eauto. Qed.
```

We may also check that the new external hint does not suffer from the complexity blow up.

```
Lemma transitivity_workaround_2 : ∀ S T,
  subtype S T.
```

Proof.

```
intros. eauto. Abort.
```

24.6 Decision Procedures

A decision procedure is able to solve proof obligations whose statement admits a particular form. This section describes three useful decision procedures. The tactic `lia` handles goals involving arithmetic and inequalities, but not general multiplications. The tactic `ring` handles goals involving arithmetic, including multiplications, but does not support inequalities. The tactic `congruence` is able to prove equalities and inequalities by exploiting equalities available in the proof context.

24.6.1 Lia

The tactic *lia* supports natural numbers (type **nat**) as well as integers (type **Z**, available by including the module **ZArith**). It supports addition, subtraction, equalities and inequalities. Before using *lia*, one needs to import the module **Lia**, as follows.

From *Stdlib* Require Import **Lia**.

Here is an example. Let *x* and *y* be two natural numbers (they cannot be negative). Assume *y* is less than 4, assume *x*+*x*+1 is less than *y*, and assume *x* is not zero. Then, it must be the case that *x* is equal to one.

```
Lemma lia_demo_1 : ∀ (x y : nat),  
  (y ≤ 4) →  
  (x + x + 1 ≤ y) →  
  (x ≠ 0) →  
  (x = 1).
```

Proof. intros. *lia*. Qed.

Another example: if *z* is the mean of *x* and *y*, and if the difference between *x* and *y* is at most 4, then the difference between *x* and *z* is at most 2.

```
Lemma lia_demo_2 : ∀ (x y z : nat),  
  (x + y = z + z) →  
  (x - y ≤ 4) →  
  (x - z ≤ 2).
```

Proof. intros. *lia*. Qed.

One can proof **False** using *lia* if the mathematical facts from the context are contradictory. In the following example, the constraints on the values *x* and *y* cannot be all satisfied in the same time.

```
Lemma lia_demo_3 : ∀ (x y : nat),  
  (x + 5 ≤ y) →  
  (y - x < 3) →  
  False.
```

Proof. intros. *lia*. Qed.

Note: *lia* can prove a goal by contradiction only if its conclusion reduces to **False**. The tactic *lia* always fails when the conclusion is an arbitrary proposition *P*, even though **False** implies any proposition *P* (by *ex_falso_quodlibet*).

```
Lemma lia_demo_4 : ∀ (x y : nat) (P : Prop),  
  (x + 5 ≤ y) →  
  (y - x < 3) →  
  P.
```

Proof.
 intros.
 lia.

Qed.

24.6.2 Ring

Compared with *lia*, the tactic *ring* adds support for multiplications, however it gives up the ability to reason on inequations. Moreover, it supports only integers (type *Z*) and not natural numbers (type *nat*). Here is an example showing how to use *ring*.

```
From Stdlib Require Import ZArith.
```

```
Module RINGDEMO.
```

```
  Open Scope Z_scope.
```

```
Lemma ring_demo :  $\forall (x\ y\ z : \mathbb{Z}),$ 
```

```
   $x \times (y + z) - z \times 3 \times x$   
   $= x \times y - 2 \times x \times z.$ 
```

```
Proof. intros. ring. Qed.
```

```
End RINGDEMO.
```

24.6.3 Congruence

The tactic *congruence* is able to exploit equalities from the proof context in order to automatically perform the rewriting operations necessary to establish a goal. It is slightly more powerful than the tactic *subst*, which can only handle equalities of the form $x = e$ where x is a variable and e an expression.

```
Lemma congruence_demo_1 :
```

```
   $\forall (f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) (g\ h : \text{nat} \rightarrow \text{nat}) (x\ y\ z : \text{nat}),$   
   $f (g\ x) (g\ y) = z \rightarrow$   
   $2 = g\ x \rightarrow$   
   $g\ y = h\ z \rightarrow$   
   $f\ 2 (h\ z) = z.$ 
```

```
Proof. intros. congruence. Qed.
```

Moreover, *congruence* is able to exploit universally quantified equalities, for example $\forall a, g\ a = h\ a$.

```
Lemma congruence_demo_2 :
```

```
   $\forall (f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) (g\ h : \text{nat} \rightarrow \text{nat}) (x\ y\ z : \text{nat}),$   
   $(\forall a, g\ a = h\ a) \rightarrow$   
   $f (g\ x) (g\ y) = z \rightarrow$   
   $g\ x = 2 \rightarrow$   
   $f\ 2 (h\ y) = z.$ 
```

```
Proof. congruence. Qed.
```

Next is an example where *congruence* is very useful.

```
Lemma congruence_demo_4 :  $\forall (f\ g : \text{nat} \rightarrow \text{nat}),$ 
```

```
   $(\forall a, f\ a = g\ a) \rightarrow$   
   $f (g (g\ 2)) = g (f (f\ 2)).$ 
```

```
Proof. congruence. Qed.
```

The tactic `congruence` is able to prove a contradiction if the goal entails an equality that contradicts an inequality available in the proof context.

Lemma `congruence_demo_3` :

```

  ∀ (f g h : nat → nat) (x : nat),
  (∀ a, f a = h a) →
  g x = f x →
  g x ≠ h x →
  False.

```

Proof. `congruence`. Qed.

One of the strengths of `congruence` is that it is a very fast tactic. So, one should not hesitate to invoke it wherever it might help.

24.7 Summary

Let us summarize the main automation tactics available.

- `auto` automatically applies `reflexivity`, `assumption`, and `apply`.
- `eauto` moreover tries `eapply`, and in particular can instantiate existentials in the conclusion.
- `info_auto`, `info_eauto`, `debug auto` and `debug eauto` can be used to obtain details on the proof search process.
- `iauto` extends `eauto` with support for negation, conjunctions, and disjunctions. However, its support for disjunction can make it exponentially slow.
- `jauto` extends `eauto` with support for negation, conjunctions, and existential at the head of hypothesis.
- `congruence` helps reasoning about equalities and inequalities.
- `lia` proves arithmetic goals with equalities and inequalities, but it does not support multiplication.
- `ring` proves arithmetic goals with multiplications, but does not support inequalities.

In order to set up automation appropriately, keep in mind the following rule of thumbs:

- automation is all about balance: not enough automation makes proofs not very robust on change, whereas too much automation makes proofs very hard to fix when they break.

- if a lemma is not goal directed (i.e., some of its variables do not occur in its conclusion), then the premises need to be ordered in such a way that proving the first premises maximizes the chances of correctly instantiating the variables that do not occur in the conclusion.
- a lemma whose conclusion is **False** should only be added as a local hint, i.e., as a hint within the current section.
- a transitivity lemma should never be considered as hint; if automation of transitivity reasoning is really necessary, an **Extern Hint** needs to be set up.
- a definition usually needs to be accompanied with a **Hint Unfold**.

Becoming a master in the black art of automation certainly requires some investment, however this investment will pay off very quickly.

Chapter 25

Library PLF.MapsTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Maps.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Maps.
Import CHECK.
Goal True.
idtac "————— t_update_same —————".
```

```

idtac " ".
idtac "#> t_update_same".
idtac "Possible points: 2".
check_type @t_update_same (
  (∀ (A : Type) (m : total_map A) (x : string),
    @eq (∀ _ : string, A) (@t_update A m x (m x)) m)).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_same.
Goal True.
idtac " ".

idtac "----- t_update_permute -----".
idtac " ".

idtac "#> t_update_permute".
idtac "Possible points: 3".
check_type @t_update_permute (
  (∀ (A : Type) (m : total_map A) (v1 v2 : A) (x1 x2 : string)
    (_ : not (@eq string x2 x1))),
  @eq (∀ _ : string, A) (@t_update A (@t_update A m x2 v2) x1 v1)
    (@t_update A (@t_update A m x1 v1) x2 v2))).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_permute.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 5".
idtac "Max points - advanced: 5".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq-congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".

```

```

idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- t_update_same -----".
Print Assumptions t_update_same.
idtac "----- t_update_permute -----".
Print Assumptions t_update_permute.
idtac "".
idtac "***** Advanced *****".
Abort.

```


Chapter 26

Library PLF.ImpTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Imp.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Imp.
Import CHECK.
Goal True.
idtac "————— optimize_0plus_b_sound —————".
```

```

idtac " ".
idtac "#> AExp.optimize_0plus_b_test1".
idtac "Possible points: 0.5".
check_type @AExp.optimize_0plus_b_test1 (
(@eq AExp.bexp
  (AExp.optimize_0plus_b
    (AExp.BNot
      (AExp.BGt (AExp.APlus (AExp.ANum 0) (AExp.ANum 4)) (AExp.ANum 8))))
    (AExp.BNot (AExp.BGt (AExp.ANum 4) (AExp.ANum 8)))))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_test1.
Goal True.
idtac " ".

idtac "#> AExp.optimize_0plus_b_test2".
idtac "Possible points: 0.5".
check_type @AExp.optimize_0plus_b_test2 (
(@eq AExp.bexp
  (AExp.optimize_0plus_b
    (AExp.BAnd
      (AExp.BLe (AExp.APlus (AExp.ANum 0) (AExp.ANum 4)) (AExp.ANum 5))
      AExp.BTrue))
    (AExp.BAnd (AExp.BLe (AExp.ANum 4) (AExp.ANum 5)) AExp.BTrue))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_test2.
Goal True.
idtac " ".

idtac "#> AExp.optimize_0plus_b_sound".
idtac "Possible points: 2".
check_type @AExp.optimize_0plus_b_sound (
(∀ b : AExp.bexp,
  @eq bool (AExp.beval (AExp.optimize_0plus_b b)) (AExp.beval b))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_sound.
Goal True.
idtac " ".

idtac "_____ bevalR _____".
idtac " ".

idtac "#> AExp.bevalR_iff_beval".

```

```

idtac "Possible points: 3".
check_type @AExp.bevalR_iff_beval (
  (∀ (b : AExp.bexp) (bv : bool),
    iff (AExp.bevalR b bv) (@eq bool (AExp.beval b) bv))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.bevalR_iff_beval.
Goal True.
idtac " ".

idtac "————- ceval_example2 —————".
idtac " ".

idtac "#> ceval_example2".
idtac "Possible points: 2".
check_type @ceval_example2 (
  (ceval (CSeq (CAsgn X (ANum 0)) (CSeq (CAsgn Y (ANum 1)) (CAsgn Z (ANum 2))))
    empty_st
    (@Maps.t_update nat
      (@Maps.t_update nat (@Maps.t_update nat empty_st X 0) Y 1) Z 2))).
idtac "Assumptions:".
Abort.
Print Assumptions ceval_example2.
Goal True.
idtac " ".

idtac "————- loop_never_stops —————".
idtac " ".

idtac "#> loop_never_stops".
idtac "Possible points: 3".
check_type @loop_never_stops ((∀ st st' : state, not (ceval loop st st'))).
idtac "Assumptions:".
Abort.
Print Assumptions loop_never_stops.
Goal True.
idtac " ".

idtac "————- no_whiles_eqv —————".
idtac " ".

idtac "#> no_whiles_eqv".
idtac "Possible points: 3".
check_type @no_whiles_eqv (
  (∀ c : com, iff (@eq bool (no_whiles c) true) (no_whilesR c))).
idtac "Assumptions:".
Abort.

```

```

Print Assumptions no_whiles_eqv.
Goal True.
idtac " ".

idtac "----- no_whiles_terminating -----".
idtac " ".

idtac "#> Manually graded: no_whiles_terminating".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_no_whiles_terminating.
idtac " ".

idtac "----- stack_compiler -----".
idtac " ".

idtac "#> s_execute1".
idtac "Possible points: 1".
check_type @s_execute1 (
  (@eq (list nat)
    (s_execute empty_st (@nil nat)
      (@cons sinstr (SPush 5)
        (@cons sinstr (SPush 3)
          (@cons sinstr (SPush 1) (@cons sinstr SMinus (@nil sinstr))))))
      (@cons nat 2 (@cons nat 5 (@nil nat)))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute1.
Goal True.
idtac " ".

idtac "#> s_execute2".
idtac "Possible points: 0.5".
check_type @s_execute2 (
  (@eq (list nat)
    (s_execute (@Maps.t_update nat empty_st X 3)
      (@cons nat 3 (@cons nat 4 (@nil nat)))
      (@cons sinstr (SPush 4)
        (@cons sinstr (SLoad X)
          (@cons sinstr SMult (@cons sinstr SPlus (@nil sinstr))))))
      (@cons nat 15 (@cons nat 4 (@nil nat)))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute2.
Goal True.
idtac " ".

idtac "#> s_compile1".

```

```

idtac "Possible points: 1.5".
check_type @s_compile1 (
  (@eq (list sinstr) (s_compile (AMinus (Ald X) (AMult (ANum 2) (Ald Y))))
    (@cons sinstr (SLoad X)
      (@cons sinstr (SPush 2)
        (@cons sinstr (SLoad Y)
          (@cons sinstr SMult (@cons sinstr SMinus (@nil sinstr)))))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile1.
Goal True.
idtac " ".

idtac "----- execute_app -----".
idtac " ".

idtac "#> execute_app".
idtac "Possible points: 3".
check_type @execute_app (
  (∀ (st : state) (p1 p2 : list sinstr) (stack : list nat),
    @eq (list nat) (s_execute st stack (@app sinstr p1 p2))
      (s_execute st (s_execute st stack p1) p2))).
idtac "Assumptions:".
Abort.
Print Assumptions execute_app.
Goal True.
idtac " ".

idtac "----- stack_compiler_correct -----".
idtac " ".

idtac "#> s_compile_correct_aux".
idtac "Possible points: 2.5".
check_type @s_compile_correct_aux (
  (∀ (st : state) (e : aexp) (stack : list nat),
    @eq (list nat) (s_execute st stack (s_compile e))
      (@cons nat (aeval st e) stack))).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile_correct_aux.
Goal True.
idtac " ".

idtac "#> s_compile_correct".
idtac "Possible points: 0.5".
check_type @s_compile_correct (

```

```

(∀ (st : state) (e : aexp),
  @eq (list nat) (s_execute st (@nil nat) (s_compile e))
    (@cons nat (aeval st e) (@nil nat)))).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile_correct.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 29".
idtac "Max points - advanced: 29".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- AExp.optimize_0plus_b_test1 -----".
Print Assumptions AExp.optimize_0plus_b_test1.
idtac "----- AExp.optimize_0plus_b_test2 -----".
Print Assumptions AExp.optimize_0plus_b_test2.
idtac "----- AExp.optimize_0plus_b_sound -----".
Print Assumptions AExp.optimize_0plus_b_sound.
idtac "----- AExp.bevalR_iff_beval -----".
Print Assumptions AExp.bevalR_iff_beval.
idtac "----- ceval_example2 -----".
Print Assumptions ceval_example2.
idtac "----- loop_never_stops -----".

```

```

Print Assumptions loop_never_stops.
idtac "----- no_whiles_eqv -----".
Print Assumptions no_whiles_eqv.
idtac "----- no_whiles_terminating -----".
idtac "MANUAL".
idtac "----- s_execute1 -----".
Print Assumptions s_execute1.
idtac "----- s_execute2 -----".
Print Assumptions s_execute2.
idtac "----- s_compile1 -----".
Print Assumptions s_compile1.
idtac "----- execute_app -----".
Print Assumptions execute_app.
idtac "----- s_compile_correct_aux -----".
Print Assumptions s_compile_correct_aux.
idtac "----- s_compile_correct -----".
Print Assumptions s_compile_correct.
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 27

Library PLF.PrefaceTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Preface.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Preface.
Import CHECK.
Goal True.
idtac " ".
```



```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 28

Library PLF.EquivTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Equiv.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Equiv.
Import CHECK.
Goal True.
idtac "————— skip_right —————".
```

```

idtac " ".
idtac "#> skip_right".
idtac "Possible points: 2".
check_type @skip_right (( $\forall$  c : com, cequiv (CSeq c CSkip) c)).
idtac "Assumptions:".
Abort.
Print Assumptions skip_right.
Goal True.
idtac " ".
idtac "----- if_false -----".
idtac " ".
idtac "#> if_false".
idtac "Possible points: 2".
check_type @if_false (
( $\forall$  (b : bexp) (c1 c2 : com) (_ : bequiv b BFalse),
  cequiv (Clf b c1 c2) c2)).
idtac "Assumptions:".
Abort.
Print Assumptions if_false.
Goal True.
idtac " ".
idtac "----- swap_if_branches -----".
idtac " ".
idtac "#> swap_if_branches".
idtac "Possible points: 3".
check_type @swap_if_branches (
( $\forall$  (b : bexp) (c1 c2 : com), cequiv (Clf b c1 c2) (Clf (BNot b) c2 c1))).
idtac "Assumptions:".
Abort.
Print Assumptions swap_if_branches.
Goal True.
idtac " ".
idtac "----- while_true -----".
idtac " ".
idtac "#> while_true".
idtac "Possible points: 2".
check_type @while_true (
( $\forall$  (b : bexp) (c : com) (_ : bequiv b BTrue),
  cequiv (CWhile b c) (CWhile BTrue CSkip))).
idtac "Assumptions:".
Abort.

```

```

Print Assumptions while_true.
Goal True.
idtac " ".

idtac "----- assign_aequiv -----".
idtac " ".

idtac "#> assign_aequiv".
idtac "Possible points: 2".
check_type @assign_aequiv (
  (∀ (X : String.string) (a : aexp) ( _ : aequiv (Ald X) a),
    cequiv CSkip (CAsgn X a))).
idtac "Assumptions:".
Abort.
Print Assumptions assign_aequiv.
Goal True.
idtac " ".

idtac "----- Clf_congruence -----".
idtac " ".

idtac "#> Clf_congruence".
idtac "Possible points: 3".
check_type @Clf_congruence (
  (∀ (b b' : bexp) (c1 c1' c2 c2' : com) ( _ : bequiv b b')
    ( _ : cequiv c1 c1') ( _ : cequiv c2 c2'),
    cequiv (Clf b c1 c2) (Clf b' c1' c2'))).
idtac "Assumptions:".
Abort.
Print Assumptions Clf_congruence.
Goal True.
idtac " ".

idtac "----- not_congr -----".
idtac " ".

idtac "#> Manually graded: not_congr".
idtac "Advanced".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_not_congr.
idtac " ".

idtac "----- fold_constants_com_sound -----".
idtac " ".

idtac "#> fold_constants_com_sound".
idtac "Possible points: 3".
check_type @fold_constants_com_sound ((ctrans_sound fold_constants_com)).

```

```

idtac "Assumptions:".
Abort.
Print Assumptions fold_constants_com_sound.
Goal True.
idtac " ".

idtac "————- inequiv_exercise —————".
idtac " ".

idtac "#> inequiv_exercise".
idtac "Possible points: 3".
check_type @inequiv_exercise ((not (cequiv (CWhile BTrue CSkip) CSkip))).
idtac "Assumptions:".
Abort.
Print Assumptions inequiv_exercise.
Goal True.
idtac " ".

idtac "————- himp_ceval —————".
idtac " ".

idtac "#> Manually graded: Himp.Check_rule_for_HAVOC".
idtac "Possible points: 2".
print_manual_grade Himp.manual_grade_for_Check_rule_for_HAVOC.
idtac " ".

idtac "————- havoc_swap —————".
idtac " ".

idtac "#> Himp.pXY_cequiv_pYX".
idtac "Possible points: 3".
check_type @Himp.pXY_cequiv_pYX (
(or (Himp.cequiv Himp.pXY Himp.pYX) (not (Himp.cequiv Himp.pXY Himp.pYX)))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.pXY_cequiv_pYX.
Goal True.
idtac " ".

idtac "————- p1_p2_term —————".
idtac " ".

idtac "#> Himp.p1_may_diverge".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.p1_may_diverge (
(∀ (st : ∀ _ : String.string, nat) (st' : state)
  (_ : not (@eq nat (st X) 0)),

```

```

    not (Himp.ceval Himp.p1 st st'))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p1_may_diverge.
Goal True.
idtac " ".

idtac "#> Himp.p2_may_diverge".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.p2_may_diverge (
  (∀ (st : ∀ _ : String.string, nat) (st' : state)
    (_ : not (@eq nat (st X) 0)),
    not (Himp.ceval Himp.p2 st st'))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p2_may_diverge.
Goal True.
idtac " ".

idtac "----- p1_p2_equiv -----".
idtac " ".

idtac "#> Himp.p1_p2_equiv".
idtac "Advanced".
idtac "Possible points: 6".
check_type @Himp.p1_p2_equiv ((Himp.cequiv Himp.p1 Himp.p2)).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p1_p2_equiv.
Goal True.
idtac " ".

idtac "----- p3_p4_inequiv -----".
idtac " ".

idtac "#> Himp.p3_p4_inequiv".
idtac "Advanced".
idtac "Possible points: 6".
check_type @Himp.p3_p4_inequiv ((not (Himp.cequiv Himp.p3 Himp.p4))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p3_p4_inequiv.
Goal True.
idtac " ".
idtac " ".

```

```

idtac "Max points - standard: 25".
idtac "Max points - advanced: 46".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- skip_right -----".
Print Assumptions skip_right.
idtac "----- if_false -----".
Print Assumptions if_false.
idtac "----- swap_if_branches -----".
Print Assumptions swap_if_branches.
idtac "----- while_true -----".
Print Assumptions while_true.
idtac "----- assign_aequiv -----".
Print Assumptions assign_aequiv.
idtac "----- CIf_congruence -----".
Print Assumptions CIf_congruence.
idtac "----- fold_constants_com_sound -----".
Print Assumptions fold_constants_com_sound.
idtac "----- inequiv_exercise -----".
Print Assumptions inequiv_exercise.
idtac "----- Check_rule_for_HAVOC -----".
idtac "MANUAL".
idtac "----- Himp.pXY_cequiv_pYX -----".
Print Assumptions Himp.pXY_cequiv_pYX.

```

```

idtac "".
idtac "***** Advanced *****".
idtac "----- not_congr -----".
idtac "MANUAL".
idtac "----- Himp.p1_may_diverge -----".
Print Assumptions Himp.p1_may_diverge.
idtac "----- Himp.p2_may_diverge -----".
Print Assumptions Himp.p2_may_diverge.
idtac "----- Himp.p1_p2_equiv -----".
Print Assumptions Himp.p1_p2_equiv.
idtac "----- Himp.p3_p4_inequiv -----".
Print Assumptions Himp.p3_p4_inequiv.
Abort.

```


Chapter 29

Library `PLF.HoareTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Hoare.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Hoare.
Import CHECK.
Goal True.
idtac "————— hoare_post_true —————".
```

```

idtac " ".
idtac "#> hoare_post_true".
idtac "Possible points: 1".
check_type @hoare_post_true (
  (∀ (P Q : Assertion) (c : com) (_ : ∀ st : state, Q st),
    valid_hoare_triple P c Q)).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_post_true.
Goal True.
idtac " ".

idtac "————— hoare_asgn_wrong —————".
idtac " ".

idtac "#> hoare_asgn_wrong".
idtac "Possible points: 2".
check_type @hoare_asgn_wrong (
  (@ex aexp
    (fun a : aexp ⇒
      not
        (valid_hoare_triple (fun _ : state ⇒ True)
          (CAsgn X a)
          (fun st : state ⇒
            @eq nat ((Aexp_of_aexp (Ald X) : Aexp) st)
              ((Aexp_of_aexp a : Aexp) st)))))).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_asgn_wrong.
Goal True.
idtac " ".

idtac "————— hoare_asgn_examples_2 —————".
idtac " ".

idtac "#> assertion_sub_ex1'".
idtac "Possible points: 1".
check_type @assertion_sub_ex1' (
  (valid_hoare_triple
    (fun st : state ⇒
      le ((Aexp_of_aexp (Ald X) : Aexp) st) ((Aexp_of_nat 5 : Aexp) st))
    (CAsgn X (AMult (ANum 2) (Ald X)))
    (fun st : state ⇒
      le ((Aexp_of_aexp (Ald X) : Aexp) st) ((Aexp_of_nat 10 : Aexp) st)))).
idtac "Assumptions:".

```

```

Abort.
Print Assumptions assertion_sub_ex1'.
Goal True.
idtac " ".

idtac "#> assertion_sub_ex2'".
idtac "Possible points: 1".
check_type @assertion_sub_ex2' (
(valid_hoare_triple
  (fun st : state ⇒
    and
      (((fun st0 : state ⇒
        le ((Aexp_of_nat 0 : Aexp) st0) ((Aexp_of_nat 3 : Aexp) st0))
        :
        Assertion) st)
      (((fun st0 : state ⇒
        le ((Aexp_of_nat 3 : Aexp) st0) ((Aexp_of_nat 5 : Aexp) st0))
        :
        Assertion) st))
  (CAsgn X (ANum 3))
  (fun st : state ⇒
    and
      (((fun st0 : state ⇒
        le ((Aexp_of_nat 0 : Aexp) st0) ((Aexp_of_aexp (Ald X) : Aexp) st0))
        :
        Assertion) st)
      (((fun st0 : state ⇒
        le ((Aexp_of_aexp (Ald X) : Aexp) st0) ((Aexp_of_nat 5 : Aexp) st0))
        :
        Assertion) st))))).
idtac "Assumptions:".
Abort.
Print Assumptions assertion_sub_ex2'.
Goal True.
idtac " ".

idtac "————- hoare_asgn_example4 —————".
idtac " ".

idtac "#> hoare_asgn_example4".
idtac "Possible points: 2".
check_type @hoare_asgn_example4 (
(valid_hoare_triple (fun _ : state ⇒ True)
  (CSeq (CAsgn X (ANum 1)) (CAsgn Y (ANum 2)))
  (fun st : state ⇒

```

```

and
  (((fun st0 : state =>
    @eq nat ((Aexp_of_aexp (Ald X) : Aexp) st0)
      ((Aexp_of_nat 1 : Aexp) st0))
    :
    Assertion) st)
  (((fun st0 : state =>
    @eq nat ((Aexp_of_aexp (Ald Y) : Aexp) st0)
      ((Aexp_of_nat 2 : Aexp) st0))
    :
    Assertion) st))))).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_asgn_example4.
Goal True.
idtac " ".

idtac "————- swap_exercise —————".
idtac " ".

idtac "#> swap_exercise".
idtac "Possible points: 3".
check_type @swap_exercise (
(valid_hoare_triple
  (fun st : state =>
    le ((Aexp_of_aexp (Ald X) : Aexp) st) ((Aexp_of_aexp (Ald Y) : Aexp) st))
  swap_program
  (fun st : state =>
    le ((Aexp_of_aexp (Ald Y) : Aexp) st) ((Aexp_of_aexp (Ald X) : Aexp) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions swap_exercise.
Goal True.
idtac " ".

idtac "————- invalid_triple —————".
idtac " ".

idtac "#> invalid_triple".
idtac "Advanced".
idtac "Possible points: 6".
check_type @invalid_triple (
(not
  (∀ (a : aexp) (n : nat),
    valid_hoare_triple

```

```

    (fun st : state ⇒
      @eq nat ((Aexp_of_aexp a : Aexp) st) ((Aexp_of_nat n : Aexp) st))
    (CSeq (CAsgn X (ANum 3)) (CAsgn Y a))
    (fun st : state ⇒
      @eq nat ((Aexp_of_aexp (Ald Y) : Aexp) st) ((Aexp_of_nat n : Aexp) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions invalid_triple.
Goal True.
idtac " ".
idtac "————— if_minus_plus —————".
idtac " ".
idtac "#> if_minus_plus".
idtac "Possible points: 2".
check_type @if_minus_plus (
(valid_hoare_triple (fun _ : state ⇒ True)
  (Clf (BLe (Ald X) (Ald Y)) (CAsgn Z (AMinus (Ald Y) (Ald X)))
    (CAsgn Y (APlus (Ald X) (Ald Z)))))
  (fun st : state ⇒
    @eq nat ((Aexp_of_aexp (Ald Y) : Aexp) st)
      (((fun st0 : state ⇒
        PeanoNat.Nat.add ((Aexp_of_aexp (Ald X) : Aexp) st0)
          ((Aexp_of_aexp (Ald Z) : Aexp) st0))
        :
        Aexp) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions if_minus_plus.
Goal True.
idtac " ".
idtac "————— if1_ceval —————".
idtac " ".
idtac "#> If1.if1true_test".
idtac "Possible points: 1".
check_type @If1.if1true_test (
(If1.ceval (If1.Clf1 (BEq (Ald X) (ANum 0)) (If1.CAsgn X (ANum 1))) empty_st
  (@Maps.t_update nat empty_st X 1))).
idtac "Assumptions:".
Abort.
Print Assumptions If1.if1true_test.
Goal True.

```

```

idtac " ".
idtac "#> If1.if1false_test".
idtac "Possible points: 1".
check_type @If1.if1false_test (
  (If1.ceval (If1.CIf1 (BEq (Ald X) (ANum 0)) (If1.CAsgn X (ANum 1)))
    (@Maps.t_update nat empty_st X 2) (@Maps.t_update nat empty_st X 2))).
idtac "Assumptions:".
Abort.
Print Assumptions If1.if1false_test.
Goal True.
idtac " ".

idtac "----- hoare_if1 -----".
idtac " ".

idtac "#> Manually graded: If1.hoare_if1".
idtac "Possible points: 2".
print_manual_grade If1.manual_grade_for_hoare_if1.
idtac " ".

idtac "----- hoare_if1_good -----".
idtac " ".

idtac "#> If1.hoare_if1_good".
idtac "Possible points: 2".
check_type @If1.hoare_if1_good (
  (If1.valid_hoare_triple
    (fun st : state =>
      @eq nat
      (((fun st0 : state =>
          PeanoNat.Nat.add ((Aexp_of_aexp (Ald X) : Aexp) st0)
            ((Aexp_of_aexp (Ald Y) : Aexp) st0))
        :
        Aexp) st)
      ((Aexp_of_aexp (Ald Z) : Aexp) st))
    (If1.CIf1 (BNeq (Ald Y) (ANum 0)) (If1.CAsgn X (APlus (Ald X) (Ald Y)))))
    (fun st : state =>
      @eq nat ((Aexp_of_aexp (Ald X) : Aexp) st)
        ((Aexp_of_aexp (Ald Z) : Aexp) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions If1.hoare_if1_good.
Goal True.
idtac " ".

idtac "----- hoare_havoc -----".

```

```

idtac " ".
idtac "#> Himp.hoare_havoc".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.hoare_havoc (
  (∀ (Q : Assertion) (X : String.string),
    Himp.valid_hoare_triple (Himp.havoc_pre X Q) (Himp.CHavoc X) Q)).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.hoare_havoc.
Goal True.
idtac " ".

idtac "----- havoc_post -----".
idtac " ".

idtac "#> Himp.havoc_post".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.havoc_post (
  (∀ (P : Assertion) (X : String.string),
    Himp.valid_hoare_triple P (Himp.CHavoc X)
      (fun st : state ⇒ @ex nat (fun n : nat ⇒ assertion_sub X (ANum n) P st)))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.havoc_post.
Goal True.
idtac " ".

idtac "----- assert_vs_assume -----".
idtac " ".

idtac "#> HoareAssertAssume.assert_assume_differ".
idtac "Possible points: 1".
check_type @HoareAssertAssume.assert_assume_differ (
  (@ex Assertion
    (fun P : Assertion ⇒
      @ex bexp
        (fun b : bexp ⇒
          @ex Assertion
            (fun Q : Assertion ⇒
              and
                (HoareAssertAssume.valid_hoare_triple P
                  (HoareAssertAssume.CAssume b) Q)
                (not

```

```

      (HoareAssertAssume.valid_hoare_triple P
        (HoareAssertAssume.CAssert b) Q)))))).
idtac "Assumptions:".
Abort.
Print Assumptions HoareAssertAssume.assert_assume_differ.
Goal True.
idtac " ".

idtac "#> HoareAssertAssume.assert_implies_assume".
idtac "Possible points: 1".
check_type @HoareAssertAssume.assert_implies_assume (
  (∀ (P : Assertion) (b : bexp) (Q : Assertion)
    (_ : HoareAssertAssume.valid_hoare_triple P (HoareAssertAssume.CAssert b)
      Q),
    HoareAssertAssume.valid_hoare_triple P (HoareAssertAssume.CAssume b) Q)).
idtac "Assumptions:".
Abort.
Print Assumptions HoareAssertAssume.assert_implies_assume.
Goal True.
idtac " ".

idtac "#> HoareAssertAssume.assert_assume_example".
idtac "Possible points: 4".
check_type @HoareAssertAssume.assert_assume_example (
  (HoareAssertAssume.valid_hoare_triple (fun _ : state ⇒ True)
    (HoareAssertAssume.CSeq (HoareAssertAssume.CAssume (BEq (Ald X) (ANum 1)))
      (HoareAssertAssume.CSeq
        (HoareAssertAssume.CAsgn X (APlus (Ald X) (ANum 1)))
        (HoareAssertAssume.CAssert (BEq (Ald X) (ANum 2))))))
    (fun _ : state ⇒ True))).
idtac "Assumptions:".
Abort.
Print Assumptions HoareAssertAssume.assert_assume_example.
Goal True.
idtac " ".
idtac " ".

idtac "Max points - standard: 24".
idtac "Max points - advanced: 36".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".

```



```

idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- hoare_post_true -----".
Print Assumptions hoare_post_true.
idtac "----- hoare_asgn_wrong -----".
Print Assumptions hoare_asgn_wrong.
idtac "----- assertion_sub_ex1' -----".
Print Assumptions assertion_sub_ex1'.
idtac "----- assertion_sub_ex2' -----".
Print Assumptions assertion_sub_ex2'.
idtac "----- hoare_asgn_example4 -----".
Print Assumptions hoare_asgn_example4.
idtac "----- swap_exercise -----".
Print Assumptions swap_exercise.
idtac "----- if_minus_plus -----".
Print Assumptions if_minus_plus.
idtac "----- If1.if1true_test -----".
Print Assumptions If1.if1true_test.
idtac "----- If1.if1false_test -----".
Print Assumptions If1.if1false_test.
idtac "----- hoare_if1 -----".
idtac "MANUAL".
idtac "----- If1.hoare_if1_good -----".
Print Assumptions If1.hoare_if1_good.
idtac "----- HoareAssertAssume.assert_assume_differ -----".
Print Assumptions HoareAssertAssume.assert_assume_differ.
idtac "----- HoareAssertAssume.assert_implies_assume -----".
Print Assumptions HoareAssertAssume.assert_implies_assume.
idtac "----- HoareAssertAssume.assert_assume_example -----".

```

```
Print Assumptions HoareAssertAssume.assert_assume_example.
idtac "".
idtac "***** Advanced *****".
idtac "----- invalid_triple -----".
Print Assumptions invalid_triple.
idtac "----- Himp.hoare_havoc -----".
Print Assumptions Himp.hoare_havoc.
idtac "----- Himp.havoc_post -----".
Print Assumptions Himp.havoc_post.
Abort.
```

Chapter 30

Library **PLF.Hoare2Test**

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Hoare2.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Hoare2.
Import CHECK.
Goal True.
idtac "————— if_minus_plus_correct —————".
```

```

idtac " ".
idtac "#> if_minus_plus_correct".
idtac "Possible points: 2".
check_type @if_minus_plus_correct ((outer_triple_valid if_minus_plus_dec)).
idtac "Assumptions:".
Abort.
Print Assumptions if_minus_plus_correct.
Goal True.
idtac " ".

idtac "————— slow_assignment —————".
idtac " ".

idtac "#> slow_assignment".
idtac "Possible points: 2".
check_type @slow_assignment (
(∀ m : nat, outer_triple_valid (slow_assignment_dec m))).
idtac "Assumptions:".
Abort.
Print Assumptions slow_assignment.
Goal True.
idtac " ".

idtac "————— factorial_correct —————".
idtac " ".

idtac "#> factorial_correct".
idtac "Advanced".
idtac "Possible points: 6".
check_type @factorial_correct ((∀ m : nat, outer_triple_valid (factorial_dec m))).
idtac "Assumptions:".
Abort.
Print Assumptions factorial_correct.
Goal True.
idtac " ".

idtac "————— minimum_correct —————".
idtac " ".

idtac "#> minimum_correct".
idtac "Advanced".
idtac "Possible points: 3".
check_type @minimum_correct ((∀ a b : nat, outer_triple_valid (minimum_dec a b))).
idtac "Assumptions:".
Abort.
Print Assumptions minimum_correct.
Goal True.

```

```

idtac " ".
idtac "————- two_loops —————".
idtac " ".
idtac "#> two_loops".
idtac "Possible points: 3".
check_type @two_loops (( $\forall$  a b c : nat, outer_triple_valid (two_loops_dec a b c))).
idtac "Assumptions:".
Abort.
Print Assumptions two_loops.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 7".
idtac "Max points - advanced: 16".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "————- if_minus_plus_correct ————".
Print Assumptions if_minus_plus_correct.
idtac "————- slow_assignment ————".
Print Assumptions slow_assignment.
idtac "————- two_loops ————".
Print Assumptions two_loops.
idtac "".

```

```
idtac "***** Advanced *****".
idtac "----- factorial_correct -----".
Print Assumptions factorial_correct.
idtac "----- minimum_correct -----".
Print Assumptions minimum_correct.
Abort.
```

Chapter 31

Library `PLF.HoareAsLogicTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import HoareAsLogic.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import HoareAsLogic.
Import CHECK.
Goal True.
idtac "————— provable_true_post —————".
```

```

idtac " ".
idtac "#> provable_true_post".
idtac "Possible points: 3".
check_type @provable_true_post (
  (∀ (c : Imp.com) (P : Hoare.Assertion),
    derivable P c (Hoare.assert_of_Prop True))).
idtac "Assumptions:".
Abort.
Print Assumptions provable_true_post.
Goal True.
idtac " ".
idtac "————- provable_false_pre —————".
idtac " ".
idtac "#> provable_false_pre".
idtac "Possible points: 3".
check_type @provable_false_pre (
  (∀ (c : Imp.com) (Q : Hoare.Assertion),
    derivable (Hoare.assert_of_Prop False) c Q)).
idtac "Assumptions:".
Abort.
Print Assumptions provable_false_pre.
Goal True.
idtac " ".
idtac "————- hoare_sound —————".
idtac " ".
idtac "#> hoare_sound".
idtac "Possible points: 3".
check_type @hoare_sound (
  (∀ (P : Hoare.Assertion) (c : Imp.com) (Q : Hoare.Assertion)
    (_ : derivable P c Q),
    valid P c Q)).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_sound.
Goal True.
idtac " ".
idtac "————- wp_seq —————".
idtac " ".
idtac "#> wp_seq".
idtac "Possible points: 1".
check_type @wp_seq (

```



```

(∀ (P Q : Hoare.Assertion) (c1 c2 : Imp.com)
  (_ : derivable P c1 (wp c2 Q)) (_ : derivable (wp c2 Q) c2 Q),
  derivable P (Imp.CSeq c1 c2) Q)).
idtac "Assumptions:".
Abort.
Print Assumptions wp_seq.
Goal True.
idtac " ".

idtac "----- wp_invariant -----".
idtac " ".

idtac "#> wp_invariant".
idtac "Possible points: 2".
check_type @wp_invariant (
  (∀ (b : Imp.bexp) (c : Imp.com) (Q : Hoare.Assertion),
    valid
      (fun st : Imp.state =>
        and ((wp (Imp.CWhile b c) Q : Hoare.Assertion) st)
          ((Hoare.bassertion b : Hoare.Assertion) st))
        c (wp (Imp.CWhile b c) Q))).
idtac "Assumptions:".
Abort.
Print Assumptions wp_invariant.
Goal True.
idtac " ".

idtac "----- hoare_complete -----".
idtac " ".

idtac "#> hoare_complete".
idtac "Possible points: 6".
check_type @hoare_complete (
  (∀ (P : Hoare.Assertion) (c : Imp.com) (Q : Hoare.Assertion)
    (_ : valid P c Q),
    derivable P c Q)).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_complete.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 18".
idtac "Max points - advanced: 18".
idtac "".

```

```

idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "——- provable_true_post ——-".
Print Assumptions provable_true_post.
idtac "——- provable_false_pre ——-".
Print Assumptions provable_false_pre.
idtac "——- hoare_sound ——-".
Print Assumptions hoare_sound.
idtac "——- wp_seq ——-".
Print Assumptions wp_seq.
idtac "——- wp_invariant ——-".
Print Assumptions wp_invariant.
idtac "——- hoare_complete ——-".
Print Assumptions hoare_complete.
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 32

Library `PLF.SmallstepTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Smallstep.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Smallstep.
Import CHECK.
Goal True.
idtac "————— test_step_2 —————".
```

```

idtac " ".
idtac "#> SimpleArith1.test_step_2".
idtac "Possible points: 1".
check_type @SimpleArith1.test_step_2 (
(SimpleArith1.step (P (C 0) (P (C 2) (P (C 1) (C 3))))
(P (C 0) (P (C 2) (C 4)))).
idtac "Assumptions:".
Abort.
Print Assumptions SimpleArith1.test_step_2.
Goal True.
idtac " ".
idtac "----- redo_determinism -----".
idtac " ".
idtac "#> step_deterministic".
idtac "Possible points: 3".
check_type @step_deterministic ((@deterministic tm step)).
idtac "Assumptions:".
Abort.
Print Assumptions step_deterministic.
Goal True.
idtac " ".
idtac "----- smallstep_bool_shortcut -----".
idtac " ".
idtac "#> Temp4.Temp5.bool_step_prop4_holds".
idtac "Possible points: 2".
check_type @Temp4.Temp5.bool_step_prop4_holds (Temp4.Temp5.bool_step_prop4).
idtac "Assumptions:".
Abort.
Print Assumptions Temp4.Temp5.bool_step_prop4_holds.
Goal True.
idtac " ".
idtac "----- test_multistep_4 -----".
idtac " ".
idtac "#> test_multistep_4".
idtac "Possible points: 2".
check_type @test_multistep_4 (
(@multi tm step (P (C 0) (P (C 2) (P (C 0) (C 3))))
(P (C 0) (C (Nat.add 2 (Nat.add 0 3))))).
idtac "Assumptions:".
Abort.
Print Assumptions test_multistep_4.

```

```

Goal True.
idtac " ".
idtac "————- multistep_congr_2 —————".
idtac " ".
idtac "#> multistep_congr_2".
idtac "Possible points: 2".
check_type @multistep_congr_2 (
(∀ (v1 t2 t2' : tm) ( _ : value v1) ( _ : @multi tm step t2 t2'),
 @multi tm step (P v1 t2) (P v1 t2')))).
idtac "Assumptions:".
Abort.
Print Assumptions multistep_congr_2.
Goal True.
idtac " ".
idtac "————- eval__multistep —————".
idtac " ".
idtac "#> eval__multistep".
idtac "Possible points: 3".
check_type @eval__multistep (
(∀ (t : tm) (n : nat) ( _ : eval t n), @multi tm step t (C n))).
idtac "Assumptions:".
Abort.
Print Assumptions eval__multistep.
Goal True.
idtac " ".
idtac "————- step__eval —————".
idtac " ".
idtac "#> step__eval".
idtac "Possible points: 3".
check_type @step__eval (
(∀ (t t' : tm) (n : nat) ( _ : step t t') ( _ : eval t' n), eval t n)).
idtac "Assumptions:".
Abort.
Print Assumptions step__eval.
Goal True.
idtac " ".
idtac "————- multistep__eval —————".
idtac " ".
idtac "#> multistep__eval".
idtac "Possible points: 3".

```

```

check_type @multistep__eval (
  (∀ (t t' : tm) ( _ : @normal_form_of tm step t t'),
    @ex nat (fun n : nat ⇒ and (@eq tm t' (C n)) (eval t n)))).
idtac "Assumptions:".
Abort.
Print Assumptions multistep__eval.
Goal True.
idtac " ".

idtac "————- combined_step_deterministic —————".
idtac " ".

idtac "#> Combined.combined_step_deterministic".
idtac "Possible points: 3".
check_type @Combined.combined_step_deterministic (
  (or (@deterministic Combined.tm Combined.step)
    (not (@deterministic Combined.tm Combined.step)))).
idtac "Assumptions:".
Abort.
Print Assumptions Combined.combined_step_deterministic.
Goal True.
idtac " ".

idtac "————- combined_strong_progress —————".
idtac " ".

idtac "#> Combined.combined_strong_progress".
idtac "Possible points: 3".
check_type @Combined.combined_strong_progress (
  (or
    (∀ t : Combined.tm,
      or (Combined.value t)
        (@ex Combined.tm (fun t' : Combined.tm ⇒ Combined.step t t'))))
    (not
      (∀ t : Combined.tm,
        or (Combined.value t)
          (@ex Combined.tm (fun t' : Combined.tm ⇒ Combined.step t t')))))).
idtac "Assumptions:".
Abort.
Print Assumptions Combined.combined_strong_progress.
Goal True.
idtac " ".

idtac "————- compiler_is_correct —————".
idtac " ".

idtac "#> compiler_is_correct".

```

```

idtac "Advanced".
idtac "Possible points: 3".
check_type @compiler_is_correct (compiler_is_correct_statement).
idtac "Assumptions:".
Abort.
Print Assumptions compiler_is_correct.
Goal True.
idtac " ".

idtac "----- normalize_ex -----".
idtac " ".

idtac "#> normalize_ex".
idtac "Possible points: 1".
check_type @normalize_ex (
  (@ex tm
    (fun e' : tm =>
      and (@multi tm step (P (C 3) (P (C 2) (C 1))) e') (value e')))).
idtac "Assumptions:".
Abort.
Print Assumptions normalize_ex.
Goal True.
idtac " ".
idtac " ".

idtac "Max points - standard: 26".
idtac "Max points - advanced: 29".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".

```

```

idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- SimpleArith1.test_step_2 -----".
Print Assumptions SimpleArith1.test_step_2.
idtac "----- step_deterministic -----".
Print Assumptions step_deterministic.
idtac "----- Temp4.Temp5.bool_step_prop4_holds -----".
Print Assumptions Temp4.Temp5.bool_step_prop4_holds.
idtac "----- test_multistep_4 -----".
Print Assumptions test_multistep_4.
idtac "----- multistep_congr_2 -----".
Print Assumptions multistep_congr_2.
idtac "----- eval__multistep -----".
Print Assumptions eval__multistep.
idtac "----- step__eval -----".
Print Assumptions step__eval.
idtac "----- multistep__eval -----".
Print Assumptions multistep__eval.
idtac "----- Combined.combined_step_deterministic -----".
Print Assumptions Combined.combined_step_deterministic.
idtac "----- Combined.combined_strong_progress -----".
Print Assumptions Combined.combined_strong_progress.
idtac "----- normalize_ex -----".
Print Assumptions normalize_ex.
idtac "".
idtac "***** Advanced *****".
idtac "----- compiler_is_correct -----".
Print Assumptions compiler_is_correct.
Abort.

```


Chapter 33

Library `PLF.TypesTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Types.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Types.
Import CHECK.
Goal True.
idtac "————— some_term_is_stuck —————".
```

```

idtac " ".
idtac "#> TM.some_term_is_stuck".
idtac "Possible points: 2".
check_type @TM.some_term_is_stuck ((@ex TM.tm (fun t : TM.tm => TM.stuck t))).
idtac "Assumptions:".
Abort.
Print Assumptions TM.some_term_is_stuck.
Goal True.
idtac " ".
idtac "----- value_is_nf -----".
idtac " ".
idtac "#> TM.value_is_nf".
idtac "Possible points: 3".
check_type @TM.value_is_nf (
  (∀ (t : TM.tm) (_ : TM.value t), @Smallstep.normal_form TM.tm TM.step t)).
idtac "Assumptions:".
Abort.
Print Assumptions TM.value_is_nf.
Goal True.
idtac " ".
idtac "----- finish_progress -----".
idtac " ".
idtac "#> TM.progress".
idtac "Possible points: 3".
check_type @TM.progress (
  (∀ (t : TM.tm) (T : TM.ty) (_ : TM.has_type t T),
    or (TM.value t) (@ex TM.tm (fun t' : TM.tm => TM.step t t')))).
idtac "Assumptions:".
Abort.
Print Assumptions TM.progress.
Goal True.
idtac " ".
idtac "----- finish_preservation -----".
idtac " ".
idtac "#> TM.preservation".
idtac "Possible points: 2".
check_type @TM.preservation (
  (∀ (t t' : TM.tm) (T : TM.ty) (_ : TM.has_type t T) (_ : TM.step t t'),
    TM.has_type t' T)).
idtac "Assumptions:".
Abort.

```

```

Print Assumptions TM.preservation.
Goal True.
idtac " ".
idtac "----- preservation_alternate_proof -----".
idtac " ".
idtac "#> TM.preservation'".
idtac "Possible points: 3".
check_type @TM.preservation' (
  (∀ (t t' : TM.tm) (T : TM.ty) ( _ : TM.has_type t T) ( _ : TM.step t t'),
    TM.has_type t' T)).
idtac "Assumptions:".
Abort.
Print Assumptions TM.preservation'.
Goal True.
idtac " ".
idtac "----- subject_expansion -----".
idtac " ".
idtac "#> TM.subject_expansion".
idtac "Possible points: 3".
check_type @TM.subject_expansion (
  (or
    (∀ (t t' : TM.tm) (T : TM.ty)
      ( _ : and (TM.step t t') (TM.has_type t' T)),
      TM.has_type t T)
    (not
      (∀ (t t' : TM.tm) (T : TM.ty)
        ( _ : and (TM.step t t') (TM.has_type t' T)),
        TM.has_type t T))))).
idtac "Assumptions:".
Abort.
Print Assumptions TM.subject_expansion.
Goal True.
idtac " ".
idtac "----- variation1 -----".
idtac " ".
idtac "#> Manually graded: TM.variation1".
idtac "Possible points: 2".
print_manual_grade TM.manual_grade_for_variation1.
idtac " ".
idtac "----- variation2 -----".
idtac " ".

```

```

idtac "#> Manually graded: TM.variation2".
idtac "Possible points: 2".
print_manual_grade TM.manual_grade_for_variation2.
idtac " ".

idtac "————- remove_pred0 —————".
idtac " ".

idtac "#> Manually graded: TM.remove_pred0".
idtac "Possible points: 1".
print_manual_grade TM.manual_grade_for_remove_pred0.
idtac " ".

idtac "————- prog_pres_bigstep —————".
idtac " ".

idtac "#> Manually graded: TM.prog_pres_bigstep".
idtac "Advanced".
idtac "Possible points: 6".
print_manual_grade TM.manual_grade_for_prog_pres_bigstep.
idtac " ".

idtac " ".

idtac "Max points - standard: 21".
idtac "Max points - advanced: 27".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "————- TM.some_term_is_stuck ————".

```

```

Print Assumptions TM.some_term_is_stuck.
idtac "——- TM.value_is_nf ——".
Print Assumptions TM.value_is_nf.
idtac "——- TM.progress ——".
Print Assumptions TM.progress.
idtac "——- TM.preservation ——".
Print Assumptions TM.preservation.
idtac "——- TM.preservation' ——".
Print Assumptions TM.preservation'.
idtac "——- TM.subject_expansion ——".
Print Assumptions TM.subject_expansion.
idtac "——- variation1 ——".
idtac "MANUAL".
idtac "——- variation2 ——".
idtac "MANUAL".
idtac "——- remove_pred0 ——".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
idtac "——- prog_pres_bigstep ——".
idtac "MANUAL".
Abort.

```

Chapter 34

Library `PLF.StlcTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Stlc.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | ""%string => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Stlc.
Import CHECK.
Goal True.
idtac "—————- substi_correct —————".
```

```

idtac " ".
idtac "#> STLC.substi_correct".
idtac "Possible points: 3".
check_type @STLC.substi_correct (
  (∀ (s : STLC.tm) (x : String.string) (t t' : STLC.tm),
    iff (@eq STLC.tm (STLC.subst x s t) t') (STLC.substi s x t t'))).
idtac "Assumptions:".
Abort.
Print Assumptions STLC.substi_correct.
Goal True.
idtac " ".

idtac "————- step_example5 —————".
idtac " ".

idtac "#> STLC.step_example5".
idtac "Possible points: 2".
check_type @STLC.step_example5 (
  (@Smallstep.multi STLC.tm STLC.step
    (STLC.tm_app
      (STLC.tm_app
        (STLC.tm_abs STLC.x
          (STLC.Ty_Arrow (STLC.Ty_Arrow STLC.Ty_Bool STLC.Ty_Bool)
            (STLC.Ty_Arrow STLC.Ty_Bool STLC.Ty_Bool))
          (STLC.tm_var STLC.x))
        (STLC.tm_abs STLC.x (STLC.Ty_Arrow STLC.Ty_Bool STLC.Ty_Bool)
          (STLC.tm_var STLC.x))))
      (STLC.tm_abs STLC.x STLC.Ty_Bool (STLC.tm_var STLC.x)))
    (STLC.tm_abs STLC.x STLC.Ty_Bool (STLC.tm_var STLC.x)))).
idtac "Assumptions:".
Abort.
Print Assumptions STLC.step_example5.
Goal True.
idtac " ".

idtac "————- typing_example_3 —————".
idtac " ".

idtac "#> STLC.typing_example_3".
idtac "Possible points: 2".
check_type @STLC.typing_example_3 (
  (@ex STLC.ty
    (fun T : STLC.ty =>
      STLC.has_type (@Maps.empty STLC.ty)
        (STLC.tm_abs STLC.x (STLC.Ty_Arrow STLC.Ty_Bool STLC.Ty_Bool))

```

```

      (STLC.tm_abs STLC.y (STLC.Ty_Arrow STLC.Ty_Bool STLC.Ty_Bool)
        (STLC.tm_abs STLC.z STLC.Ty_Bool
          (STLC.tm_app (STLC.tm_var STLC.y)
            (STLC.tm_app (STLC.tm_var STLC.x) (STLC.tm_var STLC.z))))))
    T))).
idtac "Assumptions:".
Abort.
Print Assumptions STLC.typing_example_3.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 7".
idtac "Max points - advanced: 7".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- STLC.substi_correct -----".
Print Assumptions STLC.substi_correct.
idtac "----- STLC.step_example5 -----".
Print Assumptions STLC.step_example5.
idtac "----- STLC.typing_example_3 -----".
Print Assumptions STLC.typing_example_3.
idtac "".
idtac "***** Advanced *****".
Abort.

```


Chapter 35

Library PLF.StlcPropTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import StlcProp.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | ""%string => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import StlcProp.
Import CHECK.
Goal True.
idtac "————— progress_from_term_ind —————".
```

```

idtac " ".
idtac "#> STLCProp.progress'".
idtac "Advanced".
idtac "Possible points: 3".
check_type @STLCProp.progress' (
  (∀ (t : Stlc.STLC.tm) (T : Stlc.STLC.ty)
    ( _ : Stlc.STLC.has_type (@Maps.empty Stlc.STLC.ty) t T),
    or (Stlc.STLC.value t)
      (@ex Stlc.STLC.tm (fun t' : Stlc.STLC.tm ⇒ Stlc.STLC.step t t')))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCProp.progress'.
Goal True.
idtac " ".

idtac "----- substitution_preserves_typing_from_typing_ind -----".
idtac " ".

idtac "#> STLCProp.substitution_preserves_typing_from_typing_ind".
idtac "Advanced".
idtac "Possible points: 3".
check_type @STLCProp.substitution_preserves_typing_from_typing_ind (
  (∀ (Gamma : Maps.partial_map Stlc.STLC.ty) (x : String.string)
    (U : Stlc.STLC.ty) (t v : Stlc.STLC.tm) (T : Stlc.STLC.ty)
    ( _ : Stlc.STLC.has_type (@Maps.update Stlc.STLC.ty Gamma x U) t T)
    ( _ : Stlc.STLC.has_type (@Maps.empty Stlc.STLC.ty) v U),
    Stlc.STLC.has_type Gamma (Stlc.STLC.subst x v t) T)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCProp.substitution_preserves_typing_from_typing_ind.
Goal True.
idtac " ".

idtac "----- subject_expansion_stlc -----".
idtac " ".

idtac "#> Manually graded: STLCProp.subject_expansion_stlc".
idtac "Possible points: 2".
print_manual_grade STLCProp.manual_grade_for_subject_expansion_stlc.
idtac " ".

idtac "----- unique_types -----".
idtac " ".

idtac "#> STLCProp.unique_types".
idtac "Possible points: 3".
check_type @STLCProp.unique_types (

```

```

(∀ (Gamma : Stlc.STLC.context) (e : Stlc.STLC.tm)
  (T T' : Stlc.STLC.ty) (_ : Stlc.STLC.has_type Gamma e T)
  (_ : Stlc.STLC.has_type Gamma e T'),
  @eq Stlc.STLC.ty T T')).
idtac "Assumptions:".
Abort.
Print Assumptions STLCProp.unique_types.
Goal True.
idtac " ".

idtac "—————- free_in_context —————".
idtac " ".

idtac "#> STLCProp.free_in_context".
idtac "Possible points: 2".
check_type @STLCProp.free_in_context (
  (∀ (x : String.string) (t : Stlc.STLC.tm) (T : Stlc.STLC.ty)
    (Gamma : Stlc.STLC.context) (_ : STLCProp.appears_free_in x t)
    (_ : Stlc.STLC.has_type Gamma t T),
    @ex Stlc.STLC.ty
      (fun T' : Stlc.STLC.ty =>
        @eq (option Stlc.STLC.ty) (Gamma x) (@Some Stlc.STLC.ty T')))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCProp.free_in_context.
Goal True.
idtac " ".

idtac "—————- stlc_variation1 —————".
idtac " ".

idtac "#> Manually graded: STLCProp.stlc_variation1".
idtac "Possible points: 2".
print_manual_grade STLCProp.manual_grade_for_stlc_variation1.
idtac " ".

idtac "—————- stlc_variation2 —————".
idtac " ".

idtac "#> Manually graded: STLCProp.stlc_variation2".
idtac "Possible points: 2".
print_manual_grade STLCProp.manual_grade_for_stlc_variation2.
idtac " ".

idtac "—————- stlc_variation3 —————".
idtac " ".

idtac "#> Manually graded: STLCProp.stlc_variation3".

```

```

idtac "Possible points: 2".
print_manual_grade STLCProp.manual_grade_for_stlc_variation3.
idtac " ".

idtac "————— STLCArith.subst —————".
idtac " ".

idtac "#> STLCArith.subst".
idtac "Possible points: 10".
check_type @STLCArith.subst (
  (∀ ( _ : String.string) ( _ : STLCArith.tm) ( _ : STLCArith.tm),
    STLCArith.tm)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCArith.subst.
Goal True.
idtac " ".

idtac "————— STLCArith.weakening —————".
idtac " ".

idtac "#> STLCArith.weakening".
idtac "Possible points: 6".
check_type @STLCArith.weakening (
  (∀ ( Gamma Gamma' : Maps.partial_map STLCArith.ty)
    ( t : STLCArith.tm) ( T : STLCArith.ty)
    ( _ : @Maps.includedin STLCArith.ty Gamma Gamma')
    ( _ : STLCArith.has_type Gamma t T),
    STLCArith.has_type Gamma' t T)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCArith.weakening.
Goal True.
idtac " ".

idtac "————— STLCArith.preservation —————".
idtac " ".

idtac "#> STLCArith.preservation".
idtac "Possible points: 6".
check_type @STLCArith.preservation (
  (∀ ( t t' : STLCArith.tm) ( T : STLCArith.ty)
    ( _ : STLCArith.has_type (@Maps.empty STLCArith.ty) t T)
    ( _ : STLCArith.step t t'),
    STLCArith.has_type (@Maps.empty STLCArith.ty) t' T)).
idtac "Assumptions:".
Abort.

```

```

Print Assumptions STLCArith.preservation.
Goal True.
idtac " ".
idtac "————- STLCArith.progress —————".
idtac " ".
idtac "#> STLCArith.progress".
idtac "Possible points: 6".
check_type @STLCArith.progress (
(∀ (t : STLCArith.tm) (T : STLCArith.ty)
  ( _ : STLCArith.has_type (@Maps.empty STLCArith.ty) t T),
  or (STLCArith.value t)
    (@ex STLCArith.tm (fun t' : STLCArith.tm ⇒ STLCArith.step t t')))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCArith.progress.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 41".
idtac "Max points - advanced: 47".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "————- subject_expansion_stlc ————".

```

```

idtac "MANUAL".
idtac "————- STLCProp.unique_types ————".
Print Assumptions STLCProp.unique_types.
idtac "————- STLCProp.free_in_context ————".
Print Assumptions STLCProp.free_in_context.
idtac "————- stlc_variation1 ————".
idtac "MANUAL".
idtac "————- stlc_variation2 ————".
idtac "MANUAL".
idtac "————- stlc_variation3 ————".
idtac "MANUAL".
idtac "————- STLCArith.subst ————".
Print Assumptions STLCArith.subst.
idtac "————- STLCArith.weakening ————".
Print Assumptions STLCArith.weakening.
idtac "————- STLCArith.preservation ————".
Print Assumptions STLCArith.preservation.
idtac "————- STLCArith.progress ————".
Print Assumptions STLCArith.progress.
idtac "".
idtac "***** Advanced *****".
idtac "————- STLCProp.progress' ————".
Print Assumptions STLCProp.progress'.
idtac "————- STLCProp.substitution_preserves_typing_from_typing_ind ————".
Print Assumptions STLCProp.substitution_preserves_typing_from_typing_ind.
Abort.

```

Chapter 36

Library PLF.MoreStlcTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import MoreStlc.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import MoreStlc.
Import CHECK.
Goal True.
idtac "————— STLCExtended.subst —————".
```

```

idtac " ".
idtac "#> STLCEExtended.subst".
idtac "Possible points: 3".
check_type @STLCEExtended.subst (
(∀ ( _ : String.string) ( _ : STLCEExtended.tm) ( _ : STLCEExtended.tm),
  STLCEExtended.tm)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCEExtended.subst.
Goal True.
idtac " ".
idtac "————— STLCEExtended.step —————".
idtac " ".
idtac "#> STLCEExtended.step".
idtac "Possible points: 3".
check_type @STLCEExtended.step (
(∀ ( _ : STLCEExtended.tm) ( _ : STLCEExtended.tm), Prop)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCEExtended.step.
Goal True.
idtac " ".
idtac "————— STLCEExtended.has_type —————".
idtac " ".
idtac "#> STLCEExtended.has_type".
idtac "Possible points: 3".
check_type @STLCEExtended.has_type (
(∀ ( _ : STLCEExtended.context) ( _ : STLCEExtended.tm)
  ( _ : STLCEExtended.ty),
  Prop)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCEExtended.has_type.
Goal True.
idtac " ".
idtac "————— STLCEExtended.progress —————".
idtac " ".
idtac "#> STLCEExtended.progress".
idtac "Possible points: 3".
check_type @STLCEExtended.progress (
(∀ ( t : STLCEExtended.tm) ( T : STLCEExtended.ty)

```



```

    ( _ : STLCEExtended.has_type (@Maps.empty STLCEExtended.ty) t T),
  or (STLCEExtended.value t)
  (@ex STLCEExtended.tm (fun t' : STLCEExtended.tm => STLCEExtended.step t
t')))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCEExtended.progress.
Goal True.
idtac " ".

idtac "—————- STLCEExtended.substitution_preserves_typing —————".
idtac " ".

idtac "#> STLCEExtended.substitution_preserves_typing".
idtac "Possible points: 2".
check_type @STLCEExtended.substitution_preserves_typing (
(∀ (Gamma : Maps.partial_map STLCEExtended.ty)
(x : String.string) (U : STLCEExtended.ty) (t v : STLCEExtended.tm)
(T : STLCEExtended.ty)
( _ : STLCEExtended.has_type (@Maps.update STLCEExtended.ty Gamma x U) t T)
( _ : STLCEExtended.has_type (@Maps.empty STLCEExtended.ty) v U),
STLCEExtended.has_type Gamma (STLCEExtended.subst x v t) T)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCEExtended.substitution_preserves_typing.
Goal True.
idtac " ".

idtac "—————- STLCEExtended.preservation —————".
idtac " ".

idtac "#> STLCEExtended.preservation".
idtac "Possible points: 3".
check_type @STLCEExtended.preservation (
(∀ (t t' : STLCEExtended.tm) (T : STLCEExtended.ty)
( _ : STLCEExtended.has_type (@Maps.empty STLCEExtended.ty) t T)
( _ : STLCEExtended.step t t'),
STLCEExtended.has_type (@Maps.empty STLCEExtended.ty) t' T)).
idtac "Assumptions:".
Abort.
Print Assumptions STLCEExtended.preservation.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 17".

```

```

idtac "Max points - advanced: 17".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- STLCExtended.subst -----".
Print Assumptions STLCExtended.subst.
idtac "----- STLCExtended.step -----".
Print Assumptions STLCExtended.step.
idtac "----- STLCExtended.has_type -----".
Print Assumptions STLCExtended.has_type.
idtac "----- STLCExtended.progress -----".
Print Assumptions STLCExtended.progress.
idtac "----- STLCExtended.substitution_preserves_typing -----".
Print Assumptions STLCExtended.substitution_preserves_typing.
idtac "----- STLCExtended.preservation -----".
Print Assumptions STLCExtended.preservation.
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 37

Library **PLF.SubTest**

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Sub.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | ""%string => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Sub.
Import CHECK.
Goal True.
idtac "————— arrow_sub_wrong —————".
```

```

idtac " ".
idtac "#> Manually graded: arrow_sub_wrong".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_arrow_sub_wrong.
idtac " ".
idtac "----- subtype_order -----".
idtac " ".
idtac "#> Manually graded: subtype_order".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_subtype_order.
idtac " ".
idtac "----- subtype_instances_tf_2 -----".
idtac " ".
idtac "#> Manually graded: subtype_instances_tf_2".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_subtype_instances_tf_2.
idtac " ".
idtac "----- subtype_concepts_tf -----".
idtac " ".
idtac "#> Manually graded: subtype_concepts_tf".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_subtype_concepts_tf.
idtac " ".
idtac "----- proper_subtypes -----".
idtac " ".
idtac "#> Manually graded: proper_subtypes".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_proper_subtypes.
idtac " ".
idtac "----- small_large_1 -----".
idtac " ".
idtac "#> Manually graded: small_large_1".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_small_large_1.
idtac " ".
idtac "----- small_large_2 -----".
idtac " ".
idtac "#> Manually graded: small_large_2".
idtac "Possible points: 2".

```

```

print_manual_grade manual_grade_for_small_large_2.
idtac " ".
idtac "----- small_large_4 -----".
idtac " ".
idtac "#> Manually graded: small_large_4".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_small_large_4.
idtac " ".
idtac "----- smallest_1 -----".
idtac " ".
idtac "#> Manually graded: smallest_1".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_smallest_1.
idtac " ".
idtac "----- smallest_2 -----".
idtac " ".
idtac "#> Manually graded: smallest_2".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_smallest_2.
idtac " ".
idtac "----- pair_permutation -----".
idtac " ".
idtac "#> Manually graded: pair_permutation".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_pair_permutation.
idtac " ".
idtac "----- sub_inversion_arrow -----".
idtac " ".
idtac "#> STLCSUB.sub_inversion_arrow".
idtac "Possible points: 3".
check_type @STLCSUB.sub_inversion_arrow (
(∀ (U V1 V2 : STLCSUB.ty)
  ( _ : STLCSUB.subtype U (STLCSUB.Ty_Arrow V1 V2)),
  @ex STLCSUB.ty
    (fun U1 : STLCSUB.ty ⇒
      @ex STLCSUB.ty
        (fun U2 : STLCSUB.ty ⇒
          and (@eq STLCSUB.ty U (STLCSUB.Ty_Arrow U1 U2))
            (and (STLCSUB.subtype V1 U1) (STLCSUB.subtype U2 V2)))))).
idtac "Assumptions:".

```

```

Abort.
Print Assumptions STLCSub.sub_inversion_arrow.
Goal True.
idtac " ".

idtac "————- variations —————".
idtac " ".

idtac "#> Manually graded: STLCSub.variations".
idtac "Possible points: 2".
print_manual_grade STLCSub.manual_grade_for_variations.
idtac " ".

idtac "————- products —————".
idtac " ".

idtac "#> Manually graded: STLCSub.products_value_step".
idtac "Possible points: 2".
print_manual_grade STLCSub.manual_grade_for_products_value_step.
idtac " ".

idtac "#> Manually graded: STLCSub.products_subtype_has_type".
idtac "Possible points: 2".
print_manual_grade STLCSub.manual_grade_for_products_subtype_has_type.
idtac " ".

idtac "#> Manually graded: STLCSub.products_progress".
idtac "Possible points: 3".
print_manual_grade STLCSub.manual_grade_for_products_progress.
idtac " ".

idtac "#> Manually graded: STLCSub.products_preservation".
idtac "Possible points: 3".
print_manual_grade STLCSub.manual_grade_for_products_preservation.
idtac " ".

idtac " ".

idtac "Max points - standard: 35".
idtac "Max points - advanced: 35".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".

```

```

idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- arrow_sub_wrong -----".
idtac "MANUAL".
idtac "----- subtype_order -----".
idtac "MANUAL".
idtac "----- subtype_instances_tf_2 -----".
idtac "MANUAL".
idtac "----- subtype_concepts_tf -----".
idtac "MANUAL".
idtac "----- proper_subtypes -----".
idtac "MANUAL".
idtac "----- small_large_1 -----".
idtac "MANUAL".
idtac "----- small_large_2 -----".
idtac "MANUAL".
idtac "----- small_large_4 -----".
idtac "MANUAL".
idtac "----- smallest_1 -----".
idtac "MANUAL".
idtac "----- smallest_2 -----".
idtac "MANUAL".
idtac "----- pair_permutation -----".
idtac "MANUAL".
idtac "----- STLCSub.sub_inversion_arrow -----".
Print Assumptions STLCSub.sub_inversion_arrow.
idtac "----- variations -----".
idtac "MANUAL".
idtac "----- products_value_step -----".
idtac "MANUAL".
idtac "----- products_subtype_has_type -----".
idtac "MANUAL".
idtac "----- products_progress -----".

```

```
idtac "MANUAL".
idtac "———- products_preservation ——".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
Abort.
```


Chapter 38

Library `PLF.TypecheckingTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Typechecking.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Typechecking.
Import CHECK.
Goal True.
idtac "————— type_check_defn —————".
```

```

idtac " ".
idtac "#> Manually graded: TypecheckerExtensions.type_check_defn".
idtac "Possible points: 6".
print_manual_grade TypecheckerExtensions.manual_grade_for_type_check_defn.
idtac " ".
idtac "----- ext_type_checking_sound -----".
idtac " ".
idtac "#> TypecheckerExtensions.type_checking_sound".
idtac "Possible points: 2".
check_type @TypecheckerExtensions.type_checking_sound (
  (∀ (Gamma : MoreStlc.STLCExtended.context)
    (t : MoreStlc.STLCExtended.tm) (T : MoreStlc.STLCExtended.ty)
    (_ : @eq (option MoreStlc.STLCExtended.ty)
      (TypecheckerExtensions.type_check Gamma t)
      (@Some MoreStlc.STLCExtended.ty T)),
    MoreStlc.STLCExtended.has_type Gamma t T)).
idtac "Assumptions:".
Abort.
Print Assumptions TypecheckerExtensions.type_checking_sound.
Goal True.
idtac " ".
idtac "----- ext_type_checking_complete -----".
idtac " ".
idtac "#> TypecheckerExtensions.type_checking_complete".
idtac "Possible points: 2".
check_type @TypecheckerExtensions.type_checking_complete (
  (∀ (Gamma : MoreStlc.STLCExtended.context)
    (t : MoreStlc.STLCExtended.tm) (T : MoreStlc.STLCExtended.ty)
    (_ : MoreStlc.STLCExtended.has_type Gamma t T),
    @eq (option MoreStlc.STLCExtended.ty)
      (TypecheckerExtensions.type_check Gamma t)
      (@Some MoreStlc.STLCExtended.ty T))).
idtac "Assumptions:".
Abort.
Print Assumptions TypecheckerExtensions.type_checking_complete.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 10".
idtac "Max points - advanced: 10".
idtac "".

```

```

idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- type_check_defn -----".
idtac "MANUAL".
idtac "----- TypecheckerExtensions.type_checking_sound -----".
Print Assumptions TypecheckerExtensions.type_checking_sound.
idtac "----- TypecheckerExtensions.type_checking_complete -----".
Print Assumptions TypecheckerExtensions.type_checking_complete.
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 39

Library `PLF.RecordsTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Records.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Records.
Import CHECK.
Goal True.
idtac "----- examples -----".
```

```

idtac " ".
idtac "#> STLCExtendedRecords.typing_example_2".
idtac "Possible points: 0.5".
check_type @STLCExtendedRecords.typing_example_2 (
(STLCExtendedRecords.has_type (@Maps.empty STLCExtendedRecords.ty)
  (STLCExtendedRecords.tm_app
    (STLCExtendedRecords.tm_abs
      (String.String
        (Ascii.Ascii true false false false true true false)
        String.EmptyString)
      (STLCExtendedRecords.Ty_RCons
        (String.String
          (Ascii.Ascii true false false true false true true false)
          (String.String
            (Ascii.Ascii true false false false true true false false)
            String.EmptyString))
        (STLCExtendedRecords.Ty_Arrow
          (STLCExtendedRecords.Ty_Base
            (String.String
              (Ascii.Ascii true false false false false false true
               false)
              String.EmptyString))
          (STLCExtendedRecords.Ty_Base
            (String.String
              (Ascii.Ascii true false false false false false true
               false)
              String.EmptyString))))
        (STLCExtendedRecords.Ty_RCons
          (String.String
            (Ascii.Ascii true false false true false true true false)
            (String.String
              (Ascii.Ascii false true false false true true false
               false)
              String.EmptyString))
          (STLCExtendedRecords.Ty_Arrow
            (STLCExtendedRecords.Ty_Base
              (String.String
                (Ascii.Ascii false true false false false false true
                 false)
                String.EmptyString))
            (STLCExtendedRecords.Ty_Base
              (String.String

```

```

        (Ascii.Ascii false true false false false false true
         false)
        String.EmptyString)))
    STLCExtendedRecords.Ty_RNil))
(STLCExtendedRecords.tm_rproj
 (STLCExtendedRecords.tm_var
  (String.String
   (Ascii.Ascii true false false false false true true false)
   String.EmptyString))
 (String.String
  (Ascii.Ascii true false false true false true true false)
  (String.String
   (Ascii.Ascii false true false false true true false false)
   String.EmptyString))))
(STLCExtendedRecords.tm_rcons
 (String.String
  (Ascii.Ascii true false false true false true true false)
  (String.String
   (Ascii.Ascii true false false false true true false false)
   String.EmptyString))
 (STLCExtendedRecords.tm_abs
  (String.String
   (Ascii.Ascii true false false false false true true false)
   String.EmptyString)
  (STLCExtendedRecords.Ty_Base
   (String.String
    (Ascii.Ascii true false false false false false true false)
    String.EmptyString))
  (STLCExtendedRecords.tm_var
   (String.String
    (Ascii.Ascii true false false false false true true false)
    String.EmptyString)))
(STLCExtendedRecords.tm_rcons
 (String.String
  (Ascii.Ascii true false false true false true true false)
  (String.String
   (Ascii.Ascii false true false false true true false false)
   String.EmptyString))
 (STLCExtendedRecords.tm_abs
  (String.String
   (Ascii.Ascii true false false false false true true false)
   String.EmptyString)

```

```

      (STLCExtendedRecords.Ty_Base
        (String.String
          (Ascii.Ascii false true false false false false true
            false)
          String.EmptyString)))
    (STLCExtendedRecords.tm_var
      (String.String
        (Ascii.Ascii true false false false false true true
          false)
        String.EmptyString)))
    STLCEExtendedRecords.tm_rnil)))
(STLCExtendedRecords.Ty_Arrow
  (STLCExtendedRecords.Ty_Base
    (String.String
      (Ascii.Ascii false true false false false false true false)
      String.EmptyString)))
(STLCExtendedRecords.Ty_Base
  (String.String
    (Ascii.Ascii false true false false false false true false)
    String.EmptyString))))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCExtendedRecords.typing_example_2.
Goal True.
idtac " ".
idtac "#> STLCEExtendedRecords.typing_nonexample".
idtac "Possible points: 0.5".
check_type @STLCExtendedRecords.typing_nonexample (
(not
  (@ex STLCExtendedRecords.ty
    (fun T : STLCExtendedRecords.ty ⇒
      STLCExtendedRecords.has_type
        (@Maps.update STLCExtendedRecords.ty
          (@Maps.empty STLCExtendedRecords.ty)
          (String.String
            (Ascii.Ascii true false false false false true true false)
            String.EmptyString)
          (STLCExtendedRecords.Ty_RCons
            (String.String
              (Ascii.Ascii true false false true false true true false)
              (String.String
                (Ascii.Ascii false true false false true true false)

```

```

        false)
      String.EmptyString))
    (STLCExtendedRecords.Ty_Arrow
      (STLCExtendedRecords.Ty_Base
        (String.String
          (Ascii.Ascii true false false false false false true
            false)
          String.EmptyString))
      (STLCExtendedRecords.Ty_Base
        (String.String
          (Ascii.Ascii true false false false false false true
            false)
          String.EmptyString))))
    STLCEExtendedRecords.Ty_RNil))
  (STLCExtendedRecords.tm_rcons
    (String.String
      (Ascii.Ascii true false false true false true true false)
      (String.String
        (Ascii.Ascii true false false false true true false false)
        String.EmptyString))
    (STLCExtendedRecords.tm_abs
      (String.String
        (Ascii.Ascii true false false false false true true false)
        String.EmptyString)
      (STLCExtendedRecords.Ty_Base
        (String.String
          (Ascii.Ascii false true false false false false true
            false)
          String.EmptyString))
      (STLCExtendedRecords.tm_var
        (String.String
          (Ascii.Ascii true false false false false true true
            false)
          String.EmptyString))))
    (STLCExtendedRecords.tm_var
      (String.String
        (Ascii.Ascii true false false false false true true false)
        String.EmptyString))))
  T))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCExtendedRecords.typing_nonexample.

```



```

Goal True.
idtac " ".

idtac "#> STLCExtendedRecords.typing_nonexample_2".
idtac "Possible points: 1".
check_type @STLCExtendedRecords.typing_nonexample_2 (
(∀ y : String.string,
not
  (@ex STLCExtendedRecords.ty
    (fun T : STLCExtendedRecords.ty ⇒
      STLCExtendedRecords.has_type
        (@Maps.update STLCExtendedRecords.ty
          (@Maps.empty STLCExtendedRecords.ty) y
          (STLCExtendedRecords.Ty_Base
            (String.String
              (Ascii.Ascii true false false false false true false)
              String.EmptyString))))
      (STLCExtendedRecords.tm_app
        (STLCExtendedRecords.tm_abs
          (String.String
            (Ascii.Ascii true false false false true true false)
            String.EmptyString)
          (STLCExtendedRecords.Ty_RCons
            (String.String
              (Ascii.Ascii true false false true false true true false)
              (String.String
                (Ascii.Ascii true false false false true true false
                  false)
                String.EmptyString))
            (STLCExtendedRecords.Ty_Base
              (String.String
                (Ascii.Ascii true false false false false false true
                  false)
                String.EmptyString))
            STLCExtendedRecords.Ty_RNil)
          (STLCExtendedRecords.tm_rproj
            (STLCExtendedRecords.tm_var
              (String.String
                (Ascii.Ascii true false false false false true true
                  false)
                String.EmptyString))
            (String.String
              (Ascii.Ascii true false false true false true true false)

```

```

        (String.String
          (Ascii.Ascii true false false false true true false
            false)
          String.EmptyString))))
(STLCExtendedRecords.tm_rcons
  (String.String
    (Ascii.Ascii true false false true false true true false)
    (String.String
      (Ascii.Ascii true false false false true true false
        false)
      String.EmptyString))
(STLCExtendedRecords.tm_var y)
(STLCExtendedRecords.tm_rcons
  (String.String
    (Ascii.Ascii true false false true false true true false)
    (String.String
      (Ascii.Ascii false true false false true true false
        false)
      String.EmptyString))
(STLCExtendedRecords.tm_var y) STLCExtendedRecords.tm_rnil)))
T))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCExtendedRecords.typing_nonexample_2.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 2".
idtac "Max points - advanced: 2".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".

```

```

idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- STLCExtendedRecords.typing_example_2 -----".
Print Assumptions STLCExtendedRecords.typing_example_2.
idtac "----- STLCExtendedRecords.typing_nonexample -----".
Print Assumptions STLCExtendedRecords.typing_nonexample.
idtac "----- STLCExtendedRecords.typing_nonexample_2 -----".
Print Assumptions STLCExtendedRecords.typing_nonexample_2.
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 40

Library `PLF.ReferencesTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import References.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import References.
Import CHECK.
Goal True.
idtac "———— compact_update ————".
```

```

idtac " ".
idtac "#> Manually graded: STLCRef.compact_update".
idtac "Possible points: 2".
print_manual_grade STLCRef.manual_grade_for_compact_update.
idtac " ".
idtac "—————- type_safety_violation —————".
idtac " ".
idtac "#> Manually graded: STLCRef.type_safety_violation".
idtac "Possible points: 2".
print_manual_grade STLCRef.manual_grade_for_type_safety_violation.
idtac " ".
idtac "—————- cyclic_store —————".
idtac " ".
idtac "#> STLCRef.cyclic_store".
idtac "Possible points: 3".
check_type @STLCRef.cyclic_store (
(@ex STLCRef.tm
  (fun t : STLCRef.tm ⇒
    STLCRef.multistep
      (@pair STLCRef.tm (list STLCRef.tm) t (@nil STLCRef.tm))
      (@pair STLCRef.tm (list STLCRef.tm) STLCRef.tm_unit
        (@cons STLCRef.tm
          (STLCRef.tm_abs STLCRef.x STLCRef.Ty_Nat
            (STLCRef.tm_app (STLCRef.tm_deref (STLCRef.tm_loc 1))
              (STLCRef.tm_var STLCRef.x)))
          (@cons STLCRef.tm
            (STLCRef.tm_abs STLCRef.x STLCRef.Ty_Nat
              (STLCRef.tm_app (STLCRef.tm_deref (STLCRef.tm_loc 0))
                (STLCRef.tm_var STLCRef.x)))
            (@nil STLCRef.tm)))))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCRef.cyclic_store.
Goal True.
idtac " ".
idtac "—————- store_not_unique —————".
idtac " ".
idtac "#> STLCRef.store_not_unique".
idtac "Possible points: 3".
check_type @STLCRef.store_not_unique (
(@ex STLCRef.store

```

```

(fun st : STLCRef.store ⇒
  @ex STLCRef.store_ty
  (fun ST1 : STLCRef.store_ty ⇒
    @ex STLCRef.store_ty
    (fun ST2 : STLCRef.store_ty ⇒
      and (STLCRef.store_well_typed ST1 st)
      (and (STLCRef.store_well_typed ST2 st)
        (not (@eq STLCRef.store_ty ST1 ST2))))))))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCRef.store_not_unique.
Goal True.
idtac " ".

idtac "———— factorial_ref ————".
idtac " ".

idtac "#> STLCRef.RefsAndNontermination.factorial".
idtac "Possible points: 3".
check_type @STLCRef.RefsAndNontermination.factorial (STLCRef.tm).
idtac "Assumptions:".
Abort.
Print Assumptions STLCRef.RefsAndNontermination.factorial.
Goal True.
idtac " ".

idtac "#> STLCRef.RefsAndNontermination.factorial_type".
idtac "Possible points: 3".
check_type @STLCRef.RefsAndNontermination.factorial_type (
(STLCRef.has_type (@nil STLCRef.ty) (@Maps.empty STLCRef.ty)
  STLCRef.RefsAndNontermination.factorial
  (STLCRef.Ty_Arrow STLCRef.Ty_Nat STLCRef.Ty_Nat))).
idtac "Assumptions:".
Abort.
Print Assumptions STLCRef.RefsAndNontermination.factorial_type.
Goal True.
idtac " ".
idtac " ".

idtac "Max points - standard: 16".
idtac "Max points - advanced: 16".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".

```

```

idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- compact_update -----".
idtac "MANUAL".
idtac "----- type_safety_violation -----".
idtac "MANUAL".
idtac "----- STLCRef.cyclic_store -----".
Print Assumptions STLCRef.cyclic_store.
idtac "----- STLCRef.store_not_unique -----".
Print Assumptions STLCRef.store_not_unique.
idtac "----- STLCRef.RefsAndNontermination.factorial -----".
Print Assumptions STLCRef.RefsAndNontermination.factorial.
idtac "----- STLCRef.RefsAndNontermination.factorial_type -----".
Print Assumptions STLCRef.RefsAndNontermination.factorial_type.
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 41

Library PLF.RecordSubTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import RecordSub.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import RecordSub.
Import CHECK.
Goal True.
idtac "————- subtyping_example_1 —————".
```



```

idtac " ".
idtac "#> RecordSub.Examples.subtyping_example_1".
idtac "Possible points: 2".
check_type @RecordSub.Examples.subtyping_example_1 (
(RecordSub.subtype RecordSub.Examples.TRcd_kj RecordSub.Examples.TRcd_j)).
idtac "Assumptions:".
Abort.
Print Assumptions RecordSub.Examples.subtyping_example_1.
Goal True.
idtac " ".

idtac "----- subtyping_example_2 -----".
idtac " ".

idtac "#> RecordSub.Examples.subtyping_example_2".
idtac "Possible points: 1".
check_type @RecordSub.Examples.subtyping_example_2 (
(RecordSub.subtype
  (RecordSub.Ty_Arrow RecordSub.Ty_Top RecordSub.Examples.TRcd_kj)
  (RecordSub.Ty_Arrow
    (RecordSub.Ty_Arrow
      (RecordSub.Ty_Base
        (String.String
          (Ascii.Ascii true true false false false false true false)
          String.EmptyString))
      (RecordSub.Ty_Base
        (String.String
          (Ascii.Ascii true true false false false false true false)
          String.EmptyString))))
    RecordSub.Examples.TRcd_j))).
idtac "Assumptions:".
Abort.
Print Assumptions RecordSub.Examples.subtyping_example_2.
Goal True.
idtac " ".

idtac "----- subtyping_example_3 -----".
idtac " ".

idtac "#> RecordSub.Examples.subtyping_example_3".
idtac "Possible points: 1".
check_type @RecordSub.Examples.subtyping_example_3 (
(RecordSub.subtype
  (RecordSub.Ty_Arrow RecordSub.Ty_RNil
    (RecordSub.Ty_RCons

```

```

      (String.String
        (Ascii.Ascii false true false true false true true false)
        String.EmptyString)
      (RecordSub.Ty_Base
        (String.String
          (Ascii.Ascii true false false false false false true false)
          String.EmptyString))
      RecordSub.Ty_RNil))
(RecordSub.Ty_Arrow
  (RecordSub.Ty_RCons
    (String.String
      (Ascii.Ascii true true false true false true true false)
      String.EmptyString)
    (RecordSub.Ty_Base
      (String.String
        (Ascii.Ascii false true false false false false true false)
        String.EmptyString))
      RecordSub.Ty_RNil)
    RecordSub.Ty_RNil))).
idtac "Assumptions:".
Abort.
Print Assumptions RecordSub.Examples.subtyping_example_3.
Goal True.
idtac " ".
idtac "————- subtyping_example_4 —————".
idtac " ".
idtac "#> RecordSub.Examples.subtyping_example_4".
idtac "Possible points: 2".
check_type @RecordSub.Examples.subtyping_example_4 (
(RecordSub.subtype
  (RecordSub.Ty_RCons
    (String.String
      (Ascii.Ascii false false false true true true true false)
      String.EmptyString)
    (RecordSub.Ty_Base
      (String.String
        (Ascii.Ascii true false false false false false true false)
        String.EmptyString))
    (RecordSub.Ty_RCons
      (String.String
        (Ascii.Ascii true false false true true true true false)
        String.EmptyString)

```

```

    (RecordSub.Ty_Base
      (String.String
        (Ascii.Ascii false true false false false true false)
        String.EmptyString))
  (RecordSub.Ty_RCons
    (String.String
      (Ascii.Ascii false true false true true true true false)
      String.EmptyString)
    (RecordSub.Ty_Base
      (String.String
        (Ascii.Ascii true true false false false false true false)
        String.EmptyString))
    RecordSub.Ty_RNil)))
(RecordSub.Ty_RCons
  (String.String (Ascii.Ascii false true false true true true true false)
    String.EmptyString)
  (RecordSub.Ty_Base
    (String.String
      (Ascii.Ascii true true false false false false true false)
      String.EmptyString))
  (RecordSub.Ty_RCons
    (String.String
      (Ascii.Ascii true false false true true true true false)
      String.EmptyString)
    (RecordSub.Ty_Base
      (String.String
        (Ascii.Ascii false true false false false false true false)
        String.EmptyString))
    (RecordSub.Ty_RCons
      (String.String
        (Ascii.Ascii false false false true true true true false)
        String.EmptyString)
      (RecordSub.Ty_Base
        (String.String
          (Ascii.Ascii true false false false false false true false)
          String.EmptyString))
      RecordSub.Ty_RNil))))).
idtac "Assumptions:".
Abort.
Print Assumptions RecordSub.Examples.subtyping_example_4.
Goal True.
idtac " ".

```

```

idtac "————- rcd_types_match_informal —————".
idtac " ".
idtac "#> Manually graded: RecordSub.rcd_types_match_informal".
idtac "Possible points: 3".
print_manual_grade RecordSub.manual_grade_for_rcd_types_match_informal.
idtac " ".
idtac "————- typing_example_0 —————".
idtac " ".
idtac "#> RecordSub.Examples2.typing_example_0".
idtac "Possible points: 1".
check_type @RecordSub.Examples2.typing_example_0 (
  (RecordSub.has_type (@Maps.empty RecordSub.ty) RecordSub.Examples2.trcd_kj
    RecordSub.Examples.TRcd_kj)).
idtac "Assumptions:".
Abort.
Print Assumptions RecordSub.Examples2.typing_example_0.
Goal True.
idtac " ".
idtac "————- typing_example_1 —————".
idtac " ".
idtac "#> RecordSub.Examples2.typing_example_1".
idtac "Possible points: 2".
check_type @RecordSub.Examples2.typing_example_1 (
  (RecordSub.has_type (@Maps.empty RecordSub.ty)
    (RecordSub.tm_app
      (RecordSub.tm_abs
        (String.String
          (Ascii.Ascii false false false true true true true false)
          String.EmptyString)
        RecordSub.Examples.TRcd_j
        (RecordSub.tm_rproj
          (RecordSub.tm_var
            (String.String
              (Ascii.Ascii false false false true true true true false)
              String.EmptyString))
            (String.String
              (Ascii.Ascii false true false true false true true false)
              String.EmptyString))))
        RecordSub.Examples2.trcd_kj)
    (RecordSub.Ty_Arrow
      (RecordSub.Ty_Base

```

```

      (String.String
        (Ascii.Ascii false true false false false true false)
        String.EmptyString))
    (RecordSub.Ty_Base
      (String.String
        (Ascii.Ascii false true false false false true false)
        String.EmptyString))))).
idtac "Assumptions:".
Abort.
Print Assumptions RecordSub.Examples2.typing_example_1.
Goal True.
idtac " ".
idtac "----- canonical_forms_of_arrow_types -----".
idtac " ".
idtac "#> RecordSub.canonical_forms_of_arrow_types".
idtac "Possible points: 3".
check_type @RecordSub.canonical_forms_of_arrow_types (
  (∀ (Gamma : RecordSub.context) (s : RecordSub.tm)
    (T1 T2 : RecordSub.ty)
    (_ : RecordSub.has_type Gamma s (RecordSub.Ty_Arrow T1 T2))
    (_ : RecordSub.value s),
    @ex String.string
      (fun x : String.string ⇒
        @ex RecordSub.ty
          (fun S1 : RecordSub.ty ⇒
            @ex RecordSub.tm
              (fun s2 : RecordSub.tm ⇒
                @eq RecordSub.tm s (RecordSub.tm_abs x S1 s2)))))).
idtac "Assumptions:".
Abort.
Print Assumptions RecordSub.canonical_forms_of_arrow_types.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 15".
idtac "Max points - advanced: 15".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".

```

```

idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- RecordSub.Examples.subtyping_example_1 -----".
Print Assumptions RecordSub.Examples.subtyping_example_1.
idtac "----- RecordSub.Examples.subtyping_example_2 -----".
Print Assumptions RecordSub.Examples.subtyping_example_2.
idtac "----- RecordSub.Examples.subtyping_example_3 -----".
Print Assumptions RecordSub.Examples.subtyping_example_3.
idtac "----- RecordSub.Examples.subtyping_example_4 -----".
Print Assumptions RecordSub.Examples.subtyping_example_4.
idtac "----- rcd_types_match_informal -----".
idtac "MANUAL".
idtac "----- RecordSub.Examples2.typing_example_0 -----".
Print Assumptions RecordSub.Examples2.typing_example_0.
idtac "----- RecordSub.Examples2.typing_example_1 -----".
Print Assumptions RecordSub.Examples2.typing_example_1.
idtac "----- RecordSub.canonical_forms_of_arrow_types -----".
Print Assumptions RecordSub.canonical_forms_of_arrow_types.
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 42

Library PLF.NormTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Norm.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | - => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Norm.
Import CHECK.
Goal True.
idtac "————- norm_fail —————".
```

```

idtac " ".
idtac "#> Manually graded: norm_fail".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_norm_fail.
idtac " ".
idtac "—————- norm —————".
idtac " ".
idtac "#> Manually graded: norm".
idtac "Possible points: 10".
print_manual_grade manual_grade_for_norm.
idtac " ".
idtac " ".
idtac "Max points - standard: 12".
idtac "Max points - advanced: 12".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "————- norm_fail ————".
idtac "MANUAL".
idtac "————- norm ————".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
Abort.

```


Chapter 43

Library PLF.PETest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import PE.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import PE.
Import CHECK.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 44

Library PLF.PostscriptTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Postscript.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | - => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Postscript.
Import CHECK.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 45

Library PLF.BibTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import Bib.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import Bib.
Import CHECK.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 46

Library PLF.LibTacticsTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import LibTactics.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import LibTactics.
Import CHECK.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```


Chapter 47

Library `PLF.UseTacticsTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import UseTactics.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import UseTactics.
Import CHECK.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

Chapter 48

Library `PLF.UseAutoTest`

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From PLF Require Import UseAuto.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (_ ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From PLF Require Import UseAuto.
Import CHECK.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "CSeq_congruence".
idtac "fold_constants_bexp_sound".
idtac "succ_hastype_nat__hastype_nat".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```