Contents

Chapter 1

Library Top.Symbols

1.1 Symbols: Special symbols

$Date: 2016-05-2616:17:19-0400(Thu, 26May2016)$

Chapter 2

Library Top.Preface

2.1 Preface

2.2 Welcome

This electronic book is a course on *Software Foundations*, the mathematical underpinnings of reliable software. Topics include basic concepts of logic, computer-assisted theorem proving, the Coq proof assistant, functional programming, 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.

The principal novelty of the course is that it is one hundred percent formalized and machine-checked: the entire text is literally a script for Coq. It is intended to be read alongside (or inside) an interactive session with Coq. All the details in the text are fully formalized in Coq, and most of the exercises are designed to be worked using Coq.

The files are organized into a sequence of core chapters, covering about one 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.

2.3 Overview

Building reliable software is hard. The scale and complexity of modern systems, the number of people involved in building them, and the range of demands placed on them make it extremely difficult to build software that is even more-or-less correct, much less 100% correct. At the same time, the increasing degree to which information processing is woven into every aspect of society greatly amplifies the cost of bugs and insecurities.

Computer scientists and software engineers have responded to these challenges by developing a whole host of techniques for improving software reliability, ranging from recommendations about managing software projects teams (e.g., extreme programming) to design philosophies for libraries (e.g., model-view-controller, publish-subscribe, etc.) and programming languages (e.g., object-oriented programming, aspect-oriented programming, functional programming, ...) to mathematical techniques for specifying and reasoning about properties of software and tools for helping validate these properties. The present course is focused on this last set of techniques.

The text weaves together five conceptual threads:

(1) basic tools from *logic* for making and justifying precise claims about programs;

(2) the use of *proof assistants* to construct rigorous logical arguments;

(3) *functional programming*, both as a method of programming that simplifies reasoning about programs and as a bridge between programming and logic;

(4) 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

(5) 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 all of them 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*.

2.3.1 Logic

Logic is the field of study whose subject matter is *proofs* – unassailable arguments for the truth of particular propositions. Volumes have been written about the central role of logic in computer science. Manna and Waldinger called it "the calculus of computer science," while Halpern et al.'s paper *On the Unusual Effectiveness of Logic in Computer Science* catalogs scores of ways in which logic offers critical tools and insights. Indeed, they observe that, "As a matter of fact, logic has turned out to be significiantly more effective in computer science than it has been in mathematics. This is quite remarkable, especially since much of the impetus for the development of logic during the past one hundred years came from mathematics."

In particular, the fundamental tools of *inductive proof* are ubiquitous in all of computer science. You have surely seen them before, perhaps in a course on discrete math or analysis of algorithms, but in this course we will examine them much more deeply than you have probably done so far.

2.3.2 Proof Assistants

The flow of ideas between logic and computer science has not been unidirectional: CS has also made important contributions to logic. One of these has been the development of software tools for helping construct proofs of logical propositions. These tools fall into two broad categories:

- *Automated theorem provers* provide "push-button" operation: you give them a proposition and they return either *true* or *false* (or, sometimes, *don't know: ran out of time*). Although their capabilities are still limited to specific domains, they have matured tremendously in recent years and are used now in a multitude of settings. Examples of such tools include SAT solvers, SMT solvers, and model checkers.

- *Proof assistants* are hybrid tools that automate the more routine aspects of building proofs while depending on human guidance for more difficult aspects. Widely used proof assistants include Isabelle, Agda, Twelf, ACL2, PVS, and Coq, among many others.

This course is based around Coq, a proof assistant that has been under development since 1983 and that in recent years has attracted a large community of users in both research and industry. Coq provides a rich environment for interactive development of machine-checked formal reasoning. The kernel of the Coq system is a simple proof-checker, which guarantees that only correct deduction steps are ever performed. On top of this kernel, the Coq environment provides high-level facilities for proof development, including a large library of common definitions and lemmas, powerful tactics for constructing complex proofs semi-automatically, and a special-purpose programming language for defining new proof-automation tactics for specific situations.

Coq has been a critical enabler for a huge variety of work across computer science and mathematics:

- As a *platform for modeling programming languages*, it has become a standard tool for researchers who need to describe and reason about complex language definitions. It has been used, for example, to check the security of the JavaCard platform, obtaining the highest level of common criteria certification, and for formal specifications of the x86 and LLVM instruction sets and programming languages such as C.

- As an *environment for developing formally certified software and hardware*, Coq has been used, for example, to build CompCert, a fully-verified optimizing compiler for C, and CertiKos, a fully verified hypervisor, for proving the correctness of subtle algorithms involving floating point numbers, and as the basis for CertiCrypt, an environment for reasoning about the security of cryptographic algorithms. It is also being used to build verified implementations of the open-source RISC-V processor.

- As a *realistic environment for functional programming with dependent types*, it has inspired numerous innovations. For example, the Ynot system embeds "relational Hoare reasoning" (an extension of the *Hoare Logic* we will see later in this course) in Coq.

- As a *proof assistant for higher-order logic*, it has been used to validate a number of important results in mathematics. For example, its ability to include complex computations inside proofs made it possible to develop the first formally verified proof of the 4-color theorem. This proof had previously been controversial among mathematicians because part of it included checking a large number of configurations using a program. In the Coq formalization, everything is checked, including the correctness of the computational part. More recently, an even more massive effort led to a Coq formalization of the Feit-Thompson Theorem – the first major step in the classification of finite simple groups.

By the way, in case you're wondering about the name, here's what the official Coq web site at INRIA (the French national research lab where Coq has mostly been developed) says about it: "Some French computer scientists have a tradition of naming their software as animal species: Caml, Elan, Foc or Phox are examples of this tacit convention. In French, 'coq' means rooster, and it sounds like the initials of the Calculus of Constructions (CoC) on which it is based." The rooster is also the national symbol of France, and C-o-q are the first three letters of the name of Thierry Coquand, one of Coq's early developers.

2.3.3 Functional Programming

The term *functional programming* refers both to a collection of programming idioms that can be used in almost any programming language and to a family of programming languages designed to emphasize these idioms, including Haskell, OCaml, Standard ML, F#, Scala, Scheme, Racket, Common Lisp, Clojure, Erlang, and Coq.

Functional programming has been developed over many decades – indeed, its roots go back to Church's lambda-calculus, which was invented in the 1930s, well before the first computers (at least the first electronic ones)! But since the early '90s it has enjoyed a surge of interest among industrial engineers and language designers, playing a key role in high-value systems at companies like Jane St. Capital, Microsoft, Facebook, and Ericsson.

The most basic tenet of functional programming is that, as much as possible, computation should be *pure*, in the sense that the only effect of execution should be to produce a result: it should be free from *side effects* such as I/O, assignments to mutable variables, redirecting pointers, etc. For example, whereas an *imperative* sorting function might take a list of numbers and rearrange its pointers to put the list in order, a pure sorting function would take the original list and return a *new* list containing the same numbers in sorted order.

A significant benefit of this style of programming is that it makes programs easier to understand and reason about. If every operation on a data structure yields a new data structure, leaving the old one intact, then there is no need to worry about how that structure is being shared and whether a change by one part of the program might break an invariant that another part of the program relies on. These considerations are particularly critical in concurrent systems, where every piece of mutable state that is shared between threads is a potential source of pernicious bugs. Indeed, a large part of the recent interest in functional programming in industry is due to its simpler behavior in the presence of concurrency.

Another reason for the current excitement about functional programming is related to the first: functional programs are often much easier to parallelize than their imperative counterparts. If running a computation has no effect other than producing a result, then it does not matter *where* it is run. Similarly, if a data structure is never modified destructively, then it can be copied freely, across cores or across the network. Indeed, the "Map-Reduce" idiom, which lies at the heart of massively distributed query processors like Hadoop and is used by Google to index the entire web is a classic example of functional programming.

For purposes of this course, functional programming has yet another significant attraction: it serves as a bridge between logic and computer science. Indeed, Coq itself can be viewed as a combination of a small but extremely expressive functional programming language plus a set of tools for stating and proving logical assertions. Moreover, when we come to look more closely, we find that these two sides of Coq are actually aspects of the very same underlying machinery – i.e., *proofs are programs*.

2.3.4 Program Verification

Approximately the first third of *Software Foundations* is devoted to developing the conceptual framework of logic and functional programming and gaining enough fluency with Coq to use it for modeling and reasoning about nontrivial artifacts. In the middle third, we turn our attention to 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.

2.3.5 Type Systems

Our final major topic, covering approximately the last third of the course, is *type systems*, which are 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 Coq itself!

2.3.6 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.

2.4 Practicalities

2.4.1 Chapter Dependencies

A diagram of the dependencies between chapters and some paths through the material can be found in the file *deps.html*.

2.4.2 System Requirements

Coq runs on Windows, Linux, and OS X. You will need:

- A current installation of Coq, available from the Coq home page. Everything should work with version 8.4 (or 8.5).

- An IDE for interacting with Coq. Currently, there are two choices:

- Proof General is an Emacs-based IDE. It tends to be preferred by users who are already comfortable with Emacs. It requires a separate installation (google "Proof General").

  Adventurous users of Coq within Emacs may also want to check out extensions such as *company-coq* and *control-lock*.

- CoqIDE is a simpler stand-alone IDE. It is distributed with Coq, so it should be available once you have Coq installed. It can also be compiled from scratch, but on some platforms this may involve installing additional packages for GUI libraries and such.

2.4.3 Exercises

Each chapter includes numerous exercises. Each is marked with a "star rating," which can be interpreted as follows:

- One star: easy exercises that underscore points in the text and that, for most readers, should take only a minute or two. Get in the habit of working these as you reach them.
- Two stars: straightforward exercises (five or ten minutes).
- Three stars: exercises requiring a bit of thought (ten minutes to half an hour).
- Four and five stars: more difficult exercises (half an hour and up).

Also, some exercises are marked "advanced," and some are marked "optional." Doing just the non-optional, non-advanced exercises should provide good coverage of the core material. Optional exercises provide a bit of extra practice with key concepts and introduce secondary themes that may be of interest to some readers. Advanced exercises are for readers who want an extra challenge and a deeper cut at the material.

Please do not post solutions to the exercises in a public places: Software Foundations is widely used both for self-study and for university courses. Having solutions easily available makes it much less useful for courses, which typically have graded homework assignments. We especially request that readers not post solutions to the exercises anyplace where they can be found by search engines.

2.4.4 Downloading the Coq Files

A tar file containing the full sources for the "release version" of this book (as a collection of Coq scripts and HTML files) is available here:

http://www.cis.upenn.edu/~bcpierce/sf

(If you are using the book as part of a class, your professor may give you access to a locally modified version of the files, which you should use instead of the release version.)

2.5 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!

In order to keep the legalities simple and to have a single point of responsibility in case the need should ever arise to adjust the license terms, sublicense, etc., we ask all contributors (i.e., everyone with access to the developers' repository) to assign copyright in their contributions to the appropriate "author of record," as follows:

- I hereby assign copyright in my past and future contributions to the Software Foundations project to the Author of Record of each volume or component, to be licensed under the same terms as the rest of Software Foundations. I understand that, at present, the Authors of Record are as follows: For Volumes 1 and 2, known until 2016 as "Software Foundations" and from 2016 as (respectively) "Logical Foundations" and "Programming Foundations," the Author of Record is Benjamin Pierce. For Volume 3, "Verified Functional Algorithms", the Author of Record is Andrew W. Appel. For components outside of designated Volumes (e.g., typesetting and grading tools and other software infrastructure), the Author of Record is Benjamin Pierce.

To get started, please send an email to Benjamin Pierce, describing yourself and how you plan to use the materials and including (1) the above copyright transfer text and (2) the result of doing "htpasswd -s -n NAME" where NAME is your preferred user name.

We'll set you up with access to the subversion repository and developers' mailing lists. In the repository you'll find a file *INSTRUCTORS* with further instructions.

2.6 Translations

Thanks to the efforts of a team of volunteer translators, *Software Foundations* can be enjoyed in Japanese at *http://proofcafe.org/sf*. A Chinese translation is underway.

$Date : 2016-08-2916 : 12 : 34-0400(Mon, 29Aug2016)$

Chapter 3

Library Top.Basics

3.1 Basics: Functional Programming in Coq

3.2 Introduction

The functional programming style is founded on simple, everyday mathematical intuition: If a procedure or method has no side effects, then (ignoring efficiency) all we need to understand about it is how it maps inputs to outputs – that is, we can think of it as just a concrete method for computing a mathematical function. This is one sense of the word "functional" in "functional programming." The direct connection between programs and simple mathematical objects supports both formal correctness proofs and sound informal reasoning about program behavior.

The other sense in which functional programming is "functional" is that it emphasizes the use of functions (or methods) as *first-class* values – i.e., values that can be passed as arguments to other functions, returned as results, included in data structures, etc. The recognition that functions can be treated as data gives rise to a host of useful and powerful programming idioms.

Other common features of functional languages include *algebraic data types* and *pattern matching*, which make it easy to construct and manipulate rich data structures, and sophisticated *polymorphic type systems* supporting abstraction and code reuse. Coq offers all of these features.

The first half of this chapter introduces the most essential elements of Coq's functional programming language, called *Gallina*. The second half introduces some basic *tactics* that can be used to prove properties of Coq programs.

3.3 Enumerated Types

One notable aspect of Coq is that its set of built-in features is *extremely* small. For example, instead of providing the usual palette of atomic data types (booleans, integers, strings, etc.), Coq offers a powerful mechanism for defining new data types from scratch, with all these familiar types as instances.

Naturally, the Coq distribution comes preloaded with an extensive standard library providing definitions of booleans, numbers, and many common data structures like lists and hash tables. But there is nothing magic or primitive about these library definitions. To illustrate this, we will explicitly recapitulate all the definitions we need in this course, rather than just getting them implicitly from the library.

3.3.1 Days of the Week

To see how this definition mechanism works, let's start with a very simple example. The following declaration tells Coq that we are defining a new set of data values – a *type*.

```
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

The type is called **day**, and its members are monday, tuesday, etc. The second and following lines of the definition can be read "monday is a **day**, tuesday is a **day**, etc."

Having defined **day**, we can write functions that operate on days.

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday ⇒ tuesday
  | tuesday ⇒ wednesday
  | wednesday ⇒ thursday
  | thursday ⇒ friday
  | friday ⇒ monday
  | saturday ⇒ monday
  | sunday ⇒ monday
  end.
```

One thing to note is that the argument and return types of this function are explicitly declared. Like most functional programming languages, Coq can often figure out these types for itself when they are not given explicitly – i.e., it can do *type inference* – but we'll generally include them to make reading easier.

Having defined a function, we should check that it works on some examples. There are actually three different ways to do this in Coq. First, we can use the command Compute to evaluate a compound expression involving next_weekday.

```
Compute (next_weekday friday).
```

```
Compute (next_weekday (next_weekday saturday)).
```

(We show Coq's responses in comments, but, if you have a computer handy, this would be an excellent moment to fire up the Coq interpreter under your favorite IDE – either CoqIde or Proof General – and try this for yourself. Load this file, *Basics.v*, from the book's Coq sources, find the above example, submit it to Coq, and observe the result.)

Second, we can record what we *expect* the result to be in the form of a Coq example:

```
Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.
```

This declaration does two things: it makes an assertion (that the second weekday after `saturday` is `tuesday`), and it gives the assertion a name that can be used to refer to it later. Having made the assertion, we can also ask Coq to verify it, like this:

```
Proof. simpl. reflexivity. Qed.
```

The details are not important for now (we'll come back to them in a bit), but essentially this can be read as "The assertion we've just made can be proved by observing that both sides of the equality evaluate to the same thing, after some simplification."

Third, we can ask Coq to *extract*, from our `Definition`, a program in some other, more conventional, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler. This facility is very interesting, since it gives us a way to go from proved-correct algorithms written in Gallina to efficient machine code. (Of course, we are trusting the correctness of the OCaml/Haskell/Scheme compiler, and of Coq's extraction facility itself, but this is still a big step forward from the way most software is developed today.) Indeed, this is one of the main uses for which Coq was developed. We'll come back to this topic in later chapters.

3.3.2 Homework Submission Guidelines

If you are using Software Foundations in a course, your instructor may use automatic scripts to help grade your homework assignments. In order for these scripts to work correctly (so that you get full credit for your work!), please be careful to follow these rules:

- The grading scripts work by extracting marked regions of the .v files that you submit. It is therefore important that you do not alter the "markup" that delimits exercises: the Exercise header, the name of the exercise, the "empty square bracket" marker at the end, etc. Please leave this markup exactly as you find it.
- Do not delete exercises. If you skip an exercise (e.g., because it is marked Optional, or because you can't solve it), it is OK to leave a partial proof in your .v file, but in this case please make sure it ends with *Admitted* (not, for example `Abort`).

3.3.3 Booleans

In a similar way, we can define the standard type **bool** of booleans, with members true and false.

```
Inductive bool : Type :=
  | true : bool
  | false : bool.
```

Although we are rolling our own booleans here for the sake of building up everything from scratch, Coq does, of course, provide a default implementation of the booleans, together with a multitude of useful functions and lemmas. (Take a look at Coq.Init.Datatypes in the Coq library documentation if you're interested.) Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.

Functions over booleans can be defined in the same way as above:

```
Definition negb (b:bool) : bool :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.

Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true ⇒ b2
  | false ⇒ false
  end.

Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true ⇒ true
  | false ⇒ b2
  end.
```

The last two of these illustrate Coq's syntax for multi-argument function definitions. The corresponding multi-argument application syntax is illustrated by the following "unit tests," which constitute a complete specification – a truth table – for the orb function:

```
Example test_orb1: (orb true false) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb2: (orb false false) = false.
Proof. simpl. reflexivity. Qed.
Example test_orb3: (orb false true) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb4: (orb true true) = true.
Proof. simpl. reflexivity. Qed.
```

We can also introduce some familiar syntax for the boolean operations we have just defined. The `Infix` command defines a new symbolic notation for an existing definition.

```
Infix "&&" := andb.
Infix "||" := orb.

Example test_orb5: false || false || true = true.
Proof. simpl. reflexivity. Qed.
```

A note on notation: In *.v* files, we use square brackets to delimit fragments of Coq code within comments; this convention, also used by the *coqdoc* documentation tool, keeps them visually separate from the surrounding text. In the html version of the files, these pieces of text appear in a *different font*.

The command *Admitted* can be used as a placeholder for an incomplete proof. We'll use it in exercises, to indicate the parts that we're leaving for you – i.e., your job is to replace *Admitted*s with real proofs.

Exercise: 1 star (nandb) Remove "*Admitted.*" and complete the definition of the following function; then make sure that the `Example` assertions below can each be verified by Coq. (Remove "*Admitted.*" and fill in each proof, following the model of the `orb` tests above.) The function should return `true` if either or both of its inputs are `false`.

```
Definition nandb (b1:bool) (b2:bool) : bool
  . Admitted.

Example test_nandb1: (nandb true false) = true.
   Admitted.
Example test_nandb2: (nandb false false) = true.
   Admitted.
Example test_nandb3: (nandb false true) = true.
   Admitted.
Example test_nandb4: (nandb true true) = false.
   Admitted.
```

☐

Exercise: 1 star (andb3) Do the same for the `andb3` function below. This function should return `true` when all of its inputs are `true`, and `false` otherwise.

```
Definition andb3 (b1:bool) (b2:bool) (b3:bool) : bool
  . Admitted.

Example test_andb31: (andb3 true true true) = true.
   Admitted.
Example test_andb32: (andb3 false true true) = false.
   Admitted.
Example test_andb33: (andb3 true false true) = false.
   Admitted.
Example test_andb34: (andb3 true true false) = false.
   Admitted.
```

☐

3.3.4 Function Types

Every expression in Coq has a type, describing what sort of thing it computes. The `Check` command asks Coq to print the type of an expression.

`Check` *true*.
`Check` (*negb true*).

Functions like negb itself are also data values, just like true and false. Their types are called *function types*, and they are written with arrows.

`Check` *negb*.

The type of negb, written **bool** → **bool** and pronounced "**bool** arrow **bool**," can be read, "Given an input of type **bool**, this function produces an output of type **bool**." Similarly, the type of andb, written **bool** → **bool** → **bool**, can be read, "Given two inputs, both of type **bool**, this function produces an output of type **bool**."

3.3.5 Modules

Coq provides a *module system*, to aid in organizing large developments. In this course we won't need most of its features, but one is useful: If we enclose a collection of declarations between `Module` X and `End` X markers, then, in the remainder of the file after the `End`, these definitions are referred to by names like *X.foo* instead of just foo. We will use this feature to introduce the definition of the type nat in an inner module so that it does not interfere with the one from the standard library (which we want to use in the rest because it comes with a tiny bit of convenient special notation).

`Module` *NatPlayground*.

3.3.6 Numbers

The types we have defined so far are examples of "enumerated types": their definitions explicitly enumerate a finite set of elements. A more interesting way of defining a type is to give a collection of *inductive rules* describing its elements. For example, we can define (a unary representation of) the natural numbers as follows:

`Inductive` *nat* : `Type` :=
  | *O* : *nat*
  | *S* : *nat* → *nat*.

The clauses of this definition can be read:

- O is a natural number (note that this is the letter "O," not the numeral "0").
- S is a "constructor" that takes a natural number and yields another one – that is, if n is a natural number, then S n is too.

Let's look at this in a little more detail.

Every inductively defined set (**day**, nat, **bool**, etc.) is actually a set of *expressions* built from *constructors* like O, S, true, false, monday, etc. The definition of nat says how expressions in the set nat can be built:

- O and S are constructors;
- the expression O belongs to the set nat;
- if n is an expression belonging to the set nat, then S n is also an expression belonging to the set nat; and
- expressions formed in these two ways are the only ones belonging to the set nat.

The same rules apply for our definitions of **day** and **bool**. (The annotations we used for their constructors are analogous to the one for the O constructor, indicating that they don't take any arguments.)

The above conditions are the precise force of the `Inductive` declaration. They imply that the expression O, the expression S O, the expression S (S O), the expression S (S (S O)), and so on all belong to the set nat, while other expressions built from data constructors, like true, andb true false, S (S false), and O (O (O S)) do not.

A critical point here is that what we've done so far is just to define a *representation* of numbers: a way of writing them down. The names O and S are arbitrary, and at this point they have no special meaning – they are just two different marks that we can use to write down numbers (together with a rule that says any nat will be written as some string of S marks followed by an O). If we like, we can write essentially the same definition this way:

```
Inductive nat' : Type :=
  | stop : nat'
  | tick : nat' → nat'.
```

The *interpretation* of these marks comes from how we use them to compute.

We can do this by writing functions that pattern match on representations of natural numbers just as we did above with booleans and days – for example, here is the predecessor function:

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ n'
  end.
```

The second branch can be read: "if n has the form S *n'* for some *n'*, then return *n'*."

```
End NatPlayground.
```

```
Definition minustwo (n : nat) : nat :=
  match n with
```

```
    | O ⇒ O
    | S O ⇒ O
    | S (S n') ⇒ n'
  end.
```

Because natural numbers are such a pervasive form of data, Coq provides a tiny bit of built-in magic for parsing and printing them: ordinary arabic numerals can be used as an alternative to the "unary" notation defined by the constructors S and O. Coq prints numbers in arabic form by default:

```
Check (S (S (S (S O)))).
Compute (minustwo 4).
```

The constructor S has the type nat → nat, just like the functions minustwo and pred:

```
Check S.
Check pred.
Check minustwo.
```

These are all things that can be applied to a number to yield a number. However, there is a fundamental difference between the first one and the other two: functions like pred and minustwo come with *computation rules* – e.g., the definition of pred says that pred 2 can be simplified to 1 – while the definition of S has no such behavior attached. Although it is like a function in the sense that it can be applied to an argument, it does not *do* anything at all! It is just a way of writing down numbers. (Think about standard arabic numerals: the numeral 1 is not a computation; it's a piece of data. When we write 111 to mean the number one hundred and eleven, we are using 1, three times, to write down a concrete representation of a number.)

For most function definitions over numbers, just pattern matching is not enough: we also need recursion. For example, to check that a number n is even, we may need to recursively check whether n-2 is even. To write such functions, we use the keyword `Fixpoint`.

```
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O ⇒ true
  | S O ⇒ false
  | S (S n') ⇒ evenb n'
  end.
```

We can define oddb by a similar `Fixpoint` declaration, but here is a simpler definition:

```
Definition oddb (n:nat) : bool := negb (evenb n).

Example test_oddb1: oddb 1 = true.
Proof. simpl. reflexivity. Qed.
Example test_oddb2: oddb 4 = false.
Proof. simpl. reflexivity. Qed.
```

(You will notice if you step through these proofs that `simpl` actually has no effect on the goal – all of the work is done by `reflexivity`. We'll see more about why that is shortly.)

Naturally, we can also define multi-argument functions by recursion.

```
Module NatPlayground2.

Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

Adding three to two now gives us five, as we'd expect.

```
Compute (plus 3 2).
```

The simplification that Coq performs to reach this conclusion can be visualized as follows:

As a notational convenience, if two or more arguments have the same type, they can be written together. In the following definition, (n m : nat) means just the same as if we had written (n : nat) (m : nat).

```
Fixpoint mult (n m : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ plus m (mult n' m)
  end.

Example test_mult1: (mult 3 3) = 9.
Proof. simpl. reflexivity. Qed.
```

You can match two expressions at once by putting a comma between them:

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O , _ ⇒ O
  | S _ , O ⇒ n
  | S n', S m' ⇒ minus n' m'
  end.
```

The _in the first line is a *wildcard pattern*. Writing _in a pattern is the same as writing some variable that doesn't get used on the right-hand side. This avoids the need to invent a variable name.

```
End NatPlayground2.

Fixpoint exp (base power : nat) : nat :=
  match power with
    | O ⇒ S O
    | S p ⇒ mult base (exp base p)
  end.
```

Exercise: 1 star (factorial) Recall the standard mathematical factorial function:

factorial(0) = 1 factorial(n) = n * factorial(n-1) (if n>0)

Translate this into Coq.

```
Fixpoint factorial (n:nat) : nat
  . Admitted.

Example test_factorial1: (factorial 3) = 6.
    Admitted.
Example test_factorial2: (factorial 5) = (mult 10 12).
    Admitted.
```

☐

We can make numerical expressions a little easier to read and write by introducing *notations* for addition, multiplication, and subtraction.

```
Notation "x + y" := (plus x y)
                       (at level 50, left associativity)
                       : nat_scope.
Notation "x - y" := (minus x y)
                       (at level 50, left associativity)
                       : nat_scope.
Notation "x * y" := (mult x y)
                       (at level 40, left associativity)
                       : nat_scope.

Check ((0 + 1) + 1).
```

(The `level`, `associativity`, and *nat_scope* annotations control how these notations are treated by Coq's parser. The details are not important for our purposes, but interested readers can refer to the optional "More on Notation" section at the end of this chapter.)

Note that these do not change the definitions we've already made: they are simply instructions to the Coq parser to accept x + y in place of plus x y and, conversely, to the Coq pretty-printer to display plus x y as x + y.

When we say that Coq comes with almost nothing built-in, we really mean it: even equality testing for numbers is a user-defined operation! We now define a function beq_nat, which tests natural numbers for equality, yielding a boolean. Note the use of nested `match`es (we could also have used a simultaneous match, as we did in minus.)

```
Fixpoint beq_nat (n m : nat) : bool :=
  match n with
  | O ⇒ match m with
         | O ⇒ true
         | S m' ⇒ false
         end
  | S n' ⇒ match m with
            | O ⇒ false
            | S m' ⇒ beq_nat n' m'
```

```
        end
  end.
```

The leb function tests whether its first argument is less than or equal to its second argument, yielding a boolean.

```
Fixpoint leb (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
      | O ⇒ false
      | S m' ⇒ leb n' m'
      end
  end.
Example test_leb1: (leb 2 2) = true.
Proof. simpl. reflexivity. Qed.
Example test_leb2: (leb 2 4) = true.
Proof. simpl. reflexivity. Qed.
Example test_leb3: (leb 4 2) = false.
Proof. simpl. reflexivity. Qed.
```

Exercise: 1 star (blt_nat) The blt_nat function tests natural numbers for less-*t*han, yielding a boolean. Instead of making up a new `Fixpoint` for this one, define it in terms of a previously defined function.

```
Definition blt_nat (n m : nat) : bool
  . Admitted.
Example test_blt_nat1: (blt_nat 2 2) = false.
   Admitted.
Example test_blt_nat2: (blt_nat 2 4) = true.
   Admitted.
Example test_blt_nat3: (blt_nat 4 2) = false.
   Admitted.
```

☐

3.4 Proof by Simplification

Now that we've defined a few datatypes and functions, let's turn to stating and proving properties of their behavior. Actually, we've already started doing this: each `Example` in the previous sections makes a precise claim about the behavior of some function on some particular inputs. The proofs of these claims were always the same: use `simpl` to simplify both sides of the equation, then use `reflexivity` to check that both sides contain identical values.

The same sort of "proof by simplification" can be used to prove more interesting properties as well. For example, the fact that 0 is a "neutral element" for + on the left can be proved just by observing that 0 + n reduces to n no matter what n is, a fact that can be read directly off the definition of plus.

```
Theorem plus_O_n : ∀ n : nat, 0 + n = n.
Proof.
  intros n. simpl. reflexivity. Qed.
```

(You may notice that the above statement looks different in the *.v* file in your IDE than it does in the HTML rendition in your browser, if you are viewing both. In *.v* files, we write the ∀ universal quantifier using the reserved identifier "forall." When the *.v* files are converted to HTML, this gets transformed into an upside-down-A symbol.)

This is a good place to mention that `reflexivity` is a bit more powerful than we have admitted. In the examples we have seen, the calls to `simpl` were actually not needed, because `reflexivity` can perform some simplification automatically when checking that two sides are equal; `simpl` was just added so that we could see the intermediate state – after simplification but before finishing the proof. Here is a shorter proof of the theorem:

```
Theorem plus_O_n' : ∀ n : nat, 0 + n = n.
Proof.
  intros n. reflexivity. Qed.
```

Moreover, it will be useful later to know that `reflexivity` does somewhat *more* simplification than `simpl` does – for example, it tries "unfolding" defined terms, replacing them with their right-hand sides. The reason for this difference is that, if reflexivity succeeds, the whole goal is finished and we don't need to look at whatever expanded expressions `reflexivity` has created by all this simplification and unfolding; by contrast, `simpl` is used in situations where we may have to read and understand the new goal that it creates, so we would not want it blindly expanding definitions and leaving the goal in a messy state.

The form of the theorem we just stated and its proof are almost exactly the same as the simpler examples we saw earlier; there are just a few differences.

First, we've used the keyword `Theorem` instead of `Example`. This difference is purely a matter of style; the keywords `Example` and `Theorem` (and a few others, including `Lemma`, `Fact`, and `Remark`) mean exactly the same thing to Coq.

Second, we've added the quantifier ∀ n:nat, so that our theorem talks about *all* natural numbers n. Informally, to prove theorems of this form, we generally start by saying "Suppose n is some number..." Formally, this is achieved in the proof by `intros n`, which moves n from the quantifier in the goal to a *context* of current assumptions.

The keywords `intros`, `simpl`, and `reflexivity` are examples of *tactics*. A tactic is a command that is used between `Proof` and `Qed` to guide the process of checking some claim we are making. We will see several more tactics in the rest of this chapter and yet more in future chapters.

Other similar theorems can be proved with the same pattern.

```
Theorem plus_1_l : ∀ n:nat, 1 + n = S n.
```

```
Proof.
  intros n. reflexivity. Qed.
Theorem mult_0_l : ∀ n:nat, 0 × n = 0.
Proof.
  intros n. reflexivity. Qed.
```

The *_l* suffix in the names of these theorems is pronounced "on the left."

It is worth stepping through these proofs to observe how the context and the goal change. You may want to add calls to simpl before reflexivity to see the simplifications that Coq performs on the terms before checking that they are equal.

Although simplification is powerful enough to prove some fairly general facts, there are many statements that cannot be handled by simplification alone. For instance, we cannot use it to prove that 0 is also a neutral element for + *on the right*.

```
Theorem plus_n_O : ∀ n, n = n + 0.
Proof.
  intros n. simpl.
```

(Can you explain why this happens? Step through both proofs with Coq and notice how the goal and context change.)

When stuck in the middle of a proof, we can use the Abort command to give up on it for the moment. Abort.

The next chapter will introduce *induction*, a powerful technique that can be used for proving this goal. For the moment, though, let's look at a few more simple tactics.

3.5 Proof by Rewriting

This theorem is a bit more interesting than the others we've seen:

```
Theorem plus_id_example : ∀ n m:nat,
  n = m →
  n + n = m + m.
```

Instead of making a universal claim about all numbers n and m, it talks about a more specialized property that only holds when n = m. The arrow symbol is pronounced "implies."

As before, we need to be able to reason by assuming we are given such numbers n and m. We also need to assume the hypothesis n = m. The intros tactic will serve to move all three of these from the goal into assumptions in the current context.

Since n and m are arbitrary numbers, we can't just use simplification to prove this theorem. Instead, we prove it by observing that, if we are assuming n = m, then we can replace n with m in the goal statement and obtain an equality with the same expression on both sides. The tactic that tells Coq to perform this replacement is called rewrite.

```
Proof.
  intros n m.
  intros H.
  rewrite → H.
```

```
  reflexivity. Qed.
```

The first line of the proof moves the universally quantified variables n and m into the context. The second moves the hypothesis n = m into the context and gives it the name *H*. The third tells Coq to rewrite the current goal (n + n = m + m) by replacing the left side of the equality hypothesis *H* with the right side.

(The arrow symbol in the `rewrite` has nothing to do with implication: it tells Coq to apply the rewrite from left to right. To rewrite from right to left, you can use `rewrite` ←. Try making this change in the above proof and see what difference it makes.)

Exercise: 1 star (plus_id_exercise) Remove "*Admitted.*" and fill in the proof.

```
Theorem plus_id_exercise : ∀ n m o : nat,
  n = m → m = o → n + m = m + o.
Proof.
  Admitted.
```

☐

The *Admitted* command tells Coq that we want to skip trying to prove this theorem and just accept it as a given. This can be useful for developing longer proofs, since we can state subsidiary lemmas that we believe will be useful for making some larger argument, use *Admitted* to accept them on faith for the moment, and continue working on the main argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped. Be careful, though: every time you say *Admitted* you are leaving a door open for total nonsense to enter Coq's nice, rigorous, formally checked world!

We can also use the `rewrite` tactic with a previously proved theorem instead of a hypothesis from the context. If the statement of the previously proved theorem involves quantified variables, as in the example below, Coq tries to instantiate them by matching with the current goal.

```
Theorem mult_0_plus : ∀ n m : nat,
  (0 + n) × m = n × m.
Proof.
  intros n m.
  rewrite → plus_O_n.
  reflexivity. Qed.
```

Exercise: 2 stars (mult_S_1)

```
Theorem mult_S_1 : ∀ n m : nat,
  m = S n →
  m × (1 + n) = m × m.
Proof.
  Admitted.
```

☐

3.6 Proof by Case Analysis

Of course, not everything can be proved by simple calculation and rewriting: In general, unknown, hypothetical values (arbitrary numbers, booleans, lists, etc.) can block simplification. For example, if we try to prove the following fact using the `simpl` tactic as above, we get stuck.

```
Theorem plus_1_neq_0_firsttry : ∀ n : nat,
  beq_nat (n + 1) 0 = false.
Proof.
  intros n.
  simpl. Abort.
```

The reason for this is that the definitions of both `beq_nat` and + begin by performing a `match` on their first argument. But here, the first argument to + is the unknown number `n` and the argument to `beq_nat` is the compound expression `n` + 1; neither can be simplified.

To make progress, we need to consider the possible forms of `n` separately. If `n` is `O`, then we can calculate the final result of `beq_nat` (`n` + 1) 0 and check that it is, indeed, `false`. And if `n` = `S` *n'* for some *n'*, then, although we don't know exactly what number `n` + 1 yields, we can calculate that, at least, it will begin with one `S`, and this is enough to calculate that, again, `beq_nat` (`n` + 1) 0 will yield `false`.

The tactic that tells Coq to consider, separately, the cases where `n` = `O` and where `n` = `S` *n'* is called `destruct`.

```
Theorem plus_1_neq_0 : ∀ n : nat,
  beq_nat (n + 1) 0 = false.
Proof.
  intros n. destruct n as [| n'].
  - reflexivity.
  - reflexivity. Qed.
```

The `destruct` generates *two* subgoals, which we must then prove, separately, in order to get Coq to accept the theorem. The annotation "`as` [| *n'*]" is called an *intro pattern*. It tells Coq what variable names to introduce in each subgoal. In general, what goes between the square brackets is a *list of lists* of names, separated by |. In this case, the first component is empty, since the `O` constructor is nullary (it doesn't have any arguments). The second component gives a single name, *n'*, since `S` is a unary constructor.

The - signs on the second and third lines are called *bullets*, and they mark the parts of the proof that correspond to each generated subgoal. The proof script that comes after a bullet is the entire proof for a subgoal. In this example, each of the subgoals is easily proved by a single use of `reflexivity`, which itself performs some simplification – e.g., the first one simplifies `beq_nat` (`S` *n'* + 1) 0 to `false` by first rewriting (`S` *n'* + 1) to `S` (*n'* + 1), then unfolding `beq_nat`, and then simplifying the `match`.

Marking cases with bullets is entirely optional: if bullets are not present, Coq simply asks you to prove each subgoal in sequence, one at a time. But it is a good idea to use bullets. For one thing, they make the structure of a proof apparent, making it more readable. Also, bullets instruct Coq to ensure that a subgoal is complete before trying to verify the next one, preventing proofs for different subgoals from getting mixed up. These issues become especially important in large developments, where fragile proofs lead to long debugging sessions.

There are no hard and fast rules for how proofs should be formatted in Coq – in particular, where lines should be broken and how sections of the proof should be indented to indicate their nested structure. However, if the places where multiple subgoals are generated are marked with explicit bullets at the beginning of lines, then the proof will be readable almost no matter what choices are made about other aspects of layout.

This is also a good place to mention one other piece of somewhat obvious advice about line lengths. Beginning Coq users sometimes tend to the extremes, either writing each tactic on its own line or writing entire proofs on one line. Good style lies somewhere in the middle. One reasonable convention is to limit yourself to 80-character lines.

The `destruct` tactic can be used with any inductively defined datatype. For example, we use it next to prove that boolean negation is involutive – i.e., that negation is its own inverse.

```
Theorem negb_involutive : ∀ b : bool,
  negb (negb b) = b.
Proof.
  intros b. destruct b.
  - reflexivity.
  - reflexivity. Qed.
```

Note that the `destruct` here has no `as` clause because none of the subcases of the `destruct` need to bind any variables, so there is no need to specify any names. (We could also have written `as` [|], or `as` [].) In fact, we can omit the `as` clause from *any* `destruct` and Coq will fill in variable names automatically. This is generally considered bad style, since Coq often makes confusing choices of names when left to its own devices.

It is sometimes useful to invoke `destruct` inside a subgoal, generating yet more proof obligations. In this case, we use different kinds of bullets to mark goals on different "levels." For example:

```
Theorem andb_commutative : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b.
  - destruct c.
    + reflexivity.
    + reflexivity.
  - destruct c.
    + reflexivity.
    + reflexivity.
Qed.
```

Each pair of calls to `reflexivity` corresponds to the subgoals that were generated after the execution of the `destruct c` line right above it.

Besides - and +, we can use × (asterisk) as a third kind of bullet. We can also enclose sub-proofs in curly braces, which is useful in case we ever encounter a proof that generates more than three levels of subgoals:

```
Theorem andb_commutative' : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b.
  { destruct c.
    { reflexivity. }
    { reflexivity. } }
  { destruct c.
    { reflexivity. }
    { reflexivity. } }
Qed.
```

Since curly braces mark both the beginning and the end of a proof, they can be used for multiple subgoal levels, as this example shows. Furthermore, curly braces allow us to reuse the same bullet shapes at multiple levels in a proof:

```
Theorem andb3_exchange :
  ∀ b c d, andb (andb b c) d = andb (andb b d) c.
Proof.
  intros b c d. destruct b.
  - destruct c.
    { destruct d.
      - reflexivity.
      - reflexivity. }
    { destruct d.
      - reflexivity.
      - reflexivity. }
  - destruct c.
    { destruct d.
      - reflexivity.
      - reflexivity. }
    { destruct d.
      - reflexivity.
      - reflexivity. }
Qed.
```

Before closing the chapter, let's mention one final convenience. As you may have noticed, many proofs perform case analysis on a variable right after introducing it:

intros x y. destruct y as |y.

This pattern is so common that Coq provides a shorthand for it: we can perform case analysis on a variable when introducing it by using an intro pattern instead of a variable name. For instance, here is a shorter proof of the plus_1_neq_0 theorem above.

```
Theorem plus_1_neq_0' : ∀ n : nat,
  beq_nat (n + 1) 0 = false.
Proof.
  intros [|n].
  - reflexivity.
  - reflexivity. Qed.
```

If there are no arguments to name, we can just write [].

```
Theorem andb_commutative'' :
  ∀ b c, andb b c = andb c b.
Proof.
  intros [] [].
  - reflexivity.
  - reflexivity.
  - reflexivity.
  - reflexivity.
Qed.
```

Exercise: 2 stars (andb_true_elim2) Prove the following claim, marking cases (and subcases) with bullets when you use destruct.

```
Theorem andb_true_elim2 : ∀ b c : bool,
  andb b c = true → c = true.
Proof.
   Admitted.
```

☐

Exercise: 1 star (zero_nbeq_plus_1)
```
Theorem zero_nbeq_plus_1 : ∀ n : nat,
  beq_nat 0 (n + 1) = false.
Proof.
   Admitted.
```

☐

3.6.1 More on Notation (Optional)

(In general, sections marked Optional are not needed to follow the rest of the book, except possibly other Optional sections. On a first reading, you might want to skim these sections so that you know what's there for future reference.)

Recall the notation definitions for infix plus and times:

```
Notation "x + y" := (plus x y)
                      (at level 50, left associativity)
                      : nat_scope.
```

```
Notation "x * y" := (mult x y)
                    (at level 40, left associativity)
                    : nat_scope.
```

For each notation symbol in Coq, we can specify its *precedence level* and its *associativity*. The precedence level n is specified by writing at level n; this helps Coq parse compound expressions. The associativity setting helps to disambiguate expressions containing multiple occurrences of the same symbol. For example, the parameters specified above for + and × say that the expression 1+2*3*4 is shorthand for (1+((2*3)*4)). Coq uses precedence levels from 0 to 100, and *left*, *right*, or *no* associativity. We will see more examples of this later, e.g., in the Lists chapter.

Each notation symbol is also associated with a *notation scope*. Coq tries to guess what scope is meant from context, so when it sees S(O×O) it guesses *nat_scope*, but when it sees the cartesian product (tuple) type **bool**×**bool** (which we'll see in later chapters) it guesses *type_scope*. Occasionally, it is necessary to help it out with percent-notation by writing (x×y)%nat, and sometimes in what Coq prints it will use %nat to indicate what scope a notation is in.

Notation scopes also apply to numeral notation (3, 4, 5, etc.), so you may sometimes see 0%nat, which means O (the natural number 0 that we're using in this chapter), or 0%Z, which means the Integer zero (which comes from a different part of the standard library).

Pro tip: Coq's notation mechanism is not especially powerful. Don't expect too much from it!

3.6.2 Fixpoints and Structural Recursion (Optional)

Here is a copy of the definition of addition:

```
Fixpoint plus' (n : nat) (m : nat) : nat :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (plus' n' m)
  end.
```

When Coq checks this definition, it notes that plus' is "decreasing on 1st argument." What this means is that we are performing a *structural recursion* over the argument n – i.e., that we make recursive calls only on strictly smaller values of n. This implies that all calls to plus' will eventually terminate. Coq demands that some argument of *every* Fixpoint definition is "decreasing."

This requirement is a fundamental feature of Coq's design: In particular, it guarantees that every function that can be defined in Coq will terminate on all inputs. However, because Coq's "decreasing analysis" is not very sophisticated, it is sometimes necessary to write functions in slightly unnatural ways.

Exercise: 2 stars, optional (decreasing) To get a concrete sense of this, find a way to write a sensible Fixpoint definition (of a simple function on numbers, say) that *does* terminate on all inputs, but that Coq will reject because of this restriction.

□

3.7 More Exercises

Exercise: 2 starsM (boolean_functions) Use the tactics you have learned so far to prove the following theorem about boolean functions.

```
Theorem identity_fn_applied_twice :
  ∀ (f : bool → bool),
  (∀ (x : bool), f x = x) →
  ∀ (b : bool), f (f b) = b.
Proof.
  Admitted.
```

Now state and prove a theorem *negation_fn_applied_twice* similar to the previous one but where the second hypothesis says that the function f has the property that f x = negb x.

□

Exercise: 2 stars (andb_eq_orb) Prove the following theorem. (You may want to first prove a subsidiary lemma or two. Alternatively, remember that you do not have to introduce all hypotheses at the same time.)

```
Theorem andb_eq_orb :
  ∀ (b c : bool),
  (andb b c = orb b c) →
  b = c.
Proof.
  Admitted.
```

□

Exercise: 3 starsM (binary) Consider a different, more efficient representation of natural numbers using a binary rather than unary system. That is, instead of saying that each natural number is either zero or the successor of a natural number, we can say that each binary number is either

- zero,
- twice a binary number, or
- one more than twice a binary number.

(a) First, write an inductive definition of the type *bin* corresponding to this description of binary numbers.

(Hint: Recall that the definition of nat above,

Inductive nat : Type := | O : nat | S : nat -> nat.

says nothing about what O and S "mean." It just says "O is in the set called nat, and if n is in the set then so is S n." The interpretation of O as zero and S as successor/plus one comes from the way that we *use* nat values, by writing functions to do things with them, proving things about them, and so on. Your definition of *bin* should be correspondingly simple; it is the functions you will write next that will give it mathematical meaning.)

(b) Next, write an increment function *incr* for binary numbers, and a function *bin_to_nat* to convert binary numbers to unary numbers.

(c) Write five unit tests *test_bin_incr1*, *test_bin_incr2*, etc. for your increment and binary-to-unary functions. (A "unit test" in Coq is a specific `Example` that can be proved with just `reflexivity`, as we've done for several of our definitions.) Notice that incrementing a binary number and then converting it to unary should yield the same result as first converting it to unary and then incrementing.

□

$Date : 2016 - 11 - 2216 : 39 : 52 - 0500 (Tue, 22Nov2016)$

Chapter 4

Library Top.Induction

4.1 Induction: Proof by Induction

Before getting started, we need to import all of our definitions from the previous chapter:

`Require Export` *Basics*.

For the `Require Export` to work, you first need to use *coqc* to compile *Basics.v* into *Basics.vo*. This is like making a .class file from a .java file, or a .o file from a .c file. There are two ways to do it:

- In CoqIDE:

  Open *Basics.v*. In the "Compile" menu, click on "Compile Buffer".

- From the command line:

  coqc Basics.v

If you have trouble (e.g., if you get complaints about missing identifiers later in the file), it may be because the "load path" for Coq is not set up correctly. The `Print` *LoadPath*. command may be helpful in sorting out such issues.

4.2 Proof by Induction

We proved in the last chapter that 0 is a neutral element for + on the left, using an easy argument based on simplification. We also observed that proving the fact that it is also a neutral element on the *right*...

`Theorem` *plus_n_O_firsttry* : ∀ *n*:*nat*,
  $n = n + 0$.

... can't be done in the same simple way. Just applying `reflexivity` doesn't work, since the `n` in `n` + 0 is an arbitrary unknown number, so the `match` in the definition of + can't be simplified.

```
Proof.
  intros n.
  simpl. Abort.
```

And reasoning by cases using `destruct n` doesn't get us much further: the branch of the case analysis where we assume n = 0 goes through fine, but in the branch where n = S *n'* for some *n'* we get stuck in exactly the same way.

```
Theorem plus_n_O_secondtry : ∀ n:nat,
  n = n + 0.
Proof.
  intros n. destruct n as [| n'].
  -
   reflexivity.   -
   simpl. Abort.
```

We could use `destruct` *n'* to get one step further, but, since n can be arbitrarily large, if we just go on like this we'll never finish.

To prove interesting facts about numbers, lists, and other inductively defined sets, we usually need a more powerful reasoning principle: *induction*.

Recall (from high school, a discrete math course, etc.) the *principle of induction over natural numbers*: If P(n) is some proposition involving a natural number n and we want to show that P holds for all numbers n, we can reason like this:

- show that P(O) holds;
- show that, for any *n'*, if P(*n'*) holds, then so does P(S *n'*);
- conclude that P(n) holds for all n.

In Coq, the steps are the same: we begin with the goal of proving P(n) for all n and break it down (by applying the `induction` tactic) into two separate subgoals: one where we must show P(O) and another where we must show P(*n'*) → P(S *n'*). Here's how this works for the theorem at hand:

```
Theorem plus_n_O : ∀ n:nat, n = n + 0.
Proof.
  intros n. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite ← IHn'. reflexivity. Qed.
```

Like `destruct`, the `induction` tactic takes an `as`... clause that specifies the names of the variables to be introduced in the subgoals. Since there are two subgoals, the `as`... clause has two parts, separated by |. (Strictly speaking, we can omit the `as`... clause and Coq will choose names for us. In practice, this is a bad idea, as Coq's automatic choices tend to be confusing.)

In the first subgoal, n is replaced by 0. No new variables are introduced (so the first part of the `as`... is empty), and the goal becomes $0 + 0 = 0$, which follows by simplification.

In the second subgoal, `n` is replaced by `S` *n'*, and the assumption *n'* + 0 = *n'* is added to the context with the name *IHn'* (i.e., the Induction Hypothesis for *n'*). These two names are specified in the second part of the `as`... clause. The goal in this case becomes (`S` *n'*) + 0 = `S` *n'*, which simplifies to `S` (*n'* + 0) = `S` *n'*, which in turn follows from *IHn'*.

```
Theorem minus_diag : ∀ n,
  minus n n = 0.
Proof.
  intros n. induction n as [| n' IHn'].
  -
    simpl. reflexivity.
  -
    simpl. rewrite → IHn'. reflexivity. Qed.
```

(The use of the `intros` tactic in these proofs is actually redundant. When applied to a goal that contains quantified variables, the `induction` tactic will automatically move them into the context as needed.)

Exercise: 2 stars, recommended (basic_induction) Prove the following using induction. You might need previously proven results.

```
Theorem mult_0_r : ∀ n:nat,
  n × 0 = 0.
Proof.
   Admitted.

Theorem plus_n_Sm : ∀ n m : nat,
  S (n + m) = n + (S m).
Proof.
   Admitted.

Theorem plus_comm : ∀ n m : nat,
  n + m = m + n.
Proof.
   Admitted.

Theorem plus_assoc : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
   Admitted.
   □
```

Exercise: 2 stars (double_plus) Consider the following function, which doubles its argument:

```
Fixpoint double (n:nat) :=
  match n with
```

```
  | O ⇒ O
  | S n' ⇒ S (S (double n'))
  end.
```

Use induction to prove this simple fact about double:

```
Lemma double_plus : ∀ n, double n = n + n .
Proof.
   Admitted.
```

□

Exercise: 2 stars, optional (evenb_S) One inconveninent aspect of our definition of evenb n is the recursive call on n - 2. This makes proofs about evenb n harder when done by induction on n, since we may need an induction hypothesis about n - 2. The following lemma gives an alternative characterization of evenb (S n) that works better with induction:

```
Theorem evenb_S : ∀ n : nat,
  evenb (S n) = negb (evenb n).
Proof.
   Admitted.
```

□

Exercise: 1 starM (destruct_induction) Briefly explain the difference between the tactics destruct and induction.

□

4.3 Proofs Within Proofs

In Coq, as in informal mathematics, large proofs are often broken into a sequence of theorems, with later proofs referring to earlier theorems. But sometimes a proof will require some miscellaneous fact that is too trivial and of too little general interest to bother giving it its own top-level name. In such cases, it is convenient to be able to simply state and prove the needed "sub-theorem" right at the point where it is used. The assert tactic allows us to do this. For example, our earlier proof of the mult_0_plus theorem referred to a previous theorem named plus_O_n. We could instead use assert to state and prove plus_O_n in-line:

```
Theorem mult_0_plus' : ∀ n m : nat,
  (0 + n) × m = n × m.
Proof.
  intros n m.
  assert (H: 0 + n = n). { reflexivity. }
  rewrite → H.
  reflexivity. Qed.
```

The `assert` tactic introduces two sub-goals. The first is the assertion itself; by prefixing it with H: we name the assertion H. (We can also name the assertion with `as` just as we did above with `destruct` and `induction`, i.e., `assert` $(0 + \mathsf{n} = \mathsf{n})$ `as` H.) Note that we surround the proof of this assertion with curly braces { ... }, both for readability and so that, when using Coq interactively, we can see more easily when we have finished this sub-proof. The second goal is the same as the one at the point where we invoke `assert` except that, in the context, we now have the assumption H that $0 + \mathsf{n} = \mathsf{n}$. That is, `assert` generates one subgoal where we must prove the asserted fact and a second subgoal where we can use the asserted fact to make progress on whatever we were trying to prove in the first place.

Another example of `assert`...

For example, suppose we want to prove that $(\mathsf{n} + \mathsf{m}) + (p + q) = (\mathsf{m} + \mathsf{n}) + (p + q)$. The only difference between the two sides of the $=$ is that the arguments m and n to the first inner $+$ are swapped, so it seems we should be able to use the commutativity of addition (plus_comm) to rewrite one into the other. However, the `rewrite` tactic is not very smart about *where* it applies the rewrite. There are three uses of $+$ here, and it turns out that doing `rewrite` → plus_comm will affect only the *outer* one...

```
Theorem plus_rearrange_firsttry : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  rewrite → plus_comm.
Abort.
```

To use plus_comm at the point where we need it, we can introduce a local lemma stating that $\mathsf{n} + \mathsf{m} = \mathsf{m} + \mathsf{n}$ (for the particular m and n that we are talking about here), prove this lemma using plus_comm, and then use it to do the desired rewrite.

```
Theorem plus_rearrange : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  assert (H: n + m = m + n).
  { rewrite → plus_comm. reflexivity. }
  rewrite → H. reflexivity. Qed.
```

4.4 Formal vs. Informal Proof

"*Informal proofs are algorithms; formal proofs are code.*“

What constitutes a successful proof of a mathematical claim? The question has challenged philosophers for millennia, but a rough and ready definition could be this: A proof of a mathematical proposition P is a written (or spoken) text that instills in the reader or hearer the certainty that P is true – an unassailable argument for the truth of P. That is, a proof is an act of communication.

Acts of communication may involve different sorts of readers. On one hand, the "reader" can be a program like Coq, in which case the "belief" that is instilled is that P can be mechanically derived from a certain set of formal logical rules, and the proof is a recipe that guides the program in checking this fact. Such recipes are *formal* proofs.

Alternatively, the reader can be a human being, in which case the proof will be written in English or some other natural language, and will thus necessarily be *informal*. Here, the criteria for success are less clearly specified. A "valid" proof is one that makes the reader believe P. But the same proof may be read by many different readers, some of whom may be convinced by a particular way of phrasing the argument, while others may not be. Some readers may be particularly pedantic, inexperienced, or just plain thick-headed; the only way to convince them will be to make the argument in painstaking detail. But other readers, more familiar in the area, may find all this detail so overwhelming that they lose the overall thread; all they want is to be told the main ideas, since it is easier for them to fill in the details for themselves than to wade through a written presentation of them. Ultimately, there is no universal standard, because there is no single way of writing an informal proof that is guaranteed to convince every conceivable reader.

In practice, however, mathematicians have developed a rich set of conventions and idioms for writing about complex mathematical objects that – at least within a certain community – make communication fairly reliable. The conventions of this stylized form of communication give a fairly clear standard for judging proofs good or bad.

Because we are using Coq in this course, we will be working heavily with formal proofs. But this doesn't mean we can completely forget about informal ones! Formal proofs are useful in many ways, but they are *not* very efficient ways of communicating ideas between human beings.

For example, here is a proof that addition is associative:

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof. intros n m p. induction n as [| n' IHn']. reflexivity.
  simpl. rewrite → IHn'. reflexivity. Qed.
```

Coq is perfectly happy with this. For a human, however, it is difficult to make much sense of it. We can use comments and bullets to show the structure a little more clearly...

```
Theorem plus_assoc'' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  -
    reflexivity.
  -
    simpl. rewrite → IHn'. reflexivity. Qed.
```

... and if you're used to Coq you may be able to step through the tactics one after the other in your mind and imagine the state of the context and goal stack at each point, but if the proof were even a little bit more complicated this would be next to impossible.

A (pedantic) mathematician might write the proof something like this:

- *Theorem*: For any n, m and *p*,

  n + (m + p) = (n + m) + p.

  Proof: By induction on n.

  - First, suppose n = 0. We must show

    0 + (m + p) = (0 + m) + p.

    This follows directly from the definition of +.

  - Next, suppose n = S *n'*, where

    n' + (m + p) = (n' + m) + p.

    We must show

    (S n') + (m + p) = ((S n') + m) + p.

    By the definition of +, this follows from

    S (n' + (m + p)) = S ((n' + m) + p),

    which is immediate from the induction hypothesis. *Qed*.

The overall form of the proof is basically similar, and of course this is no accident: Coq has been designed so that its `induction` tactic generates the same sub-goals, in the same order, as the bullet points that a mathematician would write. But there are significant differences of detail: the formal proof is much more explicit in some ways (e.g., the use of `reflexivity`) but much less explicit in others (in particular, the "proof state" at any given point in the Coq proof is completely implicit, whereas the informal proof reminds the reader several times where things stand).

Exercise: 2 stars, advanced, recommendedM (plus_comm_informal) Translate your solution for plus_comm into an informal proof:

Theorem: Addition is commutative.

Proof: □

Exercise: 2 stars, optionalM (beq_nat_refl_informal) Write an informal proof of the following theorem, using the informal proof of plus_assoc as a model. Don't just paraphrase the Coq tactics into English!

Theorem: true = beq_nat n n for any n.

Proof: □

4.5 More Exercises

Exercise: 3 stars, recommended (mult_comm) Use `assert` to help prove this theorem. You shouldn't need to use induction on plus_swap.

```
Theorem plus_swap : ∀ n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
  Admitted.
```

Now prove commutativity of multiplication. (You will probably need to define and prove a separate subsidiary theorem to be used in the proof of this one. You may find that plus_swap comes in handy.)

```
Theorem mult_comm : ∀ m n : nat,
  m × n = n × m.
Proof.
  Admitted.
```

□

Exercise: 3 stars, optional (more_exercises) Take a piece of paper. For each of the following theorems, first *think* about whether (a) it can be proved using only simplification and rewriting, (b) it also requires case analysis (`destruct`), or (c) it also requires induction. Write down your prediction. Then fill in the proof. (There is no need to turn in your piece of paper; this is just to encourage you to reflect before you hack!)

```
Theorem leb_refl : ∀ n:nat,
  true = leb n n.
Proof.
  Admitted.

Theorem zero_nbeq_S : ∀ n:nat,
  beq_nat 0 (S n) = false.
Proof.
  Admitted.

Theorem andb_false_r : ∀ b : bool,
  andb b false = false.
Proof.
  Admitted.

Theorem plus_ble_compat_l : ∀ n m p : nat,
  leb n m = true → leb (p + n) (p + m) = true.
Proof.
  Admitted.

Theorem S_nbeq_0 : ∀ n:nat,
  beq_nat (S n) 0 = false.
```

Proof.
  Admitted.

Theorem *mult_1_l* : $\forall$ *n*:*nat*, $1 \times n = n$.
Proof.
  Admitted.

Theorem *all3_spec* : $\forall$ *b c* : *bool*,
    orb
      (*andb b c*)
      (*orb* (*negb b*)
              (*negb c*))
  = *true*.
Proof.
  Admitted.

Theorem *mult_plus_distr_r* : $\forall$ *n m p* : *nat*,
  $(n + m) \times p = (n \times p) + (m \times p)$.
Proof.
  Admitted.

Theorem *mult_assoc* : $\forall$ *n m p* : *nat*,
  $n \times (m \times p) = (n \times m) \times p$.
Proof.
  Admitted.
  ☐

Exercise: 2 stars, optional (beq_nat_refl) Prove the following theorem. (Putting the true on the left-hand side of the equality may look odd, but this is how the theorem is stated in the Coq standard library, so we follow suit. Rewriting works equally well in either direction, so we will have no problem using the theorem no matter which way we state it.)

Theorem *beq_nat_refl* : $\forall$ *n* : *nat*,
  true = *beq_nat n n*.
Proof.
  Admitted.
  ☐

Exercise: 2 stars, optional (plus_swap') The replace tactic allows you to specify a particular subterm to rewrite and what you want it rewritten to: replace (t) with (u) replaces (all copies of) expression t in the goal by expression u, and generates $t = u$ as an additional subgoal. This is often useful when a plain rewrite acts on the wrong part of the goal.

Use the replace tactic to do a proof of plus_swap', just like plus_swap but without needing assert (n + m = m + n).

Theorem *plus_swap'* : $\forall$ *n m p* : *nat*,

$n + (m + p) = m + (n + p)$.

`Proof.`

Admitted.

☐

Exercise: 3 stars, recommendedM (binary_commute) Recall the *incr* and *bin_to_nat* functions that you wrote for the *binary* exercise in the Basics chapter. Prove that the following diagram commutes:

incr bin ———————-> bin | | bin_to_nat | | bin_to_nat | | v v nat ———————-> nat S

That is, incrementing a binary number and then converting it to a (unary) natural number yields the same result as first converting it to a natural number and then incrementing. Name your theorem *bin_to_nat_pres_incr* ("pres" for "preserves").

Before you start working on this exercise, copy the definitions from your solution to the *binary* exercise here so that this file can be graded on its own. If you want to change your original definitions to make the property easier to prove, feel free to do so!

☐

Exercise: 5 stars, advancedM (binary_inverse) This exercise is a continuation of the previous exercise about binary numbers. You will need your definitions and theorems from there to complete this one; please copy them to this file to make it self contained for grading.

(a) First, write a function to convert natural numbers to binary numbers. Then prove that starting with any natural number, converting to binary, then converting back yields the same natural number you started with.

(b) You might naturally think that we should also prove the opposite direction: that starting with a binary number, converting to a natural, and then back to binary yields the same number we started with. However, this is not true! Explain what the problem is.

(c) Define a "direct" normalization function – i.e., a function *normalize* from binary numbers to binary numbers such that, for any binary number b, converting to a natural and then back to binary yields (*normalize* `b`). Prove it. (Warning: This part is tricky!)

Again, feel free to change your earlier definitions if this helps here.

☐

$Date : 2016-10-0714:01:19-0400(Fri, 07Oct2016)$

Chapter 5

Library Top.Lists

5.1 Lists: Working with Structured Data

```
Require Export Induction.
Module NatList.
```

5.2 Pairs of Numbers

In an `Inductive` type definition, each constructor can take any number of arguments – none (as with `true` and `O`), one (as with `S`), or more than one, as here:

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.
```

This declaration can be read: "There is just one way to construct a pair of numbers: by applying the constructor `pair` to two arguments of type `nat`."

```
Check (pair 3 5).
```

Here are two simple functions for extracting the first and second components of a pair. The definitions also illustrate how to do pattern matching on two-argument constructors.

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y ⇒ x
  end.

Definition snd (p : natprod) : nat :=
  match p with
  | pair x y ⇒ y
  end.

Compute (fst (pair 3 5)).
```

Since pairs are used quite a bit, it is nice to be able to write them with the standard mathematical notation (x,y) instead of pair x y. We can tell Coq to allow this with a `Notation` declaration.

```
Notation "( x , y )" := (pair x y).
```

The new pair notation can be used both in expressions and in pattern matches (indeed, we've actually seen this already in the previous chapter, in the definition of the minus function – this works because the pair notation is also provided as part of the standard library):

```
Compute (fst (3,5)).

Definition fst' (p : natprod) : nat :=
  match p with
  | (x,y) ⇒ x
  end.

Definition snd' (p : natprod) : nat :=
  match p with
  | (x,y) ⇒ y
  end.

Definition swap_pair (p : natprod) : natprod :=
  match p with
  | (x,y) ⇒ (y,x)
  end.
```

Let's try to prove a few simple facts about pairs.

If we state things in a particular (and slightly peculiar) way, we can complete proofs with just reflexivity (and its built-in simplification):

```
Theorem surjective_pairing' : ∀ (n m : nat),
  (n,m) = (fst (n,m), snd (n,m)).
Proof.
  reflexivity. Qed.
```

But `reflexivity` is not enough if we state the lemma in a more natural way:

```
Theorem surjective_pairing_stuck : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  simpl. Abort.
```

We have to expose the structure of p so that `simpl` can perform the pattern match in fst and snd. We can do this with `destruct`.

```
Theorem surjective_pairing : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p. destruct p as [n m]. simpl. reflexivity. Qed.
```

Notice that, unlike its behavior with nats, destruct generates just one subgoal here. That's because **natprod**s can only be constructed in one way.

Exercise: 1 star (snd_fst_is_swap) Theorem *snd_fst_is_swap* : ∀ (*p* : *natprod*),
 (*snd p*, *fst p*) = *swap_pair p*.
Proof.
 Admitted.
 ☐

Exercise: 1 star, optional (fst_swap_is_snd) Theorem *fst_swap_is_snd* : ∀ (*p* : *natprod*),
 fst (*swap_pair p*) = *snd p*.
Proof.
 Admitted.
 ☐

5.3 Lists of Numbers

Generalizing the definition of pairs, we can describe the type of *lists* of numbers like this: "A list is either the empty list or else a pair of a number and another list."

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat → natlist → natlist.
```

For example, here is a three-element list:

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
```

As with pairs, it is more convenient to write lists in familiar programming notation. The following declarations allow us to use :: as an infix cons operator and square brackets as an "outfix" notation for constructing lists.

```
Notation "x :: l" := (cons x l)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

It is not necessary to understand the details of these declarations, but in case you are interested, here is roughly what's going on. The right associativity annotation tells Coq how to parenthesize expressions involving several uses of :: so that, for example, the next three declarations mean exactly the same thing:

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
```

The at level 60 part tells Coq how to parenthesize expressions that involve both :: and some other infix operator. For example, since we defined + as infix notation for the plus function at level 50,

Notation "x + y" := (plus x y) (at level 50, left associativity).

the + operator will bind tighter than ::, so 1 + 2 :: [3] will be parsed, as we'd expect, as (1 + 2) :: [3] rather than 1 + (2 :: [3]).

(Expressions like "1 + 2 :: [3]" can be a little confusing when you read them in a .v file. The inner brackets, around 3, indicate a list, but the outer brackets, which are invisible in the HTML rendering, are there to instruct the "coqdoc" tool that the bracketed part should be displayed as Coq code rather than running text.)

The second and third Notation declarations above introduce the standard square-bracket notation for lists; the right-hand side of the third one illustrates Coq's syntax for declaring n-ary notations and translating them to nested sequences of binary constructors.

Repeat

A number of functions are useful for manipulating lists. For example, the repeat function takes a number n and a count and returns a list of length count where every element is n.

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O ⇒ nil
  | S count' ⇒ n :: (repeat n count')
  end.
```

Length

The length function calculates the length of a list.

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil ⇒ O
  | h :: t ⇒ S (length t)
  end.
```

Append

The app function concatenates (appends) two lists.

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil ⇒ l2
  | h :: t ⇒ h :: (app t l2)
  end.
```

Actually, app will be used a lot in some parts of what follows, so it is convenient to have an infix operator for it.

```
Notation "x ++ y" := (app x y)
                     (right associativity, at level 60).
Example test_app1: [1;2;3] ++ [4;5] = [1;2;3;4;5].
Proof. reflexivity. Qed.
Example test_app2: nil ++ [4;5] = [4;5].
Proof. reflexivity. Qed.
Example test_app3: [1;2;3] ++ nil = [1;2;3].
Proof. reflexivity. Qed.
```

Head (with default) and Tail

Here are two smaller examples of programming with lists. The hd function returns the first element (the "head") of the list, while tl returns everything but the first element (the "tail"). Of course, the empty list has no first element, so we must pass a default value to be returned in that case.

```
Definition hd (default:nat) (l:natlist) : nat :=
  match l with
  | nil ⇒ default
  | h :: t ⇒ h
  end.
Definition tl (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ t
  end.
Example test_hd1: hd 0 [1;2;3] = 1.
Proof. reflexivity. Qed.
Example test_hd2: hd 0 [] = 0.
Proof. reflexivity. Qed.
Example test_tl: tl [1;2;3] = [2;3].
Proof. reflexivity. Qed.
```

Exercises

Exercise: 2 stars, recommended (list_funs) Complete the definitions of nonzeros, oddmembers and countoddmembers below. Have a look at the tests to understand what these functions should do.

```
Fixpoint nonzeros (l:natlist) : natlist
  . Admitted.
Example test_nonzeros:
  nonzeros [0;1;0;2;3;0;0] = [1;2;3].
```

Admitted.

Fixpoint *oddmembers* (*l*:*natlist*) : *natlist*
. *Admitted.*

Example *test_oddmembers*:
oddmembers [0;1;0;2;3;0;0] = [1;3].
Admitted.

Definition *countoddmembers* (*l*:*natlist*) : *nat*
. *Admitted.*

Example *test_countoddmembers1*:
countoddmembers [1;0;3;1;4;5] = 4.
Admitted.

Example *test_countoddmembers2*:
countoddmembers [0;2;4] = 0.
Admitted.

Example *test_countoddmembers3*:
countoddmembers nil = 0.
Admitted.
□

Exercise: 3 stars, advanced (alternate) Complete the definition of alternate, which "zips up" two lists into one, alternating between elements taken from the first list and elements from the second. See the tests below for more specific examples.

Note: one natural and elegant way of writing alternate will fail to satisfy Coq's requirement that all Fixpoint definitions be "obviously terminating." If you find yourself in this rut, look for a slightly more verbose solution that considers elements of both lists at the same time. (One possible solution requires defining a new kind of pairs, but this is not the only way.)

Fixpoint *alternate* (*l1 l2* : *natlist*) : *natlist*
. *Admitted.*

Example *test_alternate1*:
alternate [1;2;3] [4;5;6] = [1;4;2;5;3;6].
Admitted.

Example *test_alternate2*:
alternate [1] [4;5;6] = [1;4;5;6].
Admitted.

Example *test_alternate3*:
alternate [1;2;3] [4] = [1;4;2;3].
Admitted.

Example *test_alternate4*:

alternate [] [20;30] = [20;30].
Admitted.
□

Bags via Lists

A `bag` (or *multiset*) is like a set, except that each element can appear multiple times rather than just once. One possible implementation is to represent a bag of numbers as a list.

`Definition` *bag* := *natlist*.

Exercise: 3 stars, recommended (bag_functions) Complete the following definitions for the functions `count`, `sum`, `add`, and `member` for bags.

`Fixpoint` *count* (*v*:*nat*) (*s*:*bag*) : *nat*
. *Admitted.*

All these proofs can be done just by `reflexivity`.

`Example` *test_count1*: *count* 1 [1;2;3;1;4;1] = 3.
Admitted.
`Example` *test_count2*: *count* 6 [1;2;3;1;4;1] = 0.
Admitted.

Multiset `sum` is similar to set *union*: `sum a b` contains all the elements of `a` and of `b`. (Mathematicians usually define *union* on multisets a little bit differently, which is why we don't use that name for this operation.) For `sum` we're giving you a header that does not give explicit names to the arguments. Moreover, it uses the keyword `Definition` instead of `Fixpoint`, so even if you had names for the arguments, you wouldn't be able to process them recursively. The point of stating the question this way is to encourage you to think about whether `sum` can be implemented in another way – perhaps by using functions that have already been defined.

`Definition` *sum* : *bag* → *bag* → *bag*
. *Admitted.*

`Example` *test_sum1*: *count* 1 (*sum* [1;2;3] [1;4;1]) = 3.
Admitted.

`Definition` *add* (*v*:*nat*) (*s*:*bag*) : *bag*
. *Admitted.*

`Example` *test_add1*: *count* 1 (*add* 1 [1;4;1]) = 3.
Admitted.
`Example` *test_add2*: *count* 5 (*add* 1 [1;4;1]) = 0.
Admitted.

`Definition` *member* (*v*:*nat*) (*s*:*bag*) : *bool*
. *Admitted.*

Example *test_member1*: *member* 1 [1;4;1] = *true*.
Admitted.

Example *test_member2*: *member* 2 [1;4;1] = *false*.
Admitted.
□

Exercise: 3 stars, optional (bag_more_functions) Here are some more bag functions for you to practice with.

When remove_one is applied to a bag without the number to remove, it should return the same bag unchanged.

Fixpoint *remove_one* (*v*:*nat*) (*s*:*bag*) : *bag*
. *Admitted.*

Example *test_remove_one1*:
count 5 (*remove_one* 5 [2;1;5;4;1]) = 0.
Admitted.

Example *test_remove_one2*:
count 5 (*remove_one* 5 [2;1;4;1]) = 0.
Admitted.

Example *test_remove_one3*:
count 4 (*remove_one* 5 [2;1;4;5;1;4]) = 2.
Admitted.

Example *test_remove_one4*:
count 5 (*remove_one* 5 [2;1;5;4;5;1;4]) = 1.
Admitted.

Fixpoint *remove_all* (*v*:*nat*) (*s*:*bag*) : *bag*
. *Admitted.*

Example *test_remove_all1*: *count* 5 (*remove_all* 5 [2;1;5;4;1]) = 0.
Admitted.
Example *test_remove_all2*: *count* 5 (*remove_all* 5 [2;1;4;1]) = 0.
Admitted.
Example *test_remove_all3*: *count* 4 (*remove_all* 5 [2;1;4;5;1;4]) = 2.
Admitted.
Example *test_remove_all4*: *count* 5 (*remove_all* 5 [2;1;5;4;5;1;4;5;1;4]) = 0.
Admitted.

Fixpoint *subset* (*s1*:*bag*) (*s2*:*bag*) : *bool*
. *Admitted.*

Example *test_subset1*: *subset* [1;2] [2;1;4;1] = *true*.
Admitted.
Example *test_subset2*: *subset* [1;2;2] [2;1;4;1] = *false*.

Admitted.

□

Exercise: 3 stars, recommendedM (bag_theorem) Write down an interesting theorem *bag_theorem* about bags involving the functions count and add, and prove it. Note that, since this problem is somewhat open-ended, it's possible that you may come up with a theorem which is true, but whose proof requires techniques you haven't learned yet. Feel free to ask for help if you get stuck!

□

5.4 Reasoning About Lists

As with numbers, simple facts about list-processing functions can sometimes be proved entirely by simplification. For example, the simplification performed by `reflexivity` is enough for this theorem...

```
Theorem nil_app : ∀ l:natlist,
  [] ++ l = l.
Proof. reflexivity. Qed.
```

... because the [] is substituted into the "scrutinee" (the value being "scrutinized" by the match) in the definition of app, allowing the match itself to be simplified.

Also, as with numbers, it is sometimes helpful to perform case analysis on the possible shapes (empty or non-empty) of an unknown list.

```
Theorem tl_length_pred : ∀ l:natlist,
  pred (length l) = length (tl l).
Proof.
  intros l. destruct l as [| n l'].
  -
    reflexivity.
  -
    reflexivity. Qed.
```

Here, the nil case works because we've chosen to define tl nil = nil. Notice that the `as` annotation on the `destruct` tactic here introduces two names, n and *l'*, corresponding to the fact that the cons constructor for lists takes two arguments (the head and tail of the list it is constructing).

Usually, though, interesting theorems about lists require induction for their proofs.

Micro-Sermon

Simply reading example proof scripts will not get you very far! It is important to work through the details of each one, using Coq and thinking about what each step achieves.

Otherwise it is more or less guaranteed that the exercises will make no sense when you get to them. 'Nuff said.

5.4.1 Induction on Lists

Proofs by induction over datatypes like **natlist** are a little less familiar than standard natural number induction, but the idea is equally simple. Each `Inductive` declaration defines a set of data values that can be built up using the declared constructors: a boolean can be either true or false; a number can be either O or S applied to another number; a list can be either nil or cons applied to a number and a list.

Moreover, applications of the declared constructors to one another are the *only* possible shapes that elements of an inductively defined set can have, and this fact directly gives rise to a way of reasoning about inductively defined sets: a number is either O or else it is S applied to some *smaller* number; a list is either nil or else it is cons applied to some number and some *smaller* list; etc. So, if we have in mind some proposition P that mentions a list l and we want to argue that P holds for *all* lists, we can reason as follows:

- First, show that P is true of l when l is nil.
- Then show that P is true of l when l is cons n *l'* for some number n and some smaller list *l'*, assuming that P is true for *l'*.

Since larger lists can only be built up from smaller ones, eventually reaching nil, these two arguments together establish the truth of P for all lists l. Here's a concrete example:

```
Theorem app_assoc : ∀ l1 l2 l3 : natlist,
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
Proof.
  intros l1 l2 l3. induction l1 as [| n l1' IHl1'].
  -
    reflexivity.
  -
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

Notice that, as when doing induction on natural numbers, the `as`... clause provided to the `induction` tactic gives a name to the induction hypothesis corresponding to the smaller list *l1'* in the cons case. Once again, this Coq proof is not especially illuminating as a static written document – it is easy to see what's going on if you are reading the proof in an interactive Coq session and you can see the current goal and context at each point, but this state is not visible in the written-down parts of the Coq proof. So a natural-language proof – one written for human readers – will need to include more explicit signposts; in particular, it will help the reader stay oriented if we remind them exactly what the induction hypothesis is in the second case.

For comparison, here is an informal proof of the same theorem.

Theorem: For all lists l1, l2, and l3, (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).

Proof: By induction on l1.

- First, suppose l1 = []. We must show

  ($\square$ ++ l2) ++ l3 = $\square$ ++ (l2 ++ l3),

  which follows directly from the definition of ++.

- Next, suppose l1 = n::*l1'*, with

  (l1' ++ l2) ++ l3 = l1' ++ (l2 ++ l3)

  (the induction hypothesis). We must show

  ((n :: l1') ++ l2) ++ l3 = (n :: l1') ++ (l2 ++ l3).

  By the definition of ++, this follows from

  n :: ((l1' ++ l2) ++ l3) = n :: (l1' ++ (l2 ++ l3)),

  which is immediate from the induction hypothesis. $\square$

Reversing a List

For a slightly more involved example of inductive proof over lists, suppose we use app to define a list-reversing function rev:

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ rev t ++ [h]
  end.
Example test_rev1: rev [1;2;3] = [3;2;1].
Proof. reflexivity. Qed.
Example test_rev2: rev nil = nil.
Proof. reflexivity. Qed.
```

Properties of rev

Now let's prove some theorems about our newly defined rev. For something a bit more challenging than what we've seen, let's prove that reversing a list does not change its length. Our first attempt gets stuck in the successor case...

```
Theorem rev_length_firsttry : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  -
    reflexivity.
  -
```

```
    simpl.
    rewrite ← IHl'.
Abort.
```

So let's take the equation relating ++ and length that would have enabled us to make progress and prove it as a separate lemma.

```
Theorem app_length : ∀ l1 l2 : natlist,
  length (l1 ++ l2) = (length l1) + (length l2).
Proof.
  intros l1 l2. induction l1 as [| n l1' IHl1'].
  -
    reflexivity.
  -
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

Note that, to make the lemma as general as possible, we quantify over *all* **natlist**s, not just those that result from an application of rev. This should seem natural, because the truth of the goal clearly doesn't depend on the list having been reversed. Moreover, it is easier to prove the more general property.

Now we can complete the original proof.

```
Theorem rev_length : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  -
    reflexivity.
  -
    simpl. rewrite → app_length, plus_comm.
    simpl. rewrite → IHl'. reflexivity. Qed.
```

For comparison, here are informal proofs of these two theorems:

Theorem: For all lists l1 and l2, length (l1 ++ l2) = length l1 + length l2.

Proof: By induction on l1.

- First, suppose l1 = []. We must show

  length (□ ++ l2) = length □ + length l2,

  which follows directly from the definitions of length and ++.

- Next, suppose l1 = n::*l1'*, with

  length (l1' ++ l2) = length l1' + length l2.

  We must show

  length ((n::l1') ++ l2) = length (n::l1') + length l2).

This follows directly from the definitions of length and ++ together with the induction hypothesis. □

Theorem: For all lists l, length (rev l) = length l.
Proof: By induction on l.

- First, suppose l = []. We must show

  length (rev □) = length □,

  which follows directly from the definitions of length and rev.

- Next, suppose l = n::*l'*, with

  length (rev l') = length l'.

  We must show

  length (rev (n :: l')) = length (n :: l').

  By the definition of rev, this follows from

  length ((rev l') ++ n) = S (length l')

  which, by the previous lemma, is the same as

  length (rev l') + length n = S (length l').

  This follows directly from the induction hypothesis and the definition of length. □

The style of these proofs is rather longwinded and pedantic. After the first few, we might find it easier to follow proofs that give fewer details (which can easily work out in our own minds or on scratch paper if necessary) and just highlight the non-obvious steps. In this more compressed style, the above proof might look like this:

Theorem: For all lists l, length (rev l) = length l.

Proof: First, observe that length (l ++ [n]) = S (length l) for any l (this follows by a straightforward induction on l). The main property again follows by induction on l, using the observation together with the induction hypothesis in the case where l = *n'*::*l'*. □

Which style is preferable in a given situation depends on the sophistication of the expected audience and how similar the proof at hand is to ones that the audience will already be familiar with. The more pedantic style is a good default for our present purposes.

5.4.2 Search

We've seen that proofs can make use of other theorems we've already proved, e.g., using `rewrite`. But in order to refer to a theorem, we need to know its name! Indeed, it is often hard even to remember what theorems have been proven, much less what they are called.

Coq's `Search` command is quite helpful with this. Typing `Search` foo will cause Coq to display a list of all theorems involving foo. For example, try uncommenting the following line to see a list of theorems that we have proved about rev:

Keep `Search` in mind as you do the following exercises and throughout the rest of the book; it can save you a lot of time!

If you are using ProofGeneral, you can run `Search` with C-c C-a C-a. Pasting its response into your buffer can be accomplished with C-c C-;.

5.4.3 List Exercises, Part 1

Exercise: 3 starsM (list_exercises) More practice with lists:

```
Theorem app_nil_r : ∀ l : natlist,
  l ++ [] = l.
Proof.
  Admitted.

Theorem rev_app_distr: ∀ l1 l2 : natlist,
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
  Admitted.

Theorem rev_involutive : ∀ l : natlist,
  rev (rev l) = l.
Proof.
  Admitted.
```

There is a short solution to the next one. If you find yourself getting tangled up, step back and try to look for a simpler way.

```
Theorem app_assoc4 : ∀ l1 l2 l3 l4 : natlist,
  l1 ++ (l2 ++ (l3 ++ l4)) = ((l1 ++ l2) ++ l3) ++ l4.
Proof.
  Admitted.
```

An exercise about your implementation of nonzeros:

```
Lemma nonzeros_app : ∀ l1 l2 : natlist,
  nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2).
Proof.
  Admitted.
```

□

Exercise: 2 stars (beq_natlist) Fill in the definition of beq_natlist, which compares lists of numbers for equality. Prove that beq_natlist l l yields true for every list l.

```
Fixpoint beq_natlist (l1 l2 : natlist) : bool
  . Admitted.

Example test_beq_natlist1 :
  (beq_natlist nil nil = true).
  Admitted.
```

Example *test_beq_natlist2* :
  beq_natlist [1;2;3] [1;2;3] = *true*.
    Admitted.

Example *test_beq_natlist3* :
  beq_natlist [1;2;3] [1;2;4] = *false*.
    Admitted.

Theorem *beq_natlist_refl* : ∀ *l*:*natlist*,
  true = *beq_natlist l l*.
Proof.
    Admitted.
    □

5.4.4 List Exercises, Part 2

Exercise: 3 stars, advanced (bag_proofs) Here are a couple of little theorems to prove about your definitions about bags above.

Theorem *count_member_nonzero* : ∀ (*s* : *bag*),
  leb 1 (*count* 1 (1 :: *s*)) = *true*.
Proof.
    Admitted.

The following lemma about leb might help you in the next proof.

```
Theorem ble_n_Sn : ∀ n,
  leb n (S n) = true.
Proof.
  intros n. induction n as [| n' IHn'].
  -
    simpl. reflexivity.
  -
    simpl. rewrite IHn'. reflexivity. Qed.
```

Theorem *remove_decreases_count*: ∀ (*s* : *bag*),
  leb (*count* 0 (*remove_one* 0 *s*)) (*count* 0 *s*) = *true*.
Proof.
    Admitted.
    □

Exercise: 3 stars, optionalM (bag_count_sum) Write down an interesting theorem *bag_count_sum* about bags involving the functions count and sum, and prove it. (You may find that the difficulty of the proof depends on how you defined count!) □

Exercise: 4 stars, advancedM (rev_injective) Prove that the rev function is injective – that is,

forall (l1 l2 : natlist), rev l1 = rev l2 -> l1 = l2.
(There is a hard way and an easy way to do this.)

□

5.5 Options

Suppose we want to write a function that returns the nth element of some list. If we give it type nat → **natlist** → nat, then we'll have to choose some number to return when the list is too short...

```
Fixpoint nth_bad (l:natlist) (n:nat) : nat :=
  match l with
  | nil ⇒ 42
  | a :: l' ⇒ match beq_nat n O with
               | true ⇒ a
               | false ⇒ nth_bad l' (pred n)
               end
  end.
```

This solution is not so good: If nth_bad returns 42, we can't tell whether that value actually appears on the input without further processing. A better alternative is to change the return type of nth_bad to include an error value as a possible outcome. We call this type **natoption**.

```
Inductive natoption : Type :=
  | Some : nat → natoption
  | None : natoption.
```

We can then change the above definition of nth_bad to return None when the list is too short and Some a when the list has enough members and a appears at position n. We call this new function nth_error to indicate that it may result in an error.

```
Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match beq_nat n O with
               | true ⇒ Some a
               | false ⇒ nth_error l' (pred n)
               end
  end.
```

```
Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
Proof. reflexivity. Qed.
Example test_nth_error2 : nth_error [4;5;6;7] 3 = Some 7.
Proof. reflexivity. Qed.
Example test_nth_error3 : nth_error [4;5;6;7] 9 = None.
```

```
Proof. reflexivity. Qed.
```

(In the HTML version, the boilerplate proofs of these examples are elided. Click on a box if you want to see one.)

This example is also an opportunity to introduce one more small feature of Coq's programming language: conditional expressions...

```
Fixpoint nth_error' (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ if beq_nat n O then Some a
               else nth_error' l' (pred n)
  end.
```

Coq's conditionals are exactly like those found in any other language, with one small generalization. Since the boolean type is not built in, Coq actually supports conditional expressions over *any* inductively defined type with exactly two constructors. The guard is considered true if it evaluates to the first constructor in the `Inductive` definition and false if it evaluates to the second.

The function below pulls the nat out of a **natoption**, returning a supplied default in the None case.

```
Definition option_elim (d : nat) (o : natoption) : nat :=
  match o with
  | Some n' ⇒ n'
  | None ⇒ d
  end.
```

Exercise: 2 stars (hd_error) Using the same idea, fix the hd function from earlier so we don't have to pass a default element for the nil case.

```
Definition hd_error (l : natlist) : natoption
  . Admitted.

Example test_hd_error1 : hd_error [] = None.
   Admitted.

Example test_hd_error2 : hd_error [1] = Some 1.
   Admitted.

Example test_hd_error3 : hd_error [5;6] = Some 5.
   Admitted.
   □
```

Exercise: 1 star, optional (option_elim_hd) This exercise relates your new hd_error to the old hd.

```
Theorem option_elim_hd : ∀ (l:natlist) (default:nat),
  hd default l = option_elim default (hd_error l).
```

```
Proof.
  Admitted.
  □

End NatList.
```

5.6 Partial Maps

As a final illustration of how data structures can be defined in Coq, here is a simple *partial map* data type, analogous to the map or dictionary data structures found in most programming languages.

First, we define a new inductive datatype **id** to serve as the "keys" of our partial maps.

```
Inductive id : Type :=
 | Id : nat → id.
```

Internally, an **id** is just a number. Introducing a separate type by wrapping each nat with the tag Id makes definitions more readable and gives us the flexibility to change representations later if we wish.

We'll also need an equality test for **id**s:

```
Definition beq_id (x1 x2 : id) :=
  match x1, x2 with
  | Id n1, Id n2 ⇒ beq_nat n1 n2
  end.
```

Exercise: 1 star (beq_id_refl) `Theorem beq_id_refl : ∀ x, true = beq_id x x.`

```
Proof.
  Admitted.
  □
```

Now we define the type of partial maps:

```
Module PartialMap.
Export NatList.

Inductive partial_map : Type :=
  | empty : partial_map
  | record : id → nat → partial_map → partial_map.
```

This declaration can be read: "There are two ways to construct a partial_map: either using the constructor empty to represent an empty partial map, or by applying the constructor record to a key, a value, and an existing partial_map to construct a partial_map with an additional key-to-value mapping."

The update function overrides the entry for a given key in a partial map (or adds a new entry if the given key is not already present).

```
Definition update (d : partial_map)
```

```
                    (x : id) (value : nat)
                    : partial_map :=
  record x value d.
```

Last, the find function searches a partial_map for a given key. It returns None if the key was not found and Some *val* if the key was associated with *val*. If the same key is mapped to multiple values, find will return the first one it encounters.

```
Fixpoint find (x : id) (d : partial_map) : natoption :=
  match d with
  | empty ⇒ None
  | record y v d' ⇒ if beq_id x y
                      then Some v
                      else find x d'
  end.
```

Exercise: 1 star (update_eq) `Theorem` *update_eq* :

```
  ∀ (d : partial_map) (x : id) (v: nat),
    find x (update d x v) = Some v.
Proof.
  Admitted.
```

☐

Exercise: 1 star (update_neq) `Theorem` *update_neq* :

```
  ∀ (d : partial_map) (x y : id) (o: nat),
    beq_id x y = false → find x (update d y o) = find x d.
Proof.
  Admitted.
```

☐ `End` *PartialMap*.

Exercise: 2 starsM (baz_num_elts) Consider the following inductive definition:

```
Inductive baz : Type :=
  | Baz1 : baz → baz
  | Baz2 : baz → bool → baz.
```

How *many* elements does the type **baz** have? (Answer in English or the natural language of your choice.)

☐

$Date : 2016-12-1723:53:20-0500 (Sat, 17 Dec 2016)$

Chapter 6

Library Top.Poly

6.1 Poly: Polymorphism and Higher-Order Functions

```
Require Export Lists.
```

6.2 Polymorphism

In this chapter we continue our development of basic concepts of functional programming. The critical new ideas are *polymorphism* (abstracting functions over the types of the data they manipulate) and *higher-order functions* (treating functions as data). We begin with polymorphism.

6.2.1 Polymorphic Lists

For the last couple of chapters, we've been working just with lists of numbers. Obviously, interesting programs also need to be able to manipulate lists with elements from other types – lists of strings, lists of booleans, lists of lists, etc. We *could* just define a new inductive datatype for each of these, for example...

```
Inductive boollist : Type :=
  | bool_nil : boollist
  | bool_cons : bool → boollist → boollist.
```

... but this would quickly become tedious, partly because we have to make up different constructor names for each datatype, but mostly because we would also need to define new versions of all our list manipulating functions (length, rev, etc.) for each new datatype definition.

To avoid all this repetition, Coq supports *polymorphic* inductive type definitions. For example, here is a *polymorphic list* datatype.

```
Inductive list (X:Type) : Type :=
```

```
| nil : list X
| cons : X → list X → list X.
```

This is exactly like the definition of **natlist** from the previous chapter, except that the nat argument to the cons constructor has been replaced by an arbitrary type X, a binding for X has been added to the header, and the occurrences of **natlist** in the types of the constructors have been replaced by **list** X. (We can re-use the constructor names nil and cons because the earlier definition of **natlist** was inside of a `Module` definition that is now out of scope.)

What sort of thing is **list** itself? One good way to think about it is that **list** is a *function* from `Types` to `Inductive` definitions; or, to put it another way, **list** is a function from `Types` to `Types`. For any particular type X, the type **list** X is an `Inductive`ly defined set of lists whose elements are of type X.

With this definition, when we use the constructors nil and cons to build lists, we need to tell Coq the type of the elements in the lists we are building – that is, nil and cons are now *polymorphic constructors*. Observe the types of these constructors:

```
Check nil.
Check cons.
```

(Side note on notation: In .v files, the "forall" quantifier is spelled out in letters. In the generated HTML files and in the way various IDEs show .v files (with certain settings of their display controls), ∀ is usually typeset as the usual mathematical "upside down A," but you'll still see the spelled-out "forall" in a few places. This is just a quirk of typesetting: there is no difference in meaning.)

The "∀ X" in these types can be read as an additional argument to the constructors that determines the expected types of the arguments that follow. When nil and cons are used, these arguments are supplied in the same way as the others. For example, the list containing 2 and 1 is written like this:

```
Check (cons nat 2 (cons nat 1 (nil nat))).
```

(We've written nil and cons explicitly here because we haven't yet defined the [] and :: notations for the new version of lists. We'll do that in a bit.)

We can now go back and make polymorphic versions of all the list-processing functions that we wrote before. Here is `repeat`, for example:

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 ⇒ nil X
  | S count' ⇒ cons X x (repeat X x count')
  end.
```

As with nil and cons, we can use `repeat` by applying it first to a type and then to its list argument:

```
Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
Proof. reflexivity. Qed.
```

To use `repeat` to build other kinds of lists, we simply instantiate it with an appropriate type parameter:

```
Example test_repeat2 :
  repeat bool false 1 = cons bool false (nil bool).
Proof. reflexivity. Qed.

Module MumbleGrumble.
```

Exercise: 2 starsM (mumble_grumble) Consider the following two inductively defined types.

```
Inductive mumble : Type :=
  | a : mumble
  | b : mumble → nat → mumble
  | c : mumble.

Inductive grumble (X:Type) : Type :=
  | d : mumble → grumble X
  | e : X → grumble X.
```

Which of the following are well-typed elements of **grumble** X for some type X?

- d (b a 5)
- d **mumble** (b a 5)
- d **bool** (b a 5)
- e **bool** true
- e **mumble** (b c 0)
- e **bool** (b c 0)
- c

□

```
End MumbleGrumble.
```

Type Annotation Inference

Let's write the definition of `repeat` again, but this time we won't specify the types of any of the arguments. Will Coq still accept it?

```
Fixpoint repeat' X x count : list X :=
  match count with
  | 0 ⇒ nil X
  | S count' ⇒ cons X x (repeat' X x count')
```

```
  end.
```

Indeed it will. Let's see what type Coq has assigned to repeat':

```
Check repeat'.
Check repeat.
```

It has exactly the same type type as repeat. Coq was able to use *type inference* to deduce what the types of X, x, and count must be, based on how they are used. For example, since X is used as an argument to cons, it must be a Type, since cons expects a Type as its first argument; matching count with 0 and S means it must be a nat; and so on.

This powerful facility means we don't always have to write explicit type annotations everywhere, although explicit type annotations are still quite useful as documentation and sanity checks, so we will continue to use them most of the time. You should try to find a balance in your own code between too many type annotations (which can clutter and distract) and too few (which forces readers to perform type inference in their heads in order to understand your code).

Type Argument Synthesis

To use a polymorphic function, we need to pass it one or more types in addition to its other arguments. For example, the recursive call in the body of the repeat function above must pass along the type X. But since the second argument to repeat is an element of X, it seems entirely obvious that the first argument can only be X – why should we have to write it explicitly?

Fortunately, Coq permits us to avoid this kind of redundancy. In place of any type argument we can write the "implicit argument" _, which can be read as "Please try to figure out for yourself what belongs here." More precisely, when Coq encounters a _, it will attempt to *unify* all locally available information – the type of the function being applied, the types of the other arguments, and the type expected by the context in which the application appears – to determine what concrete type should replace the _.

This may sound similar to type annotation inference – indeed, the two procedures rely on the same underlying mechanisms. Instead of simply omitting the types of some arguments to a function, like

repeat' X x count : list X :=

we can also replace the types with _

repeat' (X : _) (x : _) (count : _) : list X :=

to tell Coq to attempt to infer the missing information.

Using implicit arguments, the count function can be written like this:

```
Fixpoint repeat'' X x count : list X :=
  match count with
  | 0 => nil _
  | S count' => cons _ x (repeat'' _ x count')
  end.
```

In this instance, we don't save much by writing _ instead of X. But in many cases the difference in both keystrokes and readability is nontrivial. For example, suppose we want to write down a list containing the numbers 1, 2, and 3. Instead of writing this...

```
Definition list123 :=
  cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).
```

...we can use argument synthesis to write this:

```
Definition list123' :=
  cons _ 1 (cons _ 2 (cons _ 3 (nil _))).
```

Implicit Arguments

We can go further and even avoid writing _'s in most cases by telling Coq *always* to infer the type argument(s) of a given function. The *Arguments* directive specifies the name of the function (or constructor) and then lists its argument names, with curly braces around any arguments to be treated as implicit. (If some arguments of a definition don't have a name, as is often the case for constructors, they can be marked with a wildcard pattern _.)

```
Arguments nil {X}.
Arguments cons {X} _ _.
Arguments repeat {X} x count.
```

Now, we don't have to supply type arguments at all:

```
Definition list123'' := cons 1 (cons 2 (cons 3 nil)).
```

Alternatively, we can declare an argument to be implicit when defining the function itself, by surrounding it in curly braces instead of parens. For example:

```
Fixpoint repeat''' {X : Type} (x : X) (count : nat) : list X :=
  match count with
  | 0 ⇒ nil
  | S count' ⇒ cons x (repeat''' x count')
  end.
```

(Note that we didn't even have to provide a type argument to the recursive call to repeat'''; indeed, it would be invalid to provide one!)

We will use the latter style whenever possible, but we will continue to use use explicit *Argument* declarations for Inductive constructors. The reason for this is that marking the parameter of an inductive type as implicit causes it to become implicit for the type itself, not just for its constructors. For instance, consider the following alternative definition of the **list** type:

```
Inductive list' {X:Type} : Type :=
  | nil' : list'
  | cons' : X → list' → list'.
```

Because X is declared as implicit for the *entire* inductive definition including **list'** itself, we now have to write just **list'** whether we are talking about lists of numbers or booleans or anything else, rather than **list'** nat or **list' bool** or whatever; this is a step too far.

Let's finish by re-implementing a few other standard list functions on our new polymorphic lists...

```
Fixpoint app {X : Type} (l1 l2 : list X)
             : (list X) :=
  match l1 with
  | nil ⇒ l2
  | cons h t ⇒ cons h (app t l2)
  end.

Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil ⇒ nil
  | cons h t ⇒ app (rev t) (cons h nil)
  end.

Fixpoint length {X : Type} (l : list X) : nat :=
  match l with
  | nil ⇒ 0
  | cons _ l' ⇒ S (length l')
  end.

Example test_rev1 :
  rev (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
Proof. reflexivity. Qed.

Example test_rev2:
  rev (cons true nil) = cons true nil.
Proof. reflexivity. Qed.

Example test_length1: length (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof. reflexivity. Qed.
```

Supplying Type Arguments Explicitly

One small problem with declaring arguments `Implicit` is that, occasionally, Coq does not have enough local information to determine a type argument; in such cases, we need to tell Coq that we want to give the argument explicitly just this time. For example, suppose we write this:

```
Fail Definition mynil := nil.
```

(The *Fail* qualifier that appears before `Definition` can be used with *any* command, and is used to ensure that that command indeed fails when executed. If the command does fail, Coq prints the corresponding error message, but continues processing the rest of the file.)

Here, Coq gives us an error because it doesn't know what type argument to supply to nil. We can help it by providing an explicit type declaration (so that Coq has more information available when it gets to the "application" of nil):

```
Definition mynil : list nat := nil.
```

Alternatively, we can force the implicit arguments to be explicit by prefixing the function name with @.

```
Check @nil.

Definition mynil' := @nil nat.
```

Using argument synthesis and implicit arguments, we can define convenient notation for lists, as before. Since we have made the constructor type arguments implicit, Coq will know to automatically infer these when we use the notations.

```
Notation "x :: y" := (cons x y)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y)
                     (at level 60, right associativity).
```

Now lists can be written just the way we'd hope:

```
Definition list123''' := [1; 2; 3].
```

Exercises

Exercise: 2 stars, optional (poly_exercises) Here are a few simple exercises, just like ones in the Lists chapter, for practice with polymorphism. Complete the proofs below.

```
Theorem app_nil_r : ∀ (X:Type), ∀ l:list X,
  l ++ [] = l.
Proof.
  Admitted.

Theorem app_assoc : ∀ A (l m n:list A),
  l ++ m ++ n = (l ++ m) ++ n.
Proof.
  Admitted.

Lemma app_length : ∀ (X:Type) (l1 l2 : list X),
  length (l1 ++ l2) = length l1 + length l2.
Proof.
  Admitted.
```

□

Exercise: 2 stars, optional (more_poly_exercises) Here are some slightly more interesting ones...

```
Theorem rev_app_distr: ∀ X (l1 l2 : list X),
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
   Admitted.

Theorem rev_involutive : ∀ X : Type, ∀ l : list X,
  rev (rev l) = l.
Proof.
   Admitted.
```

□

6.2.2 Polymorphic Pairs

Following the same pattern, the type definition we gave in the last chapter for pairs of numbers can be generalized to *polymorphic pairs*, often called *products*:

```
Inductive prod (X Y : Type) : Type :=
| pair : X → Y → prod X Y.

Arguments pair {X} {Y} _ _.
```

As with lists, we make the type arguments implicit and define the familiar concrete notation.

```
Notation "( x , y )" := (pair x y).
```

We can also use the `Notation` mechanism to define the standard notation for product *types*:

```
Notation "X * Y" := (prod X Y) : type_scope.
```

(The annotation : *type_scope* tells Coq that this abbreviation should only be used when parsing types. This avoids a clash with the multiplication symbol.)

It is easy at first to get (x,y) and X×Y confused. Remember that (x,y) is a *value* built from two other values, while X×Y is a *type* built from two other types. If x has type X and y has type Y, then (x,y) has type X×Y.

The first and second projection functions now look pretty much as they would in any functional programming language.

```
Definition fst {X Y : Type} (p : X × Y) : X :=
  match p with
  | (x, y) ⇒ x
  end.

Definition snd {X Y : Type} (p : X × Y) : Y :=
  match p with
  | (x, y) ⇒ y
```

```
  end.
```

The following function takes two lists and combines them into a list of pairs. In other functional languages, it is often called *zip*; we call it combine for consistency with Coq's standard library.

```
Fixpoint combine {X Y : Type} (lx : list X) (ly : list Y)
           : list (X×Y) :=
  match lx, ly with
  | [], _ ⇒ []
  | _, [] ⇒ []
  | x :: tx, y :: ty ⇒ (x, y) :: (combine tx ty)
  end.
```

Exercise: 1 star, optionalM (combine_checks) Try answering the following questions on paper and checking your answers in coq:

- What is the type of combine (i.e., what does Check @combine print?)
- What does

  Compute (combine 1;2 false;false;true;true).

  print?

□

Exercise: 2 stars, recommended (split) The function split is the right inverse of combine: it takes a list of pairs and returns a pair of lists. In many functional languages, it is called *unzip*.

Fill in the definition of split below. Make sure it passes the given unit test.

```
Fixpoint split {X Y : Type} (l : list (X×Y))
               : (list X) × (list Y)
  . Admitted.

Example test_split:
  split [(1,false);(2,false)] = ([1;2],[false;false]).
Proof.
   Admitted.
```

□

6.2.3 Polymorphic Options

One last polymorphic type for now: *polymorphic options*, which generalize **natoption** from the previous chapter:

```
Inductive option (X:Type) : Type :=
```

```
  | Some : X → option X
  | None : option X.
Arguments Some {X} _.
Arguments None {X}.
```

We can now rewrite the nth_error function so that it works with any type of lists.

```
Fixpoint nth_error {X : Type} (l : list X) (n : nat)
                   : option X :=
  match l with
  | [] ⇒ None
  | a :: l' ⇒ if beq_nat n O then Some a else nth_error l' (pred n)
  end.
Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
Proof. reflexivity. Qed.
Example test_nth_error2 : nth_error [[1];[2]] 1 = Some [2].
Proof. reflexivity. Qed.
Example test_nth_error3 : nth_error [true] 2 = None.
Proof. reflexivity. Qed.
```

Exercise: 1 star, optional (hd_error_poly) Complete the definition of a polymorphic version of the hd_error function from the last chapter. Be sure that it passes the unit tests below.

```
Definition hd_error {X : Type} (l : list X) : option X
  . Admitted.
```

Once again, to force the implicit arguments to be explicit, we can use @ before the name of the function.

```
Check @hd_error.
Example test_hd_error1 : hd_error [1;2] = Some 1.
   Admitted.
Example test_hd_error2 : hd_error [[1];[2]] = Some [1].
   Admitted.
```

□

6.3 Functions as Data

Like many other modern programming languages – including all functional languages (ML, Haskell, Scheme, Scala, Clojure, etc.) – Coq treats functions as first-class citizens, allowing them to be passed as arguments to other functions, returned as results, stored in data structures, etc.

6.3.1 Higher-Order Functions

Functions that manipulate other functions are often called *higher-order* functions. Here's a simple one:

```
Definition doit3times {X:Type} (f:X→X) (n:X) : X :=
  f (f (f n)).
```

The argument f here is itself a function (from X to X); the body of doit3times applies f three times to some value n.

```
Check @doit3times.
```

```
Example test_doit3times: doit3times minustwo 9 = 3.
Proof. reflexivity. Qed.
```

```
Example test_doit3times': doit3times negb true = false.
Proof. reflexivity. Qed.
```

6.3.2 Filter

Here is a more useful higher-order function, taking a list of Xs and a *predicate* on X (a function from X to **bool**) and "filtering" the list, returning a new list containing just those elements for which the predicate returns true.

```
Fixpoint filter {X:Type} (test: X→bool) (l:list X)
                : (list X) :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ if test h then h :: (filter test t)
                        else filter test t
  end.
```

For example, if we apply filter to the predicate evenb and a list of numbers l, it returns a list containing just the even members of l.

```
Example test_filter1: filter evenb [1;2;3;4] = [2;4].
Proof. reflexivity. Qed.
```

```
Definition length_is_1 {X : Type} (l : list X) : bool :=
  beq_nat (length l) 1.
```

```
Example test_filter2:
    filter length_is_1
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

We can use filter to give a concise version of the countoddmembers function from the Lists chapter.

```
Definition countoddmembers' (l:list nat) : nat :=
```

length (*filter oddb l*).

```
Example test_countoddmembers'1: countoddmembers' [1;0;3;1;4;5] = 4.
Proof. reflexivity. Qed.
Example test_countoddmembers'2: countoddmembers' [0;2;4] = 0.
Proof. reflexivity. Qed.
Example test_countoddmembers'3: countoddmembers' nil = 0.
Proof. reflexivity. Qed.
```

6.3.3 Anonymous Functions

It is arguably a little sad, in the example just above, to be forced to define the function length_is_1 and give it a name just to be able to pass it as an argument to filter, since we will probably never use it again. Moreover, this is not an isolated example: when using higher-order functions, we often want to pass as arguments "one-off" functions that we will never use again; having to give each of these functions a name would be tedious.

Fortunately, there is a better way. We can construct a function "on the fly" without declaring it at the top level or giving it a name.

```
Example test_anon_fun':
  doit3times (fun n ⇒ n × n) 2 = 256.
Proof. reflexivity. Qed.
```

The expression (fun n ⇒ n × n) can be read as "the function that, given a number n, yields n × n."

Here is the filter example, rewritten to use an anonymous function.

```
Example test_filter2':
    filter (fun l ⇒ beq_nat (length l) 1)
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

Exercise: 2 stars (filter_even_gt7) Use filter (instead of Fixpoint) to write a Coq function filter_even_gt7 that takes a list of natural numbers as input and returns a list of just those that are even and greater than 7.

```
Definition filter_even_gt7 (l : list nat) : list nat
  . Admitted.

Example test_filter_even_gt7_1 :
  filter_even_gt7 [1;2;6;9;10;3;12;8] = [10;12;8].
   Admitted.

Example test_filter_even_gt7_2 :
  filter_even_gt7 [5;2;6;19;129] = [].
   Admitted.
```

☐

Exercise: 3 stars (partition) Use filter to write a Coq function partition:

partition : forall X : Type, (X -> bool) -> list X -> list X * list X

Given a set X, a test function of type X → **bool** and a **list** X, partition should return a pair of lists. The first member of the pair is the sublist of the original list containing the elements that satisfy the test, and the second is the sublist containing those that fail the test. The order of elements in the two sublists should be the same as their order in the original list.

```
Definition partition {X : Type}
                     (test : X → bool)
                     (l : list X)
                   : list X × list X
  . Admitted.

Example test_partition1: partition oddb [1;2;3;4;5] = ([1;3;5], [2;4]).
  Admitted.
Example test_partition2: partition (fun x ⇒ false) [5;9;0] = ([], [5;9;0]).
  Admitted.
```

☐

6.3.4 Map

Another handy higher-order function is called map.

```
Fixpoint map {X Y:Type} (f:X→Y) (l:list X) : (list Y) :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ (f h) :: (map f t)
  end.
```

It takes a function f and a list l = [*n1*, *n2*, *n3*, ...] and returns the list [f *n1*, f *n2*, f *n3*,...] , where f has been applied to each element of l in turn. For example:

```
Example test_map1: map (fun x ⇒ plus 3 x) [2;0;2] = [5;3;5].
Proof. reflexivity. Qed.
```

The element types of the input and output lists need not be the same, since map takes *two* type arguments, X and Y; it can thus be applied to a list of numbers and a function from numbers to booleans to yield a list of booleans:

```
Example test_map2:
  map oddb [2;1;2;5] = [false;true;false;true].
Proof. reflexivity. Qed.
```

It can even be applied to a list of numbers and a function from numbers to *lists* of booleans to yield a *list of lists* of booleans:

```
Example test_map3:
    map (fun n ⇒ [evenb n;oddb n]) [2;1;2;5]
  = [[true;false];[false;true];[true;false];[false;true]].
```

```
Proof. reflexivity. Qed.
```

Exercises

Exercise: 3 stars (map_rev) Show that map and rev commute. You may need to define an auxiliary lemma.

```
Theorem map_rev : ∀ (X Y : Type) (f : X → Y) (l : list X),
  map f (rev l) = rev (map f l).
Proof.
   Admitted.
```
☐

Exercise: 2 stars, recommended (flat_map) The function map maps a **list** X to a **list** Y using a function of type X → Y. We can define a similar function, flat_map, which maps a **list** X to a **list** Y using a function f of type X → **list** Y. Your definition should work by 'flattening' the results of f, like so:

flat_map (fun n => n;n+1;n+2) 1;5;10 = 1; 2; 3; 5; 6; 7; 10; 11; 12.

```
Fixpoint flat_map {X Y:Type} (f:X → list Y) (l:list X)
                   : (list Y)
  . Admitted.

Example test_flat_map1:
  flat_map (fun n ⇒ [n;n;n]) [1;5;4]
  = [1; 1; 1; 5; 5; 5; 4; 4; 4].
   Admitted.
```
☐

Lists are not the only inductive type that we can write a map function for. Here is the definition of map for the **option** type:

```
Definition option_map {X Y : Type} (f : X → Y) (xo : option X)
                      : option Y :=
  match xo with
    | None ⇒ None
    | Some x ⇒ Some (f x)
  end.
```

Exercise: 2 stars, optional (implicit_args) The definitions and uses of filter and map use implicit arguments in many places. Replace the curly braces around the implicit arguments with parentheses, and then fill in explicit type parameters where necessary and use Coq to check that you've done so correctly. (This exercise is not to be turned in; it is probably easiest to do it on a *copy* of this file that you can throw away afterwards.) ☐

6.3.5 Fold

An even more powerful higher-order function is called `fold`. This function is the inspiration for the "*reduce*" operation that lies at the heart of Google's map/reduce distributed programming framework.

```
Fixpoint fold {X Y:Type} (f: X→Y→Y) (l:list X) (b:Y)
                         : Y :=
  match l with
  | nil ⇒ b
  | h :: t ⇒ f h (fold f t b)
  end.
```

Intuitively, the behavior of the `fold` operation is to insert a given binary operator f between every pair of elements in a given list. For example, `fold plus` [1;2;3;4] intuitively means 1+2+3+4. To make this precise, we also need a "starting element" that serves as the initial second input to f. So, for example,

fold plus 1;2;3;4 0

yields

1 + (2 + (3 + (4 + 0))).

Some more examples:

```
Check (fold andb).

Example fold_example1 :
  fold mult [1;2;3;4] 1 = 24.
Proof. reflexivity. Qed.

Example fold_example2 :
  fold andb [true;true;false;true] true = false.
Proof. reflexivity. Qed.

Example fold_example3 :
  fold app [[1];[];[2;3];[4]] [] = [1;2;3;4].
Proof. reflexivity. Qed.
```

Exercise: 1 star, advancedM (fold_types_different) Observe that the type of `fold` is parameterized by *two* type variables, X and Y, and the parameter f is a binary operator that takes an X and a Y and returns a Y. Can you think of a situation where it would be useful for X and Y to be different?

□

6.3.6 Functions That Construct Functions

Most of the higher-order functions we have talked about so far take functions as arguments. Let's look at some examples that involve *returning* functions as the results of other functions.

To begin, here is a function that takes a value x (drawn from some type X) and returns a function from nat to X that yields x whenever it is called, ignoring its nat argument.

```
Definition constfun {X: Type} (x: X) : nat→X :=
  fun (k:nat) ⇒ x.
```

```
Definition ftrue := constfun true.
```

```
Example constfun_example1 : ftrue 0 = true.
Proof. reflexivity. Qed.
```

```
Example constfun_example2 : (constfun 5) 99 = 5.
Proof. reflexivity. Qed.
```

In fact, the multiple-argument functions we have already seen are also examples of passing functions as data. To see why, recall the type of plus.

```
Check plus.
```

Each → in this expression is actually a *binary* operator on types. This operator is *right-associative*, so the type of plus is really a shorthand for nat → (nat → nat) – i.e., it can be read as saying that "plus is a one-argument function that takes a nat and returns a one-argument function that takes another nat and returns a nat." In the examples above, we have always applied plus to both of its arguments at once, but if we like we can supply just the first. This is called *partial application*.

```
Definition plus3 := plus 3.
Check plus3.
```

```
Example test_plus3 : plus3 4 = 7.
Proof. reflexivity. Qed.
Example test_plus3' : doit3times plus3 0 = 9.
Proof. reflexivity. Qed.
Example test_plus3'' : doit3times (plus 3) 0 = 9.
Proof. reflexivity. Qed.
```

6.4 Additional Exercises

```
Module Exercises.
```

Exercise: 2 stars (fold_length) Many common functions on lists can be implemented in terms of fold. For example, here is an alternative definition of length:

```
Definition fold_length {X : Type} (l : list X) : nat :=
  fold (fun _ n ⇒ S n) l 0.
```

```
Example test_fold_length1 : fold_length [4;7;0] = 3.
Proof. reflexivity. Qed.
```

Prove the correctness of fold_length.

```
Theorem fold_length_correct : ∀ X (l : list X),
  fold_length l = length l.
   Admitted.
```
□

Exercise: 3 starsM (fold_map) We can also define `map` in terms of `fold`. Finish `fold_map` below.

```
Definition fold_map {X Y:Type} (f : X → Y) (l : list X) : list Y
  . Admitted.
```

Write down a theorem *fold_map_correct* in Coq stating that `fold_map` is correct, and prove it.

□

Exercise: 2 stars, advanced (currying) In Coq, a function `f : A → B → C` really has the type `A → (B → C)`. That is, if you give `f` a value of type `A`, it will give you function *f'* : `B → C`. If you then give *f'* a value of type `B`, it will return a value of type `C`. This allows for partial application, as in `plus3`. Processing a list of arguments with functions that return functions is called *currying*, in honor of the logician Haskell Curry.

Conversely, we can reinterpret the type `A → B → C` as `(A × B) → C`. This is called *uncurrying*. With an uncurried binary function, both arguments must be given at once as a pair; there is no partial application.

We can define currying as follows:

```
Definition prod_curry {X Y Z : Type}
  (f : X × Y → Z) (x : X) (y : Y) : Z := f (x, y).
```

As an exercise, define its inverse, `prod_uncurry`. Then prove the theorems below to show that the two are inverses.

```
Definition prod_uncurry {X Y Z : Type}
  (f : X → Y → Z) (p : X × Y) : Z
  . Admitted.
```

As a (trivial) example of the usefulness of currying, we can use it to shorten one of the examples that we saw above:

```
Example test_map2: map (fun x ⇒ plus 3 x) [2;0;2] = [5;3;5].
Proof. reflexivity. Qed.
```

Thought exercise: before running the following commands, can you calculate the types of `prod_curry` and `prod_uncurry`?

```
Check @prod_curry.
Check @prod_uncurry.
```

```
Theorem uncurry_curry : ∀ (X Y Z : Type)
                        (f : X → Y → Z)
```

$x\ y,$
$prod\_curry\ (prod\_uncurry\ f)\ x\ y = f\ x\ y.$

`Proof.`

Admitted.

`Theorem` $curry\_uncurry : \forall\ (X\ Y\ Z : \texttt{Type})$
$(f : (X \times Y) \rightarrow Z)\ (p : X \times Y),$
$prod\_uncurry\ (prod\_curry\ f)\ p = f\ p.$

`Proof.`

Admitted.

☐

Exercise: 2 stars, advancedM (nth_error_informal) Recall the definition of the nth_error function:

Fixpoint nth_error {X : Type} (l : list X) (n : nat) : option X := match l with | ☐ => None | a :: l' => if beq_nat n O then Some a else nth_error l' (pred n) end.

Write an informal proof of the following theorem:

forall X n l, length l = n -> @nth_error X l n = None

☐

Exercise: 4 stars, advanced (church_numerals) This exercise explores an alternative way of defining natural numbers, using the so-called *Church numerals*, named after mathematician Alonzo Church. We can represent a natural number n as a function that takes a function f as a parameter and returns f iterated n times.

`Module` *Church.*

`Definition` $nat := \forall\ X : \texttt{Type},\ (X \rightarrow X) \rightarrow X \rightarrow X.$

Let's see how to write some numbers with this notation. Iterating a function once should be the same as just applying it. Thus:

`Definition` $one : nat :=$
`fun` $(X : \texttt{Type})\ (f : X \rightarrow X)\ (x : X) \Rightarrow f\ x.$

Similarly, two should apply f twice to its argument:

`Definition` $two : nat :=$
`fun` $(X : \texttt{Type})\ (f : X \rightarrow X)\ (x : X) \Rightarrow f\ (f\ x).$

Defining zero is somewhat trickier: how can we "apply a function zero times"? The answer is actually simple: just return the argument untouched.

`Definition` $zero : nat :=$
`fun` $(X : \texttt{Type})\ (f : X \rightarrow X)\ (x : X) \Rightarrow x.$

More generally, a number n can be written as fun X f x ⇒ f (f ... (f x) ...), with n occurrences of f. Notice in particular how the doit3times function we've defined previously is actually just the Church representation of 3.

```
Definition three : nat := @doit3times.
```

Complete the definitions of the following functions. Make sure that the corresponding unit tests pass by proving them with `reflexivity`.

Successor of a natural number:

```
Definition succ (n : nat) : nat
  . Admitted.

Example succ_1 : succ zero = one.
Proof. Admitted.

Example succ_2 : succ one = two.
Proof. Admitted.

Example succ_3 : succ two = three.
Proof. Admitted.
```

Addition of two natural numbers:

```
Definition plus (n m : nat) : nat
  . Admitted.

Example plus_1 : plus zero one = one.
Proof. Admitted.

Example plus_2 : plus two three = plus three two.
Proof. Admitted.

Example plus_3 :
  plus (plus two two) three = plus one (plus three three).
Proof. Admitted.
```

Multiplication:

```
Definition mult (n m : nat) : nat
  . Admitted.

Example mult_1 : mult one one = one.
Proof. Admitted.

Example mult_2 : mult zero (plus three three) = zero.
Proof. Admitted.

Example mult_3 : mult two three = plus three three.
Proof. Admitted.
```

Exponentiation:

(*Hint*: Polymorphism plays a crucial role here. However, choosing the right type to iterate over can be tricky. If you hit a "Universe inconsistency" error, try iterating over a different type: `nat` itself is usually problematic.)

```
Definition exp (n m : nat) : nat
  . Admitted.
```

Example *exp_1* : *exp two two* = *plus two two*.
Proof. *Admitted*.

Example *exp_2* : *exp three two* = *plus* (*mult two* (*mult two two*)) *one*.
Proof. *Admitted*.

Example *exp_3* : *exp three zero* = *one*.
Proof. *Admitted*.

End *Church*.

$\square$

End *Exercises*.

$Date : 2016 - 12 - 1723 : 53 : 20 - 0500(Sat, 17Dec2016)$

Chapter 7

Library Top.Tactics

7.1 Tactics: More Basic Tactics

This chapter introduces several additional proof strategies and tactics that allow us to begin proving more interesting properties of functional programs. We will see:

- how to use auxiliary lemmas in both ”forward-style“ and ”backward-style“ proofs;
- how to reason about data constructors (in particular, how to use the fact that they are injective and disjoint);
- how to strengthen an induction hypothesis (and when such strengthening is required); and
- more details on how to reason by case analysis.

```
Require Export Poly.
```

7.2 The apply Tactic

We often encounter situations where the goal to be proved is *exactly* the same as some hypothesis in the context or some previously proved lemma.

```
Theorem silly1 : ∀ (n m o p : nat),
     n = m →
     [n;o] = [n;p] →
     [n;o] = [m;p].
Proof.
  intros n m o p eq1 eq2.
  rewrite ← eq1.
```

Here, we could finish with ”`rewrite → eq2. reflexivity.`“ as we have done several times before. We can achieve the same effect in a single step by using the `apply` tactic instead:

```
  apply eq2. Qed.
```

The apply tactic also works with *conditional* hypotheses and lemmas: if the statement being applied is an implication, then the premises of this implication will be added to the list of subgoals needing to be proved.

```
Theorem silly2 : ∀ (n m o p : nat),
     n = m →
     (∀ (q r : nat), q = r → [q;o] = [r;p]) →
     [n;o] = [m;p].
Proof.
  intros n m o p eq1 eq2.
  apply eq2. apply eq1. Qed.
```

You may find it instructive to experiment with this proof and see if there is a way to complete it using just rewrite instead of apply.

Typically, when we use apply *H*, the statement *H* will begin with a ∀ that binds some *universal variables.* When Coq matches the current goal against the conclusion of *H*, it will try to find appropriate values for these variables. For example, when we do apply *eq2* in the following proof, the universal variable *q* in *eq2* gets instantiated with n and r gets instantiated with m.

```
Theorem silly2a : ∀ (n m : nat),
     (n,n) = (m,m) →
     (∀ (q r : nat), (q,q) = (r,r) → [q] = [r]) →
     [n] = [m].
Proof.
  intros n m eq1 eq2.
  apply eq2. apply eq1. Qed.
```

Exercise: 2 stars, optional (silly_ex) Complete the following proof without using simpl.

```
Theorem silly_ex :
     (∀ n, evenb n = true → oddb (S n) = true) →
     evenb 3 = true →
     oddb 4 = true.
Proof.
   Admitted.
```

□

To use the apply tactic, the (conclusion of the) fact being applied must match the goal exactly – for example, apply will not work if the left and right sides of the equality are swapped.

```
Theorem silly3_firsttry : ∀ (n : nat),
     true = beq_nat n 5 →
     beq_nat (S (S n)) 7 = true.
```

```
Proof.
  intros n H.
  simpl.
```

Here we cannot use apply directly, but we can use the symmetry tactic, which switches the left and right sides of an equality in the goal.

```
  symmetry.
  simpl.   apply H. Qed.
```

Exercise: 3 stars (apply_exercise1) (*Hint*: You can use apply with previously defined lemmas, not just hypotheses in the context. Remember that Search is your friend.)

```
Theorem rev_exercise1 : ∀ (l l' : list nat),
     l = rev l' →
     l' = rev l.
Proof.
   Admitted.
```

☐

Exercise: 1 star, optionalM (apply_rewrite) Briefly explain the difference between the tactics apply and rewrite. What are the situations where both can usefully be applied?

☐

7.3 The apply ... with ... Tactic

The following silly example uses two rewrites in a row to get from [a,b] to [e,f].

```
Example trans_eq_example : ∀ (a b c d e f : nat),
     [a;b] = [c;d] →
     [c;d] = [e;f] →
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  rewrite → eq1. rewrite → eq2. reflexivity. Qed.
```

Since this is a common pattern, we might like to pull it out as a lemma recording, once and for all, the fact that equality is transitive.

```
Theorem trans_eq : ∀ (X:Type) (n m o : X),
  n = m → m = o → n = o.
Proof.
  intros X n m o eq1 eq2. rewrite → eq1. rewrite → eq2.
  reflexivity. Qed.
```

Now, we should be able to use trans_eq to prove the above example. However, to do this we need a slight refinement of the apply tactic.

```
Example trans_eq_example' : ∀ (a b c d e f : nat),
     [a;b] = [c;d] →
     [c;d] = [e;f] →
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
```

If we simply tell Coq `apply trans_eq` at this point, it can tell (by matching the goal against the conclusion of the lemma) that it should instantiate X with [nat], n with [a,b], and *o* with [e,f]. However, the matching process doesn't determine an instantiation for m: we have to supply one explicitly by adding `with` (m:=[c,d]) to the invocation of `apply`.

```
  apply trans_eq with (m:=[c;d]).
  apply eq1. apply eq2. Qed.
```

Actually, we usually don't have to include the name m in the `with` clause; Coq is often smart enough to figure out which instantiation we're giving. We could instead write: `apply trans_eq with [c;d]`.

Exercise: 3 stars, optional (apply_with_exercise)
```
Example trans_eq_exercise : ∀ (n m o p : nat),
     m = (minustwo o) →
     (n + p) = m →
     (n + p) = (minustwo o).
Proof.
   Admitted.
```
☐

7.4 The `inversion` Tactic

Recall the definition of natural numbers:

Inductive nat : Type := | O : nat | S : nat -> nat.

It is obvious from this definition that every number has one of two forms: either it is the constructor O or it is built by applying the constructor S to another number. But there is more here than meets the eye: implicit in the definition (and in our informal understanding of how datatype declarations work in other programming languages) are two more facts:

- The constructor S is *injective*. That is, if S n = S m, it must be the case that n = m.
- The constructors O and S are *disjoint*. That is, O is not equal to S n for any n.

Similar principles apply to all inductively defined types: all constructors are injective, and the values built from distinct constructors are never equal. For lists, the cons constructor is injective and nil is different from every non-empty list. For booleans, true and false are different. (Since neither true nor false take any arguments, their injectivity is not interesting.) And so on.

Coq provides a tactic called `inversion` that allows us to exploit these principles in proofs. To see how to use it, let's show explicitly that the S constructor is injective:

```
Theorem S_injective : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H.
```

By writing `inversion` H at this point, we are asking Coq to generate all equations that it can infer from H as additional hypotheses, replacing variables in the goal as it goes. In the present example, this amounts to adding a new hypothesis *H1* : n = m and replacing n by m in the goal.

```
  inversion H.
  reflexivity.
Qed.
```

Here's a more interesting example that shows how multiple equations can be derived at once.

```
Theorem inversion_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
Proof.
  intros n m o H. inversion H. reflexivity. Qed.
```

We can name the equations that `inversion` generates with an `as` ... clause:

```
Theorem inversion_ex2 : ∀ (n m : nat),
  [n] = [m] →
  n = m.
Proof.
  intros n m H. inversion H as [Hnm]. reflexivity. Qed.
```

Exercise: 1 star (inversion_ex3) `Example` *inversion_ex3* : ∀ (X : `Type`) (x y z : X) (l j : *list* X),

```
  x :: y :: l = z :: j →
  y :: l = x :: j →
  x = y.
Proof.
```

Admitted.

□

When used on a hypothesis involving an equality between *different* constructors (e.g., S n = O), `inversion` solves the goal immediately. Consider the following proof:

```
Theorem beq_nat_0_l : ∀ n,
   beq_nat 0 n = true → n = 0.
Proof.
  intros n.
```

We can proceed by case analysis on n. The first case is trivial.

```
  destruct n as [| n'].
  -
    intros H. reflexivity.
```

However, the second one doesn't look so simple: assuming beq_nat 0 (S n') = true, we must show S n' = 0, but the latter clearly contradictory! The way forward lies in the assumption. After simplifying the goal state, we see that beq_nat 0 (S n') = true has become false = true:

```
  -
    simpl.
```

If we use `inversion` on this hypothesis, Coq notices that the subgoal we are working on is impossible, and therefore removes it from further consideration.

```
    intros H. inversion H. Qed.
```

This is an instance of a logical principle known as the *principle of explosion*, which asserts that a contradictory hypothesis entails anything, even false things!

```
Theorem inversion_ex4 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. inversion contra. Qed.

Theorem inversion_ex5 : ∀ (n m : nat),
  false = true →
  [n] = [m].
Proof.
  intros n m contra. inversion contra. Qed.
```

If you find the principle of explosion confusing, remember that these proofs are not actually showing that the conclusion of the statement holds. Rather, they are arguing that, if the nonsensical situation described by the premise did somehow arise, then the nonsensical conclusion would follow. We'll explore the principle of explosion of more detail in the next chapter.

Exercise: 1 star (inversion_ex6)

```
Example inversion_ex6 : ∀ (X : Type)
                          (x y z : X) (l j : list X),
  x :: y :: l = [] →
  y :: l = z :: j →
```

$x = z$.

`Proof.`

Admitted.

□

To summarize this discussion, suppose H is a hypothesis in the context or a previously proven lemma of the form

c a1 a2 ... an = d b1 b2 ... bm

for some constructors `c` and `d` and arguments *a1* ... *an* and *b1* ... *bm*. Then `inversion` H has the following effect:

- If `c` and `d` are the same constructor, then, by the injectivity of this constructor, we know that *a1* = *b1*, *a2* = *b2*, etc. The `inversion` H adds these facts to the context and tries to use them to rewrite the goal.
- If `c` and `d` are different constructors, then the hypothesis H is contradictory, and the current goal doesn't have to be considered at all. In this case, `inversion` H marks the current goal as completed and pops it off the goal stack.

The injectivity of constructors allows us to reason that ∀ (n m : nat), S n = S m → n = m. The converse of this implication is an instance of a more general fact about both constructors and functions, which we will find useful in a few places below:

`Theorem` `f_equal` : ∀ (A B : `Type`) (f: $A \to B$) (x y: A),
  $x = y \to f\ x = f\ y$.
`Proof`. `intros` A B f x y *eq*. `rewrite` *eq*. `reflexivity`. `Qed`.

7.5 Using Tactics on Hypotheses

By default, most tactics work on the goal formula and leave the context unchanged. However, most tactics also have a variant that performs a similar operation on a statement in the context.

For example, the tactic `simpl in` H performs simplification in the hypothesis named H in the context.

`Theorem` *S_inj* : ∀ (*n m* : *nat*) (*b* : *bool*),
    beq_nat (*S n*) (*S m*) = *b* →
    beq_nat n m = *b*.
`Proof`.
  `intros` *n m b H*. `simpl in` *H*. `apply` *H*. `Qed`.

Similarly, `apply` L `in` H matches some conditional statement L (of the form *L1* → *L2*, say) against a hypothesis H in the context. However, unlike ordinary `apply` (which rewrites a goal matching *L2* into a subgoal *L1*), `apply` L `in` H matches H against *L1* and, if successful, replaces it with *L2*.

In other words, `apply` L `in` H gives us a form of "forward reasoning": from $L1 \rightarrow L2$ and a hypothesis matching $L1$, it produces a hypothesis matching $L2$. By contrast, `apply` L is "backward reasoning": it says that if we know $L1 \rightarrow L2$ and we are trying to prove $L2$, it suffices to prove $L1$.

Here is a variant of a proof from above, using forward reasoning throughout instead of backward reasoning.

```
Theorem silly3' : ∀ (n : nat),
  (beq_nat n 5 = true → beq_nat (S (S n)) 7 = true) →
  true = beq_nat n 5 →
  true = beq_nat (S (S n)) 7.
Proof.
  intros n eq H.
  symmetry in H. apply eq in H. symmetry in H.
  apply H. Qed.
```

Forward reasoning starts from what is *given* (premises, previously proven theorems) and iteratively draws conclusions from them until the goal is reached. Backward reasoning starts from the *goal*, and iteratively reasons about what would imply the goal, until premises or previously proven theorems are reached. If you've seen informal proofs before (for example, in a math or computer science class), they probably used forward reasoning. In general, idiomatic use of Coq tends to favor backward reasoning, but in some situations the forward style can be easier to think about.

Exercise: 3 stars, recommended (plus_n_n_injective) Practice using "in" variants in this exercise. (Hint: use plus_n_Sm.)

```
Theorem plus_n_n_injective : ∀ n m,
     n + n = m + m →
     n = m.
Proof.
  intros n. induction n as [| n'].
   Admitted.
```

□

7.6 Varying the Induction Hypothesis

Sometimes it is important to control the exact form of the induction hypothesis when carrying out inductive proofs in Coq. In particular, we need to be careful about which of the assumptions we move (using `intros`) from the goal to the context before invoking the `induction` tactic. For example, suppose we want to show that the double function is injective – i.e., that it maps different arguments to different results:

Theorem double_injective: forall n m, double n = double m -> n = m.

The way we *start* this proof is a bit delicate: if we begin with

intros n. induction n.
all is well. But if we begin it with
intros n m. induction n.
we get stuck in the middle of the inductive case...

```
Theorem double_injective_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction n as [| n'].
  - simpl. intros eq. destruct m as [| m'].
    + reflexivity.
    + inversion eq.
  - intros eq. destruct m as [| m'].
    + inversion eq.
    + apply f_equal.
```

At this point, the induction hypothesis, *IHn'*, does *not* give us *n'* = *m'* – there is an extra S in the way – so the goal is not provable.

```
Abort.
```

What went wrong?

The problem is that, at the point we invoke the induction hypothesis, we have already introduced m into the context – intuitively, we have told Coq, "Let's consider some particular n and m..." and we now have to prove that, if double n = double m for *these particular* n and m, then n = m.

The next tactic, `induction n` says to Coq: We are going to show the goal by induction on n. That is, we are going to prove, for *all* n, that the proposition

- *P* n = "if double n = double m, then n = m"

holds, by showing

- *P* O

  (i.e., "if double O = double m then O = m") and

- *P* n → *P* (S n)

  (i.e., "if double n = double m then n = m" implies "if double (S n) = double m then S n = m").

If we look closely at the second statement, it is saying something rather strange: it says that, for a *particular* m, if we know

- "if double n = double m then n = m"

then we can prove

- "if double (S n) = double m then S n = m".

To see why this is strange, let's think of a particular m – say, 5. The statement is then saying that, if we know

- Q = "if double n = 10 then n = 5"

then we can prove

- R = "if double (S n) = 10 then S n = 5".

But knowing Q doesn't give us any help at all with proving R! (If we tried to prove R from Q, we would start with something like "Suppose double (S n) = 10..." but then we'd be stuck: knowing that double (S n) is 10 tells us nothing about whether double n is 10, so Q is useless.)

Trying to carry out this proof by induction on n when m is already in the context doesn't work because we are then trying to prove a relation involving *every* n but just a *single* m.

The successful proof of double_injective leaves m in the goal statement at the point where the `induction` tactic is invoked on n:

```
Theorem double_injective : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n. induction n as [| n'].
  - simpl. intros m eq. destruct m as [| m'].
    + reflexivity.
    + inversion eq.
  - simpl.
```

Notice that both the goal and the induction hypothesis are different this time: the goal asks us to prove something more general (i.e., to prove the statement for *every* m), but the IH is correspondingly more flexible, allowing us to choose any m we like when we apply the IH.

```
    intros m eq.
```

Now we've chosen a particular m and introduced the assumption that double n = double m. Since we are doing a case analysis on n, we also need a case analysis on m to keep the two "in sync."

```
    destruct m as [| m'].
    + simpl.
```

The 0 case is trivial:

```
    inversion eq.
  +
    apply f_equal.
```

At this point, since we are in the second branch of the `destruct m`, the *m'* mentioned in the context is the predecessor of the m we started out talking about. Since we are also in the S branch of the induction, this is perfect: if we instantiate the generic m in the IH with the current *m'* (this instantiation is performed automatically by the `apply` in the next step), then *IHn'* gives us exactly what we need to finish the proof.

```
    apply IHn'. inversion eq. reflexivity. Qed.
```

What you should take away from all this is that we need to be careful about using induction to try to prove something too specific: To prove a property of n and m by induction on n, it is sometimes important to leave m generic.

The following exercise requires the same pattern.

Exercise: 2 stars (beq_nat_true) `Theorem` *beq_nat_true* : ∀ *n m*,

```
  beq_nat n m = true → n = m.
Proof.
  Admitted.
```

☐

Exercise: 2 stars, advancedM (beq_nat_true_informal) Give a careful informal proof of beq_nat_true, being as explicit as possible about quantifiers.

☐

The strategy of doing fewer `intros` before an `induction` to obtain a more general IH doesn't always work by itself; sometimes some *rearrangement* of quantified variables is needed. Suppose, for example, that we wanted to prove double_injective by induction on m instead of n.

```
Theorem double_injective_take2_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction m as [| m'].
  - simpl. intros eq. destruct n as [| n'].
    + reflexivity.
    + inversion eq.
  - intros eq. destruct n as [| n'].
    + inversion eq.
    + apply f_equal.
Abort.
```

The problem is that, to do induction on m, we must first introduce n. (If we simply say `induction m` without introducing anything first, Coq will automatically introduce n for us!)

What can we do about this? One possibility is to rewrite the statement of the lemma so that m is quantified before n. This works, but it's not nice: We don't want to have to twist the statements of lemmas to fit the needs of a particular strategy for proving them! Rather we want to state them in the clearest and most natural way.

What we can do instead is to first introduce all the quantified variables and then *re-generalize* one or more of them, selectively taking variables out of the context and putting them back at the beginning of the goal. The `generalize dependent` tactic does this.

```
Theorem double_injective_take2 : ∀ n m,
    double n = double m →
    n = m.
Proof.
  intros n m.
  generalize dependent n.
  induction m as [| m'].
  - simpl. intros n eq. destruct n as [| n'].
    + reflexivity.
    + inversion eq.
  - intros n eq. destruct n as [| n'].
    + inversion eq.
    + apply f_equal.
      apply IHm'. inversion eq. reflexivity. Qed.
```

Let's look at an informal proof of this theorem. Note that the proposition we prove by induction leaves n quantified, corresponding to the use of generalize dependent in our formal proof.

Theorem: For any nats n and m, if double n = double m, then n = m.

Proof: Let m be a nat. We prove by induction on m that, for any n, if double n = double m then n = m.

- First, suppose m = 0, and suppose n is a number such that double n = double m. We must show that n = 0.

  Since m = 0, by the definition of double we have double n = 0. There are two cases to consider for n. If n = 0 we are done, since m = 0 = n, as required. Otherwise, if n = S *n'* for some *n'*, we derive a contradiction: by the definition of double, we can calculate double n = S (S (double *n'*)), but this contradicts the assumption that double n = 0.

- Second, suppose m = S *m'* and that n is again a number such that double n = double m. We must show that n = S *m'*, with the induction hypothesis that for every number s, if double s = double *m'* then s = *m'*.

  By the fact that m = S *m'* and the definition of double, we have double n = S (S (double *m'*)). There are two cases to consider for n.

  If n = 0, then by definition double n = 0, a contradiction.

Thus, we may assume that n = S *n'* for some *n'*, and again by the definition of double we have S (S (double *n'*)) = S (S (double *m'*)), which implies by inversion that double *n'* = double *m'*. Instantiating the induction hypothesis with *n'* thus allows us to conclude that *n'* = *m'*, and it follows immediately that S *n'* = S *m'*. Since S *n'* = n and S *m'* = m, this is just what we wanted to show. ☐

Before we close this section and move on to some exercises, let's digress briefly and use beq_nat_true to prove a similar property of identifiers that we'll need in later chapters:

```
Theorem beq_id_true : ∀ x y,
  beq_id x y = true → x = y.
Proof.
  intros [m] [n]. simpl. intros H.
  assert (H' : m = n). { apply beq_nat_true. apply H. }
  rewrite H'. reflexivity.
Qed.
```

Exercise: 3 stars, recommended (gen_dep_practice) Prove this by induction on l.

```
Theorem nth_error_after_last: ∀ (n : nat) (X : Type) (l : list X),
     length l = n →
     nth_error l n = None.
Proof.
   Admitted.
```

☐

7.7 Unfolding Definitions

It sometimes happens that we need to manually unfold a Definition so that we can manipulate its right-hand side. For example, if we define...

```
Definition square n := n × n.
```

... and try to prove a simple fact about square...

```
Lemma square_mult : ∀ n m, square (n × m) = square n × square m.
Proof.
  intros n m.
  simpl.
```

... we get stuck: simpl doesn't simplify anything at this point, and since we haven't proved any other facts about square, there is nothing we can apply or rewrite with.

To make progress, we can manually unfold the definition of square:

```
  unfold square.
```

Now we have plenty to work with: both sides of the equality are expressions involving multiplication, and we have lots of facts about multiplication at our disposal. In particular, we know that it is commutative and associative, and from these facts it is not hard to finish the proof.

```
  rewrite mult_assoc.
  assert (H : n × m × n = n × n × m).
  { rewrite mult_comm. apply mult_assoc. }
  rewrite H. rewrite mult_assoc. reflexivity.
Qed.
```

At this point, a deeper discussion of unfolding and simplification is in order.

You may already have observed that tactics like `simpl`, `reflexivity`, and `apply` will often unfold the definitions of functions automatically when this allows them to make progress. For example, if we define `foo m` to be the constant 5...

```
Definition foo (x: nat) := 5.
```

then the `simpl` in the following proof (or the `reflexivity`, if we omit the `simpl`) will unfold `foo m` to `(fun x ⇒ 5) m` and then further simplify this expression to just 5.

```
Fact silly_fact_1 : ∀ m, foo m + 1 = foo (m + 1) + 1.
Proof.
  intros m.
  simpl.
  reflexivity.
Qed.
```

However, this automatic unfolding is rather conservative. For example, if we define a slightly more complicated function involving a pattern match...

```
Definition bar x :=
  match x with
  | O ⇒ 5
  | S _ ⇒ 5
  end.
```

...then the analogous proof will get stuck:

```
Fact silly_fact_2_FAILED : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  simpl. Abort.
```

The reason that `simpl` doesn't make progress here is that it notices that, after tentatively unfolding `bar m`, it is left with a match whose scrutinee, `m`, is a variable, so the `match` cannot be simplified further. (It is not smart enough to notice that the two branches of the `match` are identical.) So it gives up on unfolding `bar m` and leaves it alone. Similarly, tentatively unfolding `bar (m+1)` leaves a `match` whose scrutinee is a function application (that, itself, cannot be simplified, even after unfolding the definition of +), so `simpl` leaves it alone.

At this point, there are two ways to make progress. One is to use `destruct m` to break the proof into two cases, each focusing on a more concrete choice of `m` (`O` vs `S _`). In each case, the `match` inside of `bar` can now make progress, and the proof is easy to complete.

```
Fact silly_fact_2 : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  destruct m.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

This approach works, but it depends on our recognizing that the `match` hidden inside `bar` is what was preventing us from making progress.

A more straightforward way to make progress is to explicitly tell Coq to unfold `bar`.

```
Fact silly_fact_2' : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  unfold bar.
```

Now it is apparent that we are stuck on the `match` expressions on both sides of the =, and we can use `destruct` to finish the proof without thinking too hard.

```
  destruct m.
  - reflexivity.
  - reflexivity.
Qed.
```

7.8 Using `destruct` on Compound Expressions

We have seen many examples where `destruct` is used to perform case analysis of the value of some variable. But sometimes we need to reason by cases on the result of some *expression*. We can also do this with `destruct`.

Here are some examples:

```
Definition sillyfun (n : nat) : bool :=
  if beq_nat n 3 then false
  else if beq_nat n 5 then false
  else false.

Theorem sillyfun_false : ∀ (n : nat),
  sillyfun n = false.
Proof.
  intros n. unfold sillyfun.
  destruct (beq_nat n 3).
    - reflexivity.
```

```
- destruct (beq_nat n 5).
  + reflexivity.
  + reflexivity. Qed.
```

After unfolding sillyfun in the above proof, we find that we are stuck on `if (beq_nat n 3) then ... else ...`. But either n is equal to 3 or it isn't, so we can use `destruct (beq_nat n 3)` to let us reason about the two cases.

In general, the `destruct` tactic can be used to perform case analysis of the results of arbitrary computations. If e is an expression whose type is some inductively defined type T, then, for each constructor c of T, `destruct e` generates a subgoal in which all occurrences of e (in the goal and in the context) are replaced by c.

Exercise: 3 stars, optional (combine_split)

```
Theorem combine_split : ∀ X Y (l : list (X × Y)) l1 l2,
  split l = (l1, l2) →
  combine l1 l2 = l.
Proof.
  Admitted.
```

☐

However, `destruct`ing compound expressions requires a bit of care, as such `destruct`s can sometimes erase information we need to complete a proof. For example, suppose we define a function sillyfun1 like this:

```
Definition sillyfun1 (n : nat) : bool :=
  if beq_nat n 3 then true
  else if beq_nat n 5 then true
  else false.
```

Now suppose that we want to convince Coq of the (rather obvious) fact that sillyfun1 n yields true only when n is odd. By analogy with the proofs we did with sillyfun above, it is natural to start the proof like this:

```
Theorem sillyfun1_odd_FAILED : ∀ (n : nat),
     sillyfun1 n = true →
     oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (beq_nat n 3).
Abort.
```

We get stuck at this point because the context does not contain enough information to prove the goal! The problem is that the substitution performed by `destruct` is too brutal – it threw away every occurrence of beq_nat n 3, but we need to keep some memory of this expression and how it was destructed, because we need to be able to reason that, since beq_nat n 3 = true in this branch of the case analysis, it must be that n = 3, from which it follows that n is odd.

What we would really like is to substitute away all existing occurences of `beq_nat n` 3, but at the same time add an equation to the context that records which case we are in. The *eqn*: qualifier allows us to introduce such an equation, giving it a name that we choose.

```
Theorem sillyfun1_odd : ∀ (n : nat),
     sillyfun1 n = true →
     oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (beq_nat n 3) eqn:Heqe3.
    - apply beq_nat_true in Heqe3.
      rewrite → Heqe3. reflexivity.
    -


      destruct (beq_nat n 5) eqn:Heqe5.
        +
          apply beq_nat_true in Heqe5.
          rewrite → Heqe5. reflexivity.
        + inversion eq. Qed.
```

Exercise: 2 stars (destruct_eqn_practice) `Theorem` *bool_fn_applied_thrice* :

```
  ∀ (f : bool → bool) (b : bool),
  f (f (f b)) = f b.
Proof.
    Admitted.
```

☐

7.9 Review

We've now seen many of Coq's most fundamental tactics. We'll introduce a few more in the coming chapters, and later on we'll see some more powerful *automation* tactics that make Coq help us with low-level details. But basically we've got what we need to get work done.

Here are the ones we've seen:

- `intros`: move hypotheses/variables from goal to context
- `reflexivity`: finish the proof (when the goal looks like `e` = `e`)
- `apply`: prove goal using a hypothesis, lemma, or constructor
- `apply`... `in` *H*: apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning)
- `apply`... `with`...: explicitly specify values for variables that cannot be determined by pattern matching

- `simpl`: simplify computations in the goal
- `simpl in` H: ... or a hypothesis
- `rewrite`: use an equality hypothesis (or lemma) to rewrite the goal
- `rewrite` ... `in` H: ... or a hypothesis
- `symmetry`: changes a goal of the form t=u into u=t
- `symmetry in` H: changes a hypothesis of the form t=u into u=t
- `unfold`: replace a defined constant by its right-hand side in the goal
- `unfold`... `in` H: ... or a hypothesis
- `destruct`... `as`...: case analysis on values of inductively defined types
- `destruct`... *eqn*:...: specify the name of an equation to be added to the context, recording the result of the case analysis
- `induction`... `as`...: induction on values of inductively defined types
- `inversion`: reason by injectivity and distinctness of constructors
- `assert` (H: e) (or `assert` (e) `as` H): introduce a "local lemma" e and call it H
- `generalize dependent` x: move the variable x (and anything else that depends on it) from the context back to an explicit hypothesis in the goal formula

7.10 Additional Exercises

Exercise: 3 stars (beq_nat_sym) `Theorem` *beq_nat_sym* : ∀ (*n m* : *nat*),
beq_nat n m = *beq_nat m n*.
`Proof`.
Admitted.
□

Exercise: 3 stars, advancedM? (beq_nat_sym_informal) Give an informal proof of this lemma that corresponds to your formal proof above:

Theorem: For any nats n m, beq_nat n m = beq_nat m n.

Proof: □

Exercise: 3 stars, optional (beq_nat_trans) Theorem *beq_nat_trans* : ∀ *n m p*,

beq_nat n m = true →
beq_nat m p = true →
beq_nat n p = true.

Proof.
Admitted.
□

Exercise: 3 stars, advancedM (split_combine) We proved, in an exercise above, that for all lists of pairs, combine is the inverse of split. How would you formalize the statement that split is the inverse of combine? When is this property true?

Complete the definition of split_combine_statement below with a property that states that split is the inverse of combine. Then, prove that the property holds. (Be sure to leave your induction hypothesis general by not doing intros on more things than necessary. Hint: what property do you need of l1 and l2 for split combine l1 l2 = (l1,l2) to be true?)

Definition *split_combine_statement* : Prop

. *Admitted.*

Theorem *split_combine* : *split_combine_statement.*
Proof.
Admitted.
□

Exercise: 3 stars, advanced (filter_exercise) This one is a bit challenging. Pay attention to the form of your induction hypothesis.

Theorem *filter_exercise* : ∀ (*X* : Type) (*test* : *X* → *bool*)
(*x* : *X*) (*l lf* : *list X*),
filter test l = x :: lf →
test x = true.

Proof.
Admitted.
□

Exercise: 4 stars, advanced, recommended (forall_exists_challenge) Define two recursive *Fixpoints*, forallb and *existsb*. The first checks whether every element in a list satisfies a given predicate:

forallb oddb 1;3;5;7;9 = true
forallb negb false;false = true
forallb evenb 0;2;4;5 = false
forallb (beq_nat 5) □ = true

The second checks whether there exists an element in the list that satisfies a given predicate:

existsb (beq_nat 5) 0;2;3;6 = false

existsb (andb true) true;true;false = true

existsb oddb 1;0;0;0;0;3 = true

existsb evenb □ = false

Next, define a *nonrecursive* version of *existsb* – call it *existsb'* – using forallb and negb.

Finally, prove a theorem *existsb_existsb'* stating that *existsb'* and *existsb* have the same behavior.

□

$Date : 2016 - 10 - 0818 : 36 : 21 - 0400(Sat, 08Oct2016)$

Chapter 8

Library Top.Logic

8.1 Logic: Logic in Coq

```
Require Export Tactics.
```

In previous chapters, we have seen many examples of factual claims (*propositions*) and ways of presenting evidence of their truth (*proofs*). In particular, we have worked extensively with *equality propositions* of the form $e1 = e2$, with implications ($P \rightarrow Q$), and with quantified propositions ($\forall$ x, P). In this chapter, we will see how Coq can be used to carry out other familiar forms of logical reasoning.

Before diving into details, let's talk a bit about the status of mathematical statements in Coq. Recall that Coq is a *typed* language, which means that every sensible expression in its world has an associated type. Logical claims are no exception: any statement we might try to prove in Coq has a type, namely Prop, the type of *propositions*. We can see this with the Check command:

```
Check 3 = 3.

Check ∀ n m : nat, n + m = m + n.
```

Note that *all* syntactically well-formed propositions have type Prop in Coq, regardless of whether they are true or not.

Simply *being* a proposition is one thing; being *provable* is something else!

```
Check ∀ n : nat, n = 2.

Check 3 = 4.
```

Indeed, propositions don't just have types: they are *first-class objects* that can be manipulated in the same ways as the other entities in Coq's world. So far, we've seen one primary place that propositions can appear: in Theorem (and Lemma and Example) declarations.

```
Theorem plus_2_2_is_4 :
  2 + 2 = 4.
Proof. reflexivity. Qed.
```

But propositions can be used in many other ways. For example, we can give a name to a proposition using a `Definition`, just as we have given names to expressions of other sorts.

```
Definition plus_fact : Prop := 2 + 2 = 4.
Check plus_fact.
```

We can later use this name in any situation where a proposition is expected – for example, as the claim in a `Theorem` declaration.

```
Theorem plus_fact_is_true :
  plus_fact.
Proof. reflexivity. Qed.
```

We can also write *parameterized* propositions – that is, functions that take arguments of some type and return a proposition.

For instance, the following function takes a number and returns a proposition asserting that this number is equal to three:

```
Definition is_three (n : nat) : Prop :=
  n = 3.
Check is_three.
```

In Coq, functions that return propositions are said to define *properties* of their arguments.

For instance, here's a (polymorphic) property defining the familiar notion of an *injective function*.

```
Definition injective {A B} (f : A → B) :=
  ∀ x y : A, f x = f y → x = y.

Lemma succ_inj : injective S.
Proof.
  intros n m H. inversion H. reflexivity.
Qed.
```

The equality operator = is also a function that returns a `Prop`.

The expression `n = m` is syntactic sugar for `eq n m`, defined using Coq's `Notation` mechanism. Because `eq` can be used with elements of any type, it is also polymorphic:

```
Check @eq.
```

(Notice that we wrote `@eq` instead of `eq`: The type argument `A` to `eq` is declared as implicit, so we need to turn off implicit arguments to see the full type of `eq`.)

8.2 Logical Connectives

8.2.1 Conjunction

The *conjunction* (or *logical and*) of propositions `A` and `B` is written `A ∧ B`, representing the claim that both `A` and `B` are true.

```
Example and_example : 3 + 4 = 7 ∧ 2 × 2 = 4.
```

To prove a conjunction, use the `split` tactic. It will generate two subgoals, one for each part of the statement:

```
Proof.
  split.
  - reflexivity.
  - reflexivity.
Qed.
```

For any propositions A and B, if we assume that A is true and we assume that B is true, we can conclude that A ∧ B is also true.

```
Lemma and_intro : ∀ A B : Prop, A → B → A ∧ B.
Proof.
  intros A B HA HB. split.
  - apply HA.
  - apply HB.
Qed.
```

Since applying a theorem with hypotheses to some goal has the effect of generating as many subgoals as there are hypotheses for that theorem, we can apply `and_intro` to achieve the same effect as `split`.

```
Example and_example' : 3 + 4 = 7 ∧ 2 × 2 = 4.
Proof.
  apply and_intro.
  - reflexivity.
  - reflexivity.
Qed.
```

Exercise: 2 stars (and_exercise)

```
Example and_exercise :
  ∀ n m : nat, n + m = 0 → n = 0 ∧ m = 0.
Proof.
   Admitted.
```

□

So much for proving conjunctive statements. To go in the other direction – i.e., to *use* a conjunctive hypothesis to help prove something else – we employ the `destruct` tactic.

If the proof context contains a hypothesis *H* of the form A ∧ B, writing `destruct H as [HA HB]` will remove *H* from the context and add two new hypotheses: *HA*, stating that A is true, and *HB*, stating that B is true.

```
Lemma and_example2 :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  intros n m H.
```

```
  destruct H as [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

As usual, we can also destruct H right when we introduce it, instead of introducing and then destructing it:

```
Lemma and_example2' :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  intros n m [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

You may wonder why we bothered packing the two hypotheses $n = 0$ and $m = 0$ into a single conjunction, since we could have also stated the theorem with two separate premises:

```
Lemma and_example2'' :
  ∀ n m : nat, n = 0 → m = 0 → n + m = 0.
Proof.
  intros n m Hn Hm.
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

For this theorem, both formulations are fine. But it's important to understand how to work with conjunctive hypotheses because conjunctions often arise from intermediate steps in proofs, especially in bigger developments. Here's a simple example:

```
Lemma and_example3 :
  ∀ n m : nat, n + m = 0 → n × m = 0.
Proof.
  intros n m H.
  assert (H' : n = 0 ∧ m = 0).
  { apply and_exercise. apply H. }
  destruct H' as [Hn Hm].
  rewrite Hn. reflexivity.
Qed.
```

Another common situation with conjunctions is that we know $A \land B$ but in some context we need just A (or just B). The following lemmas are useful in such cases:

```
Lemma proj1 : ∀ P Q : Prop,
  P ∧ Q → P.
Proof.
  intros P Q [HP HQ].
  apply HP. Qed.
```

Exercise: 1 star, optional (proj2) `Lemma` *proj2* : ∀ P Q : `Prop`,
$P \land Q \rightarrow Q$.
`Proof.`
Admitted.
☐

Finally, we sometimes need to rearrange the order of conjunctions and/or the grouping of multi-way conjunctions. The following commutativity and associativity theorems are handy in such cases.

```
Theorem and_commut : ∀ P Q : Prop,
  P ∧ Q → Q ∧ P.
Proof.
  intros P Q [HP HQ].
  split.
    - apply HQ.
    - apply HP. Qed.
```

Exercise: 2 stars (and_assoc) (In the following proof of associativity, notice how the *nested* intro pattern breaks the hypothesis H : $P \land (Q \land R)$ down into HP : P, HQ : Q, and HR : R. Finish the proof from there.)

```
Theorem and_assoc : ∀ P Q R : Prop,
  P ∧ (Q ∧ R) → (P ∧ Q) ∧ R.
Proof.
  intros P Q R [HP [HQ HR]].
   Admitted.
```
☐

By the way, the infix notation ∧ is actually just syntactic sugar for **and** A B. That is, **and** is a Coq operator that takes two propositions as arguments and yields a proposition.

```
Check and.
```

8.2.2 Disjunction

Another important connective is the *disjunction*, or *logical or* of two propositions: A ∨ B is true when either A or B is. (Alternatively, we can write **or** A B, where **or** : `Prop` → `Prop` → `Prop`.)

To use a disjunctive hypothesis in a proof, we proceed by case analysis, which, as for `nat` or other data types, can be done with `destruct` or `intros`. Here is an example:

```
Lemma or_example :
  ∀ n m : nat, n = 0 ∨ m = 0 → n × m = 0.
Proof.
  intros n m [Hn | Hm].
  -
```

```
    rewrite Hn. reflexivity.
  -
    rewrite Hm. rewrite <- mult_n_O.
    reflexivity.
Qed.
```

Conversely, to show that a disjunction holds, we need to show that one of its sides does. This is done via two tactics, `left` and `right`. As their names imply, the first one requires proving the left side of the disjunction, while the second requires proving its right side. Here is a trivial use...

```
Lemma or_intro : ∀ A B : Prop, A → A ∨ B.
Proof.
  intros A B HA.
  left.
  apply HA.
Qed.
```

... and a slightly more interesting example requiring both `left` and `right`:

```
Lemma zero_or_succ :
  ∀ n : nat, n = 0 ∨ n = S (pred n).
Proof.
  intros [|n].
  - left. reflexivity.
  - right. reflexivity.
Qed.
```

Exercise: 1 star (mult_eq_0) `Lemma` *mult_eq_0* :

```
  ∀ n m, n × m = 0 → n = 0 ∨ m = 0.
Proof.
```

Admitted.

☐

Exercise: 1 star (or_commut) `Theorem` *or_commut* : ∀ *P Q* : `Prop`,

```
  P ∨ Q → Q ∨ P.
Proof.
```

Admitted.

☐

8.2.3 Falsehood and Negation

So far, we have mostly been concerned with proving that certain things are *true* – addition is commutative, appending lists is associative, etc. Of course, we may also be interested in *negative* results, showing that certain propositions are *not* true. In Coq, such negative statements are expressed with the negation operator ¬.

To see how negation works, recall the discussion of the *principle of explosion* from the Tactics chapter; it asserts that, if we assume a contradiction, then any other proposition can be derived. Following this intuition, we could define ¬ P ("not P") as ∀ Q, P → Q. Coq actually makes a slightly different choice, defining ¬ P as P → **False**, where **False** is a *particular* contradictory proposition defined in the standard library.

```
Module MyNot.

Definition not (P:Prop) := P → False.

Notation "~ x" := (not x) : type_scope.

Check not.

End MyNot.
```

Since **False** is a contradictory proposition, the principle of explosion also applies to it. If we get **False** into the proof context, we can `destruct` it to complete any goal:

```
Theorem ex_falso_quodlibet : ∀ (P:Prop),
  False → P.
Proof.
  intros P contra.
  destruct contra. Qed.
```

The Latin *ex falso quodlibet* means, literally, "from falsehood follows whatever you like"; this is another common name for the principle of explosion.

Exercise: 2 stars, optional (not_implies_our_not) Show that Coq's definition of negation implies the intuitive one mentioned above:

```
Fact not_implies_our_not : ∀ (P:Prop),
  ¬ P → (∀ (Q:Prop), P → Q).
Proof.
   Admitted.
```

□

This is how we use `not` to state that 0 and 1 are different elements of `nat`:

```
Theorem zero_not_one : ~(0 = 1).
Proof.
  intros contra. inversion contra.
Qed.
```

Such inequality statements are frequent enough to warrant a special notation, x ≠ y:

```
Check (0 ≠ 1).

Theorem zero_not_one' : 0 ≠ 1.
Proof.
```

```
  intros H. inversion H.
Qed.
```

It takes a little practice to get used to working with negation in Coq. Even though you can see perfectly well why a statement involving negation is true, it can be a little tricky at first to get things into the right configuration so that Coq can understand it! Here are proofs of a few familiar facts to get you warmed up.

```
Theorem not_False :
  ¬ False.
Proof.
  unfold not. intros H. destruct H. Qed.

Theorem contradiction_implies_anything : ∀ P Q : Prop,
  (P ∧ ¬P) → Q.
Proof.
  intros P Q [HP HNA]. unfold not in HNA.
  apply HNA in HP. destruct HP. Qed.

Theorem double_neg : ∀ P : Prop,
  P → ~~P.
Proof.
  intros P H. unfold not. intros G. apply G. apply H. Qed.
```

Exercise: 2 stars, advanced, recommendedM (double_neg_inf) Write an informal proof of double_neg:

Theorem: P implies ~~P, for any proposition P.

□

Exercise: 2 stars, recommended (contrapositive)

```
Theorem contrapositive : ∀ (P Q : Prop),
  (P → Q) → (~Q → ¬P).
Proof.
   Admitted.
```

□

Exercise: 1 star (not_both_true_and_false)

```
Theorem not_both_true_and_false : ∀ P : Prop,
  ¬ (P ∧ ¬P).
Proof.
   Admitted.
```

□

Exercise: 1 star, advancedM (informal_not_PNP) Write an informal proof (in English) of the proposition ∀ P : Prop, ~($P \wedge \neg P$).

☐

Similarly, since inequality involves a negation, it requires a little practice to be able to work with it fluently. Here is one useful trick. If you are trying to prove a goal that is nonsensical (e.g., the goal state is false = true), apply ex_falso_quodlibet to change the goal to **False**. This makes it easier to use assumptions of the form ¬P that may be available in the context – in particular, assumptions of the form x≠y.

```
Theorem not_true_is_false : ∀ b : bool,
  b ≠ true → b = false.
Proof.
  intros [] H.
  -
    unfold not in H.
    apply ex_falso_quodlibet.
    apply H. reflexivity.
  -
    reflexivity.
Qed.
```

Since reasoning with ex_falso_quodlibet is quite common, Coq provides a built-in tactic, *exfalso*, for applying it.

```
Theorem not_true_is_false' : ∀ b : bool,
  b ≠ true → b = false.
Proof.
  intros [] H.
  -
    unfold not in H.
    exfalso.     apply H. reflexivity.
  - reflexivity.
Qed.
```

8.2.4 Truth

Besides **False**, Coq's standard library also defines **True**, a proposition that is trivially true. To prove it, we use the predefined constant I : **True**:

```
Lemma True_is_true : True.
Proof. apply I. Qed.
```

Unlike **False**, which is used extensively, **True** is used quite rarely, since it is trivial (and therefore uninteresting) to prove as a goal, and it carries no useful information as a hypothesis. But it can be quite useful when defining complex Props using conditionals or as a parameter to higher-order Props. We will see examples of such uses of **True** later on.

8.2.5 Logical Equivalence

The handy "if and only if" connective, which asserts that two propositions have the same truth value, is just the conjunction of two implications.

```
Module MyIff.

Definition iff (P Q : Prop) := (P → Q) ∧ (Q → P).

Notation "P <-> Q" := (iff P Q)
                      (at level 95, no associativity)
                      : type_scope.

End MyIff.

Theorem iff_sym : ∀ P Q : Prop,
  (P ↔ Q) → (Q ↔ P).
Proof.
  intros P Q [HAB HBA].
  split.
  - apply HBA.
  - apply HAB. Qed.

Lemma not_true_iff_false : ∀ b,
  b ≠ true ↔ b = false.
Proof.
  intros b. split.
  - apply not_true_is_false.
  -
    intros H. rewrite H. intros H'. inversion H'.
Qed.
```

Exercise: 1 star, optional (iff_properties) Using the above proof that ↔ is symmetric (iff_sym) as a guide, prove that it is also reflexive and transitive.

```
Theorem iff_refl : ∀ P : Prop,
  P ↔ P.
Proof.
   Admitted.

Theorem iff_trans : ∀ P Q R : Prop,
  (P ↔ Q) → (Q ↔ R) → (P ↔ R).
Proof.
   Admitted.
```

□

Exercise: 3 stars (or_distributes_over_and) Theorem *or_distributes_over_and* : ∀ P Q R : Prop,

$P \lor (Q \land R) \leftrightarrow (P \lor Q) \land (P \lor R)$.

```
Proof.
  Admitted.
```

□

Some of Coq's tactics treat iff statements specially, avoiding the need for some low-level proof-state manipulation. In particular, `rewrite` and `reflexivity` can be used with iff statements, not just equalities. To enable this behavior, we need to import a special Coq library that allows rewriting with other formulas besides equality:

```
Require Import Coq.Setoids.Setoid.
```

Here is a simple example demonstrating how these tactics work with iff. First, let's prove a couple of basic iff equivalences...

```
Lemma mult_0 : ∀ n m, n × m = 0 ↔ n = 0 ∨ m = 0.
Proof.
  split.
  - apply mult_eq_0.
  - apply or_example.
Qed.

Lemma or_assoc :
  ∀ P Q R : Prop, P ∨ (Q ∨ R) ↔ (P ∨ Q) ∨ R.
Proof.
  intros P Q R. split.
  - intros [H | [H | H]].
    + left. left. apply H.
    + left. right. apply H.
    + right. apply H.
  - intros [[H | H] | H].
    + left. apply H.
    + right. left. apply H.
    + right. right. apply H.
Qed.
```

We can now use these facts with `rewrite` and `reflexivity` to give smooth proofs of statements involving equivalences. Here is a ternary version of the previous `mult_0` result:

```
Lemma mult_0_3 :
  ∀ n m p, n × m × p = 0 ↔ n = 0 ∨ m = 0 ∨ p = 0.
Proof.
  intros n m p.
  rewrite mult_0. rewrite mult_0. rewrite or_assoc.
  reflexivity.
Qed.
```

The `apply` tactic can also be used with ↔. When given an equivalence as its argument, `apply` tries to guess which side of the equivalence to use.

```
Lemma apply_iff_example :
  ∀ n m : nat, n × m = 0 → n = 0 ∨ m = 0.
Proof.
  intros n m H. apply mult_0. apply H.
Qed.
```

8.2.6 Existential Quantification

Another important logical connective is *existential quantification*. To say that there is some x of type T such that some property *P* holds of x, we write ∃ x : T, *P*. As with ∀, the type annotation : T can be omitted if Coq is able to infer from the context what the type of x should be.

To prove a statement of the form ∃ x, *P*, we must show that *P* holds for some specific choice of value for x, known as the *witness* of the existential. This is done in two steps: First, we explicitly tell Coq which witness *t* we have in mind by invoking the tactic ∃ *t*. Then we prove that *P* holds after all occurrences of x are replaced by *t*.

```
Lemma four_is_even : ∃ n : nat, 4 = n + n.
Proof.
  ∃ 2. reflexivity.
Qed.
```

Conversely, if we have an existential hypothesis ∃ x, *P* in the context, we can destruct it to obtain a witness x and a hypothesis stating that *P* holds of x.

```
Theorem exists_example_2 : ∀ n,
  (∃ m, n = 4 + m) →
  (∃ o, n = 2 + o).
Proof.
  intros n [m Hm].   ∃ (2 + m).
  apply Hm. Qed.
```

Exercise: 1 star (dist_not_exists) Prove that "*P* holds for all x" implies "there is no x for which *P* does not hold."

```
Theorem dist_not_exists : ∀ (X:Type) (P : X → Prop),
  (∀ x, P x) → ¬ (∃ x, ¬ P x).
Proof.
   Admitted.
   □
```

Exercise: 2 stars (dist_exists_or) Prove that existential quantification distributes over disjunction.

```
Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
  (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
```

```
Proof.
  Admitted.
  □
```

8.3 Programming with Propositions

The logical connectives that we have seen provide a rich vocabulary for defining complex propositions from simpler ones. To illustrate, let's look at how to express the claim that an element x occurs in a list l. Notice that this property has a simple recursive structure:

- If l is the empty list, then x cannot occur on it, so the property "x appears in l" is simply false.

  - Otherwise, l has the form *x' :: l'*. In this case, x occurs in l if either it is equal to *x'* or it occurs in *l'*.

  We can translate this directly into a straightforward recursive function from taking an element and a list and returning a proposition:

```
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] ⇒ False
  | x' :: l' ⇒ x' = x ∨ In x l'
  end.
```

When In is applied to a concrete list, it expands into a concrete sequence of nested disjunctions.

```
Example In_example_1 : In 4 [1; 2; 3; 4; 5].
Proof.
  simpl. right. right. right. left. reflexivity.
Qed.
```

```
Example In_example_2 :
  ∀ n, In n [2; 4] →
  ∃ n', n = 2 × n'.
Proof.
  simpl.
  intros n [H | [H | []]].
  - ∃ 1. rewrite ← H. reflexivity.
  - ∃ 2. rewrite ← H. reflexivity.
Qed.
```

(Notice the use of the empty pattern to discharge the last case *en passant*.)

We can also prove more generic, higher-level lemmas about In.

Note, in the next, how In starts out applied to a variable and only gets expanded when we do case analysis on this variable:

```
Lemma In_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
    In x l →
    In (f x) (map f l).
Proof.
  intros A B f l x.
  induction l as [|x' l' IHl'].
  -
    simpl. intros [].
  -
    simpl. intros [H | H].
    + rewrite H. left. reflexivity.
    + right. apply IHl'. apply H.
Qed.
```

This way of defining propositions recursively, though convenient in some cases, also has some drawbacks. In particular, it is subject to Coq's usual restrictions regarding the definition of recursive functions, e.g., the requirement that they be "obviously terminating." In the next chapter, we will see how to define propositions *inductively*, a different technique with its own set of strengths and limitations.

Exercise: 2 stars (In_map_iff)

```
Lemma In_map_iff :
  ∀ (A B : Type) (f : A → B) (l : list A) (y : B),
    In y (map f l) ↔
    ∃ x, f x = y ∧ In x l.
Proof.
  Admitted.
```

□

Exercise: 2 stars (in_app_iff)

```
Lemma in_app_iff : ∀ A l l' (a:A),
  In a (l++l') ↔ In a l ∨ In a l'.
Proof.
  Admitted.
```

□

Exercise: 3 stars (All) Recall that functions returning propositions can be seen as *properties* of their arguments. For instance, if P has type nat → Prop, then P n states that property P holds of n.

Drawing inspiration from In, write a recursive function All stating that some property P holds of all elements of a list l. To make sure your definition is correct, prove the All_In lemma below. (Of course, your definition should *not* just restate the left-hand side of All_In.)

```
Fixpoint All {T : Type} (P : T → Prop) (l : list T) : Prop
  . Admitted.

Lemma All_In :
  ∀ T (P : T → Prop) (l : list T),
    (∀ x, In x l → P x) ↔
    All P l.
Proof.
   Admitted.
```
□

Exercise: 3 stars (combine_odd_even) Complete the definition of the combine_odd_even function below. It takes as arguments two properties of numbers, *Podd* and *Peven*, and it should return a property *P* such that *P* n is equivalent to *Podd* n when n is odd and equivalent to *Peven* n otherwise.

```
Definition combine_odd_even (Podd Peven : nat → Prop) : nat → Prop
  . Admitted.
```

To test your definition, prove the following facts:

```
Theorem combine_odd_even_intro :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    (oddb n = true → Podd n) →
    (oddb n = false → Peven n) →
    combine_odd_even Podd Peven n.
Proof.
   Admitted.

Theorem combine_odd_even_elim_odd :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    combine_odd_even Podd Peven n →
    oddb n = true →
    Podd n.
Proof.
   Admitted.

Theorem combine_odd_even_elim_even :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    combine_odd_even Podd Peven n →
    oddb n = false →
    Peven n.
Proof.
   Admitted.
```
□

8.4 Applying Theorems to Arguments

One feature of Coq that distinguishes it from many other proof assistants is that it treats *proofs* as first-class objects.

There is a great deal to be said about this, but it is not necessary to understand it in detail in order to use Coq. This section gives just a taste, while a deeper exploration can be found in the optional chapters ProofObjects and IndPrinciples.

We have seen that we can use the `Check` command to ask Coq to print the type of an expression. We can also use `Check` to ask what theorem a particular identifier refers to.

```
Check plus_comm.
```

Coq prints the *statement* of the plus_comm theorem in the same way that it prints the *type* of any term that we ask it to `Check`. Why?

The reason is that the identifier plus_comm actually refers to a *proof object* – a data structure that represents a logical derivation establishing of the truth of the statement ∀ n m : nat, n + m = m + n. The type of this object *is* the statement of the theorem that it is a proof of.

Intuitively, this makes sense because the statement of a theorem tells us what we can use that theorem for, just as the type of a computational object tells us what we can do with that object – e.g., if we have a term of type nat → nat → nat, we can give it two nats as arguments and get a nat back. Similarly, if we have an object of type n = m → n + n = m + m and we provide it an "argument" of type n = m, we can derive n + n = m + m.

Operationally, this analogy goes even further: by applying a theorem, as if it were a function, to hypotheses with matching types, we can specialize its result without having to resort to intermediate assertions. For example, suppose we wanted to prove the following result:

```
Lemma plus_comm3 :
  ∀ n m p, n + (m + p) = (p + m) + n.
```

It appears at first sight that we ought to be able to prove this by rewriting with plus_comm twice to make the two sides match. The problem, however, is that the second `rewrite` will undo the effect of the first.

```
Proof.
  intros n m p.
  rewrite plus_comm.
  rewrite plus_comm.
Abort.
```

One simple way of fixing this problem, using only tools that we already know, is to use `assert` to derive a specialized version of plus_comm that can be used to rewrite exactly where we want.

```
Lemma plus_comm3_take2 :
  ∀ n m p, n + (m + p) = (p + m) + n.
Proof.
```

```
  intros n m p.
  rewrite plus_comm.
  assert (H : m + p = p + m).
  { rewrite plus_comm. reflexivity. }
  rewrite H.
  reflexivity.
Qed.
```

A more elegant alternative is to apply plus_comm directly to the arguments we want to instantiate it with, in much the same way as we apply a polymorphic function to a type argument.

```
Lemma plus_comm3_take3 :
  ∀ n m p, n + (m + p) = (p + m) + n.
Proof.
  intros n m p.
  rewrite plus_comm.
  rewrite (plus_comm m).
  reflexivity.
Qed.
```

You can "use theorems as functions" in this way with almost all tactics that take a theorem name as an argument. Note also that theorem application uses the same inference mechanisms as function application; thus, it is possible, for example, to supply wildcards as arguments to be inferred, or to declare some hypotheses to a theorem as implicit by default. These features are illustrated in the proof below.

```
Example lemma_application_ex :
  ∀ {n : nat} {ns : list nat},
    In n (map (fun m ⇒ m × 0) ns) →
    n = 0.
Proof.
  intros n ns H.
  destruct (proj1 _ _ (In_map_iff _ _ _ _ _) H)
            as [m [Hm _]].
  rewrite mult_0_r in Hm. rewrite ← Hm. reflexivity.
Qed.
```

We will see many more examples of the idioms from this section in later chapters.

8.5 Coq vs. Set Theory

Coq's logical core, the *Calculus of Inductive Constructions*, differs in some important ways from other formal systems that are used by mathematicians for writing down precise and rigorous proofs. For example, in the most popular foundation for mainstream paper-and-pencil mathematics, Zermelo-Fraenkel Set Theory (ZFC), a mathematical object can potentially be a member of many different sets; a term in Coq's logic, on the other hand, is a member of at most one type. This difference often leads to slightly different ways of capturing informal mathematical concepts, but these are, by and large, quite natural and easy to work with. For example, instead of saying that a natural number `n` belongs to the set of even numbers, we would say in Coq that **ev** `n` holds, where **ev** : `nat` → `Prop` is a property describing even numbers.

However, there are some cases where translating standard mathematical reasoning into Coq can be either cumbersome or sometimes even impossible, unless we enrich the core logic with additional axioms. We conclude this chapter with a brief discussion of some of the most significant differences between the two worlds.

8.5.1 Functional Extensionality

The equality assertions that we have seen so far mostly have concerned elements of inductive types (`nat`, **bool**, etc.). But since Coq's equality operator is polymorphic, these are not the only possibilities – in particular, we can write propositions claiming that two *functions* are equal to each other:

```
Example function_equality_ex1 : plus 3 = plus (pred 4).
Proof. reflexivity. Qed.
```

In common mathematical practice, two functions `f` and `g` are considered equal if they produce the same outputs:

(forall x, f x = g x) -> f = g

This is known as the principle of *functional extensionality*.

Informally speaking, an "extensional property" is one that pertains to an object's observable behavior. Thus, functional extensionality simply means that a function's identity is completely determined by what we can observe from it – i.e., in Coq terms, the results we obtain after applying it.

Functional extensionality is not part of Coq's basic axioms. This means that some "reasonable" propositions are not provable.

```
Example function_equality_ex2 :
  (fun x ⇒ plus x 1) = (fun x ⇒ plus 1 x).
Proof.
Abort.
```

However, we can add functional extensionality to Coq's core logic using the `Axiom` command.

```
Axiom functional_extensionality : ∀ {X Y: Type}
                                    {f g : X → Y},
  (∀ (x:X), f x = g x) → f = g.
```

Using `Axiom` has the same effect as stating a theorem and skipping its proof using *Admitted*, but it alerts the reader that this isn't just something we're going to come back and fill in later!

We can now invoke functional extensionality in proofs:

```
Example function_equality_ex2 :
  (fun x ⇒ plus x 1) = (fun x ⇒ plus 1 x).
Proof.
  apply functional_extensionality. intros x.
  apply plus_comm.
Qed.
```

Naturally, we must be careful when adding new axioms into Coq's logic, as they may render it *inconsistent* – that is, they may make it possible to prove every proposition, including **False**!

Unfortunately, there is no simple way of telling whether an axiom is safe to add: hard work is generally required to establish the consistency of any particular combination of axioms.

However, it is known that adding functional extensionality, in particular, *is* consistent.

To check whether a particular proof relies on any additional axioms, use the `Print Assumptions` command.

```
Print Assumptions function_equality_ex2.
```

Exercise: 4 stars (tr_rev) One problem with the definition of the list-reversing function `rev` that we have is that it performs a call to `app` on each step; running `app` takes time asymptotically linear in the size of the list, which means that `rev` has quadratic running time. We can improve this with the following definition:

```
Fixpoint rev_append {X} (l1 l2 : list X) : list X :=
  match l1 with
  | [] ⇒ l2
  | x :: l1' ⇒ rev_append l1' (x :: l2)
  end.
Definition tr_rev {X} (l : list X) : list X :=
  rev_append l [].
```

This version is said to be *tail-recursive*, because the recursive call to the function is the last operation that needs to be performed (i.e., we don't have to execute ++ after the recursive call); a decent compiler will generate very efficient code in this case. Prove that the two definitions are indeed equivalent.

```
Lemma tr_rev_correct : ∀ X, @tr_rev X = @rev X.
  Admitted.
```

□

8.5.2 Propositions and Booleans

We've seen two different ways of encoding logical facts in Coq: with *booleans* (of type **bool**), and with *propositions* (of type `Prop`).

For instance, to claim that a number n is even, we can say either

- (1) that evenb n returns true, or
- (2) that there exists some k such that n = double k. Indeed, these two notions of evenness are equivalent, as can easily be shown with a couple of auxiliary lemmas.

We often say that the boolean evenb n *reflects* the proposition ∃ k, n = double k.

```
Theorem evenb_double : ∀ k, evenb (double k) = true.
Proof.
  intros k. induction k as [|k' IHk'].
  - reflexivity.
  - simpl. apply IHk'.
Qed.
```

Exercise: 3 stars (evenb_double_conv)

```
Theorem evenb_double_conv : ∀ n,
  ∃ k, n = if evenb n then double k
                else S (double k).
Proof.
   Admitted.
```

□

```
Theorem even_bool_prop : ∀ n,
  evenb n = true ↔ ∃ k, n = double k.
Proof.
  intros n. split.
  - intros H. destruct (evenb_double_conv n) as [k Hk].
    rewrite Hk. rewrite H. ∃ k. reflexivity.
  - intros [k Hk]. rewrite Hk. apply evenb_double.
Qed.
```

Similarly, to state that two numbers n and m are equal, we can say either (1) that beq_nat n m returns true or (2) that n = m. These two notions are equivalent.

```
Theorem beq_nat_true_iff : ∀ n1 n2 : nat,
  beq_nat n1 n2 = true ↔ n1 = n2.
Proof.
  intros n1 n2. split.
  - apply beq_nat_true.
  - intros H. rewrite H. rewrite ← beq_nat_refl. reflexivity.
Qed.
```

However, while the boolean and propositional formulations of a claim are equivalent from a purely logical perspective, they need not be equivalent *operationally*. Equality provides an extreme example: knowing that beq_nat n m = true is generally of little direct help in the middle of a proof involving n and m; however, if we convert the statement to the equivalent form n = m, we can rewrite with it.

The case of even numbers is also interesting. Recall that, when proving the backwards direction of even_bool_prop (i.e., evenb_double, going from the propositional to the boolean claim), we used a simple induction on k. On the other hand, the converse (the evenb_double_conv exercise) required a clever generalization, since we can't directly prove (∃ k, n = double k) → evenb n = true.

For these examples, the propositional claims are more useful than their boolean counterparts, but this is not always the case. For instance, we cannot test whether a general proposition is true or not in a function definition; as a consequence, the following code fragment is rejected:

```
Fail Definition is_even_prime n :=
  if n = 2 then true
  else false.
```

Coq complains that n = 2 has type Prop, while it expects an elements of **bool** (or some other inductive type with two elements). The reason for this error message has to do with the *computational* nature of Coq's core language, which is designed so that every function that it can express is computable and total. One reason for this is to allow the extraction of executable programs from Coq developments. As a consequence, Prop in Coq does *not* have a universal case analysis operation telling whether any given proposition is true or false, since such an operation would allow us to write non-computable functions.

Although general non-computable properties cannot be phrased as boolean computations, it is worth noting that even many *computable* properties are easier to express using Prop than **bool**, since recursive function definitions are subject to significant restrictions in Coq. For instance, the next chapter shows how to define the property that a regular expression matches a given string using Prop. Doing the same with **bool** would amount to writing a regular expression matcher, which would be more complicated, harder to understand, and harder to reason about.

Conversely, an important side benefit of stating facts using booleans is enabling some proof automation through computation with Coq terms, a technique known as *proof by reflection*. Consider the following statement:

```
Example even_1000 : ∃ k, 1000 = double k.
```

The most direct proof of this fact is to give the value of k explicitly.

```
Proof. ∃ 500. reflexivity. Qed.
```

On the other hand, the proof of the corresponding boolean statement is even simpler:

```
Example even_1000' : evenb 1000 = true.
Proof. reflexivity. Qed.
```

What is interesting is that, since the two notions are equivalent, we can use the boolean formulation to prove the other one without mentioning the value 500 explicitly:

```
Example even_1000'' : ∃ k, 1000 = double k.
```

```
Proof. apply even_bool_prop. reflexivity. Qed.
```

Although we haven't gained much in terms of proof size in this case, larger proofs can often be made considerably simpler by the use of reflection. As an extreme example, the Coq proof of the famous *4-color theorem* uses reflection to reduce the analysis of hundreds of different cases to a boolean computation. We won't cover reflection in great detail, but it serves as a good example showing the complementary strengths of booleans and general propositions.

Exercise: 2 stars (logical_connectives) The following lemmas relate the propositional connectives studied in this chapter to the corresponding boolean operations.

```
Lemma andb_true_iff : ∀ b1 b2:bool,
  b1 && b2 = true ↔ b1 = true ∧ b2 = true.
Proof.
  Admitted.

Lemma orb_true_iff : ∀ b1 b2,
  b1 || b2 = true ↔ b1 = true ∨ b2 = true.
Proof.
  Admitted.
```

□

Exercise: 1 star (beq_nat_false_iff) The following theorem is an alternate "negative" formulation of beq_nat_true_iff that is more convenient in certain situations (we'll see examples in later chapters).

```
Theorem beq_nat_false_iff : ∀ x y : nat,
  beq_nat x y = false ↔ x ≠ y.
Proof.
  Admitted.
```

□

Exercise: 3 stars (beq_list) Given a boolean operator *beq* for testing equality of elements of some type A, we can define a function beq_list *beq* for testing equality of lists with elements in A. Complete the definition of the beq_list function below. To make sure that your definition is correct, prove the lemma beq_list_true_iff.

```
Fixpoint beq_list {A : Type} (beq : A → A → bool)
                  (l1 l2 : list A) : bool
  . Admitted.

Lemma beq_list_true_iff :
  ∀ A (beq : A → A → bool),
    (∀ a1 a2, beq a1 a2 = true ↔ a1 = a2) →
    ∀ l1 l2, beq_list beq l1 l2 = true ↔ l1 = l2.
```

```
Proof.
  Admitted.
```
□

Exercise: 2 stars, recommended (All_forallb) Recall the function forallb, from the exercise *forall_exists_challenge* in chapter Tactics:

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool :=
  match l with
  | [] ⇒ true
  | x :: l' ⇒ andb (test x) (forallb test l')
  end.
```

Prove the theorem below, which relates forallb to the All property of the above exercise.

```
Theorem forallb_true_iff : ∀ X test (l : list X),
  forallb test l = true ↔ All (fun x ⇒ test x = true) l.
Proof.
  Admitted.
```

Are there any important properties of the function forallb which are not captured by this specification?

□

8.5.3 Classical vs. Constructive Logic

We have seen that it is not possible to test whether or not a proposition P holds while defining a Coq function. You may be surprised to learn that a similar restriction applies to *proofs*! In other words, the following intuitive reasoning principle is not derivable in Coq:

```
Definition excluded_middle := ∀ P : Prop,
  P ∨ ¬ P.
```

To understand operationally why this is the case, recall that, to prove a statement of the form $P \vee Q$, we use the `left` and `right` tactics, which effectively require knowing which side of the disjunction holds. But the universally quantified P in excluded_middle is an *arbitrary* proposition, which we know nothing about. We don't have enough information to choose which of `left` or `right` to apply, just as Coq doesn't have enough information to mechanically decide whether P holds or not inside a function.

However, if we happen to know that P is reflected in some boolean term b, then knowing whether it holds or not is trivial: we just have to check the value of b.

```
Theorem restricted_excluded_middle : ∀ P b,
  (P ↔ b = true) → P ∨ ¬ P.
Proof.
  intros P [] H.
  - left. rewrite H. reflexivity.
```

```
  - right. rewrite H. intros contra. inversion contra.
Qed.
```

In particular, the excluded middle is valid for equations $n = m$, between natural numbers n and m.

```
Theorem restricted_excluded_middle_eq : ∀ (n m : nat),
  n = m ∨ n ≠ m.
Proof.
  intros n m.
  apply (restricted_excluded_middle (n = m) (beq_nat n m)).
  symmetry.
  apply beq_nat_true_iff.
Qed.
```

It may seem strange that the general excluded middle is not available by default in Coq; after all, any given claim must be either true or false. Nonetheless, there is an advantage in not assuming the excluded middle: statements in Coq can make stronger claims than the analogous statements in standard mathematics. Notably, if there is a Coq proof of $\exists$ x, P x, it is possible to explicitly exhibit a value of x for which we can prove P x – in other words, every proof of existence is necessarily *constructive*.

Logics like Coq's, which do not assume the excluded middle, are referred to as *constructive logics*.

More conventional logical systems such as ZFC, in which the excluded middle does hold for arbitrary propositions, are referred to as *classical*.

The following example illustrates why assuming the excluded middle may lead to non-constructive proofs:

Claim: There exist irrational numbers a and b such that a ^ b is rational.

Proof: It is not difficult to show that *sqrt* 2 is irrational. If *sqrt* 2 ^ *sqrt* 2 is rational, it suffices to take a = b = *sqrt* 2 and we are done. Otherwise, *sqrt* 2 ^ *sqrt* 2 is irrational. In this case, we can take a = *sqrt* 2 ^ *sqrt* 2 and b = *sqrt* 2, since a ^ b = *sqrt* 2 ^ (*sqrt* 2 × *sqrt* 2) = *sqrt* 2 ^ 2 = 2. □

Do you see what happened here? We used the excluded middle to consider separately the cases where *sqrt* 2 ^ *sqrt* 2 is rational and where it is not, without knowing which one actually holds! Because of that, we wind up knowing that such a and b exist but we cannot determine what their actual values are (at least, using this line of argument).

As useful as constructive logic is, it does have its limitations: There are many statements that can easily be proven in classical logic but that have much more complicated constructive proofs, and there are some that are known to have no constructive proof at all! Fortunately, like functional extensionality, the excluded middle is known to be compatible with Coq's logic, allowing us to add it safely as an axiom. However, we will not need to do so in this book: the results that we cover can be developed entirely within constructive logic at negligible extra cost.

It takes some practice to understand which proof techniques must be avoided in constructive reasoning, but arguments by contradiction, in particular, are infamous for leading to non-constructive proofs. Here's a typical example: suppose that we want to show that there exists x with some property P, i.e., such that P x. We start by assuming that our conclusion is false; that is, $\neg \exists$ x, P x. From this premise, it is not hard to derive $\forall$ x, $\neg$ P x. If we manage to show that this intermediate fact results in a contradiction, we arrive at an existence proof without ever exhibiting a value of x for which P x holds!

The technical flaw here, from a constructive standpoint, is that we claimed to prove $\exists$ x, P x using a proof of $\neg\neg$ ($\exists$ x, P x). Allowing ourselves to remove double negations from arbitrary statements is equivalent to assuming the excluded middle, as shown in one of the exercises below. Thus, this line of reasoning cannot be encoded in Coq without assuming additional axioms.

Exercise: 3 stars (excluded_middle_irrefutable) The consistency of Coq with the general excluded middle axiom requires complicated reasoning that cannot be carried out within Coq itself. However, the following theorem implies that it is always safe to assume a decidability axiom (i.e., an instance of excluded middle) for any *particular* Prop P. Why? Because we cannot prove the negation of such an axiom; if we could, we would have both $\neg$ ($P \lor \neg P$) and $\neg\neg$ ($P \lor \neg P$), a contradiction.

```
Theorem excluded_middle_irrefutable: ∀ (P:Prop),
  ¬ ¬ (P ∨ ¬ P).
Proof.
  Admitted.
  □
```

Exercise: 3 stars, advanced (not_exists_dist) It is a theorem of classical logic that the following two assertions are equivalent:

˜ (exists x, ˜ P x) forall x, P x

The dist_not_exists theorem above proves one side of this equivalence. Interestingly, the other direction cannot be proved in constructive logic. Your job is to show that it is implied by the excluded middle.

```
Theorem not_exists_dist :
  excluded_middle →
  ∀ (X:Type) (P : X → Prop),
    ¬ (∃ x, ¬ P x) → (∀ x, P x).
Proof.
  Admitted.
  □
```

Exercise: 5 stars, optional (classical_axioms) For those who like a challenge, here is an exercise taken from the Coq'Art book by Bertot and Casteran (p. 123). Each of the following four statements, together with excluded_middle, can be considered as characterizing classical logic. We can't prove any of them in Coq, but we can consistently add any one of them as an axiom if we wish to work in classical logic.

Prove that all five propositions (these four plus excluded_middle) are equivalent.

Definition *peirce* := $\forall$ P Q: Prop,
$((P \rightarrow Q)$->$P)$->P.

Definition *double_negation_elimination* := $\forall$ P:Prop,
~~$P \rightarrow P$.

Definition *de_morgan_not_and_not* := $\forall$ P Q:Prop,
~(~$P \land \neg Q) \rightarrow P \lor Q$.

Definition *implies_to_or* := $\forall$ P Q:Prop,
$(P \rightarrow Q) \rightarrow$ (~$P \lor Q)$.

$\square$

$Date : 2016 - 12 - 1816 : 20 : 32 - 0500(Sun, 18Dec2016)$

Chapter 9

Library Top.IndProp

9.1 IndProp: Inductively Defined Propositions

```
Require Export Logic.
```

9.2 Inductively Defined Propositions

In the Logic chapter, we looked at several ways of writing propositions, including conjunction, disjunction, and quantifiers. In this chapter, we bring a new tool into the mix: *inductive definitions*.

Recall that we have seen two ways of stating that a number n is even: We can say (1) evenb n = true, or (2) ∃ k, n = double k. Yet another possibility is to say that n is even if we can establish its evenness from the following rules:

- Rule ev_0: The number 0 is even.
- Rule ev_SS: If n is even, then S (S n) is even.

To illustrate how this definition of evenness works, let's imagine using it to show that 4 is even. By rule ev_SS, it suffices to show that 2 is even. This, in turn, is again guaranteed by rule ev_SS, as long as we can show that 0 is even. But this last fact follows directly from the ev_0 rule.

We will see many definitions like this one during the rest of the course. For purposes of informal discussions, it is helpful to have a lightweight notation that makes them easy to read and write. *Inference rules* are one such notation:

(ev_0) ev 0

ev n

(ev_SS) ev (S (S n))

Each of the textual rules above is reformatted here as an inference rule; the intended reading is that, if the *premises* above the line all hold, then the *conclusion* below the line follows. For example, the rule ev_SS says that, if n satisfies **ev**, then S (S n) also does. If a rule has no premises above the line, then its conclusion holds unconditionally.

We can represent a proof using these rules by combining rule applications into a *proof tree*. Here's how we might transcribe the above proof that 4 is even:

(ev_0) ev 0

(ev_SS) ev 2

(ev_SS) ev 4

Why call this a "tree" (rather than a "stack", for example)? Because, in general, inference rules can have multiple premises. We will see examples of this below.

Putting all of this together, we can translate the definition of evenness into a formal Coq definition using an `Inductive` declaration, where each constructor corresponds to an inference rule:

```
Inductive ev : nat → Prop :=
| ev_0 : ev 0
| ev_SS : ∀ n : nat, ev n → ev (S (S n)).
```

This definition is different in one crucial respect from previous uses of `Inductive`: its result is not a `Type`, but rather a function from nat to `Prop` – that is, a property of numbers. Note that we've already seen other inductive definitions that result in functions, such as **list**, whose type is `Type → Type`. What is new here is that, because the nat argument of **ev** appears *unnamed*, to the *right* of the colon, it is allowed to take different values in the types of different constructors: 0 in the type of ev_0 and S (S n) in the type of ev_SS.

In contrast, the definition of **list** names the X parameter *globally*, to the *left* of the colon, forcing the result of nil and cons to be the same (**list** X). Had we tried to bring nat to the left in defining **ev**, we would have seen an error:

```
Fail Inductive wrong_ev (n : nat) : Prop :=
| wrong_ev_0 : wrong_ev 0
| wrong_ev_SS : ∀ n, wrong_ev n → wrong_ev (S (S n)).
```

("Parameter" here is Coq jargon for an argument on the left of the colon in an `Inductive` definition; "index" is used to refer to arguments on the right of the colon.)

We can think of the definition of **ev** as defining a Coq property **ev** : nat → `Prop`, together with theorems ev_0 : **ev** 0 and ev_SS : ∀ n, **ev** n → **ev** (S (S n)). Such "constructor theorems" have the same status as proven theorems. In particular, we can use Coq's `apply` tactic with the rule names to prove **ev** for particular numbers...

```
Theorem ev_4 : ev 4.
Proof. apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

... or we can use function application syntax:

```
Theorem ev_4' : ev 4.
Proof. apply (ev_SS 2 (ev_SS 0 ev_0)). Qed.
```

We can also prove theorems that have hypotheses involving **ev**.

```
Theorem ev_plus4 : ∀ n, ev n → ev (4 + n).
Proof.
  intros n. simpl. intros Hn.
  apply ev_SS. apply ev_SS. apply Hn.
Qed.
```

More generally, we can show that any number multiplied by 2 is even:

Exercise: 1 star (ev_double) `Theorem ev_double : ∀ n,`

```
  ev (double n).
Proof.
   Admitted.
```

☐

9.3 Using Evidence in Proofs

Besides *constructing* evidence that numbers are even, we can also *reason about* such evidence.

Introducing **ev** with an `Inductive` declaration tells Coq not only that the constructors ev_0 and ev_SS are valid ways to build evidence that some number is even, but also that these two constructors are the *only* ways to build evidence that numbers are even (in the sense of **ev**).

In other words, if someone gives us evidence E for the assertion **ev** n, then we know that E must have one of two shapes:

- E is ev_0 (and n is O), or
- E is ev_SS *n' E'* (and n is S (S *n'*), where *E'* is evidence for **ev** *n'*).

This suggests that it should be possible to analyze a hypothesis of the form **ev** n much as we do inductively defined data structures; in particular, it should be possible to argue by *induction* and *case analysis* on such evidence. Let's look at a few examples to see what this means in practice.

9.3.1 Inversion on Evidence

Suppose we are proving some fact involving a number n, and we are given **ev** n as a hypothesis. We already know how to perform case analysis on n using the `inversion` tactic, generating separate subgoals for the case where n = O and the case where n = S *n'* for some *n'*. But for some proofs we may instead want to analyze the evidence that **ev** n *directly*.

By the definition of **ev**, there are two cases to consider:

- If the evidence is of the form ev_0, we know that n = 0.
- Otherwise, the evidence must have the form ev_SS *n' E'*, where n = S (S *n'*) and *E'* is evidence for **ev** *n'*.

We can perform this kind of reasoning in Coq, again using the `inversion` tactic. Besides allowing us to reason about equalities involving constructors, `inversion` provides a case-analysis principle for inductively defined propositions. When used in this way, its syntax is similar to `destruct`: We pass it a list of identifiers separated by | characters to name the arguments to each of the possible constructors.

```
Theorem ev_minus2 : ∀ n,
  ev n → ev (pred (pred n)).
Proof.
  intros n E.
  inversion E as [| n' E'].
  - simpl. apply ev_0.
  - simpl. apply E'. Qed.
```

In words, here is how the inversion reasoning works in this proof:

- If the evidence is of the form ev_0, we know that n = 0. Therefore, it suffices to show that **ev** (pred (pred 0)) holds. By the definition of pred, this is equivalent to showing that **ev** 0 holds, which directly follows from ev_0.
- Otherwise, the evidence must have the form ev_SS *n' E'*, where n = S (S *n'*) and *E'* is evidence for **ev** *n'*. We must then show that **ev** (pred (pred (S (S *n'*)))) holds, which, after simplification, follows directly from *E'*.

This particular proof also works if we replace `inversion` by `destruct`:

```
Theorem ev_minus2' : ∀ n,
  ev n → ev (pred (pred n)).
Proof.
  intros n E.
  destruct E as [| n' E'].
  - simpl. apply ev_0.
  - simpl. apply E'. Qed.
```

The difference between the two forms is that `inversion` is more convenient when used on a hypothesis that consists of an inductive property applied to a complex expression (as opposed to a single variable). Here's is a concrete example. Suppose that we wanted to prove the following variation of ev_minus2:

```
Theorem evSS_ev : ∀ n,
  ev (S (S n)) → ev n.
```

Intuitively, we know that evidence for the hypothesis cannot consist just of the ev_0 constructor, since O and S are different constructors of the type nat; hence, ev_SS is the only case that applies. Unfortunately, destruct is not smart enough to realize this, and it still generates two subgoals. Even worse, in doing so, it keeps the final goal unchanged, failing to provide any useful information for completing the proof.

```
Proof.
  intros n E.
  destruct E as [| n' E'].
  -


Abort.
```

What happened, exactly? Calling destruct has the effect of replacing all occurrences of the property argument by the values that correspond to each constructor. This is enough in the case of ev_minus2' because that argument, n, is mentioned directly in the final goal. However, it doesn't help in the case of evSS_ev since the term that gets replaced (S (S n)) is not mentioned anywhere.

The inversion tactic, on the other hand, can detect (1) that the first case does not apply, and (2) that the n' that appears on the ev_SS case must be the same as n. This allows us to complete the proof:

```
Theorem evSS_ev : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n E.
  inversion E as [| n' E'].
  apply E'.
Qed.
```

By using inversion, we can also apply the principle of explosion to "obviously contradictory" hypotheses involving inductive properties. For example:

```
Theorem one_not_even : ¬ ev 1.
Proof.
  intros H. inversion H. Qed.
```

Exercise: 1 star (inversion_practice) Prove the following results using inversion.

```
Theorem SSSSev__even : ∀ n,
  ev (S (S (S (S n)))) → ev n.
Proof.
   Admitted.

Theorem even5_nonsense :
  ev 5 → 2 + 2 = 9.
Proof.
   Admitted.
```

□

The way we've used `inversion` here may seem a bit mysterious at first. Until now, we've only used `inversion` on equality propositions, to utilize injectivity of constructors or to discriminate between different constructors. But we see here that `inversion` can also be applied to analyzing evidence for inductively defined propositions.

Here's how `inversion` works in general. Suppose the name I refers to an assumption *P* in the current context, where *P* has been defined by an `Inductive` declaration. Then, for each of the constructors of *P*, `inversion` I generates a subgoal in which I has been replaced by the exact, specific conditions under which this constructor could have been used to prove *P*. Some of these subgoals will be self-contradictory; `inversion` throws these away. The ones that are left represent the cases that must be proved to establish the original goal. For those, `inversion` adds all equations into the proof context that must hold of the arguments given to *P* (e.g., S (S *n'*) = n in the proof of evSS_ev).

The ev_double exercise above shows that our new notion of evenness is implied by the two earlier ones (since, by even_bool_prop in chapter Logic, we already know that those are equivalent to each other). To show that all three coincide, we just need the following lemma:

```
Lemma ev_even_firsttry : ∀ n,
  ev n → ∃ k, n = double k.
Proof.
```

We could try to proceed by case analysis or induction on n. But since **ev** is mentioned in a premise, this strategy would probably lead to a dead end, as in the previous section. Thus, it seems better to first try inversion on the evidence for **ev**. Indeed, the first case can be solved trivially.

```
  intros n E. inversion E as [| n' E'].
  -
    ∃ 0. reflexivity.
  - simpl.
```

Unfortunately, the second case is harder. We need to show ∃ k, S (S *n'*) = double k, but the only available assumption is *E'*, which states that **ev** *n'* holds. Since this isn't directly useful, it seems that we are stuck and that performing case analysis on *E* was a waste of time.

If we look more closely at our second goal, however, we can see that something interesting happened: By performing case analysis on *E*, we were able to reduce the original result to an similar one that involves a *different* piece of evidence for **ev**: *E'*. More formally, we can finish our proof by showing that

exists k', n' = double k',

which is the same as the original statement, but with *n'* instead of n. Indeed, it is not difficult to convince Coq that this intermediate result suffices.

```
    assert (I : (∃ k', n' = double k') →
                (∃ k, S (S n') = double k)).
    { intros [k' Hk']. rewrite Hk'. ∃ (S k'). reflexivity. }
    apply I.
```

Admitted.

9.3.2 Induction on Evidence

If this looks familiar, it is no coincidence: We've encountered similar problems in the `Induction` chapter, when trying to use case analysis to prove results that required induction. And once again the solution is... induction!

The behavior of `induction` on evidence is the same as its behavior on data: It causes Coq to generate one subgoal for each constructor that could have used to build that evidence, while providing an induction hypotheses for each recursive occurrence of the property in question.

Let's try our current lemma again:

```
Lemma ev_even : ∀ n,
  ev n → ∃ k, n = double k.
Proof.
  intros n E.
  induction E as [|n' E' IH].
  -
    ∃ 0. reflexivity.
  -
    destruct IH as [k' Hk'].
    rewrite Hk'. ∃ (S k'). reflexivity.
Qed.
```

Here, we can see that Coq produced an *IH* that corresponds to *E'*, the single recursive occurrence of **ev** in its own definition. Since *E'* mentions *n'*, the induction hypothesis talks about *n'*, as opposed to `n` or some other number.

The equivalence between the second and third definitions of evenness now follows.

```
Theorem ev_even_iff : ∀ n,
  ev n ↔ ∃ k, n = double k.
Proof.
  intros n. split.
  - apply ev_even.
  - intros [k Hk]. rewrite Hk. apply ev_double.
Qed.
```

As we will see in later chapters, induction on evidence is a recurring technique across many areas, and in particular when formalizing the semantics of programming languages, where many properties of interest are defined inductively.

The following exercises provide simple examples of this technique, to help you familiarize yourself with it.

Exercise: 2 stars (ev_sum) `Theorem ev_sum : ∀ n m, ev n → ev m → ev (n + m).`

```
Proof.
  Admitted.
```
□

Exercise: 4 stars, advanced, optional (ev_alternate) In general, there may be multiple ways of defining a property inductively. For example, here's a (slightly contrived) alternative definition for **ev**:

```
Inductive ev' : nat → Prop :=
| ev'_0 : ev' 0
| ev'_2 : ev' 2
| ev'_sum : ∀ n m, ev' n → ev' m → ev' (n + m).
```

Prove that this definition is logically equivalent to the old one. (You may want to look at the previous theorem when you get to the induction step.)

```
Theorem ev'_ev : ∀ n, ev' n ↔ ev n.
Proof.
  Admitted.
```
□

Exercise: 3 stars, advanced, recommended (ev_ev__ev) Finding the appropriate thing to do induction on is a bit tricky here:

```
Theorem ev_ev__ev : ∀ n m,
  ev (n+m) → ev n → ev m.
Proof.
  Admitted.
```
□

Exercise: 3 stars, optional (ev_plus_plus) This exercise just requires applying existing lemmas. No induction or even case analysis is needed, though some of the rewriting may be tedious.

```
Theorem ev_plus_plus : ∀ n m p,
  ev (n+m) → ev (n+p) → ev (m+p).
Proof.
  Admitted.
```
□

9.4 Inductive Relations

A proposition parameterized by a number (such as **ev**) can be thought of as a *property* – i.e., it defines a subset of nat, namely those numbers for which the proposition is provable.

In the same way, a two-argument proposition can be thought of as a *relation* – i.e., it defines a set of pairs for which the proposition is provable.

```
Module Playground.
```

One useful example is the "less than or equal to" relation on numbers.

The following definition should be fairly intuitive. It says that there are two ways to give evidence that one number is less than or equal to another: either observe that they are the same number, or give evidence that the first is less than or equal to the predecessor of the second.

```
Inductive le : nat → nat → Prop :=
  | le_n : ∀ n, le n n
  | le_S : ∀ n m, (le n m) → (le n (S m)).
```

```
Notation "m <= n" := (le m n).
```

Proofs of facts about ≤ using the constructors `le_n` and `le_S` follow the same patterns as proofs about properties, like **ev** above. We can `apply` the constructors to prove ≤ goals (e.g., to show that 3<=3 or 3<=6), and we can use tactics like `inversion` to extract information from ≤ hypotheses in the context (e.g., to prove that $(2 \leq 1) \to 2+2=5$.)

Here are some sanity checks on the definition. (Notice that, although these are the same kind of simple "unit tests" as we gave for the testing functions we wrote in the first few lectures, we must construct their proofs explicitly – `simpl` and `reflexivity` don't do the job, because the proofs aren't just a matter of simplifying computations.)

```
Theorem test_le1 :
  3 ≤ 3.
Proof.
  apply le_n. Qed.
```

```
Theorem test_le2 :
  3 ≤ 6.
Proof.
  apply le_S. apply le_S. apply le_S. apply le_n. Qed.
```

```
Theorem test_le3 :
  (2 ≤ 1) → 2 + 2 = 5.
Proof.
  intros H. inversion H. inversion H2. Qed.
```

The "strictly less than" relation `n < m` can now be defined in terms of **le**.

```
End Playground.
```

```
Definition lt (n m:nat) := le (S n) m.
```

```
Notation "m < n" := (lt m n).
```

Here are a few more simple relations on numbers:

```
Inductive square_of : nat → nat → Prop :=
```

| sq : ∀ n:nat, $square\_of$ n ($n \times n$).

```
Inductive next_nat : nat → nat → Prop :=
  | nn : ∀ n:nat, next_nat n (S n).
```

```
Inductive next_even : nat → nat → Prop :=
  | ne_1 : ∀ n, ev (S n) → next_even n (S n)
  | ne_2 : ∀ n, ev (S (S n)) → next_even n (S (S n)).
```

Exercise: 2 stars, optional (total_relation) Define an inductive binary relation *total_relation* that holds between every pair of natural numbers.

☐

Exercise: 2 stars, optional (empty_relation) Define an inductive binary relation *empty_relation* (on numbers) that never holds.

☐

Exercise: 3 stars, optional (le_exercises) Here are a number of facts about the ≤ and < relations that we are going to need later in the course. The proofs make good practice exercises.

```
Lemma le_trans : ∀ m n o, m ≤ n → n ≤ o → m ≤ o.
Proof.
    Admitted.

Theorem O_le_n : ∀ n,
  0 ≤ n.
Proof.
    Admitted.

Theorem n_le_m__Sn_le_Sm : ∀ n m,
  n ≤ m → S n ≤ S m.
Proof.
    Admitted.

Theorem Sn_le_Sm__n_le_m : ∀ n m,
  S n ≤ S m → n ≤ m.
Proof.
    Admitted.

Theorem le_plus_l : ∀ a b,
  a ≤ a + b.
Proof.
    Admitted.

Theorem plus_lt : ∀ n1 n2 m,
  n1 + n2 < m →
```

$n1 < m \land n2 < m$.

```
Proof.
 unfold lt.
   Admitted.
```

`Theorem` *lt_S* : $\forall$ n m,
  $n < m \to$
  $n < S\ m$.

```
Proof.
   Admitted.
```

`Theorem` *leb_complete* : $\forall$ n m,
  $leb\ n\ m = true \to n \leq m$.

```
Proof.
   Admitted.
```

Hint: The next one may be easiest to prove by induction on `m`.

`Theorem` *leb_correct* : $\forall$ n m,
  $n \leq m \to$
  $leb\ n\ m = true$.

```
Proof.
   Admitted.
```

Hint: This theorem can easily be proved without using `induction`.

`Theorem` *leb_true_trans* : $\forall$ n m o,
  $leb\ n\ m = true \to leb\ m\ o = true \to leb\ n\ o = true$.

```
Proof.
   Admitted.
```

Exercise: 2 stars, optional (leb_iff) `Theorem` *leb_iff* : $\forall$ n m,
  $leb\ n\ m = true \leftrightarrow n \leq m$.

```
Proof.
   Admitted.
```

☐

`Module` *R*.

Exercise: 3 stars, recommendedM (R_provability) We can define three-place relations, four-place relations, etc., in just the same way as binary relations. For example, consider the following three-place relation on numbers:

`Inductive` R : $nat \to nat \to nat \to$ `Prop` :=
  | $c1$: $R\ 0\ 0\ 0$
  | $c2$: $\forall$ $m\ n\ o$, $R\ m\ n\ o \to R\ (S\ m)\ n\ (S\ o)$
  | $c3$: $\forall$ $m\ n\ o$, $R\ m\ n\ o \to R\ m\ (S\ n)\ (S\ o)$
  | $c4$: $\forall$ $m\ n\ o$, $R\ (S\ m)\ (S\ n)\ (S\ (S\ o)) \to R\ m\ n\ o$

| $c5 : \forall\ m\ n\ o,\ R\ m\ n\ o \rightarrow R\ n\ m\ o.$

- Which of the following propositions are provable?
  - R 1 1 2
  - R 2 2 6
- If we dropped constructor c5 from the definition of R, would the set of provable propositions change? Briefly (1 sentence) explain your answer.
- If we dropped constructor c4 from the definition of R, would the set of provable propositions change? Briefly (1 sentence) explain your answer.

☐

Exercise: 3 stars, optional (R_fact) The relation R above actually encodes a familiar function. Figure out which function; then state and prove this equivalence in Coq?

```
Definition fR : nat → nat → nat
  . Admitted.

Theorem R_equiv_fR : ∀ m n o, R m n o ↔ fR m n = o.
Proof.
   Admitted.
   ☐

End R.
```

Exercise: 4 stars, advanced (subsequence) A list is a *subsequence* of another list if all of the elements in the first list occur in the same order in the second list, possibly with some extra elements in between. For example,

1;2;3
is a subsequence of each of the lists
1;2;3 1;1;1;2;2;3 1;2;7;3 5;6;1;9;9;2;7;3;8
but it is *not* a subsequence of any of the lists
1;2 1;3 5;6;2;1;7;3;8.

- Define an inductive proposition *subseq* on **list** nat that captures what it means to be a subsequence. (Hint: You'll need three cases.)
- Prove *subseq_refl* that subsequence is reflexive, that is, any list is a subsequence of itself.
- Prove *subseq_app* that for any lists l1, l2, and l3, if l1 is a subsequence of l2, then l1 is also a subsequence of l2 ++ l3.

- (Optional, harder) Prove *subseq_trans* that subsequence is transitive – that is, if l1 is a subsequence of l2 and l2 is a subsequence of l3, then l1 is a subsequence of l3. Hint: choose your induction carefully!

□

Exercise: 2 stars, optionalM (R_provability2) Suppose we give Coq the following definition:

Inductive R : nat -> list nat -> Prop := | c1 : R 0 □ | c2 : forall n l, R n l -> R (S n) (n :: l) | c3 : forall n l, R (S n) l -> R n l.

Which of the following propositions are provable?

- R 2 [1;0]
- R 1 [1;2;1;0]
- R 6 [3;2;1;0]

□

9.5 Case Study: Regular Expressions

The **ev** property provides a simple example for illustrating inductive definitions and the basic techniques for reasoning about them, but it is not terribly exciting – after all, it is equivalent to the two non-inductive of evenness that we had already seen, and does not seem to offer any concrete benefit over them. To give a better sense of the power of inductive definitions, we now show how to use them to model a classic concept in computer science: *regular expressions.*

Regular expressions are a simple language for describing strings, defined as follows:

```
Inductive reg_exp (T : Type) : Type :=
| EmptySet : reg_exp T
| EmptyStr : reg_exp T
| Char : T → reg_exp T
| App : reg_exp T → reg_exp T → reg_exp T
| Union : reg_exp T → reg_exp T → reg_exp T
| Star : reg_exp T → reg_exp T.

Arguments EmptySet {T}.
Arguments EmptyStr {T}.
Arguments Char {T} _.
Arguments App {T} _ _.
Arguments Union {T} _ _.
Arguments Star {T} _.
```

Note that this definition is *polymorphic*: Regular expressions in **reg_exp** T describe strings with characters drawn from T – that is, lists of elements of T.

(We depart slightly from standard practice in that we do not require the type T to be finite. This results in a somewhat different theory of regular expressions, but the difference is not significant for our purposes.)

We connect regular expressions and strings via the following rules, which define when a regular expression *matches* some string:

- The expression EmptySet does not match any string.
- The expression EmptyStr matches the empty string [].
- The expression Char x matches the one-character string [x].
- If *re1* matches *s1*, and *re2* matches *s2*, then App *re1 re2* matches *s1* ++ *s2*.
- If at least one of *re1* and *re2* matches s, then Union *re1 re2* matches s.
- Finally, if we can write some string s as the concatenation of a sequence of strings s = *s_1* ++ ... ++ *s_k*, and the expression *re* matches each one of the strings *s_i*, then Star *re* matches s.

  As a special case, the sequence of strings may be empty, so Star *re* always matches the empty string [] no matter what *re* is.

We can easily translate this informal definition into an `Inductive` one as follows:

```
Inductive exp_match {T} : list T → reg_exp T → Prop :=
| MEmpty : exp_match [] EmptyStr
| MChar : ∀ x, exp_match [x] (Char x)
| MApp : ∀ s1 re1 s2 re2,
            exp_match s1 re1 →
            exp_match s2 re2 →
            exp_match (s1 ++ s2) (App re1 re2)
| MUnionL : ∀ s1 re1 re2,
              exp_match s1 re1 →
              exp_match s1 (Union re1 re2)
| MUnionR : ∀ re1 s2 re2,
              exp_match s2 re2 →
              exp_match s2 (Union re1 re2)
| MStar0 : ∀ re, exp_match [] (Star re)
| MStarApp : ∀ s1 s2 re,
               exp_match s1 re →
               exp_match s2 (Star re) →
               exp_match (s1 ++ s2) (Star re).
```

Again, for readability, we can also display this definition using inference-rule notation. At the same time, let's introduce a more readable infix notation.

```
Notation "s =~ re" := (exp_match s re) (at level 80).
```

(MEmpty) □ =˜ EmptyStr

(MChar) x =˜ Char x
s1 =˜ re1 s2 =˜ re2

(MApp) s1 ++ s2 =˜ App re1 re2
s1 =˜ re1

(MUnionL) s1 =˜ Union re1 re2
s2 =˜ re2

(MUnionR) s2 =˜ Union re1 re2

(MStar0) □ =˜ Star re
s1 =˜ re s2 =˜ Star re

(MStarApp) s1 ++ s2 =˜ Star re

Notice that these rules are not *quite* the same as the informal ones that we gave at the beginning of the section. First, we don't need to include a rule explicitly stating that no string matches EmptySet; we just don't happen to include any rule that would have the effect of some string matching EmptySet. (Indeed, the syntax of inductive definitions doesn't even *allow* us to give such a "negative rule.")

Second, the informal rules for Union and Star correspond to two constructors each: MUnionL / MUnionR, and MStar0 / MStarApp. The result is logically equivalent to the original rules but more convenient to use in Coq, since the recursive occurrences of **exp_match** are given as direct arguments to the constructors, making it easier to perform induction on evidence. (The *exp_match_ex1* and *exp_match_ex2* exercises below ask you to prove that the constructors given in the inductive declaration and the ones that would arise from a more literal transcription of the informal rules are indeed equivalent.)

Let's illustrate these rules with a few examples.

```
Example reg_exp_ex1 : [1] =~ Char 1.
Proof.
  apply MChar.
Qed.
```

```
Example reg_exp_ex2 : [1; 2] =~ App (Char 1) (Char 2).
Proof.
  apply (MApp [1] _ [2]).
```

```
  - apply MChar.
  - apply MChar.
Qed.
```

(Notice how the last example applies MApp to the strings [1] and [2] directly. Since the goal mentions [1; 2] instead of [1] ++ [2], Coq wouldn't be able to figure out how to split the string on its own.)

Using inversion, we can also show that certain strings do *not* match a regular expression:

```
Example reg_exp_ex3 : ¬ ([1; 2] =~ Char 1).
Proof.
  intros H. inversion H.
Qed.
```

We can define helper functions to help write down regular expressions. The reg_exp_of_list function constructs a regular expression that matches exactly the list that it receives as an argument:

```
Fixpoint reg_exp_of_list {T} (l : list T) :=
  match l with
  | [] ⇒ EmptyStr
  | x :: l' ⇒ App (Char x) (reg_exp_of_list l')
  end.

Example reg_exp_ex4 : [1; 2; 3] =~ reg_exp_of_list [1; 2; 3].
Proof.
  simpl. apply (MApp [1]).
  { apply MChar. }
  apply (MApp [2]).
  { apply MChar. }
  apply (MApp [3]).
  { apply MChar. }
  apply MEmpty.
Qed.
```

We can also prove general facts about **exp_match**. For instance, the following lemma shows that every string s that matches *re* also matches Star *re*.

```
Lemma MStar1 :
  ∀ T s (re : reg_exp T) ,
    s =~ re →
    s =~ Star re.
Proof.
  intros T s re H.
  rewrite ← (app_nil_r _ s).
  apply (MStarApp s [] re).
  - apply H.
```

```
  - apply MStar0.
Qed.
```

(Note the use of app_nil_r to change the goal of the theorem to exactly the same shape expected by MStarApp.)

Exercise: 3 stars (exp_match_ex1) The following lemmas show that the informal matching rules given at the beginning of the chapter can be obtained from the formal inductive definition.

```
Lemma empty_is_empty : ∀ T (s : list T),
  ¬ (s =~ EmptySet).
Proof.
   Admitted.

Lemma MUnion' : ∀ T (s : list T) (re1 re2 : reg_exp T),
  s =~ re1 ∨ s =~ re2 →
  s =~ Union re1 re2.
Proof.
   Admitted.
```

The next lemma is stated in terms of the fold function from the Poly chapter: If *ss* : **list** (**list** T) represents a sequence of strings *s1*, ..., *sn*, then fold app *ss* [] is the result of concatenating them all together.

```
Lemma MStar' : ∀ T (ss : list (list T)) (re : reg_exp T),
  (∀ s, In s ss → s =~ re) →
  fold app ss [] =~ Star re.
Proof.
   Admitted.
```

□

Exercise: 4 stars (reg_exp_of_list) Prove that reg_exp_of_list satisfies the following specification:

```
Lemma reg_exp_of_list_spec : ∀ T (s1 s2 : list T),
  s1 =~ reg_exp_of_list s2 ↔ s1 = s2.
Proof.
   Admitted.
```

□

Since the definition of **exp_match** has a recursive structure, we might expect that proofs involving regular expressions will often require induction on evidence. For example, suppose that we wanted to prove the following intuitive result: If a regular expression *re* matches some string s, then all elements of s must occur somewhere in *re*. To state this theorem, we first define a function re_chars that lists all characters that occur in a regular expression:

```
Fixpoint re_chars {T} (re : reg_exp T) : list T :=
```

```
  match re with
  | EmptySet ⇒ []
  | EmptyStr ⇒ []
  | Char x ⇒ [x]
  | App re1 re2 ⇒ re_chars re1 ++ re_chars re2
  | Union re1 re2 ⇒ re_chars re1 ++ re_chars re2
  | Star re ⇒ re_chars re
  end.
```

We can then phrase our theorem as follows:

```
Theorem in_re_match : ∀ T (s : list T) (re : reg_exp T) (x : T),
  s =~ re →
  In x s →
  In x (re_chars re).
Proof.
  intros T s re x Hmatch Hin.
  induction Hmatch
    as [
        |x'
        |s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2 re2 Hmatch IH
        |re|s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
  -
    apply Hin.
  -
    apply Hin.
  - simpl. rewrite in_app_iff in *.
    destruct Hin as [Hin | Hin].
    +
      left. apply (IH1 Hin).
    +
      right. apply (IH2 Hin).
  -
    simpl. rewrite in_app_iff.
    left. apply (IH Hin).
  -
    simpl. rewrite in_app_iff.
    right. apply (IH Hin).
  -
    destruct Hin.
```

Something interesting happens in the MStarApp case. We obtain *two* induction hypotheses: One that applies when x occurs in *s1* (which matches *re*), and a second one that applies when x occurs in *s2* (which matches Star *re*). This is a good illustration of why we need induction on evidence for **exp_match**, as opposed to *re*: The latter would only provide an induction hypothesis for strings that match *re*, which would not allow us to reason about the case In x *s2*.

```
-
  simpl. rewrite in_app_iff in Hin.
  destruct Hin as [Hin | Hin].
  +
    apply (IH1 Hin).
  +
    apply (IH2 Hin).
Qed.
```

Exercise: 4 stars (re_not_empty) Write a recursive function re_not_empty that tests whether a regular expression matches some string. Prove that your function is correct.

```
Fixpoint re_not_empty {T : Type} (re : reg_exp T) : bool
  . Admitted.

Lemma re_not_empty_correct : ∀ T (re : reg_exp T),
  (∃ s, s =~ re) ↔ re_not_empty re = true.
Proof.
   Admitted.
   □
```

9.5.1 The *remember* Tactic

One potentially confusing feature of the `induction` tactic is that it happily lets you try to set up an induction over a term that isn't sufficiently general. The effect of this is to lose information (much as `destruct` can do), and leave you unable to complete the proof. Here's an example:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
```

Just doing an `inversion` on *H1* won't get us very far in the recursive cases. (Try it!). So we need induction. Here is a naive first attempt:

```
  induction H1
    as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
        |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

But now, although we get seven cases (as we would expect from the definition of **exp_match**), we have lost a very important bit of information from *H1*: the fact that *s1* matched something of the form Star *re*. This means that we have to give proofs for *all* seven constructors of this definition, even though all but two of them (MStar0 and MStarApp) are contradictory. We can still get the proof to go through for a few constructors, such as MEmpty...

-

```
simpl. intros H. apply H.
```

... but most cases get stuck. For MChar, for instance, we must show that
s2 =~ Char x' -> x' :: s2 =~ Char x',
which is clearly impossible.

-

```
Abort.
```

The problem is that `induction` over a Prop hypothesis only works properly with hypotheses that are completely general, i.e., ones in which all the arguments are variables, as opposed to more complex expressions, such as Star *re*.

(In this respect, `induction` on evidence behaves more like `destruct` than like `inversion`.)

We can solve this problem by generalizing over the problematic expressions with an explicit equality:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re re' : reg_exp T),
  s1 =~ re' →
  re' = Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
```

We can now proceed by performing induction over evidence directly, because the argument to the first hypothesis is sufficiently general, which means that we can discharge most cases by inverting the *re'* = Star *re* equality in the context.

This idiom is so common that Coq provides a tactic to automatically generate such equations for us, avoiding thus the need for changing the statements of our theorems.

Invoking the tactic *remember* `e as x` causes Coq to (1) replace all occurrences of the expression `e` by the variable `x`, and (2) add an equation `x` = `e` to the context. Here's how we can use it to show the above result: `Abort.`

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
  remember (Star re) as re'.
```

We now have *Heqre'* : *re'* = Star *re*.

```
  generalize dependent s2.
```

```
induction H1
  as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
      |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
      |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

The *Heqre'* is contradictory in most cases, which allows us to conclude immediately.

```
- inversion Heqre'.
- inversion Heqre'.
- inversion Heqre'.
- inversion Heqre'.
- inversion Heqre'.
```

The interesting cases are those that correspond to Star. Note that the induction hypothesis *IH2* on the MStarApp case mentions an additional premise Star *re''* = Star *re'*, which results from the equality generated by *remember*.

```
-
  inversion Heqre'. intros s H. apply H.

-
  inversion Heqre'. rewrite H0 in IH2, Hmatch1.
  intros s2 H1. rewrite <- app_assoc.
  apply MStarApp.
  + apply Hmatch1.
  + apply IH2.
    * reflexivity.
    * apply H1.
Qed.
```

Exercise: 4 stars (exp_match_ex2) The MStar'' lemma below (combined with its converse, the MStar' exercise above), shows that our definition of **exp_match** for Star is equivalent to the informal one given previously.

```
Lemma MStar'' : ∀ T (s : list T) (re : reg_exp T),
  s =~ Star re ->
  ∃ ss : list (list T),
    s = fold app ss []
    ∧ ∀ s', In s' ss -> s' =~ re.
Proof.
  Admitted.
```

□

Exercise: 5 stars, advanced (pumping) One of the first really interesting theorems in the theory of regular expressions is the so-called *pumping lemma*, which states, informally, that any sufficiently long string s matching a regular expression *re* can be "pumped" by repeating some middle section of s an arbitrary number of times to produce a new string also matching *re*.

To begin, we need to define "sufficiently long." Since we are working in a constructive logic, we actually need to be able to calculate, for each regular expression *re*, the minimum length for strings s to guarantee "pumpability."

```
Module Pumping.

Fixpoint pumping_constant {T} (re : reg_exp T) : nat :=
  match re with
  | EmptySet ⇒ 0
  | EmptyStr ⇒ 1
  | Char _ ⇒ 2
  | App re1 re2 ⇒
      pumping_constant re1 + pumping_constant re2
  | Union re1 re2 ⇒
      pumping_constant re1 + pumping_constant re2
  | Star _ ⇒ 1
  end.
```

Next, it is useful to define an auxiliary function that repeats a string (appends it to itself) some number of times.

```
Fixpoint napp {T} (n : nat) (l : list T) : list T :=
  match n with
  | 0 ⇒ []
  | S n' ⇒ l ++ napp n' l
  end.

Lemma napp_plus: ∀ T (n m : nat) (l : list T),
  napp (n + m) l = napp n l ++ napp m l.
Proof.
  intros T n m l.
  induction n as [|n IHn].
  - reflexivity.
  - simpl. rewrite IHn, app_assoc. reflexivity.
Qed.
```

Now, the pumping lemma itself says that, if s =~ *re* and if the length of s is at least the pumping constant of *re*, then s can be split into three substrings *s1* ++ *s2* ++ *s3* in such a way that *s2* can be repeated any number of times and the result, when combined with *s1* and *s3* will still match *re*. Since *s2* is also guaranteed not to be the empty string, this gives us a (constructive!) way to generate strings matching *re* that are as long as we like.

```
Lemma pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
```

```
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
```

To streamline the proof (which you are to fill in), the `omega` tactic, which is enabled by the following `Require`, is helpful in several places for automatically completing tedious low-level arguments involving equalities or inequalities over natural numbers. We'll return to `omega` in a later chapter, but feel free to experiment with it now if you like. The first case of the induction gives an example of how it is used.

```
Require Import Coq.omega.Omega.

Proof.
  intros T re s Hmatch.
  induction Hmatch
    as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
       | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
       | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
  -
    simpl. omega.
   Admitted.

End Pumping.
```

☐

9.6 Case Study: Improving Reflection

We've seen in the `Logic` chapter that we often need to relate boolean computations to statements in `Prop`. But performing this conversion in the way we did it there can result in tedious proof scripts. Consider the proof of the following theorem:

```
Theorem filter_not_empty_In : ∀ n l,
  filter (beq_nat n) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (beq_nat n m) eqn:H.
    +
      intros _. rewrite beq_nat_true_iff in H. rewrite H.
      left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
Qed.
```

In the first branch after destruct, we explicitly apply the beq_nat_true_iff lemma to the equation generated by destructing beq_nat n m, to convert the assumption beq_nat n m = true into the assumption n = m; then we had to rewrite using this assumption to complete the case.

We can streamline this by defining an inductive proposition that yields a better case-analysis principle for beq_nat n m. Instead of generating an equation such as beq_nat n m = true, which is generally not directly useful, this principle gives us right away the assumption we really need: n = m.

We'll actually define something a bit more general, which can be used with arbitrary properties (and not just equalities):

```
Module FirstTry.

Inductive reflect : Prop → bool → Prop :=
| ReflectT : ∀ (P:Prop), P → reflect P true
| ReflectF : ∀ (P:Prop), ¬ P → reflect P false.
```

Before explaining this, let's rearrange it a little: Since the types of both ReflectT and ReflectF begin with ∀ (*P*:Prop), we can make the definition a bit more readable and easier to work with by making *P* a parameter of the whole Inductive declaration.

```
End FirstTry.

Inductive reflect (P : Prop) : bool → Prop :=
| ReflectT : P → reflect P true
| ReflectF : ¬ P → reflect P false.
```

The **reflect** property takes two arguments: a proposition *P* and a boolean b. Intuitively, it states that the property *P* is *reflected* in (i.e., equivalent to) the boolean b: *P* holds if and only if b = true. To see this, notice that, by definition, the only way we can produce evidence that **reflect** *P* true holds is by showing that *P* is true and using the ReflectT constructor. If we invert this statement, this means that it should be possible to extract evidence for *P* from a proof of **reflect** *P* true. Conversely, the only way to show **reflect** *P* false is by combining evidence for ¬ *P* with the ReflectF constructor.

It is easy to formalize this intuition and show that the two statements are indeed equivalent:

```
Theorem iff_reflect : ∀ P b, (P ↔ b = true) → reflect P b.
Proof.
  intros P b H. destruct b.
  - apply ReflectT. rewrite H. reflexivity.
  - apply ReflectF. rewrite H. intros H'. inversion H'.
Qed.
```

Exercise: 2 stars, recommended (reflect_iff)

```
Theorem reflect_iff : ∀ P b, reflect P b → (P ↔ b = true).
Proof.
```

Admitted.

□

The advantage of **reflect** over the normal "if and only if" connective is that, by destructing a hypothesis or lemma of the form **reflect** P b, we can perform case analysis on b while at the same time generating appropriate hypothesis in the two branches (P in the first subgoal and $\neg P$ in the second).

```
Lemma beq_natP : ∀ n m, reflect (n = m) (beq_nat n m).
Proof.
  intros n m.
  apply iff_reflect. rewrite beq_nat_true_iff. reflexivity.
Qed.
```

The new proof of filter_not_empty_In now goes as follows. Notice how the calls to `destruct` and `apply` are combined into a single call to `destruct`.

(To see this clearly, look at the two proofs of filter_not_empty_In with Coq and observe the differences in proof state at the beginning of the first case of the `destruct`.)

```
Theorem filter_not_empty_In' : ∀ n l,
  filter (beq_nat n) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (beq_natP n m) as [H | H].
    +
      intros _. rewrite H. left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
Qed.
```

Exercise: 3 stars, recommended (beq_natP_practice) Use beq_natP as above to prove the following:

```
Fixpoint count n l :=
  match l with
  | [] ⇒ 0
  | m :: l' ⇒ (if beq_nat n m then 1 else 0) + count n l'
  end.

Theorem beq_natP_practice : ∀ n l,
  count n l = 0 → ~(In n l).
Proof.
  Admitted.
```

□

This technique gives us only a small gain in convenience for the proofs we've seen here, but using **reflect** consistently often leads to noticeably shorter and clearer scripts as proofs get larger. We'll see many more examples in later chapters.

The use of the **reflect** property was popularized by *SSReflect*, a Coq library that has been used to formalize important results in mathematics, including as the 4-color theorem and the Feit-Thompson theorem. The name SSReflect stands for *small-scale reflection*, i.e., the pervasive use of reflection to simplify small proof steps with boolean computations.

9.7 Additional Exercises

Exercise: 3 stars, recommended (nostutter) Formulating inductive definitions of properties is an important skill you'll need in this course. Try to solve this exercise without any help at all.

We say that a list "stutters" if it repeats the same element consecutively. The property "**nostutter** mylist" means that mylist does not stutter. Formulate an inductive definition for **nostutter**. (This is different from the *NoDup* property in the exercise above; the sequence 1;4;1 repeats but does not stutter.)

```
Inductive nostutter {X:Type} : list X → Prop :=

.
```

Make sure each of these tests succeeds, but feel free to change the suggested proof (in comments) if the given one doesn't work for you. Your definition might be different from ours and still be correct, in which case the examples might need a different proof. (You'll notice that the suggested proofs use a number of tactics we haven't talked about, to make them more robust to different possible ways of defining **nostutter**. You can probably just uncomment and use them as-is, but you can also prove each example with more basic tactics.)

```
Example test_nostutter_1: nostutter [3;1;4;1;5;6].
  Admitted.

Example test_nostutter_2: nostutter (@nil nat).
  Admitted.

Example test_nostutter_3: nostutter [5].
  Admitted.

Example test_nostutter_4: not (nostutter [3;1;1;4]).
  Admitted.
```

□

Exercise: 4 stars, advanced (filter_challenge) Let's prove that our definition of filter from the Poly chapter matches an abstract specification. Here is the specification, written out informally in English:

A list l is an "in-order merge" of l1 and l2 if it contains all the same elements as l1 and l2, in the same order as l1 and l2, but possibly interleaved. For example,

1;4;6;2;3

is an in-order merge of

1;6;2

and

4;3.

Now, suppose we have a set X, a function test: X→**bool**, and a list l of type **list** X. Suppose further that l is an in-order merge of two lists, l1 and l2, such that every item in l1 satisfies test and no item in l2 satisfies test. Then filter test l = l1.

Translate this specification into a Coq theorem and prove it. (You'll need to begin by defining what it means for one list to be a merge of two others. Do this with an inductive relation, not a `Fixpoint`.)

☐

Exercise: 5 stars, advanced, optional (filter_challenge_2) A different way to characterize the behavior of filter goes like this: Among all subsequences of l with the property that test evaluates to true on all their members, filter test l is the longest. Formalize this claim and prove it.

☐

Exercise: 4 stars, optional (palindromes) A palindrome is a sequence that reads the same backwards as forwards.

- Define an inductive proposition *pal* on **list** X that captures what it means to be a palindrome. (Hint: You'll need three cases. Your definition should be based on the structure of the list; just having a single constructor like

  c : forall l, l = rev l -> pal l

  may seem obvious, but will not work very well.)

- Prove (*pal_app_rev*) that

  forall l, pal (l ++ rev l).

- Prove (*pal_rev* that)

  forall l, pal l -> l = rev l.

☐

Exercise: 5 stars, optional (palindrome_converse) Again, the converse direction is significantly more difficult, due to the lack of evidence. Using your definition of *pal* from the previous exercise, prove that

forall l, l = rev l -> pal l.

□

Exercise: 4 stars, advanced, optional (NoDup) Recall the definition of the In property from the Logic chapter, which asserts that a value x appears at least once in a list l:

Your first task is to use In to define a proposition *disjoint* X l1 l2, which should be provable exactly when l1 and l2 are lists (with elements of type X) that have no elements in common.

Next, use In to define an inductive proposition *NoDup* X l, which should be provable exactly when l is a list (with elements of type X) where every member is different from every other. For example, *NoDup* nat [1;2;3;4] and *NoDup* **bool** [] should be provable, while *NoDup* nat [1;2;1] and *NoDup* **bool** [true;true] should not be.

Finally, state and prove one or more interesting theorems relating *disjoint*, *NoDup* and ++ (list append).

□

Exercise: 4 stars, advanced, optional (pigeonhole principle) The *pigeonhole principle* states a basic fact about counting: if we distribute more than n items into n pigeonholes, some pigeonhole must contain at least two items. As often happens, this apparently trivial fact about numbers requires non-trivial machinery to prove, but we now have enough...

First prove an easy useful lemma.

```
Lemma in_split : ∀ (X:Type) (x:X) (l:list X),
  In x l →
  ∃ l1 l2, l = l1 ++ x :: l2.
Proof.
  Admitted.
```

Now define a property **repeats** such that **repeats** X l asserts that l contains at least one repeated element (of type X).

```
Inductive repeats {X:Type} : list X → Prop :=

.
```

Now, here's a way to formalize the pigeonhole principle. Suppose list l2 represents a list of pigeonhole labels, and list l1 represents the labels assigned to a list of items. If there are more items than labels, at least two items must have the same label – i.e., list l1 must contain repeats.

This proof is much easier if you use the excluded_middle hypothesis to show that In is decidable, i.e., ∀ x l, (In x l) ∨ ¬ (In x l). However, it is also possible to make the proof go through *without* assuming that In is decidable; if you manage to do this, you will not need the excluded_middle hypothesis.

```
Theorem pigeonhole_principle: ∀ (X:Type) (l1 l2:list X),
   excluded_middle →
   (∀ x, In x l1 → In x l2) →
   length l2 < length l1 →
   repeats l1.
Proof.
   intros X l1. induction l1 as [|x l1' IHl1'].
   Admitted.
```

□

$Date : 2016 - 12 - 1723 : 53 : 20 - 0500(Sat, 17Dec2016)$

Chapter 10

Library Top.Maps

10.1 Maps: Total and Partial Maps

Maps (or dictionaries) are ubiquitous data structures both generally and in the theory of programming languages in particular; 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 return an **option** to indicate success or failure. The latter is defined in terms of the former, using None as the default element.

10.2 The Coq Standard Library

One small digression before we get to maps.

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (and, 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 Coq's standard library stuff. 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.

```
Require Import Coq.Arith.Arith.
Require Import Coq.Bool.Bool.
Require Import Coq.Strings.String.
Require Import Coq.Logic.FunctionalExtensionality.
```

Documentation for the standard library can be found at http://coq.inria.fr/library/.

The `Search` command is a good way to look for theorems involving objects of specific types. Take a minute now to experiment with it.

10.3 Identifiers

First, we need a type for the keys that we use to index into our maps. For this purpose, we again use the type **id** from the Lists chapter. To make this chapter self contained, we repeat its definition here, together with the equality comparison function for **id**s and its fundamental property.

```
Inductive id : Type :=
  | Id : string → id.

Definition beq_id x y :=
  match x,y with
    | Id n1, Id n2 ⇒ if string_dec n1 n2 then true else false
  end.
```

(The function string_dec comes from Coq's string library. If you check its result type, you'll see that it does not actually return a **bool**, but rather a type that looks like {x = y} + {x ≠ y}, called a **sumbool**, which can be thought of as an "evidence-carrying boolean." Formally, an element of **sumbool** is either a proof that two things are equal or a proof that they are unequal, together with a tag indicating which. But for present purposes you can think of it as just a fancy **bool**.)

```
Theorem beq_id_refl : ∀ id, true = beq_id id id.
Proof.
  intros [n]. simpl. destruct (string_dec n n).
  - reflexivity.
  - destruct n0. reflexivity.
Qed.
```

The following useful property of beq_id follows from an analogous lemma about strings:

```
Theorem beq_id_true_iff : ∀ x y : id,
  beq_id x y = true ↔ x = y.
Proof.
   intros [n1] [n2].
   unfold beq_id.
   destruct (string_dec n1 n2).
   - subst. split. reflexivity. reflexivity.
   - split.
     + intros contra. inversion contra.
     + intros H. inversion H. subst. destruct n. reflexivity.
Qed.
```

Similarly:

```
Theorem beq_id_false_iff : ∀ x y : id,
  beq_id x y = false
  ↔ x ≠ y.
```

```
Proof.
  intros x y. rewrite ← beq_id_true_iff.
  rewrite not_true_iff_false. reflexivity. Qed.
```

This useful variant follows just by rewriting:

```
Theorem false_beq_id : ∀ x y : id,
    x ≠ y
    → beq_id x y = false.
Proof.
  intros x y. rewrite beq_id_false_iff.
  intros H. apply H. Qed.
```

10.4 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, where two maps that respond to queries in the same way will be represented as literally the same thing (the very same function), rather than just "equivalent" data structures. This, in turn, simplifies proofs that use maps.

We build 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) := id → A.
```

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

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

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

More interesting is the update function, which (as before) 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.

```
Definition t_update {A:Type} (m : total_map A)
                    (x : id) (v : A) :=
  fun x' ⇒ if beq_id 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 **id**s to **bool**s, where Id 3 is mapped to true and every other key is mapped to false, like this:

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

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

```
Example update_example1 : examplemap (Id "baz") = false.
Proof. reflexivity. Qed.

Example update_example2 : examplemap (Id "foo") = false.
Proof. reflexivity. Qed.

Example update_example3 : examplemap (Id "quux") = false.
Proof. reflexivity. Qed.

Example update_example4 : examplemap (Id "bar") = true.
Proof. reflexivity. Qed.
```

To 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 is discussed in the Logic chapter.)

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

```
Lemma t_apply_empty: ∀ A x v, @t_empty A v x = v.
Proof.
  Admitted.
```

□

Exercise: 2 stars, 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 : ∀ A (m: total_map A) x v,
  (t_update m x v) x = v.
Proof.
  Admitted.
```

□

Exercise: 2 stars, 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 : ∀ (X:Type) v x1 x2
                          (m : total_map X),
  x1 ≠ x2 →
```

$(t\_update\ m\ x1\ v)\ x2 = m\ x2.$

`Proof.`

Admitted.

☐

Exercise: 2 stars, 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* : ∀ *A* (*m*: *total_map A*) *v1 v2 x*,
t_update (*t_update m x v1*) *x v2*
= *t_update m x v2*.

`Proof.`

Admitted.

☐

For the final two lemmas about total maps, it's convenient to use the reflection idioms introduced in chapter IndProp. We begin by proving a fundamental *reflection lemma* relating the equality proposition on **id**s with the boolean function beq_id.

Exercise: 2 stars, optional (beq_idP) Use the proof of beq_natP in chapter IndProp as a template to prove the following:

`Lemma` *beq_idP* : ∀ *x y*, *reflect* (*x* = *y*) (*beq_id x y*).

`Proof.`

Admitted.

☐

Now, given **id**s *x1* and *x2*, we can use the `destruct` (beq_idP *x1 x2*) to simultaneously perform case analysis on the result of beq_id *x1 x2* and generate hypotheses about the equality (in the sense of =) of *x1* and *x2*.

Exercise: 2 stars (t_update_same) With the example in chapter IndProp as a template, use beq_idP 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* : ∀ *X x* (*m* : *total_map X*),
t_update m x (*m x*) = *m*.

`Proof.`

Admitted.

☐

Exercise: 3 stars, recommended (t_update_permute) Use beq_idP 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 : ∀ (X:Type) v1 v2 x1 x2
                                   (m : total_map X),
  x2 ≠ x1 →
    (t_update (t_update m x2 v2) x1 v1)
  = (t_update (t_update m x1 v1) x2 v2).
Proof.
   Admitted.
   □
```

10.5 Partial maps

Finally, 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:Type} : partial_map A :=
  t_empty None.

Definition update {A:Type} (m : partial_map A)
                      (x : id) (v : A) :=
  t_update m x (Some v).
```

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

```
Lemma apply_empty : ∀ A x, @empty A x = None.
Proof.
  intros. unfold empty. rewrite t_apply_empty.
  reflexivity.
Qed.

Lemma update_eq : ∀ A (m: partial_map A) x v,
  (update m x v) x = Some v.
Proof.
  intros. unfold update. rewrite t_update_eq.
  reflexivity.
Qed.

Theorem update_neq : ∀ (X:Type) v x1 x2
                          (m : partial_map X),
  x2 ≠ x1 →
  (update m x2 v) x1 = m x1.
Proof.
  intros X v x1 x2 m H.
  unfold update. rewrite t_update_neq. reflexivity.
  apply H. Qed.

Lemma update_shadow : ∀ A (m: partial_map A) v1 v2 x,
```

```
  update (update m x v1) x v2 = update m x v2.
Proof.
  intros A m v1 v2 x1. unfold update. rewrite t_update_shadow.
  reflexivity.
Qed.

Theorem update_same : ∀ X v x (m : partial_map X),
  m x = Some v →
  update m x v = m.
Proof.
  intros X v x m H. unfold update. rewrite ← H.
  apply t_update_same.
Qed.

Theorem update_permute : ∀ (X:Type) v1 v2 x1 x2
                                        (m : partial_map X),
  x2 ≠ x1 →
    (update (update m x2 v2) x1 v1)
  = (update (update m x1 v1) x2 v2).
Proof.
  intros X v1 v2 x1 x2 m. unfold update.
  apply t_update_permute.
Qed.
```

$Date : 2016-11-2216 : 39 : 52-0500 (Tue, 22Nov2016)$

Chapter 11

Library Top.ProofObjects

11.1 ProofObjects: The Curry-Howard Correspondence

"*Algorithms are the computational content of proofs.*" –Robert Harper

`Require Export` *IndProp*.

We have seen that Coq has mechanisms both for *programming*, using inductive data types like nat or **list** and functions over these types, and for *proving* properties of these programs, using inductive propositions (like **ev**), implication, universal quantification, and the like. So far, we have mostly treated these mechanisms as if they were quite separate, and for many purposes this is a good way to think. But we have also seen hints that Coq's programming and proving facilities are closely related. For example, the keyword `Inductive` is used to declare both data types and propositions, and → is used both to describe the type of functions on data and logical implication. This is not just a syntactic accident! In fact, programs and proofs in Coq are almost the same thing. In this chapter we will study how this works.

We have already seen the fundamental idea: provability in Coq is represented by concrete *evidence*. When we construct the proof of a basic proposition, we are actually building a tree of evidence, which can be thought of as a data structure.

If the proposition is an implication like A → B, then its proof will be an evidence *transformer*: a recipe for converting evidence for A into evidence for B. So at a fundamental level, proofs are simply programs that manipulate evidence.

Question: If evidence is data, what are propositions themselves?

Answer: They are types!

Look again at the formal definition of the **ev** property.

`Print` *ev*.

Suppose we introduce an alternative pronunciation of ":". Instead of "has type," we can say "is a proof of." For example, the second line in the definition of **ev** declares that ev_0 : **ev** 0. Instead of "ev_0 has type **ev** 0," we can say that "ev_0 is a proof of **ev** 0."

This pun between types and propositions – between : as "has type" and : as "is a proof of" or "is evidence for" – is called the *Curry-Howard correspondence*. It proposes a deep connection between the world of logic and the world of computation:

propositions ~ types proofs ~ data values

See *Wadler* 2015 for a brief history and an up-to-date exposition.

Many useful insights follow from this connection. To begin with, it gives us a natural interpretation of the type of the ev_SS constructor:

```
Check ev_SS.
```

This can be read "ev_SS is a constructor that takes two arguments – a number n and evidence for the proposition **ev** n – and yields evidence for the proposition **ev** (S (S n))."

Now let's look again at a previous proof involving **ev**.

```
Theorem ev_4 : ev 4.
Proof.
  apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

As with ordinary data values and functions, we can use the Print command to see the *proof object* that results from this proof script.

```
Print ev_4.
```

As a matter of fact, we can also write down this proof object *directly*, without the need for a separate proof script:

```
Check (ev_SS 2 (ev_SS 0 ev_0)).
```

The expression ev_SS 2 (ev_SS 0 ev_0) can be thought of as instantiating the parameterized constructor ev_SS with the specific arguments 2 and 0 plus the corresponding proof objects for its premises **ev** 2 and **ev** 0. Alternatively, we can think of ev_SS as a primitive "evidence constructor" that, when applied to a particular number, wants to be further applied to evidence that that number is even; its type,

forall n, ev n -> ev (S (S n)),

expresses this functionality, in the same way that the polymorphic type ∀ X, **list** X expresses the fact that the constructor nil can be thought of as a function from types to empty lists with elements of that type.

We saw in the Logic chapter that we can use function application syntax to instantiate universally quantified variables in lemmas, as well as to supply evidence for assumptions that these lemmas impose. For instance:

```
Theorem ev_4': ev 4.
Proof.
  apply (ev_SS 2 (ev_SS 0 ev_0)).
Qed.
```

We can now see that this feature is a trivial consequence of the status the Coq grants to proofs and propositions: Lemmas and hypotheses can be combined in expressions (i.e., proof objects) according to the same basic rules used for programs in the language.

11.2 Proof Scripts

The *proof objects* we've been discussing lie at the core of how Coq operates. When Coq is following a proof script, what is happening internally is that it is gradually constructing a proof object – a term whose type is the proposition being proved. The tactics between `Proof` and `Qed` tell it how to build up a term of the required type. To see this process in action, let's use the `Show Proof` command to display the current state of the proof tree at various points in the following tactic proof.

```
Theorem ev_4'' : ev 4.
Proof.
  Show Proof.
  apply ev_SS.
  Show Proof.
  apply ev_SS.
  Show Proof.
  apply ev_0.
  Show Proof.
Qed.
```

At any given moment, Coq has constructed a term with a "hole" (indicated by ?`Goal` here, and so on), and it knows what type of evidence is needed to fill this hole.

Each hole corresponds to a subgoal, and the proof is finished when there are no more subgoals. At this point, the evidence we've built stored in the global context under the name given in the `Theorem` command.

Tactic proofs are useful and convenient, but they are not essential: in principle, we can always construct the required evidence by hand, as shown above. Then we can use `Definition` (rather than `Theorem`) to give a global name directly to a piece of evidence.

```
Definition ev_4''' : ev 4 :=
  ev_SS 2 (ev_SS 0 ev_0).
```

All these different ways of building the proof lead to exactly the same evidence being saved in the global environment.

```
Print ev_4.
Print ev_4'.
Print ev_4''.
Print ev_4'''.
```

Exercise: 1 star (eight_is_even) Give a tactic proof and a proof object showing that **ev** 8.

```
Theorem ev_8 : ev 8.
Proof.
  Admitted.

Definition ev_8' : ev 8
```

. *Admitted.*
□

11.3 Quantifiers, Implications, Functions

In Coq's computational universe (where data structures and programs live), there are two sorts of values with arrows in their types: *constructors* introduced by `Inductive`-ly defined data types, and *functions*.

Similarly, in Coq's logical universe (where we carry out proofs), there are two ways of giving evidence for an implication: constructors introduced by `Inductive`-ly defined propositions, and... functions!

For example, consider this statement:

```
Theorem ev_plus4 : ∀ n, ev n → ev (4 + n).
Proof.
  intros n H. simpl.
  apply ev_SS.
  apply ev_SS.
  apply H.
Qed.
```

What is the proof object corresponding to ev_plus4?

We're looking for an expression whose *type* is ∀ n, **ev** n → **ev** (4 + n) – that is, a *function* that takes two arguments (one number and a piece of evidence) and returns a piece of evidence! Here it is:

```
Definition ev_plus4' : ∀ n, ev n → ev (4 + n) :=
  fun (n : nat) ⇒ fun (H : ev n) ⇒
    ev_SS (S (S n)) (ev_SS n H).
```

Recall that `fun` n ⇒ *blah* means "the function that, given n, yields *blah*," and that Coq treats 4 + n and S (S (S (S n))) as synonyms. Another equivalent way to write this definition is:

```
Definition ev_plus4'' (n : nat) (H : ev n) : ev (4 + n) :=
  ev_SS (S (S n)) (ev_SS n H).
```

```
Check ev_plus4''.
```

When we view the proposition being proved by ev_plus4 as a function type, one aspect of it may seem a little unusual. The second argument's type, **ev** n, mentions the *value* of the first argument, n. While such *dependent types* are not found in conventional programming languages, they can be useful in programming too, as the recent flurry of activity in the functional programming community demonstrates.

Notice that both implication (→) and quantification (∀) correspond to functions on evidence. In fact, they are really the same thing: → is just a shorthand for a degenerate use of ∀ where there is no dependency, i.e., no need to give a name to the type on the left-hand side of the arrow.

For example, consider this proposition:

```
Definition ev_plus2 : Prop :=
  ∀ n, ∀ (E : ev n), ev (n + 2).
```

A proof term inhabiting this proposition would be a function with two arguments: a number n and some evidence E that n is even. But the name E for this evidence is not used in the rest of the statement of ev_plus2, so it's a bit silly to bother making up a name for it. We could write it like this instead, using the dummy identifier _ in place of a real name:

```
Definition ev_plus2' : Prop :=
  ∀ n, ∀ (_ : ev n), ev (n + 2).
```

Or, equivalently, we can write it in more familiar notation:

```
Definition ev_plus2'' : Prop :=
  ∀ n, ev n → ev (n + 2).
```

In general, "P → Q" is just syntactic sugar for "∀ (_:P), Q".

11.4 Programming with Tactics

If we can build proofs by giving explicit terms rather than executing tactic scripts, you may be wondering whether we can build *programs* using *tactics* rather than explicit terms. Naturally, the answer is yes!

```
Definition add1 : nat → nat.
intro n.
Show Proof.
apply S.
Show Proof.
apply n. Defined.

Print add1.

Compute add1 2.
```

Notice that we terminate the Definition with a . rather than with := followed by a term. This tells Coq to enter *proof scripting mode* to build an object of type nat → nat. Also, we terminate the proof with Defined rather than Qed; this makes the definition *transparent* so that it can be used in computation like a normally-defined function. (Qed-defined objects are opaque during computation.)

This feature is mainly useful for writing functions with dependent types, which we won't explore much further in this book. But it does illustrate the uniformity and orthogonality of the basic ideas in Coq.

11.5 Logical Connectives as Inductive Types

Inductive definitions are powerful enough to express most of the connectives and quantifiers we have seen so far. Indeed, only universal quantification (and thus implication) is built into Coq; all the others are defined inductively. We'll see these definitions in this section.

```
Module Props.
```

11.5.1 Conjunction

To prove that $P \wedge Q$ holds, we must present evidence for both P and Q. Thus, it makes sense to define a proof object for $P \wedge Q$ as consisting of a pair of two proofs: one for P and another one for Q. This leads to the following definition.

```
Module And.

Inductive and (P Q : Prop) : Prop :=
| conj : P → Q → and P Q.

End And.
```

Notice the similarity with the definition of the **prod** type, given in chapter Poly; the only difference is that **prod** takes `Type` arguments, whereas **and** takes `Prop` arguments.

```
Print prod.
```

This should clarify why `destruct` and `intros` patterns can be used on a conjunctive hypothesis. Case analysis allows us to consider all possible ways in which $P \wedge Q$ was proved – here just one (the `conj` constructor). Similarly, the `split` tactic actually works for any inductively defined proposition with only one constructor. In particular, it works for **and**:

```
Lemma and_comm : ∀ P Q : Prop, P ∧ Q ↔ Q ∧ P.
Proof.
  intros P Q. split.
  - intros [HP HQ]. split.
    + apply HQ.
    + apply HP.
  - intros [HP HQ]. split.
    + apply HQ.
    + apply HP.
Qed.
```

This shows why the inductive definition of **and** can be manipulated by tactics as we've been doing. We can also use it to build proofs directly, using pattern-matching. For instance:

```
Definition and_comm'_aux P Q (H : P ∧ Q) :=
  match H with
  | conj HP HQ ⇒ conj HQ HP
  end.

Definition and_comm' P Q : P ∧ Q ↔ Q ∧ P :=
```

conj (*and_comm'_aux* P Q) (*and_comm'_aux* Q P).

Exercise: 2 stars, optional (conj_fact) Construct a proof object demonstrating the following proposition.

Definition *conj_fact* : $\forall$ P Q R, $P \wedge Q \rightarrow Q \wedge R \rightarrow P \wedge R$
. *Admitted.*
☐

11.5.2 Disjunction

The inductive definition of disjunction uses two constructors, one for each side of the disjunct:

```
Module Or.
Inductive or (P Q : Prop) : Prop :=
| or_introl : P → or P Q
| or_intror : Q → or P Q.
End Or.
```

This declaration explains the behavior of the `destruct` tactic on a disjunctive hypothesis, since the generated subgoals match the shape of the or_introl and or_intror constructors.

Once again, we can also directly write proof objects for theorems involving **or**, without resorting to tactics.

Exercise: 2 stars, optional (or_commut") Try to write down an explicit proof object for or_commut (without using `Print` to peek at the ones we already defined!).

Definition *or_comm* : $\forall$ P Q, $P \vee Q \rightarrow Q \vee P$
. *Admitted.*
☐

11.5.3 Existential Quantification

To give evidence for an existential quantifier, we package a witness x together with a proof that x satisfies the property *P*:

```
Module Ex.
Inductive ex {A : Type} (P : A → Prop) : Prop :=
| ex_intro : ∀ x : A, P x → ex P.
End Ex.
```

This may benefit from a little unpacking. The core definition is for a type former **ex** that can be used to build propositions of the form **ex** *P*, where *P* itself is a *function* from witness values in the type A to propositions. The ex_intro constructor then offers a way of constructing evidence for **ex** *P*, given a witness x and a proof of *P* x.

The more familiar form ∃ x, *P* x desugars to an expression involving **ex**:

```
Check ex (fun n ⇒ ev n).
```

Here's how to define an explicit proof object involving **ex**:

```
Definition some_nat_is_even : ∃ n, ev n :=
  ex_intro ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

Exercise: 2 stars, optional (ex_ev_Sn) Complete the definition of the following proof object:

```
Definition ex_ev_Sn : ex (fun n ⇒ ev (S n))
  . Admitted.
```

□

11.5.4 True and False

The inductive definition of the **True** proposition is simple:

```
Inductive True : Prop :=
  | I : True.
```

It has one constructor (so every proof of **True** is the same, so being given a proof of **True** is not informative.)

False is equally simple – indeed, so simple it may look syntactically wrong at first glance!

```
Inductive False : Prop :=.
```

That is, **False** is an inductive type with *no* constructors – i.e., no way to build evidence for it.

```
End Props.
```

11.6 Equality

Even Coq's equality relation is not built in. It has the following inductive definition. (Actually, the definition in the standard library is a small variant of this, which gives an induction principle that is slightly easier to use.)

```
Module MyEquality.

Inductive eq {X:Type} : X → X → Prop :=
| eq_refl : ∀ x, eq x x.

Notation "x = y" := (eq x y)
                    (at level 70, no associativity)
                    : type_scope.
```

The way to think about this definition is that, given a set X, it defines a *family* of propositions "x is equal to y," indexed by pairs of values (x and y) from X. There is just one way of constructing evidence for each member of this family: applying the constructor eq_refl to a type X and a value x : X yields evidence that x is equal to x.

Exercise: 2 stars (leibniz_equality) The inductive definition of equality corresponds to *Leibniz equality*: what we mean when we say "x and y are equal" is that every property on *P* that is true of x is also true of y.

```
Lemma leibniz_equality : ∀ (X : Type) (x y: X),
  x = y → ∀ P:X→Prop, P x → P y.
Proof.
   Admitted.
```

☐

We can use eq_refl to construct evidence that, for example, 2 = 2. Can we also use it to construct evidence that 1 + 1 = 2? Yes, we can. Indeed, it is the very same piece of evidence! The reason is that Coq treats as "the same" any two terms that are *convertible* according to a simple set of computation rules. These rules, which are similar to those used by `Compute`, include evaluation of function application, inlining of definitions, and simplification of `match`es.

```
Lemma four: 2 + 2 = 1 + 3.
Proof.
  apply eq_refl.
Qed.
```

The `reflexivity` tactic that we have used to prove equalities up to now is essentially just short-hand for `apply` refl_equal.

In tactic-based proofs of equality, the conversion rules are normally hidden in uses of `simpl` (either explicit or implicit in other tactics such as `reflexivity`). But you can see them directly at work in the following explicit proof objects:

```
Definition four' : 2 + 2 = 1 + 3 :=
  eq_refl 4.

Definition singleton : ∀ (X:Set) (x:X), []++[x] = x::[] :=
  fun (X:Set) (x:X) ⇒ eq_refl [x].

End MyEquality.

Definition quiz6 : ∃ x, x + 3 = 4
  := ex_intro (fun z ⇒ (z + 3 = 4)) 1 (refl_equal 4).
```

11.6.1 Inversion, Again

We've seen `inversion` used with both equality hypotheses and hypotheses about inductively defined propositions. Now that we've seen that these are actually the same thing, we're in a position to take a closer look at how `inversion` behaves.

In general, the `inversion` tactic...

- takes a hypothesis H whose type P is inductively defined, and
- for each constructor C in P's definition,
    - generates a new subgoal in which we assume H was built with C,
    - adds the arguments (premises) of C to the context of the subgoal as extra hypotheses,
    - matches the conclusion (result type) of C against the current goal and calculates a set of equalities that must hold in order for C to be applicable,
    - adds these equalities to the context (and, for convenience, rewrites them in the goal), and
    - if the equalities are not satisfiable (e.g., they involve things like S n = O), immediately solves the subgoal.

Example: If we invert a hypothesis built with **or**, there are two constructors, so two subgoals get generated. The conclusion (result type) of the constructor ($P \vee Q$) doesn't place any restrictions on the form of P or Q, so we don't get any extra equalities in the context of the subgoal.

Example: If we invert a hypothesis built with **and**, there is only one constructor, so only one subgoal gets generated. Again, the conclusion (result type) of the constructor ($P \wedge Q$) doesn't place any restrictions on the form of P or Q, so we don't get any extra equalities in the context of the subgoal. The constructor does have two arguments, though, and these can be seen in the context in the subgoal.

Example: If we invert a hypothesis built with eq, there is again only one constructor, so only one subgoal gets generated. Now, though, the form of the refl_equal constructor does give us some extra information: it tells us that the two arguments to eq must be the same! The `inversion` tactic adds this fact to the context.

$Date : 2017 - 01 - 3018 : 47 : 45 - 0500(Mon, 30Jan2017)$

Chapter 12

Library Top.IndPrinciples

12.1 IndPrinciples: Induction Principles

With the Curry-Howard correspondence and its realization in Coq in mind, we can now take a deeper look at induction principles.

```
Require Export ProofObjects.
```

12.2 Basics

Every time we declare a new `Inductive` datatype, Coq automatically generates an *induction principle* for this type. This induction principle is a theorem like any other: If t is defined inductively, the corresponding induction principle is called *t_ind*. Here is the one for natural numbers:

```
Check nat_ind.
```

The `induction` tactic is a straightforward wrapper that, at its core, simply performs `apply` *t_ind*. To see this more clearly, let's experiment with directly using `apply nat_ind`, instead of the `induction` tactic, to carry out some proofs. Here, for example, is an alternate proof of a theorem that we saw in the Basics chapter.

```
Theorem mult_0_r' : ∀ n:nat,
  n × 0 = 0.
Proof.
  apply nat_ind.
  - reflexivity.
  - simpl. intros n' IHn'. rewrite → IHn'.
    reflexivity. Qed.
```

This proof is basically the same as the earlier one, but a few minor differences are worth noting.

First, in the induction step of the proof (the "S" case), we have to do a little bookkeeping manually (the `intros`) that `induction` does automatically.

Second, we do not introduce n into the context before applying nat_ind – the conclusion of nat_ind is a quantified formula, and apply needs this conclusion to exactly match the shape of the goal state, including the quantifier. By contrast, the induction tactic works either with a variable in the context or a quantified variable in the goal.

These conveniences make induction nicer to use in practice than applying induction principles like nat_ind directly. But it is important to realize that, modulo these bits of bookkeeping, applying nat_ind is what we are really doing.

Exercise: 2 stars, optional (plus_one_r') Complete this proof without using the induction tactic.

```
Theorem plus_one_r' : ∀ n:nat,
  n + 1 = S n.
Proof.
  Admitted.
```

□

Coq generates induction principles for every datatype defined with Inductive, including those that aren't recursive. Although of course we don't need induction to prove properties of non-recursive datatypes, the idea of an induction principle still makes sense for them: it gives a way to prove that a property holds for all values of the type.

These generated principles follow a similar pattern. If we define a type *t* with constructors c1 ... *cn*, Coq generates a theorem with this shape:

t_ind : forall P : t -> Prop, ... case for c1 ... -> ... case for c2 ... -> ... ... case for cn ... -> forall n : t, P n

The specific shape of each case depends on the arguments to the corresponding constructor. Before trying to write down a general rule, let's look at some more examples. First, an example where the constructors take no arguments:

```
Inductive yesno : Type :=
  | yes : yesno
  | no : yesno.

Check yesno_ind.
```

Exercise: 1 star, optional (rgb) Write out the induction principle that Coq will generate for the following datatype. Write down your answer on paper or type it into a comment, and then compare it with what Coq prints.

```
Inductive rgb : Type :=
  | red : rgb
  | green : rgb
  | blue : rgb.
Check rgb_ind.
```

□

Here's another example, this time with one of the constructors taking some arguments.

```
Inductive natlist : Type :=
  | nnil : natlist
  | ncons : nat → natlist → natlist.
Check natlist_ind.
```

Exercise: 1 star, optional (natlist1) Suppose we had written the above definition a little differently:

```
Inductive natlist1 : Type :=
  | nnil1 : natlist1
  | nsnoc1 : natlist1 → nat → natlist1.
```

Now what will the induction principle look like? □

From these examples, we can extract this general rule:

- The type declaration gives several constructors; each corresponds to one clause of the induction principle.
- Each constructor `c` takes argument types *a1* ... *an*.
- Each *ai* can be either *t* (the datatype we are defining) or some other type `s`.
- The corresponding case of the induction principle says:
  - "For all values *x1*...*xn* of types *a1*...*an*, if *P* holds for each of the inductive arguments (each *xi* of type *t*), then *P* holds for `c` *x1* ... *xn*".

Exercise: 1 star, optional (byntree_ind) Write out the induction principle that Coq will generate for the following datatype. (Again, write down your answer on paper or type it into a comment, and then compare it with what Coq prints.)

```
Inductive byntree : Type :=
 | bempty : byntree
 | bleaf : yesno → byntree
 | nbranch : yesno → byntree → byntree → byntree.
```

□

Exercise: 1 star, optional (ex_set) Here is an induction principle for an inductively defined set.

ExSet_ind : forall P : ExSet -> Prop, (forall b : bool, P (con1 b)) -> (forall (n : nat) (e : ExSet), P e -> P (con2 n e)) -> forall e : ExSet, P e

Give an `Inductive` definition of **ExSet**:

```
Inductive ExSet : Type :=

.
```

□

12.3 Polymorphism

Next, what about polymorphic datatypes?

The inductive definition of polymorphic lists

Inductive list (X:Type) : Type := | nil : list X | cons : X -> list X -> list X.

is very similar to that of **natlist**. The main difference is that, here, the whole definition is *parameterized* on a set X: that is, we are defining a *family* of inductive types **list** X, one for each X. (Note that, wherever **list** appears in the body of the declaration, it is always applied to the parameter X.) The induction principle is likewise parameterized on X:

list_ind : forall (X : Type) (P : list X -> Prop), P □ -> (forall (x : X) (l : list X), P l -> P (x :: l)) -> forall l : list X, P l

Note that the *whole* induction principle is parameterized on X. That is, *list_ind* can be thought of as a polymorphic function that, when applied to a type X, gives us back an induction principle specialized to the type **list** X.

Exercise: 1 star, optional (tree) Write out the induction principle that Coq will generate for the following datatype. Compare your answer with what Coq prints.

`Inductive` *tree* $(X$:`Type`$)$: `Type` :=
| *leaf* : $X \rightarrow$ *tree* X
| *node* : *tree* $X \rightarrow$ *tree* $X \rightarrow$ *tree* X.
`Check` *tree_ind*.

□

Exercise: 1 star, optional (mytype) Find an inductive definition that gives rise to the following induction principle:

mytype_ind : forall (X : Type) (P : mytype X -> Prop), (forall x : X, P (constr1 X x)) -> (forall n : nat, P (constr2 X n)) -> (forall m : mytype X, P m -> forall n : nat, P (constr3 X m n)) -> forall m : mytype X, P m □

Exercise: 1 star, optional (foo) Find an inductive definition that gives rise to the following induction principle:

foo_ind : forall (X Y : Type) (P : foo X Y -> Prop), (forall x : X, P (bar X Y x)) -> (forall y : Y, P (baz X Y y)) -> (forall f1 : nat -> foo X Y, (forall n : nat, P (f1 n)) -> P (quux X Y f1)) -> forall f2 : foo X Y, P f2 □

Exercise: 1 star, optional (foo') Consider the following inductive definition:

`Inductive` *foo'* $(X$:`Type`$)$: `Type` :=
| *C1* : *list* $X \rightarrow$ *foo'* $X \rightarrow$ *foo'* X
| *C2* : *foo'* X.

What induction principle will Coq generate for foo'? Fill in the blanks, then check your answer with Coq.)

foo'_ind : forall (X : Type) (P : foo' X -> Prop), (forall (l : list X) (f : foo' X), ______________________ -> ______________________) -> ____ -> forall f : foo' X, ______________________

□

12.4 Induction Hypotheses

Where does the phrase "induction hypothesis" fit into this story?

The induction principle for numbers

forall P : nat -> Prop, P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n

is a generic statement that holds for all propositions *P* (or rather, strictly speaking, for all families of propositions *P* indexed by a number n). Each time we use this principle, we are choosing *P* to be a particular expression of type nat→Prop.

We can make proofs by induction more explicit by giving this expression a name. For example, instead of stating the theorem mult_0_r as "∀ n, n × 0 = 0," we can write it as "∀ n, P_m0r n", where P_m0r is defined as...

```
Definition P_m0r (n:nat) : Prop :=
  n × 0 = 0.
```

... or equivalently:

```
Definition P_m0r' : nat→Prop :=
  fun n ⇒ n × 0 = 0.
```

Now it is easier to see where P_m0r appears in the proof.

```
Theorem mult_0_r'' : ∀ n:nat,
  P_m0r n.
Proof.
  apply nat_ind.
  - reflexivity.
  -

    intros n IHn.
    unfold P_m0r in IHn. unfold P_m0r. simpl. apply IHn. Qed.
```

This extra naming step isn't something that we do in normal proofs, but it is useful to do it explicitly for an example or two, because it allows us to see exactly what the induction hypothesis is. If we prove ∀ n, P_m0r n by induction on n (using either induction or apply nat_ind), we see that the first subgoal requires us to prove P_m0r 0 ("*P* holds for zero"), while the second subgoal requires us to prove ∀ *n'*, P_m0r *n'* → P_m0r (S *n'*) (that is "*P* holds of S *n'* if it holds of *n'*" or, more elegantly, "*P* is preserved by S"). The *induction hypothesis* is the premise of this latter implication – the assumption that *P* holds of *n'*, which we are allowed to use in proving that *P* holds for S *n'*.

12.5 More on the `induction` Tactic

The `induction` tactic actually does even more low-level bookkeeping for us than we discussed above.

Recall the informal statement of the induction principle for natural numbers:

- If *P* `n` is some proposition involving a natural number n, and we want to show that P holds for *all* numbers n, we can reason like this:
  - show that *P* `O` holds
  - show that, if *P* *n'* holds, then so does *P* (`S` *n'*)
  - conclude that *P* `n` holds for all n.

So, when we begin a proof with `intros n` and then `induction n`, we are first telling Coq to consider a *particular* `n` (by introducing it into the context) and then telling it to prove something about *all* numbers (by using induction).

What Coq actually does in this situation, internally, is to "re-generalize" the variable we perform induction on. For example, in our original proof that `plus` is associative...

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p.
  induction n as [| n'].
  - reflexivity.
  -

    simpl. rewrite → IHn'. reflexivity. Qed.
```

It also works to apply `induction` to a variable that is quantified in the goal.

```
Theorem plus_comm' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction n as [| n'].
  - intros m. rewrite ← plus_n_O. reflexivity.
  - intros m. simpl. rewrite → IHn'.
    rewrite ← plus_n_Sm. reflexivity. Qed.
```

Note that `induction n` leaves `m` still bound in the goal – i.e., what we are proving inductively is a statement beginning with ∀ `m`.

If we do `induction` on a variable that is quantified in the goal *after* some other quantifiers, the `induction` tactic will automatically introduce the variables bound by these quantifiers into the context.

```
Theorem plus_comm'' : ∀ n m : nat,
```

$n + m = m + n$.

```
Proof.
  induction m as [| m'].
  - simpl. rewrite <- plus_n_O. reflexivity.
  - simpl. rewrite <- IHm'.
    rewrite <- plus_n_Sm. reflexivity. Qed.
```

Exercise: 1 star, optional (plus_explicit_prop) Rewrite both plus_assoc' and plus_comm' and their proofs in the same style as mult_0_r'' above – that is, for each theorem, give an explicit Definition of the proposition being proved by induction, and state the theorem and proof in terms of this defined proposition.

□

12.6 Induction Principles in Prop

Earlier, we looked in detail at the induction principles that Coq generates for inductively defined *sets*. The induction principles for inductively defined *propositions* like **ev** are a tiny bit more complicated. As with all induction principles, we want to use the induction principle on **ev** to prove things by inductively considering the possible shapes that something in **ev** can have. Intuitively speaking, however, what we want to prove are not statements about *evidence* but statements about *numbers*: accordingly, we want an induction principle that lets us prove properties of numbers by induction on evidence.

For example, from what we've said so far, you might expect the inductive definition of **ev**...

Inductive ev : nat -> Prop := | ev_0 : ev 0 | ev_SS : forall n : nat, ev n -> ev (S (S n)).

...to give rise to an induction principle that looks like this...

ev_ind_max : forall P : (forall n : nat, ev n -> Prop), P O ev_0 -> (forall (m : nat) (E : ev m), P m E -> P (S (S m)) (ev_SS m E)) -> forall (n : nat) (E : ev n), P n E

... because:

- Since **ev** is indexed by a number n (every **ev** object E is a piece of evidence that some particular number n is even), the proposition P is parameterized by both n and E – that is, the induction principle can be used to prove assertions involving both an even number and the evidence that it is even.

- Since there are two ways of giving evidence of evenness (**ev** has two constructors), applying the induction principle generates two subgoals:

  - We must prove that P holds for O and ev_0.
  - We must prove that, whenever n is an even number and E is an evidence of its evenness, if P holds of n and E, then it also holds of S (S n) and ev_SS n E.

- If these subgoals can be proved, then the induction principle tells us that P is true for *all* even numbers n and evidence E of their evenness.

This is more flexibility than we normally need or want: it is giving us a way to prove logical assertions where the assertion involves properties of some piece of *evidence* of evenness, while all we really care about is proving properties of *numbers* that are even – we are interested in assertions about numbers, not about evidence. It would therefore be more convenient to have an induction principle for proving propositions P that are parameterized just by n and whose conclusion establishes P for all even numbers n:

forall P : nat -> Prop, ... -> forall n : nat, even n -> P n

For this reason, Coq actually generates the following simplified induction principle for **ev**:

```
Check ev_ind.
```

In particular, Coq has dropped the evidence term E as a parameter of the the proposition P.

In English, ev_ind says:

- Suppose, P is a property of natural numbers (that is, P n is a Prop for every n). To show that P n holds whenever n is even, it suffices to show:

  - P holds for 0,
  - for any n, if n is even and P holds for n, then P holds for S (S n).

As expected, we can apply ev_ind directly instead of using induction. For example, we can use it to show that **ev'** (the slightly awkward alternate definition of evenness that we saw in an exercise in the \chap{IndProp} chapter) is equivalent to the cleaner inductive definition **ev**:

```
Theorem ev_ev' : ∀ n, ev n → ev' n.
Proof.
  apply ev_ind.
  -
    apply ev'_0.
  -
    intros m Hm IH.
    apply (ev'_sum 2 m).
    + apply ev'_2.
    + apply IH.
Qed.
```

The precise form of an Inductive definition can affect the induction principle Coq generates.

For example, in chapter IndProp, we defined $\leq$ as:

This definition can be streamlined a little by observing that the left-hand argument n is the same everywhere in the definition, so we can actually make it a "general parameter" to the whole definition, rather than an argument to each constructor.

```
Inductive le (n:nat) : nat → Prop :=
  | le_n : le n n
  | le_S : ∀ m, (le n m) → (le n (S m)).
Notation "m <= n" := (le m n).
```

The second one is better, even though it looks less symmetric. Why? Because it gives us a simpler induction principle.

```
Check le_ind.
```

12.7 Formal vs. Informal Proofs by Induction

Question: What is the relation between a formal proof of a proposition P and an informal proof of the same proposition P?

Answer: The latter should *teach* the reader how to produce the former.

Question: How much detail is needed??

Unfortunately, there is no single right answer; rather, there is a range of choices.

At one end of the spectrum, we can essentially give the reader the whole formal proof (i.e., the "informal" proof will amount to just transcribing the formal one into words). This may give the reader the ability to reproduce the formal one for themselves, but it probably doesn't *teach* them anything much.

At the other end of the spectrum, we can say "The theorem is true and you can figure out why for yourself if you think about it hard enough." This is also not a good teaching strategy, because often writing the proof requires one or more significant insights into the thing we're proving, and most readers will give up before they rediscover all the same insights as we did.

In the middle is the golden mean – a proof that includes all of the essential insights (saving the reader the hard work that we went through to find the proof in the first place) plus high-level suggestions for the more routine parts to save the reader from spending too much time reconstructing these (e.g., what the IH says and what must be shown in each case of an inductive proof), but not so much detail that the main ideas are obscured.

Since we've spent much of this chapter looking "under the hood" at formal proofs by induction, now is a good moment to talk a little about *informal* proofs by induction.

In the real world of mathematical communication, written proofs range from extremely longwinded and pedantic to extremely brief and telegraphic. Although the ideal is somewhere in between, while one is getting used to the style it is better to start out at the pedantic end. Also, during the learning phase, it is probably helpful to have a clear standard to compare against. With this in mind, we offer two templates – one for proofs by induction over *data* (i.e., where the thing we're doing induction on lives in `Type`) and one for proofs by induction over *evidence* (i.e., where the inductively defined thing lives in `Prop`).

12.7.1 Induction Over an Inductively Defined Set

Template:

- *Theorem*: <Universally quantified proposition of the form "For all n:S, *P*(n)," where S is some inductively defined set.>

  Proof: By induction on n.

  <one case for each constructor c of S...>

  - Suppose n = c *a1* ... *ak*, where <...and here we state the IH for each of the a's that has type S, if any>. We must show <...and here we restate *P*(c *a1* ... *ak*)>.

    <go on and prove *P*(n) to finish the case...>

  - <other cases similarly...> ☐

Example:

- *Theorem*: For all sets X, lists l : **list** X, and numbers n, if length l = n then *index* (S n) l = None.

  Proof: By induction on l.

  - Suppose l = []. We must show, for all numbers n, that, if length [] = n, then *index* (S n) [] = None.

    This follows immediately from the definition of *index*.

  - Suppose l = x :: *l'* for some x and *l'*, where length *l'* = *n'* implies *index* (S *n'*) *l'* = None, for any number *n'*. We must show, for all n, that, if length (x::*l'*) = n then *index* (S n) (x::*l'*) = None.

    Let n be a number with length l = n. Since

    length l = length (x::l') = S (length l'),

    it suffices to show that

    index (S (length l')) l' = None.

    But this follows directly from the induction hypothesis, picking *n'* to be length *l'*.
    ☐

12.7.2 Induction Over an Inductively Defined Proposition

Since inductively defined proof objects are often called "derivation trees," this form of proof is also known as *induction on derivations*.

Template:

- *Theorem*: <Proposition of the form "$Q \to P$," where Q is some inductively defined proposition (more generally, "For all x y z, Q x y $\mathsf{z} \to P$ x y z")>

  Proof: By induction on a derivation of Q. <Or, more generally, "Suppose we are given x, y, and z. We show that Q x y z implies P x y z, by induction on a derivation of Q x y z"...>

  <one case for each constructor c of Q...>

  - Suppose the final rule used to show Q is c. Then <...and here we state the types of all of the a's together with any equalities that follow from the definition of the constructor and the IH for each of the a's that has type Q, if there are any>. We must show <...and here we restate P>.

    <go on and prove P to finish the case...>

  - <other cases similarly...> □

Example

- *Theorem*: The $\leq$ relation is transitive – i.e., for all numbers n, m, and o, if $\mathsf{n} \leq \mathsf{m}$ and $\mathsf{m} \leq o$, then $\mathsf{n} \leq o$.

  Proof: By induction on a derivation of $\mathsf{m} \leq o$.

  - Suppose the final rule used to show $\mathsf{m} \leq o$ is $\mathsf{le\_n}$. Then $\mathsf{m} = o$ and we must show that $\mathsf{n} \leq \mathsf{m}$, which is immediate by hypothesis.

  - Suppose the final rule used to show $\mathsf{m} \leq o$ is $\mathsf{le\_S}$. Then $o = \mathsf{S}\ o'$ for some o' with $\mathsf{m} \leq o'$. We must show that $\mathsf{n} \leq \mathsf{S}\ o'$. By induction hypothesis, $\mathsf{n} \leq o'$.

    But then, by $\mathsf{le\_S}$, $\mathsf{n} \leq \mathsf{S}\ o'$. □

$Date : 2016 - 09 - 2711 : 35 : 13 - 0400 (Tue, 27 Sep 2016)$

Chapter 13

Library Top.Rel

13.1 Rel: Properties of Relations

This short (and optional) chapter develops some basic definitions and a few theorems about binary relations in Coq. The key definitions are repeated where they are actually used (in the Smallstep chapter), so readers who are already comfortable with these ideas can safely skim or skip this chapter. However, relations are also a good source of exercises for developing facility with Coq's basic reasoning facilities, so it may be useful to look at this material just after the IndProp chapter.

```
Require Export IndProp.
```

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: Type) := X → X → Prop.
```

Confusingly, the Coq standard library hijacks the generic term "relation" for this specific instance of the idea. To maintain consistency with the library, we will do the same. So, henceforth the Coq identifier relation will always refer to a binary relation between some set and itself, whereas the English word "relation" can refer either to the specific Coq concept or the more general concept of a relation between any number of possibly different sets. The context of the discussion should always make clear which is meant.

An example relation on nat is le, the less-than-or-equal-to relation, which we usually write $n1 \leq n2$.

```
Print le.
Check le : nat → nat → Prop.
Check le : relation nat.
```

(Why did we write it this way instead of starting with Inductive le : relation nat...? Because we wanted to put the first nat to the left of the :, which makes Coq generate a somewhat nicer induction principle for reasoning about ≤.)

13.2 Basic Properties

As anyone knows who has taken an undergraduate discrete math course, there is a lot to be said about relations in general, including ways of classifying relations (as reflexive, transitive, etc.), theorems that can be proved generically about certain sorts of relations, constructions that build one relation from another, etc. For example...

Partial Functions

A relation R on a set X is a *partial function* if, for every x, there is at most one y such that R x y – i.e., R x *y1* and R x *y2* together imply *y1* = *y2*.

```
Definition partial_function {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.
```

For example, the **next_nat** relation defined earlier is a partial function.

```
Print next_nat.
Check next_nat : relation nat.

Theorem next_nat_partial_function :
   partial_function next_nat.
Proof.
  unfold partial_function.
  intros x y1 y2 H1 H2.
  inversion H1. inversion H2.
  reflexivity. Qed.
```

However, the $\leq$ relation on numbers is not a partial function. (Assume, for a contradiction, that $\leq$ is a partial function. But then, since $0 \leq 0$ and $0 \leq 1$, it follows that $0 = 1$. This is nonsense, so our assumption was contradictory.)

```
Theorem le_not_a_partial_function :
  ¬ (partial_function le).
Proof.
  unfold not. unfold partial_function. intros Hc.
  assert (0 = 1) as Nonsense. {
    apply Hc with (x := 0).
    - apply le_n.
    - apply le_S. apply le_n. }
  inversion Nonsense. Qed.
```

Exercise: 2 stars, optional Show that the *total_relation* defined in earlier is not a partial function.

□

Exercise: 2 stars, optional Show that the *empty_relation* that we defined earlier is a partial function.

□

Reflexive Relations

A *reflexive* relation on a set X is one for which every element of X is related to itself.

```
Definition reflexive {X: Type} (R: relation X) :=
  ∀ a : X, R a a.

Theorem le_reflexive :
  reflexive le.
Proof.
  unfold reflexive. intros n. apply le_n. Qed.
```

Transitive Relations

A relation R is *transitive* if R a c holds whenever R a b and R b c do.

```
Definition transitive {X: Type} (R: relation X) :=
  ∀ a b c : X, (R a b) → (R b c) → (R a c).

Theorem le_trans :
  transitive le.
Proof.
  intros n m o Hnm Hmo.
  induction Hmo.
  - apply Hnm.
  - apply le_S. apply IHHmo. Qed.

Theorem lt_trans:
  transitive lt.
Proof.
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  apply le_S in Hnm.
  apply le_trans with (a := (S n)) (b := (S m)) (c := o).
  apply Hnm.
  apply Hmo. Qed.
```

Exercise: 2 stars, optional We can also prove lt_trans more laboriously by induction, without using le_trans. Do this.

```
Theorem lt_trans' :
  transitive lt.
Proof.
```

```
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  induction Hmo as [| m' Hm'o].
   Admitted.
```
□

Exercise: 2 stars, optional Prove the same thing again by induction on *o*.

```
Theorem lt_trans'' :
  transitive lt.
Proof.
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  induction o as [| o'].
   Admitted.
```
□

The transitivity of **le**, in turn, can be used to prove some facts that will be useful later (e.g., for the proof of antisymmetry below)...

```
Theorem le_Sn_le : ∀ n m, S n ≤ m → n ≤ m.
Proof.
  intros n m H. apply le_trans with (S n).
  - apply le_S. apply le_n.
  - apply H.
Qed.
```

Exercise: 1 star, optional
```
Theorem le_S_n : ∀ n m,
  (S n ≤ S m) → (n ≤ m).
Proof.
   Admitted.
```
□

Exercise: 2 stars, optional (le_Sn_n_inf) Provide an informal proof of the following theorem:

Theorem: For every n, ¬ (S n ≤ n)

A formal proof of this is an optional exercise below, but try writing an informal proof without doing the formal proof first.

Proof: □

Exercise: 1 star, optional
```
Theorem le_Sn_n : ∀ n,
  ¬ (S n ≤ n).
Proof.
   Admitted.
```

□

Reflexivity and transitivity are the main concepts we'll need for later chapters, but, for a bit of additional practice working with relations in Coq, let's look at a few other common ones...

Symmetric and Antisymmetric Relations

A relation R is *symmetric* if R a b implies R b a.

Definition *symmetric* $\{X$: Type$\}$ $(R$: *relation* $X)$:=
  $\forall$ a b : X, $(R\ a\ b) \rightarrow (R\ b\ a)$.

Exercise: 2 stars, optional Theorem *le_not_symmetric* :
  $\neg$ (*symmetric le*).
Proof.
  Admitted.

□

A relation R is *antisymmetric* if R a b and R b a together imply a = b – that is, if the only "cycles" in R are trivial ones.

Definition *antisymmetric* $\{X$: Type$\}$ $(R$: *relation* $X)$:=
  $\forall$ a b : X, $(R\ a\ b) \rightarrow (R\ b\ a) \rightarrow a = b$.

Exercise: 2 stars, optional Theorem *le_antisymmetric* :
  antisymmetric le.
Proof.
  Admitted.

□

Exercise: 2 stars, optional Theorem *le_step* : $\forall$ n m p,
  $n < m \rightarrow$
  $m \leq S\ p \rightarrow$
  $n \leq p$.
Proof.
  Admitted.

□

Equivalence Relations

A relation is an *equivalence* if it's reflexive, symmetric, and transitive.

Definition *equivalence* $\{X$:Type$\}$ $(R$: *relation* $X)$:=
  (*reflexive* R) $\wedge$ (*symmetric* R) $\wedge$ (*transitive* R).

Partial Orders and Preorders

A relation is a *partial order* when it's reflexive, *anti*-symmetric, and transitive. In the Coq standard library it's called just "order" for short.

```
Definition order {X:Type} (R: relation X) :=
  (reflexive R) ∧ (antisymmetric R) ∧ (transitive R).
```

A preorder is almost like a partial order, but doesn't have to be antisymmetric.

```
Definition preorder {X:Type} (R: relation X) :=
  (reflexive R) ∧ (transitive R).

Theorem le_order :
  order le.
Proof.
  unfold order. split.
    - apply le_reflexive.
    - split.
      + apply le_antisymmetric.
      + apply le_trans. Qed.
```

13.3 Reflexive, Transitive Closure

The *reflexive, transitive closure* of a relation R is the smallest relation that contains R and that is both reflexive and transitive. Formally, it is defined like this in the Relations module of the Coq standard library:

```
Inductive clos_refl_trans {A: Type} (R: relation A) : relation A :=
    | rt_step : ∀ x y, R x y → clos_refl_trans R x y
    | rt_refl : ∀ x, clos_refl_trans R x x
    | rt_trans : ∀ x y z,
          clos_refl_trans R x y →
          clos_refl_trans R y z →
          clos_refl_trans R x z.
```

For example, the reflexive and transitive closure of the **next_nat** relation coincides with the **le** relation.

```
Theorem next_nat_closure_is_le : ∀ n m,
  (n ≤ m) ↔ ((clos_refl_trans next_nat) n m).
Proof.
  intros n m. split.
  -
    intro H. induction H.
    + apply rt_refl.
    +
```

```
      apply rt_trans with m. apply IHle. apply rt_step.
      apply nn.
  -
    intro H. induction H.
    + inversion H. apply le_S. apply le_n.
    + apply le_n.
    +
      apply le_trans with y.
      apply IHclos_refl_trans1.
      apply IHclos_refl_trans2. Qed.
```

The above definition of reflexive, transitive closure is natural: it says, explicitly, that the reflexive and transitive closure of R is the least relation that includes R and that is closed under rules of reflexivity and transitivity. But it turns out that this definition is not very convenient for doing proofs, since the "nondeterminism" of the rt_trans rule can sometimes lead to tricky inductions. Here is a more useful definition:

```
Inductive clos_refl_trans_1n {A : Type}
                             (R : relation A) (x : A)
                             : A → Prop :=
  | rt1n_refl : clos_refl_trans_1n R x x
  | rt1n_trans (y z : A) :
      R x y → clos_refl_trans_1n R y z →
      clos_refl_trans_1n R x z.
```

Our new definition of reflexive, transitive closure "bundles" the rt_step and rt_trans rules into the single rule step. The left-hand premise of this step is a single use of R, leading to a much simpler induction principle.

Before we go on, we should check that the two definitions do indeed define the same relation...

First, we prove two lemmas showing that **clos_refl_trans_1n** mimics the behavior of the two "missing" **clos_refl_trans** constructors.

```
Lemma rsc_R : ∀ (X:Type) (R:relation X) (x y : X),
       R x y → clos_refl_trans_1n R x y.
Proof.
  intros X R x y H.
  apply rt1n_trans with y. apply H. apply rt1n_refl. Qed.
```

Exercise: 2 stars, optional (rsc_trans) Lemma *rsc_trans* :

```
  ∀ (X:Type) (R: relation X) (x y z : X),
      clos_refl_trans_1n R x y →
      clos_refl_trans_1n R y z →
      clos_refl_trans_1n R x z.
Proof.
```

Admitted.

☐

Then we use these facts to prove that the two definitions of reflexive, transitive closure do indeed define the same relation.

Exercise: 3 stars, optional (rtc_rsc_coincide) `Theorem` *rtc_rsc_coincide* :

$\forall$ (X:`Type`) (R: *relation* X) (x y : X),

clos_refl_trans R x y $\leftrightarrow$ *clos_refl_trans_1n* R x y.

`Proof`.

Admitted.

☐

$Date : 2016-05-2616:17:19-0400 (Thu, 26 May 2016)$

Chapter 14

Library Top.Imp

14.1 Imp: Simple Imperative Programs

In this chapter, we begin a new direction that will continue for the rest of the course. Up to now most of our attention has been focused on various aspects of Coq itself, while from now on we'll mostly be using Coq to formalize other things. (We'll continue to pause from time to time to introduce a few additional aspects of Coq.)

Our first 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;; WHILE not (Z = 0) DO Y ::= Y * Z;; Z ::= Z - 1 END

This chapter looks at how to define the *syntax* and *semantics* of Imp; the chapters that follow develop a theory of *program equivalence* and introduce *Hoare Logic*, a widely used logic for reasoning about imperative programs.

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Lists.List.
Import ListNotations.

Require Import Maps.
```

14.2 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 expressions with *variables*, and finally a language of *commands* including assignment, conditions, sequencing, and loops.

14.2.1 Syntax

Module *AExp*.

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

Inductive *aexp* : Type :=
  | *ANum* : *nat* → *aexp*
  | *APlus* : *aexp* → *aexp* → *aexp*
  | *AMinus* : *aexp* → *aexp* → *aexp*
  | *AMult* : *aexp* → *aexp* → *aexp*.

Inductive *bexp* : Type :=
  | *BTrue* : *bexp*
  | *BFalse* : *bexp*
  | *BEq* : *aexp* → *aexp* → *bexp*
  | *BLe* : *aexp* → *aexp* → *bexp*
  | *BNot* : *bexp* → *bexp*
  | *BAnd* : *bexp* → *bexp* → *bexp*.

In this chapter, we'll 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 implementation of a 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 taken a course where these techniques are covered (e.g., a compilers course) 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 | not b | b and b

Compared to the Coq 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 + and is an infix symbol) while leaving other aspects of lexical analysis and parsing (like the relative precedence of +, -, and ×, the use of parens to explicitly group subexpressions, etc.) unspecified. Some additional information (and human intelligence) would be required to turn this description into a formal definition, for example when implementing a compiler.

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

- On the other hand, 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 getting every detail nailed down precisely.
>
> Indeed, there are dozens of BNF-like notations and people switch freely among them, usually without bothering to say which form 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.

14.2.2 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_aeval1:
  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 ⇒ beq_nat (aeval a1) (aeval a2)
  | BLe a1 a2 ⇒ leb (aeval a1) (aeval a2)
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

14.2.3 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 make sure 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 sure the optimization is correct – i.e., that evaluating an optimized expression gives the same result as the original – we should prove it.

```
Theorem optimize_0plus_sound: ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a. induction a.
  - reflexivity.
  - destruct a1.
    + destruct n.
      × 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.
```

14.3 Coq 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 Coq'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 – Coq'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.

14.3.1 Tacticals

Tacticals is Coq'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 (instead of failing).

```
Theorem silly1 : ∀ ae, aeval ae = aeval ae.
Proof. try reflexivity. Qed.

Theorem silly2 : ∀ (P : Prop), P → P.
Proof.
  intros P HP.
  try reflexivity.   apply HP. Qed.
```

There is no real 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 : ∀ n, leb 0 n = true.
Proof.
  intros.
  destruct n.
```

```
    - simpl. reflexivity.
    - simpl. reflexivity.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma foo' : ∀ n, leb 0 n = 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': ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;

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

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

Coq experts often use this "...; `try`... " idiom after a tactic like `induction` to take care of many similar cases all at once. Naturally, 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`,

aeval (optimize_0plus a) = 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

  aeval (optimize_0plus (ANum n)) = aeval (ANum n).

This is immediate from the definition of optimize_0plus.

- Suppose a = APlus *a1 a2* for some *a1* and *a2*. We must show

  aeval (optimize_0plus (APlus a1 a2)) = aeval (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* = ANum n for some n. If n = ANum 0, then

  optimize_0plus (APlus a1 a2) = optimize_0plus a2

  and the IH for *a2* is exactly what we need. On the other hand, if n = S *n'* for some *n'*, then again optimize_0plus simply calls itself recursively, and the result follows from the IH. ☐

However, this proof can still be improved: the first case (for a = 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'': ∀ a,
  aeval (optimize_0plus a) = 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* | *T2* | ... | *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'* | *T'* | ... | *T'*

The `repeat` Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails. Here is an example showing 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 still succeeds without changing the original goal (i.e., it repeats zero times).

```
Theorem In10' : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (left; reflexivity).
  repeat (right; try (left; reflexivity)).
Qed.
```

The tactic `repeat` T also does not have any upper bound on the number of times it applies T. If T is a tactic that always succeeds, then repeat T will loop forever (e.g., `repeat simpl` loops forever, since `simpl` always succeeds). While evaluation in Coq's term language, Gallina, is guaranteed to terminate, tactic evaluation is not! This does not affect Coq's logical consistency, however, since the job of `repeat` and other tactics is to guide Coq in constructing proofs; if the construction process diverges, this simply means that we have failed to construct a proof, not that we have constructed a wrong one.

Exercise: 3 stars (optimize_0plus_b) Since the optimize_0plus transformation doesn't change the value of **aexp**s, we should be able to apply it to all the **aexp**s that appear in a **bexp** without changing the **bexp**'s value. Write a function which performs that transformation on **bexp**s, and prove it is sound. Use the tacticals we've just seen to make the proof as elegant as possible.

```
Fixpoint optimize_0plus_b (b : bexp) : bexp
  . Admitted.

Theorem optimize_0plus_b_sound : ∀ b,
  beval (optimize_0plus_b b) = beval b.
Proof.
  Admitted.
```

☐

Exercise: 4 stars, optional (optimizer) *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 prove it correct – and build up to something more interesting incrementially.)

☐

14.3.2 Defining New Tactic Notations

Coq also provides several ways of "programming" tactic scripts.

- The `Tactic Notation` idiom illustrated below gives a handy way to define "shorthand tactics" that bundle several tactics into a single command.
- For more sophisticated programming, Coq offers a built-in programming language called `Ltac` with primitives that can examine and modify the proof state. The details are a bit too complicated to get into here (and it is generally agreed that `Ltac` is not the most beautiful part of Coq's design!), but they can be found in the reference manual and other books on Coq, and there are many examples of `Ltac` definitions in the Coq standard library that you can use as examples.
- There is also an OCaml API, which can be used to build tactics that access Coq's internal structures at a lower level, but this is seldom worth the trouble for ordinary Coq users.

The `Tactic Notation` mechanism is the easiest to come to grips with, and it offers plenty of power for many purposes. Here's an example.

```
Tactic Notation "simpl_and_try" tactic(c) :=
  simpl;
  try c.
```

This defines a new tactical called *simpl_and_try* that takes one tactic `c` as an argument and is defined to be equivalent to the tactic `simpl; try c`. Now writing "*simpl_and_try* `reflexivity`." in a proof will be the same as writing "`simpl; try reflexivity`."

14.3.3 The `omega` Tactic

The `omega` tactic implements a decision procedure for a subset of first-order logic called *Presburger arithmetic*. It is based on the Omega algorithm invented in 1991 by William Pugh *Pugh* 1991.

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 inequality ($\leq$), and
- the logical connectives $\wedge$, $\vee$, $\neg$, and $\rightarrow$,

then invoking `omega` will either solve the goal or tell you that it is actually false.

```
Require Import Coq.omega.Omega.

Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. omega.
Qed.
```

14.3.4 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`: 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`.
- `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, behave like `apply` H.
- *contradiction*: Try to find a hypothesis H in the current 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 below.

14.4 Evaluation as a Relation

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

```
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).
```

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`. (This notation is one place where the limitation to ASCII symbols becomes a little bothersome. The standard notation for the evaluation relation is a double down-arrow. We'll typeset it like this in the HTML version of the notes and use a double slash as the closest approximation in *.v* files.)

```
Notation "e '\\' n"
          := (aevalR e n)
             (at level 50, left associativity)
          : type_scope.

End aevalR_first_try.
```

In fact, Coq provides a way to use this notation in the definition of **aevalR** itself. This reduces confusion by avoiding situations where we're working on a proof involving statements in the form `e \\ n` but we have to refer back to a definition written using the form **aevalR** `e n`.

We do this by first "reserving" the notation, then giving the definition together with a declaration of what the notation means.

```
Reserved Notation "e '\\' n" (at level 50, left associativity).

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

| E_ANum : $\forall$ (n:nat),
    ($ANum$ n) \\ n
| E_APlus : $\forall$ ($e1$ $e2$: $aexp$) ($n1$ $n2$: nat),
    ($e1$ \\ $n1$) $\rightarrow$ ($e2$ \\ $n2$) $\rightarrow$ ($APlus$ $e1$ $e2$) \\ ($n1$ + $n2$)
| E_AMinus : $\forall$ ($e1$ $e2$: $aexp$) ($n1$ $n2$: nat),
    ($e1$ \\ $n1$) $\rightarrow$ ($e2$ \\ $n2$) $\rightarrow$ ($AMinus$ $e1$ $e2$) \\ ($n1$ - $n2$)
| E_AMult : $\forall$ ($e1$ $e2$: $aexp$) ($n1$ $n2$: nat),
    ($e1$ \\ $n1$) $\rightarrow$ ($e2$ \\ $n2$) $\rightarrow$ ($AMult$ $e1$ $e2$) \\ ($n1$ $\times$ $n2$)

where "e '\\' n" := ($aevalR$ e n) : $type\_scope$.

14.4.1 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 (we have already seen them in the `Prop` chapter).

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)

...would 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 either elided or else indicated by saying something like "Let **aevalR** be the smallest relation closed under the following rules...".

For example, \\ is 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 e2 \\ n2

(E_AMinus) AMinus e1 e2 \\ n1-n2

e1 \\ n1 e2 \\ n2

(E_AMult) AMult e1 e2 \\ n1*n2

14.4.2 Equivalence of the Definitions

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

```
Theorem aeval_iff_aevalR : ∀ a n,
  (a \\ n) ↔ 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.
```

We can make the proof quite a bit shorter by making more use of tacticals.

```
Theorem aeval_iff_aevalR' : ∀ 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 (bevalR) Write a relation **bevalR** in the same style as **aevalR**, and prove that it is equivalent to beval.

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

.

Lemma beval_iff_bevalR : ∀ b bv,
  bevalR b bv ↔ beval b = bv.
Proof.
   Admitted.
   □

End AExp.
```

14.4.3 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.

However, there are circumstances 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 by considering also a division operation:

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp
  | ADiv : aexp → aexp → 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 straightforward.

```
Reserved Notation "e '\\' n"
                  (at level 50, 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.

End aevalR_division.

Module aevalR_extended.
```

Suppose, instead, 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 distribution of results, but just saying what results are *possible*.)

```
Reserved Notation "e '\\' n" (at level 50, left associativity).

Inductive aexp : Type :=
  | AAny : aexp
  | ANum : nat → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → 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 style should I use by default? The examples above show that relational definitions are fundamentally more powerful 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 choice. But what about when both styles are workable?

One point in favor of relational definitions is that some people feel they are more elegant and easier to understand. Another is that Coq automatically generates nice inversion and induction principles from `Inductive` definitions.

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

- Functions are by definition deterministic and defined on all arguments; for a relation we have to show these properties explicitly if we need them.

- With functions we can also take advantage of Coq's computation mechanism to simplify expressions during proofs.

Furthermore, functions can be directly "extracted" 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 Coq 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.

14.5 Expressions With Variables

Let's turn our attention back to defining Imp. 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.

14.5.1 States

Since we'll want to look variables up to find out their current values, we'll reuse the type **id** from the Maps chapter for the type of variables in Imp.

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 going to mention a finite number of them. The state captures all of the information stored in memory. For Imp programs, because each variable stores a natural number, we can represent the state as a mapping from identifiers to `nat`. For more complex programming languages, the state might have more structure.

```
Definition state := total_map nat.

Definition empty_state : state :=
  t_empty 0.
```

14.5.2 Syntax

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

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : id → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.
```

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

```
Definition W : id := Id "W".
Definition X : id := Id "X".
Definition Y : id := Id "Y".
Definition Z : id := Id "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 devoped to Imp, this overloading should not cause confusion.)

The definition of **bexp**s is unchanged (except for using the new **aexp**s):

```
Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp → aexp → bexp
  | BLe : aexp → aexp → bexp
  | BNot : bexp → bexp
  | BAnd : bexp → bexp → bexp.
```

14.5.3 Evaluation

The arith and boolean evaluators are extended to handle variables in the obvious way, taking a state as an extra argument:

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

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ beq_nat (aeval st a1) (aeval st a2)
  | BLe a1 a2 ⇒ leb (aeval st a1) (aeval st a2)
  | BNot b1 ⇒ negb (beval st b1)
  | BAnd b1 b2 ⇒ andb (beval st b1) (beval st b2)
  end.

Example aexp1 :
  aeval (t_update empty_state X 5)
        (APlus (ANum 3) (AMult (AId X) (ANum 2)))
  = 13.
Proof. reflexivity. Qed.

Example bexp1 :
  beval (t_update empty_state X 5)
        (BAnd BTrue (BNot (BLe (AId X) (ANum 4))))
  = true.
Proof. reflexivity. Qed.
```

14.6 Commands

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

14.6.1 Syntax

Informally, commands `c` are described by the following BNF grammar. (We choose this slightly awkward concrete syntax for the sake of being able to define Imp syntax using Coq's Notation mechanism. In particular, we use *IFB* to avoid conflicting with the `if` notation from the standard library.)

c ::= SKIP | x ::= a | c ;; c | IFB b THEN c ELSE c FI | WHILE b DO c END

For example, here's factorial in Imp:

Z ::= X;; Y ::= 1;; WHILE not (Z = 0) DO Y ::= Y * Z;; Z ::= Z - 1 END

When this command terminates, the variable Y will contain the factorial of the initial value of X.

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

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.
```

As usual, we can use a few `Notation` declarations to make things more readable. To avoid conflicts with Coq's built-in notations, we keep this light – in particular, we don't introduce any notations for *aexps* and *bexps* to avoid confusion with the numeric and boolean operators we've already defined.

```
Notation "'SKIP'" :=
  CSkip.
Notation "x '::=' a" :=
  (CAss x a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity).
```

For example, here is the factorial function again, written as a formal definition to Coq:

```
Definition fact_in_coq : com :=
  Z ::= AId X;;
  Y ::= ANum 1;;
  WHILE BNot (BEq (AId Z) (ANum 0)) DO
    Y ::= AMult (AId Y) (AId Z);;
    Z ::= AMinus (AId Z) (ANum 1)
  END.
```

14.6.2 More Examples

Assignment:

```
Definition plus2 : com :=
  X ::= (APlus (AId X) (ANum 2)).
```

```
Definition XtimesYinZ : com :=
  Z ::= (AMult (AId X) (AId Y)).
```

```
Definition subtract_slowly_body : com :=
  Z ::= AMinus (AId Z) (ANum 1) ;;
  X ::= AMinus (AId X) (ANum 1).
```

Loops

```
Definition subtract_slowly : com :=
  WHILE BNot (BEq (AId X) (ANum 0)) DO
    subtract_slowly_body
  END.

Definition subtract_3_from_5_slowly : com :=
  X ::= ANum 3 ;;
  Z ::= ANum 5 ;;
  subtract_slowly.
```

An infinite loop:

```
Definition loop : com :=
  WHILE BTrue DO
    SKIP
  END.
```

14.7 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...

14.7.1 Evaluation as a Function (Failed Attempt)

Here's an attempt at defining an evaluation function for commands, omitting the *WHILE* case.

```
Fixpoint ceval_fun_no_while (st : state) (c : com)
                            : state :=
  match c with
    | SKIP ⇒
        st
    | x ::= a1 ⇒
        t_update st x (aeval st a1)
    | c1 ;; c2 ⇒
        let st' := ceval_fun_no_while st c1 in
        ceval_fun_no_while st' c2
```

```
  | IFB b THEN c1 ELSE c2 FI ⇒
      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 traditional 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.

Coq 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 Coq is not just a functional programming language but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an (invalid!) program showing what would go wrong if Coq 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 Coq's logical consistency.

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

14.7.2 Evaluation as a Relation

Here's a better way: define **ceval** as a *relation* rather than a *function* – i.e., define it in `Prop` instead of `Type`, as we did for **aevalR** above.

This is an important change. Besides freeing us from awkward workarounds, it gives us a lot 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 `c` / *st* \\ *st'* for the **ceval** relation: `c` / *st* \\ *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) SKIP / st \\ st

aeval st a1 = n

(E_Ass) x := a1 / st \\ (t_update st x n)
c1 / st \\ st' c2 / st' \\ st''

(E_Seq) c1;;c2 / st \\ st''
beval st b1 = true c1 / st \\ st'

(E_IfTrue) IF b1 THEN c1 ELSE c2 FI / st \\ st'
beval st b1 = false c2 / st \\ st'

(E_IfFalse) IF b1 THEN c1 ELSE c2 FI / st \\ st'
beval st b = false

(E_WhileEnd) WHILE b DO c END / st \\ st
beval st b = true c / st \\ st' WHILE b DO c END / st' \\ st''

(E_WhileLoop) WHILE b DO c END / st \\ st''

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

```
Reserved Notation "c1 '/' st '\\' st'"
                  (at level 40, st at level 39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      SKIP / st \\ st
  | E_Ass : ∀ st a1 n x,
      aeval st a1 = n →
      (x ::= a1) / st \\ (t_update st x n)
  | E_Seq : ∀ c1 c2 st st' st'',
      c1 / st \\ st' →
      c2 / st' \\ st'' →
      (c1 ;; c2) / st \\ st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      c1 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      c2 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileEnd : ∀ b st c,
      beval st b = false →
      (WHILE b DO c END) / st \\ st
```

```
  | E_WhileLoop : ∀ st st' st'' b c,
      beval st b = true →
      c / st \\ st' →
      (WHILE b DO c END) / st' \\ st'' →
      (WHILE b DO c END) / st \\ st''

  where "c1 '/' st '\\' st'" := (ceval c1 st st').
```

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

```
Example ceval_example1:
    (X ::= ANum 2;;
     IFB BLe (AId X) (ANum 1)
       THEN Y ::= ANum 3
       ELSE Z ::= ANum 4
     FI)
   / empty_state
   \\ (t_update (t_update empty_state X 2) Z 4).
Proof.
  apply E_Seq with (t_update empty_state X 2).
  -
    apply E_Ass. reflexivity.
  -
    apply E_IfFalse.
      reflexivity.
      apply E_Ass. reflexivity. Qed.
```

Exercise: 2 stars (ceval_example2) `Example` *ceval_example2*:

```
    (X ::= ANum 0;; Y ::= ANum 1;; Z ::= ANum 2) / empty_state \\
    (t_update (t_update (t_update empty_state X 0) Y 1) Z 2).
Proof.
   Admitted.
```

☐

Exercise: 3 stars, advanced (pup_to_n) Write an Imp program that sums the numbers from 1 to X (inclusive: 1 + 2 + ... + X) in the variable Y. Prove that this program executes as intended for X = 2 (this is trickier than you might expect).

```
Definition pup_to_n : com
  . Admitted.

Theorem pup_to_2_ceval :
  pup_to_n / (t_update empty_state X 2) \\
```

t_update (*t_update* (*t_update* (*t_update* (*t_update* (*t_update* *empty_state*
  X 2) *Y* 0) *Y* 2) *X* 1) *Y* 3) *X* 0.

```
Proof.
  Admitted.
```

☐

14.7.3 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: ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     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.
```

14.8 Reasoning About Imp Programs

We'll get deeper into systematic techniques for reasoning about Imp programs in the following chapters, but we can do quite a bit just working with the bare definitions. This section explores some examples.

```
Theorem plus2_spec : ∀ st n st',
  st X = n →
  plus2 / st \\ st' →
  st' X = n + 2.
Proof.
  intros st n st' HX Heval.
```

Inverting *Heval* essentially forces Coq 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, recommendedM (XtimesYinZ_spec) State and prove a specification of XtimesYinZ.

□

Exercise: 3 stars, recommended (loop_never_stops) Theorem *loop_never_stops* : ∀ *st st'*,

```
  ~(loop / st \\ st').
Proof.
  intros st st' contra. unfold loop in contra.
  remember (WHILE BTrue DO SKIP END) as loopdef
             eqn:Heqloopdef.
```

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 inversion).

Admitted.

□

Exercise: 3 stars (no_whilesR) 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)
  | IFB _ THEN ct ELSE cf FI ⇒
      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 starsM (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.

□

14.9 Additional Exercises

Exercise: 3 stars (stack_compiler) 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 entered 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 task of this exercise is to write a small compiler that translates **aexp**s 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.
- SMult: Similar, but multiply.

```
Inductive sinstr : Type :=
| SPush : nat → sinstr
| SLoad : id → sinstr
| SPlus : sinstr
| SMinus : sinstr
| SMult : sinstr.
```

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 the specification leaves unspecified what to do when encountering an SPlus, SMinus, or SMult instruction if the stack contains less than two elements. In a sense, it is immaterial what we do, since our compiler will never emit such a malformed program.

```
Fixpoint s_execute (st : state) (stack : list nat)
                   (prog : list sinstr)
                 : list nat
  . Admitted.

Example s_execute1 :
     s_execute empty_state []
       [SPush 5; SPush 3; SPush 1; SMinus]
   = [2; 5].
   Admitted.

Example s_execute2 :
     s_execute (t_update empty_state 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 (AMinus (AId X) (AMult (ANum 2) (AId Y)))
  = [SLoad X; SPush 2; SLoad Y; SMult; SMinus].
  Admitted.
```

□

Exercise: 4 stars, advanced (stack_compiler_correct) Now we'll prove the correctness of the compiler implemented in the previous exercise. Remember that the specification left unspecified what to do when encountering an SPlus, SMinus, or SMult instruction if the stack contains less than two elements. (In order to make your correctness proof easier you might find it helpful to go back and change your implementation!)

Prove the following theorem. You will need to start by stating a more general lemma to get a usable induction hypothesis; the main theorem will then be a simple corollary of this lemma.

```
Theorem s_compile_correct : ∀ (st : state) (e : aexp),
  s_execute st [] (s_compile e) = [ aeval st e ].
Proof.
  Admitted.
```

□

Exercise: 3 stars, 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.

□

```
Module BreakImp.
```

Exercise: 4 stars, advanced (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 : com
  | CBreak : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Notation "'SKIP'" :=
  CSkip.
Notation "'BREAK'" :=
  CBreak.
Notation "x '::=' a" :=
  (CAss x a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity).
```

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 : result
  | SBreak : result.

Reserved Notation "c1 '/' st '\\' s '/' st'"
                  (at level 40, st, s at level 39).
```

Intuitively, c / *st* \\ s / *st'* 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 = SBreak) or that execution should continue normally (s = SContinue).

The definition of the "c / *st* \\ s / *st'*" relation is very similar to the one we gave above for the regular evaluation relation (c / *st* \\ *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 *IFB* b *THEN* c1 *ELSE* c2 *FI*, 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 *WHILE* b *DO* c *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 → state → result → state → Prop :=
  | E_Skip : ∀ st,
      CSkip / st \\ SContinue / st


  where "c1 '/' st '\\' s '/' st'" := (ceval c1 st s st').
```

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

```
Theorem break_ignore : ∀ c st st' s,
     (BREAK;; c) / st \\ s / st' →
     st = st'.
Proof.
  Admitted.

Theorem while_continue : ∀ b c st st' s,
```

(*WHILE b DO c END*) / *st* \\ *s* / *st'* →
s = *SContinue*.
`Proof.`
Admitted.

`Theorem` *while_stops_on_break* : ∀ *b c st st'*,
beval st b = *true* →
c / *st* \\ *SBreak* / *st'* →
(*WHILE b DO c END*) / *st* \\ *SContinue* / *st'*.
`Proof.`
Admitted.
☐

Exercise: 3 stars, advanced, optional (while_break_true) `Theorem` *while_break_true* : ∀ *b c st st'*,
(*WHILE b DO c END*) / *st* \\ *SContinue* / *st'* →
beval st' b = *true* →
∃ *st''*, *c* / *st''* \\ *SBreak* / *st'*.
`Proof.`
Admitted.
☐

Exercise: 4 stars, advanced, optional (ceval_deterministic) `Theorem` *ceval_deterministic*: ∀ (*c*:*com*) *st st1 st2 s1 s2*,
c / *st* \\ *s1* / *st1* →
c / *st* \\ *s2* / *st2* →
st1 = *st2* ∧ *s1* = *s2*.
`Proof.`
Admitted.

☐ `End` *BreakImp*.

Exercise: 4 stars, 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 Coq.

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 15

Library Top.ImpParser

15.1 ImpParser: Lexing and Parsing in Coq

The development of the Imp language in *Imp.v* completely ignores issues of concrete syntax – how an ascii string that a programmer might write gets translated into abstract syntax trees defined by the datatypes **aexp**, **bexp**, and **com**. In this chapter, we illustrate how the rest of the story can be filled in by building a simple lexical analyzer and parser using Coq's functional programming facilities.

It is not important to understand all the details here (and accordingly, the explanations are fairly terse and there are no exercises). The main point is simply to demonstrate that it can be done. You are invited to look through the code – most of it is not very complicated, though the parser relies on some "monadic" programming idioms that may require a little work to make out – but most readers will probably want to just skim down to the Examples section at the very end to get the punchline.

15.2 Internals

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.Lists.List.
Import ListNotations.
Require Import Maps.
Require Import Imp.
```

15.2.1 Lexical Analysis

```
Definition isWhite (c : ascii) : bool :=
```

```
  let n := nat_of_ascii c in
  orb (orb (beq_nat n 32)
           (beq_nat n 9))
      (orb (beq_nat n 10)
           (beq_nat n 13)).

Notation "x '<=?' y" := (leb x y)
  (at level 70, no associativity) : nat_scope.

Definition isLowerAlpha (c : ascii) : bool :=
  let n := nat_of_ascii c in
    andb (97 <=? n) (n <=? 122).

Definition isAlpha (c : ascii) : bool :=
  let n := nat_of_ascii c in
    orb (andb (65 <=? n) (n <=? 90))
        (andb (97 <=? n) (n <=? 122)).

Definition isDigit (c : ascii) : bool :=
  let n := nat_of_ascii c in
     andb (48 <=? n) (n <=? 57).

Inductive chartype := white | alpha | digit | other.

Definition classifyChar (c : ascii) : chartype :=
  if isWhite c then
    white
  else if isAlpha c then
    alpha
  else if isDigit c then
    digit
  else
    other.

Fixpoint list_of_string (s : string) : list ascii :=
  match s with
  | EmptyString => []
  | String c s => c :: (list_of_string s)
  end.

Fixpoint string_of_list (xs : list ascii) : string :=
  fold_right String EmptyString xs.

Definition token := string.

Fixpoint tokenize_helper (cls : chartype) (acc xs : list ascii)
                         : list (list ascii) :=
  let tk := match acc with [] => [] | _::_ => [rev acc] end in
  match xs with
  | [] => tk
```

```
  | (x::xs') ⇒
    match cls, classifyChar x, x with
    | _, _, "(" ⇒
      tk ++ ["("]::(tokenize_helper other [] xs')
    | _, _, ")" ⇒
      tk ++ [")"]::(tokenize_helper other [] xs')
    | _, white, _ ⇒
      tk ++ (tokenize_helper white [] xs')
    | alpha,alpha,x ⇒
      tokenize_helper alpha (x::acc) xs'
    | digit,digit,x ⇒
      tokenize_helper digit (x::acc) xs'
    | other,other,x ⇒
      tokenize_helper other (x::acc) xs'
    | _,tp,x ⇒
      tk ++ (tokenize_helper tp [x] xs')
    end
  end %char.

Definition tokenize (s : string) : list string :=
  map string_of_list (tokenize_helper white [] (list_of_string s)).

Example tokenize_ex1 :
    tokenize "abc12==3 223*(3+(a+c))" %string
  = ["abc"; "12"; "=="; "3"; "223";
       "*"; "("; "3"; "+"; "(";
       "a"; "+"; "c"; ")"; ")"]%string.
Proof. reflexivity. Qed.
```

15.2.2 Parsing

Options With Errors

An **option** type with error messages:

```
Inductive optionE (X:Type) : Type :=
  | SomeE : X → optionE X
  | NoneE : string → optionE X.

Implicit Arguments SomeE [[X]].
Implicit Arguments NoneE [[X]].
```

Some syntactic sugar to make writing nested match-expressions on optionE more convenient.

```
Notation "'DO' ( x , y ) <== e1 ; e2"
   := (match e1 with
```

```
            | SomeE (x,y) ⇒ e2
            | NoneE err ⇒ NoneE err
         end)
   (right associativity, at level 60).
Notation "'DO' ( x , y ) <- e1 ; e2 'OR' e3"
   := (match e1 with
            | SomeE (x,y) ⇒ e2
            | NoneE err ⇒ e3
         end)
   (right associativity, at level 60, e2 at next level).
```

Generic Combinators for Building Parsers

```
Open Scope string_scope.
Definition parser (T : Type) :=
  list token → optionE (T × list token).
Fixpoint many_helper {T} (p : parser T) acc steps xs :=
  match steps, p xs with
  | 0, _ ⇒
      NoneE "Too many recursive calls"
  | _, NoneE _ ⇒
      SomeE ((rev acc), xs)
  | S steps', SomeE (t, xs') ⇒
      many_helper p (t::acc) steps' xs'
  end.
```

A (step-indexed) parser that expects zero or more *p*s:

```
Fixpoint many {T} (p : parser T) (steps : nat) : parser (list T) :=
  many_helper p [] steps.
```

A parser that expects a given token, followed by *p*:

```
Definition firstExpect {T} (t : token) (p : parser T)
                     : parser T :=
  fun xs ⇒ match xs with
              | x::xs' ⇒
                 if string_dec x t
                 then p xs'
                 else NoneE ("expected '" ++ t ++ "'.")
              | [] ⇒
                 NoneE ("expected '" ++ t ++ "'.")
              end.
```

A parser that expects a particular token:

```
Definition expect (t : token) : parser unit :=
  firstExpect t (fun xs ⇒ SomeE(tt, xs)).
```

A Recursive-Descent Parser for Imp

Identifiers:

```
Definition parseIdentifier (xs : list token)
                         : optionE (id × list token) :=
match xs with
| [] ⇒ NoneE "Expected identifier"
| x::xs' ⇒
    if forallb isLowerAlpha (list_of_string x) then
      SomeE (Id x, xs')
    else
      NoneE ("Illegal identifier:'" ++ x ++ "'")
end.
```

Numbers:

```
Definition parseNumber (xs : list token)
                     : optionE (nat × list token) :=
match xs with
| [] ⇒ NoneE "Expected number"
| x::xs' ⇒
    if forallb isDigit (list_of_string x) then
      SomeE (fold_left
               (fun n d ⇒
                  10 × n + (nat_of_ascii d -
                            nat_of_ascii "0"%char))
               (list_of_string x)
               0,
             xs')
    else
      NoneE "Expected number"
end.
```

Parse arithmetic expressions

```
Fixpoint parsePrimaryExp (steps:nat)
                         (xs : list token)
                       : optionE (aexp × list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
      DO (i, rest) <-- parseIdentifier xs ;
```

```
          SomeE (AId i, rest)
      OR DO (n, rest) <- parseNumber xs ;
          SomeE (ANum n, rest)
                OR (DO (e, rest) <== firstExpect "("
                       (parseSumExp steps') xs;
          DO (u, rest') <== expect ")" rest ;
          SomeE(e,rest'))
  end

with parseProductExp (steps:nat)
                     (xs : list token) :=
  match steps with
  | 0 => NoneE "Too many recursive calls"
  | S steps' =>
    DO (e, rest) <==
      parsePrimaryExp steps' xs ;
    DO (es, rest') <==
       many (firstExpect "*" (parsePrimaryExp steps'))
            steps' rest;
    SomeE (fold_left AMult es e, rest')
  end

with parseSumExp (steps:nat) (xs : list token) :=
  match steps with
  | 0 => NoneE "Too many recursive calls"
  | S steps' =>
    DO (e, rest) <==
      parseProductExp steps' xs ;
    DO (es, rest') <==
      many (fun xs =>
        DO (e,rest') <-
           firstExpect "+"
             (parseProductExp steps') xs;
           SomeE ( (true, e), rest')
        OR DO (e,rest') <==
        firstExpect "-"
           (parseProductExp steps') xs;
            SomeE ( (false, e), rest'))
        steps' rest;
      SomeE (fold_left (fun e0 term =>
                          match term with
                            (true, e) => APlus e0 e
```

```
                              | (false, e) ⇒ AMinus e0 e
                              end)
                          es e,
          rest')
  end.
```

```
Definition parseAExp := parseSumExp.
```

Parsing boolean expressions:

```
Fixpoint parseAtomicExp (steps:nat)
                        (xs : list token) :=
match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
     DO (u,rest) <- expect "true" xs;
        SomeE (BTrue,rest)
     OR DO (u,rest) <- expect "false" xs;
        SomeE (BFalse,rest)
     OR DO (e,rest) <-
           firstExpect "not"
              (parseAtomicExp steps')
              xs;
        SomeE (BNot e, rest)
     OR DO (e,rest) <-
             firstExpect "("
               (parseConjunctionExp steps') xs;
          (DO (u,rest') <== expect ")" rest;
             SomeE (e, rest'))
     OR DO (e, rest) <== parseProductExp steps' xs;
           (DO (e', rest') <-
             firstExpect "=="
               (parseAExp steps') rest;
             SomeE (BEq e e', rest')
            OR DO (e', rest') <-
              firstExpect "<="
                (parseAExp steps') rest;
              SomeE (BLe e e', rest')
            OR
              NoneE
      "Expected '==' or '<=' after arithmetic expression")
end

with parseConjunctionExp (steps:nat)
                         (xs : list token) :=
```

```
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    DO (e, rest) <==
      parseAtomicExp steps' xs ;
    DO (es, rest') <==
        many (firstExpect "&&"
                (parseAtomicExp steps'))
             steps' rest;
    SomeE (fold_left BAnd es e, rest')
  end.
```

```
Definition parseBExp := parseConjunctionExp.
```

```
Check parseConjunctionExp.
```

```
Definition testParsing {X : Type}
           (p : nat →
                list token →
                optionE (X × list token))
           (s : string) :=
  let t := tokenize s in
  p 100 t.
```

Parsing commands:

```
Fixpoint parseSimpleCommand (steps:nat)
                            (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    DO (u, rest) <-- expect "SKIP" xs;
      SomeE (SKIP, rest)
    OR DO (e,rest) <--
         firstExpect "IF" (parseBExp steps') xs;
       DO (c,rest') <==
         firstExpect "THEN"
           (parseSequencedCommand steps') rest;
       DO (c',rest'') <==
         firstExpect "ELSE"
           (parseSequencedCommand steps') rest';
       DO (u,rest''') <==
         expect "END" rest'';
       SomeE(IFB e THEN c ELSE c' FI, rest''')
    OR DO (e,rest) <--
         firstExpect "WHILE"
```

```
              (parseBExp steps') xs;
          DO (c,rest') <==
            firstExpect "DO"
              (parseSequencedCommand steps') rest;
          DO (u,rest'') <==
            expect "END" rest';
          SomeE(WHILE e DO c END, rest'')
       OR DO (i, rest) <==
            parseIdentifier xs;
          DO (e, rest') <==
            firstExpect ":=" (parseAExp steps') rest;
          SomeE(i ::= e, rest')
  end

with parseSequencedCommand (steps:nat)
                           (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
        DO (c, rest) <==
          parseSimpleCommand steps' xs;
        DO (c', rest') <--
          firstExpect ";"
            (parseSequencedCommand steps') rest;
          SomeE(c ;; c', rest')
        OR
          SomeE(c, rest)
  end.

Definition bignumber := 1000.

Definition parse (str : string) : optionE (com × list token) :=
  let tokens := tokenize str in
  parseSequencedCommand bignumber tokens.
```

15.3 Examples

$Date: 2017-01-31 19:12:59 -0500 (Tue, 31 Jan 2017)$

Chapter 16

Library Top.ImpCEvalFun

16.1 ImpCEvalFun: Evaluation Function for Imp

We saw in the Imp chapter how a naive approach to defining a function representing evaluation for Imp runs into difficulties. There, we adopted the solution of changing from a functional to a relational definition of evaluation. In this optional chapter, we consider strategies for getting the functional approach to work.

16.2 A Broken Evaluator

```
Require Import Coq.omega.Omega.
Require Import Coq.Arith.Arith.
Require Import Imp.
Require Import Maps.
```

Here was our first try at an evaluation function for commands, omitting *WHILE*.

```
Fixpoint ceval_step1 (st : state) (c : com) : state :=
  match c with
    | SKIP ⇒
        st
    | l ::= a1 ⇒
        t_update st l (aeval st a1)
    | c1 ;; c2 ⇒
        let st' := ceval_step1 st c1 in
        ceval_step1 st' c2
    | IFB b THEN c1 ELSE c2 FI ⇒
        if (beval st b)
          then ceval_step1 st c1
          else ceval_step1 st c2
    | WHILE b1 DO c1 END ⇒
```

```
      st
  end.
```

As we remarked in chapter Imp, in a traditional functional programming language like ML or Haskell we could write the WHILE case as follows:

| WHILE b1 DO c1 END => if (beval st b1) then ceval_step1 st (c1;; WHILE b1 DO c1 END) else st

Coq 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, the changed ceval_step1 function applied to the loop program from *Imp.v* would never terminate. Since Coq is not just a functional programming language, but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an invalid(!) Coq program showing what would go wrong if Coq allowed non-terminating recursive functions:

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

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

Thus, because it doesn't terminate on all inputs, the full version of ceval_step1 cannot be written in Coq – at least not without one additional trick...

16.3 A Step-Indexed Evaluator

The trick we need is to pass an *additional* parameter to the evaluation function that tells it how long to run. Informally, we start the evaluator with a certain amount of "gas" in its tank, and we allow it to run until either it terminates in the usual way *or* it runs out of gas, at which point we simply stop evaluating and say that the final result is the empty memory. (We could also say that the result is the current state at the point where the evaluator runs out fo gas – it doesn't really matter because the result is going to be wrong in either case!)

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) : state :=
  match i with
  | O ⇒ empty_state
  | S i' ⇒
    match c with
      | SKIP ⇒
          st
      | l ::= a1 ⇒
          t_update st l (aeval st a1)
      | c1 ;; c2 ⇒
          let st' := ceval_step2 st c1 i' in
          ceval_step2 st' c2 i'
      | IFB b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step2 st c1 i'
```

```
        else ceval_step2 st c2 i'
    | WHILE b1 DO c1 END ⇒
        if (beval st b1)
        then let st' := ceval_step2 st c1 i' in
             ceval_step2 st' c i'
        else st
  end
end.
```

Note: It is tempting to think that the index i here is counting the "number of steps of evaluation." But if you look closely you'll see that this is not the case: for example, in the rule for sequencing, the same i is passed to both recursive calls. Understanding the exact way that i is treated will be important in the proof of ceval__ceval_step, which is given as an exercise below.

One thing that is not so nice about this evaluator is that we can't tell, from its result, whether it stopped because the program terminated normally or because it ran out of gas. Our next version returns an **option** state instead of just a state, so that we can distinguish between normal and abnormal termination.

```
Fixpoint ceval_step3 (st : state) (c : com) (i : nat)
                    : option state :=
  match i with
  | O ⇒ None
  | S i' ⇒
    match c with
      | SKIP ⇒
          Some st
      | l ::= a1 ⇒
          Some (t_update st l (aeval st a1))
      | c1 ;; c2 ⇒
          match (ceval_step3 st c1 i') with
          | Some st' ⇒ ceval_step3 st' c2 i'
          | None ⇒ None
          end
      | IFB b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step3 st c1 i'
            else ceval_step3 st c2 i'
      | WHILE b1 DO c1 END ⇒
          if (beval st b1)
          then match (ceval_step3 st c1 i') with
               | Some st' ⇒ ceval_step3 st' c i'
               | None ⇒ None
               end
```

```
          else Some st
      end
    end.
```

We can improve the readability of this version by introducing a bit of auxiliary notation to hide the plumbing involved in repeatedly matching against optional states.

```
Notation "'LETOPT' x <== e1 'IN' e2"
   := (match e1 with
           | Some x ⇒ e2
           | None ⇒ None
        end)
   (right associativity, at level 60).

Fixpoint ceval_step (st : state) (c : com) (i : nat)
                        : option state :=
  match i with
  | O ⇒ None
  | S i' ⇒
    match c with
      | SKIP ⇒
          Some st
      | l ::= a1 ⇒
          Some (t_update st l (aeval st a1))
      | c1 ;; c2 ⇒
          LETOPT st' <== ceval_step st c1 i' IN
          ceval_step st' c2 i'
      | IFB b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step st c1 i'
            else ceval_step st c2 i'
      | WHILE b1 DO c1 END ⇒
          if (beval st b1)
          then LETOPT st' <== ceval_step st c1 i' IN
               ceval_step st' c i'
          else Some st
    end
  end.

Definition test_ceval (st:state) (c:com) :=
  match ceval_step st c 500 with
  | None ⇒ None
  | Some st ⇒ Some (st X, st Y, st Z)
  end.
```

Exercise: 2 stars, recommended (pup_to_n) Write an Imp program that sums the numbers from 1 to X (inclusive: 1 + 2 + ... + X) in the variable Y. Make sure your solution satisfies the test that follows.

```
Definition pup_to_n : com
  . Admitted.
```

☐

Exercise: 2 stars, optional (peven) Write a *While* program that sets Z to 0 if X is even and sets Z to 1 otherwise. Use *ceval_test* to test your program.

☐

16.4 Relational vs. Step-Indexed Evaluation

As for arithmetic and boolean expressions, we'd hope that the two alternative definitions of evaluation would actually amount to the same thing in the end. This section shows that this is the case.

```
Theorem ceval_step__ceval: ∀ c st st',
      (∃ i, ceval_step st c i = Some st') →
      c / st \\ st'.
Proof.
  intros c st st' H.
  inversion H as [i E].
  clear H.
  generalize dependent st'.
  generalize dependent st.
  generalize dependent c.
  induction i as [| i' ].
  -
    intros c st st' H. inversion H.
  -
    intros c st st' H.
    destruct c;
           simpl in H; inversion H; subst; clear H.
      + apply E_Skip.
      + apply E_Ass. reflexivity.
      +
        destruct (ceval_step st c1 i') eqn:Heqr1.
        ×
          apply E_Seq with s.
            apply IHi'. rewrite Heqr1. reflexivity.
```

```
      apply IHi'. simpl in H1. assumption.
    ×
      inversion H1.
  +
    destruct (beval st b) eqn:Heqr.
    ×
      apply E_IfTrue. rewrite Heqr. reflexivity.
      apply IHi'. assumption.
    ×
      apply E_IfFalse. rewrite Heqr. reflexivity.
      apply IHi'. assumption.
  + destruct (beval st b) eqn :Heqr.
    ×
     destruct (ceval_step st c i') eqn:Heqr1.
     {
       apply E_WhileLoop with s. rewrite Heqr.
       reflexivity.
       apply IHi'. rewrite Heqr1. reflexivity.
       apply IHi'. simpl in H1. assumption. }
     { inversion H1. }
    ×
      inversion H1.
      apply E_WhileEnd.
      rewrite ← Heqr. subst. reflexivity. Qed.
```

Exercise: 4 stars (ceval_step__ceval_inf) Write an informal proof of ceval_step__ceval, following the usual template. (The template for case analysis on an inductively defined value should look the same as for induction, except that there is no induction hypothesis.) Make your proof communicate the main ideas to a human reader; do not simply transcribe the steps of the formal proof.

☐

```
Theorem ceval_step_more: ∀ i1 i2 st st' c,
  i1 ≤ i2 →
  ceval_step st c i1 = Some st' →
  ceval_step st c i2 = Some st'.
Proof.
induction i1 as [|i1']; intros i2 st st' c Hle Hceval.
  -
    simpl in Hceval. inversion Hceval.
  -
    destruct i2 as [|i2']. inversion Hle.
    assert (Hle': i1' ≤ i2') by omega.
```

```
destruct c.
+
  simpl in Hceval. inversion Hceval.
  reflexivity.
+
  simpl in Hceval. inversion Hceval.
  reflexivity.
+
  simpl in Hceval. simpl.
  destruct (ceval_step st c1 i1') eqn:Heqst1'o.
  ×
    apply (IHi1' i2') in Heqst1'o; try assumption.
    rewrite Heqst1'o. simpl. simpl in Hceval.
    apply (IHi1' i2') in Hceval; try assumption.
  ×
    inversion Hceval.

+
  simpl in Hceval. simpl.
  destruct (beval st b); apply (IHi1' i2') in Hceval;
    assumption.

+
  simpl in Hceval. simpl.
  destruct (beval st b); try assumption.
  destruct (ceval_step st c i1') eqn: Heqst1'o.
  ×
    apply (IHi1' i2') in Heqst1'o; try assumption.
    rewrite → Heqst1'o. simpl. simpl in Hceval.
    apply (IHi1' i2') in Hceval; try assumption.
  ×
    simpl in Hceval. inversion Hceval. Qed.
```

Exercise: 3 stars, recommended (ceval__ceval_step) Finish the following proof. You'll need ceval_step_more in a few places, as well as some basic facts about ≤ and plus.

```
Theorem ceval__ceval_step: ∀ c st st',
      c / st \\ st' →
      ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st' Hce.
  induction Hce.
   Admitted.
```

□

```
Theorem ceval_and_ceval_step_coincide: ∀ c st st',
      c / st \\ st'
  ↔ ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st'.
  split. apply ceval__ceval_step. apply ceval_step__ceval.
Qed.
```

16.5 Determinism of Evaluation Again

Using the fact that the relational and step-indexed definition of evaluation are the same, we can give a slicker proof that the evaluation *relation* is deterministic.

```
Theorem ceval_deterministic' : ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 He1 He2.
  apply ceval__ceval_step in He1.
  apply ceval__ceval_step in He2.
  inversion He1 as [i1 E1].
  inversion He2 as [i2 E2].
  apply ceval_step_more with (i2 := i1 + i2) in E1.
  apply ceval_step_more with (i2 := i1 + i2) in E2.
  rewrite E1 in E2. inversion E2. reflexivity.
  omega. omega. Qed.
```

$Date : 2016-10-2220:19:37-0400(Sat, 22Oct2016)$

Chapter 17

Library Top.Extraction

17.1 Extraction: Extracting ML from Coq

17.2 Basic Extraction

In its simplest form, extracting an efficient program from one written in Coq is completely straightforward.

First we say what language we want to extract into. Options are OCaml (the most mature), Haskell (which mostly works), and Scheme (a bit out of date).

`Extraction` *Language Ocaml*.

Now we load up the Coq environment with some definitions, either directly or by importing them from other modules.

`Require Import` *Coq.Arith.Arith*.
`Require Import` *Coq.Arith.EqNat*.
`Require Import` *ImpCEvalFun*.

Finally, we tell Coq the name of a definition to extract and the name of a file to put the extracted code into.

`Extraction` "imp1.ml" *ceval_step*.

When Coq processes this command, it generates a file *imp1.ml* containing an extracted version of `ceval_step`, together with everything that it recursively depends on. Compile the present *.v* file and have a look at *imp1.ml* now.

17.3 Controlling Extraction of Specific Types

We can tell Coq to extract certain `Inductive` definitions to specific OCaml types. For each one, we must say

- how the Coq type itself should be represented in OCaml, and

- how each constructor should be translated.

```
Extract Inductive bool ⇒ "bool" [ "true" "false" ].
```

Also, for non-enumeration types (where the constructors take arguments), we give an OCaml expression that can be used as a “recursor” over elements of the type. (Think Church numerals.)

```
Extract Inductive nat ⇒ "int"
  [ "0" "(fun x -> x + 1)" ]
  "(fun zero succ n -> if n=0 then zero () else succ (n-1))".
```

We can also extract defined constants to specific OCaml terms or operators.

```
Extract Constant plus ⇒ "( + )".
Extract Constant mult ⇒ "( * )".
Extract Constant beq_nat ⇒ "( = )".
```

Important: It is entirely *your responsibility* to make sure that the translations you're proving make sense. For example, it might be tempting to include this one

Extract Constant minus => “(-)”.

but doing so could lead to serious confusion! (Why?)

```
Extraction "imp2.ml" ceval_step.
```

Have a look at the file *imp2.ml*. Notice how the fundamental definitions have changed from *imp1.ml*.

17.4 A Complete Example

To use our extracted evaluator to run Imp programs, all we need to add is a tiny driver program that calls the evaluator and prints out the result.

For simplicity, we'll print results by dumping out the first four memory locations in the final state.

Also, to make it easier to type in examples, let's extract a parser from the ImpParser Coq module. To do this, we need a few magic declarations to set up the right correspondence between Coq strings and lists of OCaml characters.

```
Require Import Ascii String.
Extract Inductive ascii ⇒ char
[
"(* If this appears, you're using Ascii internals. Please don't *) (fun (b0,b1,b2,b3,b4,b5,b6,b7)
-> let f b i = if b then 1 lsl i else 0 in Char.chr (f b0 0 + f b1 1 + f b2 2 + f b3 3 + f b4 4
+ f b5 5 + f b6 6 + f b7 7))"
]
"(* If this appears, you're using Ascii internals. Please don't *) (fun f c -> let n = Char.code
c in let h i = (n land (1 lsl i)) <> 0 in f (h 0) (h 1) (h 2) (h 3) (h 4) (h 5) (h 6) (h 7))".
```

```
Extract Constant zero ⇒ "'\000'".
Extract Constant one ⇒ "'\001'".
Extract Constant shift ⇒
 "fun b c -> Char.chr (((Char.code c) lsl 1) land 255 + if b then 1 else 0)".
Extract Inlined Constant ascii_dec ⇒ "(=)".
```

We also need one more variant of booleans.

```
Extract Inductive sumbool ⇒ "bool" ["true" "false"].
```

The extraction is the same as always.

```
Require Import Imp.
Require Import ImpParser.
Extraction "imp.ml" empty_state ceval_step parse.
```

Now let's run our generated Imp evaluator. First, have a look at *impdriver.ml*. (This was written by hand, not extracted.)

Next, compile the driver together with the extracted code and execute it, as follows.

ocamlc -w -20 -w -26 -o impdriver imp.mli imp.ml impdriver.ml ./impdriver

(The -*w* flags to *ocamlc* are just there to suppress a few spurious warnings.)

17.5 Discussion

Since we've proved that the ceval_step function behaves the same as the **ceval** relation in an appropriate sense, the extracted program can be viewed as a *certified* Imp interpreter. Of course, the parser we're using is not certified, since we didn't prove anything about it!

$Date : 2017 - 01 - 3119 : 12 : 59 - 0500 (Tue, 31 Jan 2017)$

Chapter 18

Library Top.Equiv

18.1 Equiv: Program Equivalence

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Lists.List.
Require Import Coq.Logic.FunctionalExtensionality.
Import ListNotations.
Require Import Maps.
Require Import Imp.
```

Some Advice for Working on Exercises:

- Most of the Coq proofs we ask you to do are similar to proofs that we've provided. Before starting to work on exercises problems, take the time to work through our proofs (both informally, on paper, and in Coq) and make sure you understand them in detail. This will save you a lot of time.

- The Coq proofs we're doing now are sufficiently complicated that it is more or less impossible to complete them simply 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's exercises can get pretty long if you try to write out all the cases explicitly.

18.2 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, including assignment and other commands, we need to consider the role of variables and state.

18.2.1 Definitions

For **aexp**s and **bexp**s with variables, the definition we want is clear. We say that two **aexp**s or **bexp**s are *behaviorally equivalent* if they evaluate to the same result in every state.

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

Definition bequiv (b1 b2 : bexp) : 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 (AMinus (AId X) (AId X)) (ANum 0).
Proof.
  intros st. simpl. omega.
Qed.

Theorem bequiv_example:
  bequiv (BEq (AMinus (AId X) (AId X)) (ANum 0)) BTrue.
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 (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),
  (c1 / st \\ st') ↔ (c2 / st \\ st').
```

18.2.2 Simple Examples

For examples of command equivalence, let's start by looking at some trivial program transformations 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 (skip_right) Prove that adding a *SKIP* after a command results in an equivalent program

```
Theorem skip_right: ∀ c,
  cequiv
    (c ;; SKIP)
    c.
Proof.
   Admitted.
```

☐

Similarly, here is a simple transformation that optimizes *IFB* commands:

```
Theorem IFB_true_simple: ∀ c1 c2,
  cequiv
    (IFB BTrue THEN c1 ELSE c2 FI)
    c1.
Proof.
  intros c1 c2.
  split; intros H.
  -
```

```
    inversion H; subst. assumption. inversion H5.
  -
    apply E_IfTrue. reflexivity. assumption. Qed.
```

Of course, few programmers would be tempted to write a conditional whose guard is literally BTrue. A more interesting case is when the guard is *equivalent* to true: *Theorem*: If b is equivalent to BTrue, then *IFB* b *THEN* c1 *ELSE* c2 *FI* is equivalent to c1.

Proof:

- (→) We must show, for all *st* and *st'*, that if *IFB* b *THEN* c1 *ELSE* c2 *FI* / *st* \\ *st'* then c1 / *st* \\ *st'*.

  Proceed by cases on the rules that could possibly have been used to show *IFB* b *THEN* c1 *ELSE* c2 *FI* / *st* \\ *st'*, namely E_IfTrue and E_IfFalse.

  - Suppose the final rule rule in the derivation of *IFB* b *THEN* c1 *ELSE* c2 *FI* / *st* \\ *st'* was E_IfTrue. We then have, by the premises of E_IfTrue, that c1 / *st* \\ *st'*. This is exactly what we set out to prove.
  - On the other hand, suppose the final rule in the derivation of *IFB* b *THEN* c1 *ELSE* c2 *FI* / *st* \\ *st'* was E_IfFalse. We then know that beval *st* b = false and c2 / *st* \\ *st'*.

    Recall that b is equivalent to BTrue, i.e., forall *st*, beval *st* b = beval *st* BTrue. In particular, this means that beval *st* b = true, since beval *st* BTrue = true. But this is a contradiction, since E_IfFalse requires that beval *st* b = false. Thus, the final rule could not have been E_IfFalse.

- (←) We must show, for all *st* and *st'*, that if c1 / *st* \\ *st'* then *IFB* b *THEN* c1 *ELSE* c2 *FI* / *st* \\ *st'*.

  Since b is equivalent to BTrue, we know that beval *st* b = beval *st* BTrue = true. Together with the assumption that c1 / *st* \\ *st'*, we can apply E_IfTrue to derive *IFB* b *THEN* c1 *ELSE* c2 *FI* / *st* \\ *st'*. □

Here is the formal version of this proof:

```
Theorem IFB_true: ∀ b c1 c2,
     bequiv b BTrue →
     cequiv
       (IFB b THEN c1 ELSE c2 FI)
       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.
      inversion H5.
  -
    apply E_IfTrue; try assumption.
    unfold bequiv in Hb. simpl in Hb.
    rewrite Hb. reflexivity. Qed.
```

Exercise: 2 stars, recommended (IFB_false)

```
Theorem IFB_false: ∀ b c1 c2,
  bequiv b BFalse →
  cequiv
    (IFB b THEN c1 ELSE c2 FI)
    c2.
Proof.
   Admitted.
```

☐

Exercise: 3 stars (swap_if_branches) Show that we can swap the branches of an IF if we also negate its guard.

```
Theorem swap_if_branches: ∀ b e1 e2,
  cequiv
    (IFB b THEN e1 ELSE e2 FI)
    (IFB BNot b THEN e2 ELSE e1 FI).
Proof.
   Admitted.
```

☐

For *WHILE* loops, we can give a similar pair of theorems. A loop whose guard is equivalent to BFalse is equivalent to *SKIP*, while a loop whose guard is equivalent to BTrue is equivalent to *WHILE* BTrue *DO SKIP END* (or any other non-terminating program). The first of these facts is easy.

```
Theorem WHILE_false : ∀ b c,
  bequiv b BFalse →
  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. inversion H2.
  -
    inversion H; subst.
    apply E_WhileEnd.
    rewrite Hb.
    reflexivity. 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 BTrue never terminate.

Lemma: If b is equivalent to BTrue, then it cannot be the case that (*WHILE* b *DO* c *END*) / *st* \\ *st'*.

Proof: Suppose that (*WHILE* b *DO* c *END*) / *st* \\ *st'*. We show, by induction on a derivation of (*WHILE* b *DO* c *END*) / *st* \\ *st'*, that this assumption leads to a contradiction.

- Suppose (*WHILE* b *DO* c *END*) / *st* \\ *st'* is proved using rule E_WhileEnd. Then by assumption beval *st* b = false. But this contradicts the assumption that b is equivalent to BTrue.
- Suppose (*WHILE* b *DO* c *END*) / *st* \\ *st'* is proved using rule E_WhileLoop. Then we are given the induction hypothesis that (*WHILE* b *DO* c *END*) / *st* \\ *st'* is contradictory, which is exactly what we are trying to prove!
- Since these are the only rules that could have been used to prove (*WHILE* b *DO* c *END*) / *st* \\ *st'*, the other cases of the induction are immediately contradictory. □

```
Lemma WHILE_true_nonterm : ∀ b c st st',
  bequiv b BTrue →
  ~( (WHILE b DO c END) / st \\ st' ).
Proof.
  intros b c st st' Hb.
  intros H.
  remember (WHILE b DO c END) as cw eqn:Heqcw.
  induction H;

    inversion Heqcw; subst; clear Heqcw.
  -
    unfold bequiv in Hb.
    rewrite Hb in H. inversion H.
```

-

```
apply IHceval2. reflexivity. Qed.
```

Exercise: 2 stars, optional (WHILE_true_nonterm_informal) Explain what the lemma WHILE_true_nonterm means in English.

☐

Exercise: 2 stars, recommended (WHILE_true) Prove the following theorem. *Hint*: You'll want to use WHILE_true_nonterm here.

```
Theorem WHILE_true: ∀ b c,
  bequiv b BTrue →
  cequiv
    (WHILE b DO c END)
    (WHILE BTrue DO SKIP END).
Proof.
   Admitted.
```

☐

A more interesting fact about *WHILE* commands is that any finite number of copies of the body can be "unrolled" without changing meaning. Unrolling is a common transformation in real compilers.

```
Theorem loop_unrolling: ∀ b c,
  cequiv
    (WHILE b DO c END)
    (IFB b THEN (c ;; WHILE b DO c END) ELSE SKIP FI).
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_WhileLoop with (st' := st'0).
      assumption. assumption. assumption.
    +
      inversion H5; subst. apply E_WhileEnd. assumption. Qed.
```

Exercise: 2 stars, optional (seq_assoc) `Theorem` *seq_assoc* : ∀ *c1 c2 c3*,

```
  cequiv ((c1;;c2);;c3) (c1;;(c2;;c3)).
Proof.
  Admitted.
```

☐

Proving program properties involving assignments is one place where the Functional Extensionality axiom often comes in handy.

```
Theorem identity_assignment : ∀ (X:id),
  cequiv
    (X ::= AId X)
    SKIP.
Proof.
  intros. split; intro H.
    -
      inversion H; subst. simpl.
      replace (t_update st X (st X)) with st.
      + constructor.
      + apply functional_extensionality. intro.
        rewrite t_update_same; reflexivity.
    -
      replace st' with (t_update st' X (aeval st' (AId X))).
      + inversion H. subst. apply E_Ass. reflexivity.
      + apply functional_extensionality. intro.
        rewrite t_update_same. reflexivity.
Qed.
```

Exercise: 2 stars, recommended (assign_aequiv) `Theorem` *assign_aequiv* : ∀ *X e*,

```
  aequiv (AId X) e →
  cequiv SKIP (X ::= e).
Proof.
  Admitted.
```

☐

Exercise: 2 stars (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 BNot (BLe (AId X) (ANum 0)) DO
    X ::= APlus (AId X) (ANum 1)
```

```
  END.
Definition prog_b : com :=
  IFB BEq (AId X) (ANum 0) THEN
    X ::= APlus (AId X) (ANum 1);;
    Y ::= ANum 1
  ELSE
    Y ::= ANum 0
  FI;;
  X ::= AMinus (AId X) (AId Y);;
  Y ::= ANum 0.
Definition prog_c : com :=
  SKIP.
Definition prog_d : com :=
  WHILE BNot (BEq (AId X) (ANum 0)) DO
    X ::= APlus (AMult (AId X) (AId Y)) (ANum 1)
  END.
Definition prog_e : com :=
  Y ::= ANum 0.
Definition prog_f : com :=
  Y ::= APlus (AId X) (ANum 1);;
  WHILE BNot (BEq (AId X) (AId Y)) DO
    Y ::= APlus (AId X) (ANum 1)
  END.
Definition prog_g : com :=
  WHILE BTrue DO
    SKIP
  END.
Definition prog_h : com :=
  WHILE BNot (BEq (AId X) (AId X)) DO
    X ::= APlus (AId X) (ANum 1)
  END.
Definition prog_i : com :=
  WHILE BNot (BEq (AId X) (AId Y)) DO
    X ::= APlus (AId Y) (ANum 1)
  END.
Definition equiv_classes : list (list com)
  . Admitted.
```

□

18.3 Properties of Behavioral Equivalence

We next consider some fundamental properties of the program equivalence relations.

18.3.1 Behavioral Equivalence Is an Equivalence

First, we verify that the equivalences on *aexps*, *bexps*, and **com**s 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 : ∀ (b1 b2 : bexp),
  bequiv b1 b2 → bequiv b2 b1.
Proof.
  unfold bequiv. intros b1 b2 H. intros st. symmetry. apply H. Qed.

Lemma trans_bequiv : ∀ (b1 b2 b3 : bexp),
  bequiv b1 b2 → bequiv b2 b3 → bequiv b1 b3.
Proof.
  unfold bequiv. intros b1 b2 b3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.

Lemma refl_cequiv : ∀ (c : com), cequiv c c.
Proof.
  unfold cequiv. intros c st st'. apply iff_refl. Qed.

Lemma sym_cequiv : ∀ (c1 c2 : com),
  cequiv c1 c2 → cequiv c2 c1.
Proof.
  unfold cequiv. intros c1 c2 H st st'.
  assert (c1 / st \\ st' ↔ c2 / st \\ st') as H'.
```

```
  { apply H. }
  apply iff_sym. assumption.
Qed.

Lemma iff_trans : ∀ (P1 P2 P3 : Prop),
  (P1 ↔ P2) → (P2 ↔ P3) → (P1 ↔ P3).
Proof.
  intros P1 P2 P3 H12 H23.
  inversion H12. inversion H23.
  split; intros A.
    apply H1. apply H. apply A.
    apply H0. apply H2. apply A. Qed.

Lemma trans_cequiv : ∀ (c1 c2 c3 : com),
  cequiv c1 c2 → cequiv c2 c3 → cequiv c1 c3.
Proof.
  unfold cequiv. intros c1 c2 c3 H12 H23 st st'.
  apply iff_trans with (c2 / st \\ st'). apply H12. apply H23. Qed.
```

18.3.2 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 a1 a1'

---

cequiv (i ::= a1) (i ::= a1')

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 a 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 non-varying parts – 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 CAss_congruence : ∀ i a1 a1',
  aequiv a1 a1' →
  cequiv (CAss i a1) (CAss i a1').
Proof.
  intros i a1 a2 Heqv st st'.
```

```
  split; intros Hceval.
  -
    inversion Hceval. subst. apply E_Ass.
    rewrite Heqv. reflexivity.
  -
    inversion Hceval. subst. apply E_Ass.
    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 *b1* is equivalent to *b1'* and c1 is equivalent to *c1'*, then *WHILE b1 DO* c1 *END* is equivalent to *WHILE b1' DO c1' END*.

Proof: Suppose *b1* is equivalent to *b1'* and c1 is equivalent to *c1'*. We must show, for every *st* and *st'*, that *WHILE b1 DO* c1 *END / st \\ st'* iff *WHILE b1' DO c1' END / st \\ st'*. We consider the two directions separately.

- (→) We show that *WHILE b1 DO* c1 *END / st \\ st'* implies *WHILE b1' DO c1' END / st \\ st'*, by induction on a derivation of *WHILE b1 DO* c1 *END / st \\ st'*. The only nontrivial cases are when the final rule in the derivation is E_WhileEnd or E_WhileLoop.

  - E_WhileEnd: In this case, the form of the rule gives us beval *st b1* = false and *st* = *st'*. But then, since *b1* and *b1'* are equivalent, we have beval *st b1'* = false, and *E-WhileEnd* applies, giving us *WHILE b1' DO c1' END / st \\ st'*, as required.
  - E_WhileLoop: The form of the rule now gives us beval *st b1* = true, with c1 / *st* \\ *st'0* and *WHILE b1 DO* c1 *END / st'0 \\ st'* for some state *st'0*, with the induction hypothesis *WHILE b1' DO c1' END / st'0 \\ st'*.

    Since c1 and *c1'* are equivalent, we know that *c1' / st \\ st'0*. And since *b1* and *b1'* are equivalent, we have beval *st b1'* = true. Now *E-WhileLoop* applies, giving us *WHILE b1' DO c1' END / st \\ st'*, as required.

- (←) Similar. □

```
Theorem CWhile_congruence : ∀ b1 b1' c1 c1',
  bequiv b1 b1' → cequiv c1 c1' →
  cequiv (WHILE b1 DO c1 END) (WHILE b1' DO c1' END).
Proof.
  unfold bequiv,cequiv.
  intros b1 b1' c1 c1' Hb1e Hc1e st st'.
  split; intros Hce.
  -
    remember (WHILE b1 DO c1 END) as cwhile
      eqn:Heqcwhile.
```

```
    induction Hce; inversion Heqcwhile; subst.
    +
      apply E_WhileEnd. rewrite ← Hb1e. apply H.
    +
      apply E_WhileLoop with (st' := st').
      × rewrite ← Hb1e. apply H.
      ×
        apply (Hc1e st st'). apply Hce1.
      ×
        apply IHHce2. reflexivity.
  -
    remember (WHILE b1' DO c1' END) as c'while
      eqn:Heqc'while.
    induction Hce; inversion Heqc'while; subst.
    +
      apply E_WhileEnd. rewrite → Hb1e. apply H.
    +
      apply E_WhileLoop with (st' := st').
      × rewrite → Hb1e. apply H.
      ×
        apply (Hc1e st st'). apply Hce1.
      ×
        apply IHHce2. reflexivity. Qed.
```

Exercise: 3 stars, optional (CSeq_congruence) `Theorem` *CSeq_congruence* : ∀ *c1 c1' c2 c2'*,

```
  cequiv c1 c1' → cequiv c2 c2' →
  cequiv (c1;;c2) (c1';;c2').
Proof.
   Admitted.
```

☐

Exercise: 3 stars (CIf_congruence) `Theorem` *CIf_congruence* : ∀ *b b' c1 c1' c2 c2'*,

```
  bequiv b b' → cequiv c1 c1' → cequiv c2 c2' →
  cequiv (IFB b THEN c1 ELSE c2 FI)
         (IFB b' THEN c1' ELSE c2' FI).
Proof.
   Admitted.
```

☐

For example, here are two equivalent programs and a proof of their equivalence...

```
Example congruence_example:
  cequiv
```

```
    (X ::= ANum 0;;
     IFB (BEq (AId X) (ANum 0))
     THEN
       Y ::= ANum 0
     ELSE
       Y ::= ANum 42
     FI)

    (X ::= ANum 0;;
     IFB (BEq (AId X) (ANum 0))
     THEN
       Y ::= AMinus (AId X) (AId X)
     ELSE
       Y ::= ANum 42
     FI).
Proof.
  apply CSeq_congruence.
    apply refl_cequiv.
    apply CIf_congruence.
      apply refl_bequiv.
      apply CAss_congruence. unfold aequiv. simpl.
        symmetry. apply minus_diag.
      apply refl_cequiv.
Qed.
```

Exercise: 3 stars, advanced, optional (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?

☐

18.4 Program Transformations

A *program transformation* is a function that takes a program as input and produces some variant of the program as output. Compiler optimizations such as constant folding are a canonical example, 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).
```

18.4.1 The Constant-Folding Transformation

An expression is *constant* when 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
  | AId i ⇒ AId i
  | APlus a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
    | (a1', a2') ⇒ APlus a1' a2'
    end
  | AMinus a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
    | (a1', a2') ⇒ AMinus a1' a2'
    end
  | AMult a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 × n2)
    | (a1', a2') ⇒ AMult a1' a2'
    end
  end.
Example fold_aexp_ex1 :
    fold_constants_aexp
      (AMult (APlus (ANum 1) (ANum 2)) (AId X))
  = AMult (ANum 3) (AId X).
Proof. reflexivity. Qed.
```

Note that this version of constant folding doesn't eliminate trivial additions, etc. – we are focusing attention 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.

```
Example fold_aexp_ex2 :
    fold_constants_aexp
      (AMinus (AId X) (APlus (AMult (ANum 0) (ANum 6))
                             (AId Y)))
  = AMinus (AId X) (APlus (ANum 0) (AId Y)).
Proof. reflexivity. Qed.
```

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

```
Fixpoint fold_constants_bexp (b : bexp) : bexp :=
  match b with
  | BTrue ⇒ BTrue
  | BFalse ⇒ BFalse
  | BEq a1 a2 ⇒
      match (fold_constants_aexp a1, fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if beq_nat n1 n2 then BTrue else BFalse
      | (a1', a2') ⇒
          BEq a1' a2'
      end
  | BLe a1 a2 ⇒
      match (fold_constants_aexp a1, fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if leb n1 n2 then BTrue else BFalse
      | (a1', a2') ⇒
          BLe a1' a2'
      end
  | BNot b1 ⇒
      match (fold_constants_bexp b1) with
      | BTrue ⇒ BFalse
      | BFalse ⇒ BTrue
      | b1' ⇒ BNot b1'
      end
  | BAnd b1 b2 ⇒
      match (fold_constants_bexp b1, fold_constants_bexp b2) with
      | (BTrue, BTrue) ⇒ BTrue
      | (BTrue, BFalse) ⇒ BFalse
      | (BFalse, BTrue) ⇒ BFalse
      | (BFalse, BFalse) ⇒ BFalse
      | (b1', b2') ⇒ BAnd b1' b2'
      end
  end.

Example fold_bexp_ex1 :
```

```
    fold_constants_bexp (BAnd BTrue (BNot (BAnd BFalse BTrue)))
  = BTrue.
Proof. reflexivity. Qed.

Example fold_bexp_ex2 :
    fold_constants_bexp
      (BAnd (BEq (AId X) (AId Y))
            (BEq (ANum 0)
                 (AMinus (ANum 2) (APlus (ANum 1)
                                          (ANum 1)))))
  = BAnd (BEq (AId X) (AId Y)) BTrue.
Proof. reflexivity. Qed.
```

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

```
Fixpoint fold_constants_com (c : com) : com :=
  match c with
  | SKIP ⇒
      SKIP
  | i ::= a ⇒
      CAss i (fold_constants_aexp a)
  | c1 ;; c2 ⇒
      (fold_constants_com c1) ;; (fold_constants_com c2)
  | IFB b THEN c1 ELSE c2 FI ⇒
      match fold_constants_bexp b with
      | BTrue ⇒ fold_constants_com c1
      | BFalse ⇒ fold_constants_com c2
      | b' ⇒ IFB b' THEN fold_constants_com c1
                    ELSE fold_constants_com c2 FI
      end
  | WHILE b DO c END ⇒
      match fold_constants_bexp b with
      | BTrue ⇒ WHILE BTrue DO SKIP END
      | BFalse ⇒ SKIP
      | b' ⇒ WHILE b' DO (fold_constants_com c) END
      end
  end.

Example fold_com_ex1 :
  fold_constants_com

    (X ::= APlus (ANum 4) (ANum 5);;
     Y ::= AMinus (AId X) (ANum 3);;
     IFB BEq (AMinus (AId X) (AId Y))
```

```
            (APlus (ANum 2) (ANum 4)) THEN
      SKIP
    ELSE
      Y ::= ANum 0
    FI;;
    IFB BLe (ANum 0)
            (AMinus (ANum 4) (APlus (ANum 2) (ANum 1)))
    THEN
      Y ::= ANum 0
    ELSE
      SKIP
    FI;;
    WHILE BEq (AId Y) (ANum 0) DO
      X ::= APlus (AId X) (ANum 1)
    END)
  =
   (X ::= ANum 9;;
    Y ::= AMinus (AId X) (ANum 3);;
    IFB BEq (AMinus (AId X) (AId Y)) (ANum 6) THEN
      SKIP
    ELSE
      (Y ::= ANum 0)
    FI;;
    Y ::= ANum 0;;
    WHILE BEq (AId Y) (ANum 0) DO
      X ::= APlus (AId X) (ANum 1)
    END).
Proof. reflexivity. Qed.
```

18.4.2 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, 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, for all boolean expressions b. Proceed by induction on b. We show just the case where b has the form BEq *a1* *a2*.

In this case, we must show
beval st (BEq a1 a2) = beval st (fold_constants_bexp (BEq a1 a2)).
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

  fold_constants_bexp (BEq a1 a2) = if beq_nat n1 n2 then BTrue else BFalse

  and

  beval st (BEq a1 a2) = beq_nat (aeval st a1) (aeval st a2).

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

  aeval st a1 = aeval st (fold_constants_aexp a1) = aeval st (ANum n1) = n1

  and

  aeval st a2 = aeval st (fold_constants_aexp a2) = aeval st (ANum n2) = n2,

  so

  beval st (BEq a1 a2) = beq_nat (aeval a1) (aeval a2) = beq_nat n1 n2.

  Also, it is easy to see (by considering the cases *n1* = *n2* and *n1* ≠ *n2* separately) that

  beval st (if beq_nat n1 n2 then BTrue else BFalse) = if beq_nat n1 n2 then beval st BTrue else beval st BFalse = if beq_nat n1 n2 then true else false = beq_nat n1 n2.

  So

  beval st (BEq a1 a2) = beq_nat n1 n2. = beval st (if beq_nat n1 n2 then BTrue else BFalse),

  as required.

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

  beval st (BEq a1 a2) = beval st (BEq (fold_constants_aexp a1) (fold_constants_aexp a2)),

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

  beq_nat (aeval st a1) (aeval st a2) = beq_nat (aeval st (fold_constants_aexp a1)) (aeval st (fold_constants_aexp a2)).

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

  aeval st a1 = aeval st (fold_constants_aexp a1) aeval st a2 = aeval st (fold_constants_aexp a2),

  completing the case. □

```
Theorem fold_constants_bexp_sound:
  btrans_sound fold_constants_bexp.
Proof.
  unfold btrans_sound. intros b. unfold bequiv. intros st.
  induction b;

    try reflexivity.
  -
    rename a into a1. rename a0 into a2. simpl.
```

(Doing induction when there are a lot of constructors makes specifying variable names a chore, but Coq doesn't always choose nice variable names. We can rename entries in the context with the `rename` tactic: `rename a` *into a1* will change `a` to *a1* in the current goal and context.)

```
    remember (fold_constants_aexp a1) as a1' eqn:Heqa1'.
    remember (fold_constants_aexp a2) as a2' eqn:Heqa2'.
    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 (beq_nat n n0); reflexivity.
  -
    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 (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 CAss_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 IFB_true; assumption.
    +
      apply trans_cequiv with c2; try assumption.
      apply IFB_false; assumption.
  -
    Admitted.
```

☐

Soundness of (0 + n) Elimination, Redux

Exercise: 4 stars, advanced, optional (optimize_0plus) Recall the definition optimize_0plus from the Imp chapter:

Fixpoint optimize_0plus (e:aexp) : aexp := match e 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.

Note that this function is defined over the old **aexp**s, without states.

Write a new version of this function that accounts for variables, plus analogous ones for **bexp**s and commands:

optimize_0plus_aexp optimize_0plus_bexp optimize_0plus_com

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*!

Then define an optimizer on commands that first folds constants (using fold_constants_com) and then eliminates 0 + n terms (using *optimize_0plus_com*).

- Give a meaningful example of this optimizer's output.

- Prove that the optimizer is sound. (This part should be *very* easy.)

□

18.5 Proving That Programs 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 for each occurrence of a given variable in another expression:

```
Fixpoint subst_aexp (i : id) (u : aexp) (a : aexp) : aexp :=
  match a with
  | ANum n ⇒
      ANum n
  | AId i' ⇒
      if beq_id i i' then u else AId i'
  | APlus a1 a2 ⇒
      APlus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMinus a1 a2 ⇒
      AMinus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMult a1 a2 ⇒
      AMult (subst_aexp i u a1) (subst_aexp i u a2)
  end.
```

```
Example subst_aexp_ex :
  subst_aexp X (APlus (ANum 42) (ANum 53))
```

```
                (APlus (AId Y) (AId X))
= (APlus (AId Y) (APlus (ANum 42) (ANum 53))).
Proof. 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 := ∀ i1 i2 a1 a2,
  cequiv (i1 ::= a1;; i2 ::= a2)
         (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2).
```

Sadly, the property does *not* always hold – i.e., it is not the case that, for all i1, i2, *a1*, and *a2*,

cequiv (i1 ::= a1;; i2 ::= a2) (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2).

To see this, suppose (for a contradiction) that for all i1, i2, *a1*, and *a2*, we have

cequiv (i1 ::= a1;; i2 ::= a2) (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2).

Consider the following program:

X ::= APlus (AId X) (ANum 1);; Y ::= AId X

Note that

(X ::= APlus (AId X) (ANum 1);; Y ::= AId X) / empty_state \\ st1,

where *st1* = { X |-> 1, Y |-> 1 }.

By assumption, we know that

cequiv (X ::= APlus (AId X) (ANum 1);; Y ::= AId X) (X ::= APlus (AId X) (ANum 1);; Y ::= APlus (AId X) (ANum 1))

so, by the definition of cequiv, we have

(X ::= APlus (AId X) (ANum 1);; Y ::= APlus (AId X) (ANum 1)) / empty_state \\ st1.

But we can also derive

(X ::= APlus (AId X) (ANum 1);; Y ::= APlus (AId X) (ANum 1)) / empty_state \\ st2,

where *st2* = { X |-> 1, Y |-> 2 }. But *st1* ≠ *st2*, which is a contradiction, since **ceval** is deterministic! □

```
Theorem subst_inequiv :
  ¬ subst_equiv_property.
Proof.
  unfold subst_equiv_property.
  intros Contra.

  remember (X ::= APlus (AId X) (ANum 1);;
            Y ::= AId X)
      as c1.
  remember (X ::= APlus (AId X) (ANum 1);;
            Y ::= APlus (AId X) (ANum 1))
      as c2.
  assert (cequiv c1 c2) by (subst; apply Contra).
```

```
remember (t_update (t_update empty_state X 1) Y 1) as st1.
remember (t_update (t_update empty_state X 1) Y 2) as st2.
assert (H1: c1 / empty_state \\ st1);
assert (H2: c2 / empty_state \\ st2);
try (subst;
     apply E_Seq with (st' := (t_update empty_state X 1));
     apply E_Ass; reflexivity).
apply H in H1.

assert (Hcontra: st1 = st2)
  by (apply (ceval_deterministic c2 empty_state); assumption).
assert (Hcontra': st1 Y = st2 Y)
  by (rewrite Hcontra; reflexivity).
subst. inversion Hcontra'. Qed.
```

Exercise: 4 stars, optional (better_subst_equiv) The equivalence we had in mind above was not complete nonsense – 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:id) : aexp → Prop :=
  | VNUNum: ∀ n, var_not_used_in_aexp X (ANum n)
  | VNUId: ∀ Y, X ≠ Y → var_not_used_in_aexp X (AId Y)
  | VNUPlus: ∀ a1 a2,
      var_not_used_in_aexp X a1 →
      var_not_used_in_aexp X a2 →
      var_not_used_in_aexp X (APlus 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 (AMinus 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 (AMult a1 a2).

Lemma aeval_weakening : ∀ i st a ni,
  var_not_used_in_aexp i a →
  aeval (t_update st i ni) a = aeval st a.
Proof.
   Admitted.
```

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

☐

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

```
Theorem inequiv_exercise:
  ¬ cequiv (WHILE BTrue DO SKIP END) SKIP.
Proof.
  Admitted.
  □
```

18.6 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 the compiler writer 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 unitialized 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
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
```

```
  | CWhile : bexp → com → com
  | CHavoc : id → com.
Notation "'SKIP'" :=
  CSkip.
Notation "X '::=' a" :=
  (CAss X a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'HAVOC' l" := (CHavoc l) (at level 60).
```

Exercise: 2 stars (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 "c1 '/' st '\\' st'"
                  (at level 40, st at level 39).
Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st : state, SKIP / st \\ st
  | E_Ass : ∀ (st : state) (a1 : aexp) (n : nat) (X : id),
      aeval st a1 = n →
      (X ::= a1) / st \\ t_update st X n
  | E_Seq : ∀ (c1 c2 : com) (st st' st'' : state),
      c1 / st \\ st' →
      c2 / st' \\ st'' →
      (c1 ;; c2) / st \\ st''
  | E_IfTrue : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
      beval st b1 = true →
      c1 / st \\ st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
      beval st b1 = false →
      c2 / st \\ st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileEnd : ∀ (b1 : bexp) (st : state) (c1 : com),
      beval st b1 = false →
      (WHILE b1 DO c1 END) / st \\ st
  | E_WhileLoop : ∀ (st st' st'' : state) (b1 : bexp) (c1 : com),
      beval st b1 = true →
```

    c1 / st \\ st' →
    (WHILE b1 DO c1 END) / st' \\ st'' →
    (WHILE b1 DO c1 END) / st \\ st''

where "c1 '/' st '\\' st'" := (ceval c1 st st').

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

```
Example havoc_example1 : (HAVOC X) / empty_state \\ t_update empty_state X 0.
Proof.
  Admitted.

Example havoc_example2 :
  (SKIP;; HAVOC Z) / empty_state \\ t_update empty_state Z 42.
Proof.
  Admitted.
```

☐

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

```
Definition cequiv (c1 c2 : com) : Prop := ∀ st st' : state,
  c1 / st \\ st' ↔ c2 / st \\ st'.
```

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

Exercise: 3 stars (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, optional (havoc_copy) Are the following two programs equivalent?

```
Definition ptwice :=
  HAVOC X;; HAVOC Y.

Definition pcopy :=
  HAVOC X;; Y ::= AId 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 $ptwice\_cequiv\_pcopy$:
  $cequiv\ ptwice\ pcopy \lor \neg cequiv\ ptwice\ pcopy$.
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 (BNot (BEq (AId X) (ANum 0))) DO
    HAVOC Y;;
    X ::= APlus (AId X) (ANum 1)
  END.
```

```
Definition p2 : com :=
  WHILE (BNot (BEq (AId X) (ANum 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 $p1\_may\_diverge$: $\forall\ st\ st',\ st\ X \neq 0 \rightarrow$
  $\neg\ p1\ /\ st\ \backslash\backslash\ st'$.
Proof. *Admitted.*

Lemma $p2\_may\_diverge$: $\forall\ st\ st',\ st\ X \neq 0 \rightarrow$
  $\neg\ p2\ /\ st\ \backslash\backslash\ st'$.
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, advancedM (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 ::= ANum 1;;
  WHILE (BNot (BEq (AId X) (ANum 0))) DO
    HAVOC X;;
    HAVOC Z
  END.
```

```
Definition p4 : com :=
  X ::= (ANum 0);;
  Z ::= (ANum 1).
```

```
Theorem p3_p4_inequiv : ¬ 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 when given a same starting state *st*, the set of possible terminating states is the same for both programs. If p5 terminates, what should the final state be? Conversely, is it always possible to make p5 terminate?)

```
Definition p5 : com :=
  WHILE (BNot (BEq (AId X) (ANum 1))) DO
    HAVOC X
  END.
```

```
Definition p6 : com :=
  X ::= ANum 1.
```

```
Theorem p5_p6_equiv : cequiv p5 p6.
Proof. Admitted.
```

□

```
End Himp.
```

18.7 Additional Exercises

Exercise: 4 stars, 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: 3 stars, optional (swap_noninterfering_assignments) (Hint: You'll need *functional_extensionality* for this one.)

```
Theorem swap_noninterfering_assignments: ∀ l1 l2 a1 a2,
  l1 ≠ l2 →
  var_not_used_in_aexp l1 a2 →
  var_not_used_in_aexp l2 a1 →
  cequiv
    (l1 ::= a1;; l2 ::= a2)
    (l2 ::= a2;; l1 ::= a1).
Proof.
    Admitted.
```
□

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 : com) : Prop := ∀ (st st' : state),
  c1 / st \\ st' → c2 / st \\ 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 : ¬ capprox c3 c4 ∧ ¬ capprox c4 c3.
Proof. Admitted.
```

Find a program cmin that approximates every other program.

```
Definition cmin : com
  . Admitted.

Theorem cmin_minimal : ∀ c, 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* : ∀ *c* *c'*,
zprop *c* → *capprox* *c* *c'* → *zprop* *c'*.

Proof. *Admitted.*

□

$Date : 2016 - 12 - 2010 : 47 : 46 - 0500 (Tue, 20 Dec 2016)$

Chapter 19

Library Top.Hoare

19.1 Hoare: Hoare Logic, Part I

`Require Import` *Coq.Bool.Bool.*
`Require Import` *Coq.Arith.Arith.*
`Require Import` *Coq.Arith.EqNat.*
`Require Import` *Coq.omega.Omega.*
`Require Import` *Imp.*
`Require Import` *Maps.*

In the past couple of chapters, we've begun 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 the book when we discuss *types* and *type soundness*. In this chapter, though, we turn to a different set of issues.

Our goal is to carry out 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 *compositional proof technique* for proving that programs are correct with respect to such specifications – where by "compositional" we mean that the structure of proofs directly mirrors the structure of the programs that they are about.

This chapter:

- A systematic method for reasoning about the correctness of particular programs in Imp

Goals:

- a natural notation for *program specifications* and
- a *compositional* proof technique for program correctness

Plan:

- assertions (Hoare Triples)
- proof rules
- decorated programs
- loop invariants
- examples

19.2 Assertions

To talk about specifications of programs, the first thing we need is a way of making *assertions* about properties that hold at particular points during a program's execution – i.e., claims about the current state of the memory when execution reaches that point. Formally, an assertion is just a family of propositions indexed by a state.

Definition $Assertion := state \to$ Prop.

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

```
Module ExAssertions.
Definition as1 : Assertion := fun st ⇒ st X = 3.
Definition as2 : Assertion := fun st ⇒ st X ≤ st Y.
Definition as3 : Assertion :=
  fun st ⇒ st X = 3 ∨ st X ≤ st Y.
Definition as4 : Assertion :=
  fun st ⇒ st Z × st Z ≤ st X ∧
            ¬ (((S (st Z)) × (S (st Z))) ≤ st X).
Definition as5 : Assertion := fun st ⇒ True.
Definition as6 : Assertion := fun st ⇒ False.
End ExAssertions.
```

☐

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 $st \Rightarrow$; and (2) this state st is the only one that we ever use to look up variables in assertions (we will 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 $st \Rightarrow$, and we'll write just X to mean st X. Thus, instead of writing

fun st => (st Z) * (st Z) <= m /\ ~ ((S (st Z)) * (S (st Z)) <= m)

we'll write just

Z * Z <= m /\ ~((S Z) * (S Z) <= m).

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 Coq variables (of type nat). This is why, when translating from informal to formal, we replace X with st X but leave m alone.

Given two assertions P and Q, we say that P *implies* Q, written P -» Q (in ASCII, P -» Q), if, whenever P holds in some state st, Q also holds.

```
Definition assert_implies (P Q : Assertion) : Prop :=
  ∀ st, P st → Q st.

Notation "P -» Q" := (assert_implies P Q)
                    (at level 80) : hoare_spec_scope.
```

```
Open Scope hoare_spec_scope.
```

(The *hoare_spec_scope* annotation here tells Coq that this notation is not global but is intended to be used in particular contexts. The `Open Scope` tells Coq that this file is one such context.)

We'll also want the "iff" variant of implication between assertions:

```
Notation "P «-» Q" :=
  (P -» Q ∧ Q -» P) (at level 80) : hoare_spec_scope.
```

19.3 Hoare Triples

Next, we need a way of making formal claims about the behavior of commands.

In general, the behavior of a command is to transform one state to another, so it is natural to express claims about commands in terms of assertions that are true before and after the command executes:

- "If command `c` is started in a state satisfying assertion *P*, and if `c` eventually terminates in some final state, then this final state will satisfy the assertion *Q*."

Such a claim is called a *Hoare Triple*. The property *P* is called the *precondition* of `c`, while *Q* is the *postcondition*. Formally:

```
Definition hoare_triple
           (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st',
     c / st \\ st' →
     P st →
     Q st'.
```

Since we'll be working a lot with Hoare triples, it's useful to have a compact notation:

[1] c [2]. (The traditional notation is $\{P\}$ `c` $\{Q\}$, but single braces are already used for other things in Coq.)

```
Notation "{{ P }} c {{ Q }}" :=
  (hoare_triple P c Q) (at level 90, c at next level)
  : hoare_spec_scope.
```

Exercise: 1 star, optional (triples) Paraphrase the following Hoare triples in English.

1) [3] c [4]

2) [5] c [6]

[1] `P`
[2] `Q`
[3] `True`
[4] `X=5`
[5] `X=m`
[6] `X=m+5)`

3) [7] c [8]
4) [9] c [10]
5) [11] c [12]
6) [13] c [14]
□

Exercise: 1 star, optional (valid_triples) Which of the following Hoare triples are *valid* – i.e., the claimed relation between *P*, c, and *Q* is true?

1) [15] X ::= 5 [16]
2) [17] X ::= X + 1 [18]
3) [19] X ::= 5; Y ::= 0 [20]
4) [21] X ::= 5 [22]
5) [23] SKIP [24]
6) [25] SKIP [26]
7) [27] WHILE True DO SKIP END [28]
8) [29] WHILE X == 0 DO X ::= X + 1 END [30]
9) [31] WHILE X <> 0 DO X ::= X + 1 END [32] □

(Note that we're using informal mathematical notations for expressions inside of commands, for readability, rather than their formal **aexp** and **bexp** encodings. We'll continue doing so throughout the chapter.)

---

[7] `X<=Y`
[8] `Y<=X`
[9] `True`
[10] `False`
[11] `X=m`
[12] `Y=real_factm`
[13] `True`
[14] `(Z*Z)<=m/\~(((SZ)*(SZ))<=m)`
[15] `True`
[16] `X=5`
[17] `X=2`
[18] `X=3`
[19] `True`
[20] `X=5`
[21] `X=2/\X=3`
[22] `X=0`
[23] `True`
[24] `False`
[25] `False`
[26] `True`
[27] `True`
[28] `False`
[29] `X=0`
[30] `X=1`
[31] `X=1`
[32] `X=100`

To get us warmed up for what's coming, here are two simple facts about Hoare triples.

```
Theorem hoare_post_true : ∀ (P Q : Assertion) c,
  (∀ st, Q st) →
  {{P}} c {{Q}}.
Proof.
  intros P Q c H. unfold hoare_triple.
  intros st st' Heval HP.
  apply H. Qed.

Theorem hoare_pre_false : ∀ (P Q : Assertion) c,
  (∀ st, ~(P st)) →
  {{P}} c {{Q}}.
Proof.
  intros P Q c H. unfold hoare_triple.
  intros st st' Heval HP.
  unfold not in H. apply H in HP.
  inversion HP. Qed.
```

19.4 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 hoare_triple.

19.4.1 Assignment

The rule for assignment is the most fundamental of the Hoare logic proof rules. Here's how it works.

Consider this valid Hoare triple:

[33] X ::= Y [34]

In English: if we start out in a state where the value of Y is 1 and we assign Y to X, then we'll finish in a state where X is 1. That is, the property of being equal to 1 gets transferred from Y to X.

Similarly, in

[35] X ::= Y + Z [36]

[33] Y=1

[34] X=1

[35] Y+Z=1

[36] X=1

the same property (being equal to one) gets transferred to X from the expression Y + Z on the right-hand side of the assignment.

More generally, if a is *any* arithmetic expression, then

[37] X ::= a [38]

is a valid Hoare triple.

This can be made even more general. To conclude that an arbitrary property Q holds after X ::= a, we need to assume that Q holds before X ::= a, but *with all occurrences of* X replaced by a in Q. This leads to the Hoare rule for assignment

[39] X ::= a [40]

where "Q [X |-> a]" is pronounced "Q where a is substituted for X".

For example, these are valid applications of the assignment rule:

[41] X ::= X + 1 [42]

[43] X ::= 3 [44]

[45] X ::= 3 [46]

To formalize the rule, we must first formalize the idea of "substituting an expression for an Imp variable in an assertion." That is, 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, wherever P mentions X, *P'* should instead mention a.

Since P is an arbitrary Coq proposition, we can't directly "edit" its text. Instead, we can achieve the same effect by evaluating P in an updated state:

```
Definition assn_sub X a P : Assertion :=
  fun (st : state) ⇒
    P (t_update st X (aeval st a)).

Notation "P [ X |-> a ]" := (assn_sub X a P) (at level 10).
```

That is, P [X |-> a] stands for an assertion – let's call it *P'* – that is 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, let's calculate what happens with a couple of examples. First, suppose *P'* is (X ≤ 5) [X |-> 3] – that is, more formally, *P'* is the Coq expression

fun st => (fun st' => st' X <= 5) (t_update st X (aeval st (ANum 3))),

which simplifies to

fun st => (fun st' => st' X <= 5) (t_update st X 3)

and further simplifies to

---

[37] a=1

[38] X=1

[39] Q[X|->a]

[40] Q

[41] (X<=5)[X|->X+1]i.e.,X+1<=5

[42] X<=5

[43] (X=3)[X|->3]i.e.,3=3

[44] X=3

[45] (0<=X/\X<=5)[X|->3]i.e.,(0<=3/\3<=5)

[46] 0<=X/\X<=5

fun st => ((t_update st X 3) 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 |-> X+1]. Formally, *P'* is the Coq expression

fun st => (fun st' => st' X <= 5) (t_update st X (aeval st (APlus (AId X) (ANum 1)))),

which simplifies to

fun st => (((t_update st X (aeval st (APlus (AId X) (ANum 1))))) X) <= 5

and further simplifies to

fun st => (aeval st (APlus (AId X) (ANum 1))) <= 5.

That is, *P'* is the assertion that X+1 is at most 5.

Now, using the concept of substitution, we can give the precise proof rule for assignment:

(hoare_asgn) [47] X ::= a [48]

We can prove formally that this rule is indeed valid.

```
Theorem hoare_asgn : ∀ Q X a,
  {{Q [X |-> a]}} (X ::= a) {{Q}}.
Proof.
  unfold hoare_triple.
  intros Q X a st st' HE HQ.
  inversion HE. subst.
  unfold assn_sub in HQ. assumption. Qed.
```

Here's a first formal proof using this rule.

```
Example assn_sub_example :
  {{(fun st ⇒ st X = 3) [X |-> ANum 3]}}
  (X ::= (ANum 3))
  {{fun st ⇒ st X = 3}}.
Proof.
  apply hoare_asgn. Qed.
```

Exercise: 2 starsM (hoare_asgn_examples) Translate these informal Hoare triples...

1) [49] X ::= X + 1 [50]

2) [51] X ::= 3 [52]

---

[47] `Q[X|->a]`

[48] `Q`

[49] `(X<=5)[X|->X+1]`

[50] `X<=5`

[51] `(0<=X/\X<=5)[X|->3]`

[52] `0<=X/\X<=5`

...into formal statements (use the names *assn_sub_ex1* and *assn_sub_ex2*) and use hoare_asgn to prove them.

☐

Exercise: 2 stars, recommendedM (hoare_asgn_wrong) The assignment rule looks backward to almost everyone the first time they see it. If it still seems puzzling, it may help to think a little about alternative "forward" rules. Here is a seemingly natural one:

(hoare_asgn_wrong) [53] X ::= a [54]

Give a counterexample showing that this rule is incorrect and argue informally that it is really a counterexample. (Hint: The rule universally quantifies over the arithmetic expression a, and your counterexample needs to exhibit an a for which the rule doesn't work.)

☐

Exercise: 3 stars, advanced (hoare_asgn_fwd) However, by using a *parameter* m (a Coq number) to remember the original value of X we can define a Hoare rule for assignment that does, intuitively, "work forwards" rather than backwards.

(hoare_asgn_fwd) [55] X ::= a [56] (where st' = t_update st X m)

Note that we use the original value of X to reconstruct the state *st'* before the assignment took place. Prove that this rule is correct. (Also note that this rule is more complicated than hoare_asgn.)

```
Theorem hoare_asgn_fwd :
  (∀ {X Y: Type} {f g : X → Y},
     (∀ (x: X), f x = g x) → f = g) →
  ∀ m a P,
  {{fun st ⇒ P st ∧ st X = m}}
    X ::= a
  {{fun st ⇒ P (t_update st X m)
             ∧ st X = aeval (t_update st X m) a }}.
Proof.
  Admitted.
```

☐

Exercise: 2 stars, advanced (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.

---

[53] True
[54] X=a
[55] funst=>Pst/\stX=m
[56] funst=>Pst'/\stX=aevalst'a

(hoare_asgn_fwd_exists) [57] X ::= a [58]

```
Theorem hoare_asgn_fwd_exists :
  (∀ {X Y: Type} {f g : X → Y},
     (∀ (x: X), f x = g x) → f = g) →
  ∀ a P,
  {{fun st ⇒ P st}}
    X ::= a
  {{fun st ⇒ ∃ m, P (t_update st X m) ∧
                    st X = aeval (t_update st X m) a }}.
Proof.
  intros functional_extensionality a P.
   Admitted.
   □
```

19.4.2 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, or they actually may be logically weaker (for preconditions) or stronger (for postconditions) than what we need.

For instance, while

[59] X ::= 3 [60],

follows directly from the assignment rule,

[61] X ::= 3 [62]

does not. This triple is valid, but it is not an instance of hoare_asgn because **True** and (X = 3) [X |-> 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:

[63] c [64] P - P'

(hoare_consequence_pre_equiv) [65] c [66]

[57] `funst=>Pst`
[58] `funst=>existsm,P(t_updatestXm)/\stX=aeval(t_updatestXm)a`
[59] `(X=3)[X|->3]`
[60] `X=3`
[61] `True`
[62] `X=3`
[63] `P'`
[64] `Q`
[65] `P`
[66] `Q`

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*.

[67] c [68] P -» P'

(hoare_consequence_pre) [69] c [70]

[71] c [72] Q' -» Q

(hoare_consequence_post) [73] c [74]

Here are the formal versions:

```
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{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 : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' -» Q →
  {{P}} c {{Q}}.
Proof.
  intros P Q Q' c Hhoare Himp.
  intros st st' Hc HP.
  apply Himp.
  apply (Hhoare st st').
  assumption. assumption. Qed.
```

For example, we can use the first consequence rule like this:

[75] -» [76] X ::= 1 [77]

Or, formally...

```
Example hoare_asgn_example1 :
```

---

[67] `P'`
[68] `Q`
[69] `P`
[70] `Q`
[71] `P`
[72] `Q'`
[73] `P`
[74] `Q`
[75] `True`
[76] `1=1`
[77] `X=1`

```
  {{fun st ⇒ True}} (X ::= (ANum 1)) {{fun st ⇒ st X = 1}}.
Proof.
  apply hoare_consequence_pre
    with (P' := (fun st ⇒ st X = 1) [X |-> ANum 1]).
  apply hoare_asgn.
  intros st H. unfold assn_sub, t_update. simpl. reflexivity.
Qed.
```

Finally, for convenience in proofs, we can state a combined rule of consequence that allows us to vary both the precondition and the postcondition at the same time.

[78] c [79] P -» P' Q' -» Q

(hoare_consequence) [80] c [81]

```
Theorem hoare_consequence : ∀ (P P' Q Q' : Assertion) c,
  {{P'}} c {{Q'}} →
  P -» P' →
  Q' -» Q →
  {{P}} c {{Q}}.
Proof.
  intros P P' Q Q' c Hht HPP' HQ'Q.
  apply hoare_consequence_pre with (P' := P').
  apply hoare_consequence_post with (Q' := Q').
  assumption. assumption. assumption. Qed.
```

19.4.3 Digression: The `eapply` Tactic

This is a good moment to take another look at the `eapply` tactic, which we introduced briefly in the Auto chapter.

We had to write "`with` (*P'* := ...)" explicitly in the proof of hoare_asgn_example1 and hoare_consequence above, to make sure that all of the metavariables in the premises to the hoare_consequence_pre rule would be set to specific values. (Since *P'* doesn't appear in the conclusion of hoare_consequence_pre, the process of unifying the conclusion with the current goal doesn't constrain *P'* to a specific assertion.)

This is annoying, both because the assertion is a bit long and also because, in hoare_asgn_example1, the very next thing we are going to do – applying the hoare_asgn rule – will tell us exactly what it should be! We can use `eapply` instead of `apply` to tell Coq, essentially, "Be patient: The missing part is going to be filled in later in the proof."

```
Example hoare_asgn_example1' :
  {{fun st ⇒ True}}
```

---

[78] P'
[79] Q'
[80] P
[81] Q

```
  (X ::= (ANum 1))
  {{fun st ⇒ st X = 1}}.
Proof.
  eapply hoare_consequence_pre.
  apply hoare_asgn.
  intros st H. reflexivity. Qed.
```

In general, `eapply` *H* tactic works just like `apply` *H* except that, instead of failing if unifying the goal with the conclusion of *H* does not determine how to instantiate all of the variables appearing in the premises of *H*, `eapply` *H* will replace these variables with *existential variables* (written ?*nnn*), which function as placeholders for expressions that will be determined (by further unification) later in the proof.

In order for `Qed` to succeed, all existential variables need to be determined by the end of the proof. Otherwise Coq will (rightly) refuse to accept the proof. Remember that the Coq tactics build proof objects, and proof objects containing existential variables are not complete.

```
Lemma silly1 : ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∀ x y : nat, P x y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
Proof.
  intros P Q HP HQ. eapply HQ. apply HP.
```

Coq gives a warning after `apply` *HP*. (The warnings look different between Coq 8.4 and Coq 8.5. In 8.4, the warning says "No more subgoals but non-instantiated existential variables." In 8.5, it says "All the remaining goals are on the shelf," meaning that we've finished all our top-level proof obligations but along the way we've put some aside to be done later, and we have not finished those.) Trying to close the proof with `Qed` gives an error.

```
Abort.
```

An additional constraint is that existential variables cannot be instantiated with terms containing ordinary variables that did not exist at the time the existential variable was created. (The reason for this technical restriction is that allowing such instantiation would lead to inconsistency of Coq's logic.)

```
Lemma silly2 :
  ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
Proof.
  intros P Q HP HQ. eapply HQ. destruct HP as [y HP'].
```

Doing `apply` *HP'* above fails with the following error:
Error: Impossible to unify "?175" with "y".

In this case there is an easy fix: doing `destruct` *HP* *before* doing `eapply` *HQ*.

```
Abort.
Lemma silly2_fixed :
  ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
Proof.
  intros P Q HP HQ. destruct HP as [y HP'].
  eapply HQ. apply HP'.
Qed.
```

The `apply` *HP'* in the last step unifies the existential variable in the goal with the variable `y`.

Note that the `assumption` tactic doesn't work in this case, since it cannot handle existential variables. However, Coq also provides an *eassumption* tactic that solves the goal if one of the premises matches the goal up to instantiations of existential variables. We can use it instead of `apply` *HP'* if we like.

```
Lemma silly2_eassumption : ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
Proof.
  intros P Q HP HQ. destruct HP as [y HP']. eapply HQ. eassumption.
Qed.
```

Exercise: 2 starsM (hoare_asgn_examples_2) Translate these informal Hoare triples...

[82] X ::= X + 1 [83] [84] X ::= 3 [85]

...into formal statements (name them *assn_sub_ex1'* and *assn_sub_ex2'*) and use hoare_asgn and hoare_consequence_pre to prove them.

□

19.4.4 Skip

Since *SKIP* doesn't change the state, it preserves any property *P*:

(hoare_skip) [86] SKIP [87]

---

[82] `X+1<=5`
[83] `X<=5`
[84] `0<=3/\3<=5`
[85] `0<=X/\X<=5`
[86] `P`
[87] `P`

```
Theorem hoare_skip : ∀ P,
     {{P}} SKIP {{P}}.
Proof.
  intros P st st' H HP. inversion H. subst.
  assumption. Qed.
```

19.4.5 Sequencing

More interestingly, if the 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:

[88] c1 [89] [90] c2 [91]

(hoare_seq) [92] c1;;c2 [93]

```
Theorem hoare_seq : ∀ P Q R c1 c2,
     {{Q}} c2 {{R}} →
     {{P}} c1 {{Q}} →
     {{P}} c1;;c2 {{R}}.
Proof.
  intros P Q R c1 c2 H1 H2 st st' H12 Pre.
  inversion H12; subst.
  apply (H1 st'0 st'); try assumption.
  apply (H2 st st'0); assumption. 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.

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:

[94] X ::= a;; [95] <—- decoration for Q SKIP [96]

Here's an example of a program involving both assignment and sequencing.

```
Example hoare_asgn_example3 : ∀ a n,
  {{fun st ⇒ aeval st a = n}}
```

---

[88] P
[89] Q
[90] Q
[91] R
[92] P
[93] R
[94] a=n
[95] X=n
[96] X=n

```
  (X ::= a;; SKIP)
  {{fun st ⇒ st X = n}}.
Proof.
  intros a n. eapply hoare_seq.
  -
    apply hoare_skip.
  -
    eapply hoare_consequence_pre. apply hoare_asgn.
    intros st H. subst. reflexivity.
Qed.
```

We typically use hoare_seq in conjunction with hoare_consequence_pre and the eapply tactic, as in this example.

Exercise: 2 stars, recommended (hoare_asgn_example4) Translate this "decorated program" into a formal proof:

[97] -» [98] X ::= 1;; [99] -» [100] Y ::= 2 [101]

(Note the use of "-»" decorations, each marking a use of hoare_consequence_pre.)

```
Example hoare_asgn_example4 :
  {{fun st ⇒ True}} (X ::= (ANum 1);; Y ::= (ANum 2))
  {{fun st ⇒ st X = 1 ∧ st Y = 2}}.
Proof.
   Admitted.
```

□

Exercise: 3 stars (swap_exercise) Write an Imp program c that swaps the values of X and Y and show that it satisfies the following specification:

[102] c [103]

Your proof should not need to use unfold hoare_triple.

```
Definition swap_program : com
  . Admitted.
```

```
Theorem swap_exercise :
  {{fun st ⇒ st X ≤ st Y}}
  swap_program
  {{fun st ⇒ st Y ≤ st X}}.
Proof.
```

[97] True
[98] 1=1
[99] X=1
[100] X=1/\2=2
[101] X=1/\Y=2
[102] X<=Y
[103] Y<=X

Admitted.

□

Exercise: 3 starsM (hoarestate1) Explain why the following proposition can't be proven:

forall (a : aexp) (n : nat), [104] (X ::= (ANum 3);; Y ::= a) [105].

□

19.4.6 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:

[106] c1 [107] [108] c2 [109]

---

[110] IFB b THEN c1 ELSE c2 [111]

However, this is rather weak. For example, using this rule, we cannot show

[112] IFB X == 0 THEN Y ::= 2 ELSE Y ::= X + 1 FI [113]

since the rule tells us nothing 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).

[114] c1 [115] [116] c2 [117]

(hoare_if) [118] IFB b THEN c1 ELSE c2 FI [119]

---

[104] `funst=>aevalsta=n`
[105] `funst=>stY=n`
[106] `P`
[107] `Q`
[108] `P`
[109] `Q`
[110] `P`
[111] `Q`
[112] `True`
[113] `X<=Y`
[114] `P/\b`
[115] `Q`
[116] `P/\~b`
[117] `Q`
[118] `P`
[119] `Q`

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 bassn b for the assertion "the boolean expression b evaluates to true (in the given state)."

```
Definition bassn b : Assertion :=
  fun st ⇒ (beval st b = true).
```

A couple of useful facts about bassn:

```
Lemma bexp_eval_true : ∀ b st,
  beval st b = true → (bassn b) st.
Proof.
  intros b st Hbe.
  unfold bassn. assumption. Qed.

Lemma bexp_eval_false : ∀ b st,
  beval st b = false → ¬ ((bassn b) st).
Proof.
  intros b st Hbe contra.
  unfold bassn in contra.
  rewrite → contra in Hbe. inversion Hbe. Qed.
```

Now we can formalize the Hoare proof rule for conditionals and prove it correct.

```
Theorem hoare_if : ∀ P Q b c1 c2,
  {{fun st ⇒ P st ∧ bassn b st}} c1 {{Q}} →
  {{fun st ⇒ P st ∧ ~(bassn b st)}} c2 {{Q}} →
  {{P}} (IFB b THEN c1 ELSE c2 FI) {{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 bexp_eval_true. assumption.
  -
    apply (HFalse st st').
      assumption.
      split. assumption.
             apply bexp_eval_false. assumption. Qed.
```

Example

Here is a formal proof that the program we used to motivate the rule satisfies the specification we gave.

```
Example if_example :
    {{fun st ⇒ True}}
  IFB (BEq (AId X) (ANum 0))
    THEN (Y ::= (ANum 2))
    ELSE (Y ::= APlus (AId X) (ANum 1))
  FI
    {{fun st ⇒ st X ≤ st Y}}.
Proof.
  apply hoare_if.
  -
    eapply hoare_consequence_pre. apply hoare_asgn.
    unfold bassn, assn_sub, t_update, assert_implies.
    simpl. intros st [_ H].
    apply beq_nat_true in H.
    rewrite H. omega.
  -
    eapply hoare_consequence_pre. apply hoare_asgn.
    unfold assn_sub, t_update, assert_implies.
    simpl; intros st _. omega.
Qed.
```

Exercise: 2 stars (if_minus_plus) Prove the following hoare triple using hoare_if. Do not use `unfold` hoare_triple.

```
Theorem if_minus_plus :
  {{fun st ⇒ True}}
  IFB (BLe (AId X) (AId Y))
    THEN (Z ::= AMinus (AId Y) (AId X))
    ELSE (Y ::= APlus (AId X) (AId Z))
  FI
  {{fun st ⇒ st Y = st X + st Z}}.
Proof.
   Admitted.
```

☐

Exercise: One-sided conditionals

Exercise: 4 starsM (if1_hoare) In this exercise we consider extending Imp with "one-sided conditionals" of the form *IF1* b *THEN* c *FI*. 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 *FI* does nothing.

We recommend that you do this exercise before 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. We use a separate module to prevent polluting the global name space.)

```
Module If1.

Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CIf1 : bexp → com → com.

Notation "'SKIP'" :=
  CSkip.
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "X '::=' a" :=
  (CAss X a) (at level 60).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'IF1' b 'THEN' c 'FI'" :=
  (CIf1 b c) (at level 80, right associativity).
```

Next we need to extend the evaluation relation to accommodate *IF1* branches. This is for you to do... What rule(s) need to be added to **ceval** to evaluate one-sided conditionals?

```
Reserved Notation "c1 '/' st '\\' st'" (at level 40, st at level 39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st : state, SKIP / st \\ st
  | E_Ass : ∀ (st : state) (a1 : aexp) (n : nat) (X : id),
             aeval st a1 = n → (X ::= a1) / st \\ t_update st X n
  | E_Seq : ∀ (c1 c2 : com) (st st' st'' : state),
             c1 / st \\ st' → c2 / st' \\ st'' → (c1 ;; c2) / st \\ st''
  | E_IfTrue : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
                beval st b1 = true →
                c1 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
                 beval st b1 = false →
```

```
                    c2 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileEnd : ∀ (b1 : bexp) (st : state) (c1 : com),
                     beval st b1 = false → (WHILE b1 DO c1 END) / st \\ st
  | E_WhileLoop : ∀ (st st' st'' : state) (b1 : bexp) (c1 : com),
                     beval st b1 = true →
                     c1 / st \\ st' →
                     (WHILE b1 DO c1 END) / st' \\ st'' →
                     (WHILE b1 DO c1 END) / st \\ st''

  where "c1 '/' st '\\' st'" := (ceval c1 st st').
```

Now we repeat (verbatim) the definition and notation of Hoare triples.

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st',
       c / st \\ st' →
       P st →
       Q st'.
Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q)
                                      (at level 90, c at next level)
                                      : hoare_spec_scope.
```

Finally, we (i.e., you) need to state and prove a theorem, *hoare_if1*, that expresses an appropriate Hoare logic proof rule for one-sided conditionals. Try to come up with a rule that is both sound and as precise as possible.

For full credit, prove formally hoare_if1_good that your rule is precise enough to show the following valid Hoare triple:

[120] IF1 Y <> 0 THEN X ::= X + Y FI [121]

Hint: Your proof of this triple may need to use the other proof rules also. Because we're working in a separate module, you'll need to copy here the rules you find necessary.

```
Lemma hoare_if1_good :
  {{ fun st ⇒ st X + st Y = st Z }}
  IF1 BNot (BEq (AId Y) (ANum 0)) THEN
    X ::= APlus (AId X) (AId Y)
  FI
  {{ fun st ⇒ st X = st Z }}.
Proof. Admitted.
End If1.
```

□

[120] X+Y=Z

[121] X=Z

19.4.7 Loops

Finally, we need a rule for reasoning about while loops.

Suppose we have a loop

WHILE b DO c END

and we want to find a pre-condition P and a post-condition Q such that

[122] WHILE b DO c END [123]

is a valid triple.

First of all, let's think about the case where `b` is false at the beginning – i.e., let's assume that the loop body never executes at all. In this case, the loop behaves like *SKIP*, so we might be tempted to write:

[124] WHILE b DO c END [125].

But, as we remarked above for the conditional, we know a little more at the end – not just P, but also the fact that `b` is false in the current state. So we can enrich the postcondition a little:

[126] WHILE b DO c END [127]

What about the case where the loop body *does* get executed? In order to ensure that P holds when the loop finally exits, we certainly need to make sure that the command `c` guarantees that P holds whenever `c` is finished. Moreover, since P holds at the beginning of the first execution of `c`, and since each execution of `c` re-establishes P when it finishes, we can always assume that P holds at the beginning of `c`. This leads us to the following rule:

[128] c [129]

---

[130] WHILE b DO c END [131] This is almost the rule we want, but again it can be improved a little: at the beginning of the loop body, we know not only that P holds, but also that the guard `b` is true in the current state. This gives us a little more information to use in reasoning about `c` (showing that it establishes the invariant by the time it finishes). This gives us the final version of the rule:

[132] c [133]

(hoare_while) [134] WHILE b DO c END [135]

---

[122] `P`

[123] `Q`

[124] `P`

[125] `P`

[126] `P`

[127] `P/\~b`

[128] `P`

[129] `P`

[130] `P`

[131] `P/\~b`

[132] `P/\b`

[133] `P`

[134] `P`

[135] `P/\~b`

The proposition P is called an *invariant* of the loop.

```
Lemma hoare_while : ∀ P b c,
  {{fun st ⇒ P st ∧ bassn b st}} c {{P}} →
  {{P}} WHILE b DO c END {{fun st ⇒ P st ∧ ¬ (bassn b st)}}.
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.
  -
    split. assumption. apply bexp_eval_false. assumption.
  -
    apply IHHe2. reflexivity.
    apply (Hhoare st st'). assumption.
      split. assumption. apply bexp_eval_true. assumption.
Qed.
```

One subtlety in the terminology is that calling some assertion P a "loop invariant" doesn't just mean that it is preserved by the body of the loop in question (i.e., {{P}} c {{P}}, where c is the loop body), but rather that P *together with the fact that the loop's guard is true* is a sufficient precondition for c to ensure P as a postcondition.

This is a slightly (but significantly) weaker requirement. For example, if P is the assertion X = 0, then P *is* an invariant of the loop

WHILE X = 2 DO X := 1 END

although it is clearly *not* preserved by the body of the loop.

```
Example while_example :
    {{fun st ⇒ st X ≤ 3}}
  WHILE (BLe (AId X) (ANum 2))
  DO X ::= APlus (AId X) (ANum 1) END
    {{fun st ⇒ st X = 3}}.
Proof.
  eapply hoare_consequence_post.
  apply hoare_while.
  eapply hoare_consequence_pre.
  apply hoare_asgn.
  unfold bassn, assn_sub, assert_implies, t_update. simpl.
    intros st [H1 H2]. apply leb_complete in H2. omega.
  unfold bassn, assert_implies. intros st [Hle Hb].
    simpl in Hb. destruct (leb (st X) 2) eqn : Heqle.
    exfalso. apply Hb; reflexivity.
    apply leb_iff_conv in Heqle. omega.
Qed.
```

We can use the WHILE rule to prove the following Hoare triple...

```
Theorem always_loop_hoare : ∀ P Q,
  {{P}} WHILE BTrue DO SKIP END {{Q}}.
Proof.
  intros P Q.
  apply hoare_consequence_pre with (P' := fun st : state ⇒ True).
  eapply hoare_consequence_post.
  apply hoare_while.
  -
    apply hoare_post_true. intros st. apply I.
  -
    simpl. intros st [Hinv Hguard].
    exfalso. apply Hguard. reflexivity.
  -
    intros st H. constructor. Qed.
```

Of course, this result is not surprising if we remember that the definition of 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 only talk about terminating commands are often said to describe a logic of "partial" correctness. It is also possible to give Hoare rules for "total" correctness, which build in the fact that the commands terminate. However, in this course we will only talk about partial correctness.

Exercise: *REPEAT*

Exercise: 4 stars, advancedM (hoare_repeat) In this exercise, we'll add a new command to our language of commands: *REPEAT* c *UNTIL* a *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 : id → aexp → com
  | CSeq : com → com → com
  | CIf : 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 "'SKIP'" :=
```

```
  CSkip.
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "X '::=' a" :=
  (CAsgn X a) (at level 60).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'REPEAT' e1 'UNTIL' b2 'END'" :=
  (CRepeat e1 b2) (at level 80, right associativity).
```

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. Then update the *ceval_cases* tactic to handle these added cases.

```
Inductive ceval : state → com → state → Prop :=
  | E_Skip : ∀ st,
      ceval st SKIP st
  | E_Ass : ∀ st a1 n X,
      aeval st a1 = n →
      ceval st (X ::= a1) (t_update st X n)
  | E_Seq : ∀ c1 c2 st st' st'',
      ceval st c1 st' →
      ceval st' c2 st'' →
      ceval st (c1 ;; c2) st''
  | E_IfTrue : ∀ st st' b1 c1 c2,
      beval st b1 = true →
      ceval st c1 st' →
      ceval st (IFB b1 THEN c1 ELSE c2 FI) st'
  | E_IfFalse : ∀ st st' b1 c1 c2,
      beval st b1 = false →
      ceval st c2 st' →
      ceval st (IFB b1 THEN c1 ELSE c2 FI) st'
  | E_WhileEnd : ∀ b1 st c1,
      beval st b1 = false →
      ceval st (WHILE b1 DO c1 END) st
  | E_WhileLoop : ∀ st st' st'' b1 c1,
      beval st b1 = true →
      ceval st c1 st' →
      ceval st' (WHILE b1 DO c1 END) st'' →
      ceval st (WHILE b1 DO c1 END) st''
```

.

A couple of definitions from above, copied here so they use the new **ceval**.

```
Notation "c1 '/' st '\\' st'" := (ceval st c1 st')
                                  (at level 40, st at level 39).
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion)
                        : Prop :=
  ∀ st st', (c / st \\ st') → P st → Q st'.
Notation "{{ P }} c {{ Q }}" :=
  (hoare_triple P c Q) (at level 90, c at next level).
```

To make sure you've got the evaluation rules for *REPEAT* right, prove that `ex1_repeat` evaluates correctly.

```
Definition ex1_repeat :=
  REPEAT
    X ::= ANum 1;;
    Y ::= APlus (AId Y) (ANum 1)
  UNTIL (BEq (AId X) (ANum 1)) END.
Theorem ex1_repeat_works :
  ex1_repeat / empty_state \\
               t_update (t_update empty_state X 1) Y 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:
[136] REPEAT Y ::= X;; X ::= X - 1 UNTIL X = 0 END [137]

```
End RepeatExercise.
```

□

19.5 Summary

So far, we've introduced Hoare Logic as a tool for reasoning about Imp programs. In the reminder of this chapter we'll explore a systematic way to use Hoare Logic to prove properties about programs. The rules of Hoare Logic are:

---

[136] `X>0`
[137] `X=0/\Y>0`

(hoare_asgn) [138] X::=a [139]

(hoare_skip) [140] SKIP [141]
[142] c1 [143] [144] c2 [145]

(hoare_seq) [146] c1;;c2 [147]
[148] c1 [149] [150] c2 [151]

(hoare_if) [152] IFB b THEN c1 ELSE c2 FI [153]
[154] c [155]

(hoare_while) [156] WHILE b DO c END [157]
[158] c [159] P -» P' Q' -» Q

(hoare_consequence) [160] c [161]

In the next chapter, we'll see how these rules are used to prove that programs satisfy specifications of their behavior.

---

[138] `Q[X|->a]`
[139] `Q`
[140] `P`
[141] `P`
[142] `P`
[143] `Q`
[144] `Q`
[145] `R`
[146] `P`
[147] `R`
[148] `P/\b`
[149] `Q`
[150] `P/\~b`
[151] `Q`
[152] `P`
[153] `Q`
[154] `P/\b`
[155] `P`
[156] `P`
[157] `P/\~b`
[158] `P'`
[159] `Q'`
[160] `P`
[161] `Q`

19.6 Additional Exercises

Exercise: 3 stars (himp_hoare) In this exercise, we will derive proof rules for the *HAVOC* command, which we studied in the last 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 : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CHavoc : id → com.

Notation "'SKIP'" :=
  CSkip.
Notation "X '::=' a" :=
  (CAsgn X a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'HAVOC' X" := (CHavoc X) (at level 60).

Reserved Notation "c1 '/' st '\\' st'" (at level 40, st at level 39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st : state, SKIP / st \\ st
  | E_Ass : ∀ (st : state) (a1 : aexp) (n : nat) (X : id),
            aeval st a1 = n → (X ::= a1) / st \\ t_update st X n
  | E_Seq : ∀ (c1 c2 : com) (st st' st'' : state),
            c1 / st \\ st' → c2 / st' \\ st'' → (c1 ;; c2) / st \\ st''
  | E_IfTrue : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
               beval st b1 = true →
               c1 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
                beval st b1 = false →
                c2 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileEnd : ∀ (b1 : bexp) (st : state) (c1 : com),
                 beval st b1 = false → (WHILE b1 DO c1 END) / st \\ st
  | E_WhileLoop : ∀ (st st' st'' : state) (b1 : bexp) (c1 : com),
```

```
                    beval st b1 = true →
                    c1 / st \\ st' →
                    (WHILE b1 DO c1 END) / st' \\ st'' →
                    (WHILE b1 DO c1 END) / st \\ st''
  | E_Havoc : ∀ (st : state) (X : id) (n : nat),
                (HAVOC X) / st \\ t_update st X n

  where "c1 '/' st '\\' st'" := (ceval c1 st st').
```

The definition of Hoare triples is exactly as before.

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st', c / st \\ st' → P st → Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q)
                                  (at level 90, c at next level)
                                  : hoare_spec_scope.
```

Complete the Hoare rule for *HAVOC* commands below by defining havoc_pre and prove that the resulting rule is correct.

```
Definition havoc_pre (X : id) (Q : Assertion) : Assertion
  . Admitted.

Theorem hoare_havoc : ∀ (Q : Assertion) (X : id),
  {{ havoc_pre X Q }} HAVOC X {{ Q }}.
Proof.
  Admitted.

End Himp.
```

☐

$Date : 2017-02-0210:32:28-0500 (Thu, 02 Feb 2017)$

Chapter 20

Library Top.Hoare2

20.1 Hoare2: Hoare Logic, Part II

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Maps.
Require Import Imp.
Require Import Hoare.
```

20.2 Decorated Programs

The beauty of Hoare Logic is that it is *compositional*: the structure of proofs exactly follows the structure of programs. This suggests that we can record the essential ideas of a proof informally (leaving out some low-level calculational details) by decorating a program with appropriate assertions on each of its commands. Such a *decorated program* carries with it an (informal) proof of its own correctness.

For example, consider the program:

X ::= m;; Z ::= p; WHILE X <> 0 DO Z ::= Z - 1;; X ::= X - 1 END

(Note the *parameters* `m` and p, which stand for fixed-but-arbitrary numbers. Formally, they are simply Coq variables of type `nat`.) One possible specification for this program:

[1] X ::= m;; Z ::= p; WHILE X <> 0 DO Z ::= Z - 1;; X ::= X - 1 END [2] Finally, here is a decorated version of the program, embodying a proof of this specification:

---

[1] `True`

[2] `Z=p-m`

[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 text interleaved with assertions (either a single assertion or possibly two assertions separated by an implication). To check that a decorated program represents a valid proof, we check that each individual command is *locally consistent* with its nearby assertions in the following sense:

- *SKIP* is locally consistent if its precondition and postcondition are the same:

  [15] SKIP [16]

- The sequential composition of `c1` and `c2` is locally consistent (with respect to assertions P and R) if `c1` is locally consistent (with respect to P and Q) and `c2` is locally consistent (with respect to Q and R):

  [17] c1;; [18] c2 [19]

- An assertion is locally consistent if its precondition is the appropriate substitution of its postcondition:

  [20] X ::= a [21]

- A conditional is locally consistent (with respect to assertions P and Q) if the assertions at the top of its "then" and "else" branches are exactly $P \wedge \mathsf{b}$ and $P \wedge \neg \mathsf{b}$ and if its "then" branch is locally consistent (with respect to $P \wedge \mathsf{b}$ and Q) and its "else" branch is locally consistent (with respect to $P \wedge \neg \mathsf{b}$ and Q):

[3] `True`
[4] `m=m`
[5] `X=m`
[6] `X=m/\p=p`
[7] `X=m/\Z=p`
[8] `Z-X=p-m`
[9] `Z-X=p-m/\X<>0`
[10] `(Z-1)-(X-1)=p-m`
[11] `Z-(X-1)=p-m`
[12] `Z-X=p-m`
[13] `Z-X=p-m/\~(X<>0)`
[14] `Z=p-m`
[15] `P`
[16] `P`
[17] `P`
[18] `Q`
[19] `R`
[20] `P[X|->a]`
[21] `P`

[22] IFB b THEN [23] c1 [24] ELSE [25] c2 [26] FI [27]

- A while loop with precondition P is locally consistent if its postcondition is $P \wedge \neg \mathsf{b}$, if the pre- and postconditions of its body are exactly $P \wedge \mathsf{b}$ and P, and if its body is locally consistent:

  [28] WHILE b DO [29] c1 [30] END [31]

- A pair of assertions separated by -» is locally consistent if the first implies the second:

  [32] -» [33]

  This corresponds to the application of hoare_consequence and is the only place in a decorated program where checking whether decorations are correct is not fully mechanical and syntactic, but rather may involve logical and/or arithmetic reasoning.

The above essentially describes a procedure for *verifying* the correctness of a given proof involves checking that every single command is locally consistent with the accompanying assertions. If we are instead interested in *finding* a proof for a given specification, we need to discover the right assertions. This can be done in an almost mechanical way, with the exception of finding loop invariants, which is the subject of the next section. In the remainder of this section we explain in detail how to construct decorations for several simple programs that don't involve non-trivial loop invariants.

20.2.1 Example: Swapping Using Addition and Subtraction

Here is a program that 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 prove using decorations that this program is correct – i.e., it always swaps the values of variables X and Y.

[(1) {{ X = m ∧ Y = n }} -» (2) {{ (X + Y) - ((X + Y) - Y) = n ∧ (X + Y) - Y = m }} X ::= X + Y;; (3) {{ X - (X - Y) = n ∧ X - Y = m }} Y ::= X - Y;; (4) {{ X - Y = n ∧ Y = m }} X ::= X - Y (5) {{ X = n ∧ Y = m }}

[22] `P`

[23] `P/\b`

[24] `Q`

[25] `P/\~b`

[26] `Q`

[27] `Q`

[28] `P`

[29] `P/\b`

[30] `P`

[31] `P/\~b`

[32] `P`

[33] `P'`

These decorations can be constructed as *follows*: - *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** (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. For this we substitute* [X] by [m] **and** [Y] by [n] **and** *calculate* as *follows*:

(m + n) - ((m + n) - n) = n ∧ (m + n) - n = m (m + n) - m = n ∧ m = m n = n ∧ m = m

Note that, since we are working with *natural numbers rather than fixed-width machine integers, we don't need to worry about the possibility of arithmetic overflow anywhere* in *this argument. This makes life quite* a *bit simpler*!

20.2.2 Example: Simple Conditionals

Here is a simple decorated program using conditionals:

(1) [34] IFB X <= Y THEN (2) [35] -» (3) [36] Z ::= Y - X (4) [37] ELSE (5) [38] -» (6) [39] Z ::= X - Y (7) [40] FI (8) [41]

These decorations were constructed as follows:

- We start with the outer precondition (1) and postcondition (8).
- We follow the format dictated by the hoare_if rule and 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 ≤ 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 ¬ (X ≤

[34] `True`
[35] `True/\X<=Y`
[36] `(Y-X)+X=Y\/(Y-X)+Y=X`
[37] `Z+X=Y\/Z+Y=X`
[38] `True/\~(X<=Y)`
[39] `(X-Y)+X=Y\/(X-Y)+Y=X`
[40] `Z+X=Y\/Z+Y=X`
[41] `Z+X=Y\/Z+Y=X`

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).

Exercise: 2 starsM (if_minus_plus_reloaded) Fill in valid decorations for the following program:

[42] IFB X <= Y THEN [43] -» [44] Z ::= Y - X [45] ELSE [46] -» [47] Y ::= X + Z [48] FI [49] □

20.2.3 Example: Reduce to Zero

Here is a *WHILE* loop that is so simple it needs no invariant (i.e., the invariant **True** will do the job).

(1) [50] WHILE X <> 0 DO (2) [51] -» (3) [52] X ::= X - 1 (4) [53] END (5) [54] -» (6) [55]

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) and with the negation of the guard to obtain (5). Note that, because the outer postcondition (6) does not syntactically match (5), we need a trivial use of the consequence rule from (5) to (6).
- Assertion (3) is the same as (4), because X does not appear in 4, so the substitution in the assignment rule is trivial.
- Finally, the implication between (2) and (3) is also trivial.

From this informal proof, it is easy to read off a formal proof using the Coq versions of the Hoare rules. Note that we do *not* unfold the definition of hoare_triple anywhere in this proof – the idea is to use the Hoare rules as a "self-contained" logic for reasoning about programs.

---

[42] `True`
[43]
[44]
[45]
[46]
[47]
[48]
[49] `Y=X+Z`
[50] `True`
[51] `True/\X<>0`
[52] `True`
[53] `True`
[54] `True/\X=0`
[55] `X=0`

```
Definition reduce_to_zero' : com :=
  WHILE BNot (BEq (AId X) (ANum 0)) DO
    X ::= AMinus (AId X) (ANum 1)
  END.

Theorem reduce_to_zero_correct' :
  {{fun st ⇒ True}}
  reduce_to_zero'
  {{fun st ⇒ st X = 0}}.
Proof.
  unfold reduce_to_zero'.
  eapply hoare_consequence_post.
  apply hoare_while.
  -

    eapply hoare_consequence_pre. apply hoare_asgn.
    intros st [HT Hbp]. unfold assn_sub. apply I.
  -
    intros st [Inv GuardFalse].
    unfold bassn in GuardFalse. simpl in GuardFalse.
    rewrite not_true_iff_false in GuardFalse.
    rewrite negb_false_iff in GuardFalse.
    apply beq_nat_true in GuardFalse.
    apply GuardFalse. Qed.
```

20.2.4 Example: Division

The following Imp program calculates the integer quotient and remainder of two numbers m and n that are arbitrary constants in the program.

X ::= m;; Y ::= 0;; WHILE n <= X DO X ::= X - n;; Y ::= Y + 1 END;

In 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 reminder X and a quotient Y such that $n \times Y + X = m \wedge X < n$.

It turns out that we get lucky with this program and don't have to think very hard about the loop invariant: the invariant is just the first conjunct $n \times Y + X = m$, and we can use this to decorate the program.

(1) [56] ->> (2) [57] X ::= m;; (3) [58] Y ::= 0;; (4) [59] WHILE n <= X DO (5) [60] ->> (6) [61] X

[56] True

[57] n*0+m=m

[58] n*0+X=m

[59] n*Y+X=m

[60] n*Y+X=m/\n<=X

[61] n*(Y+1)+(X-n)=m

::= X - n;; (7) [62] Y ::= Y + 1 (8) [63] END (9) [64]

Assertions (4), (5), (8), and (9) are derived mechanically from the 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 two uses of the consequence rule are correct – i.e., that (1) implies (2) and that (5) implies (6). This is indeed the case, so we have a valid decorated program.

20.3 Finding Loop Invariants

Once the outermost precondition and postcondition are chosen, the only creative part in verifying programs 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 the loop invariant (or the induction hypothesis) means that you have a stronger assumption to work with when trying to establish the postcondition of the loop body (or complete the induction step of the proof), but it also means that the loop body's postcondition (or the statement being proved inductively) is stronger and thus harder to prove!

This section explains how to approach the challenge of finding loop invariants through a series of examples and exercises.

20.3.1 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 following specification:

[65] WHILE X <> 0 DO Y ::= Y - 1;; X ::= X - 1 END [66]

To verify this program, we need to find an invariant I for the loop. As a first step we can leave I as an unknown and build a *skeleton* for the proof by applying (backward) the rules for local consistency. This process leads to the following skeleton:

(1) [67] -» (a) (2) [68] WHILE X <> 0 DO (3) [69] -» (c) (4) [70] Y ::= Y - 1;; (5) [71] X ::= X -

[62] `n*(Y+1)+X=m`

[63] `n*Y+X=m`

[64] `n*Y+X=m/\X<n`

[65] `X=m/\Y=n`

[66] `Y=n-m`

[67] `X=m/\Y=n`

[68] `I`

[69] `I/\X<>0`

[70] `I[X|->X-1][Y|->Y-1]`

[71] `I[X|->X-1]`

1 (6) [72] END (7) [73] -» (b) (8) [74]

By examining this skeleton, we can see that any valid I 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 one iteration of the loop, i.e., (3) must imply (4).

These conditions are actually independent of the particular program and specification we are considering. Indeed, every loop invariant has to satisfy them. One way to find an 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 the process.

For instance, in the reduce-to-zero example above, we saw that, for a very simple loop, choosing **True** as an invariant did the job. So let's try instantiating I with **True** in the skeleton above see what we get...

(1) [75] -» (a - OK) (2) [76] WHILE X <> 0 DO (3) [77] -» (c - OK) (4) [78] Y ::= Y - 1;; (5) [79] X ::= X - 1 (6) [80] END (7) [81] -» (b - WRONG!) (8) [82]

While conditions (a) and (c) are trivially satisfied, condition (b) is wrong, i.e., it is not the case that (7) $\mathsf{True} \land \mathsf{X} = 0$ implies (8) $\mathsf{Y} = \mathsf{n} - \mathsf{m}$. In fact, the two assertions are completely unrelated, so it is very easy to find a counterexample to the implication (say, $\mathsf{Y} = \mathsf{X} = \mathsf{m} = 0$ and $\mathsf{n} = 1$).

If we want (b) to hold, we need to strengthen the invariant so that it implies the postcondition (8). One simple way to do this is to let the invariant *be* the postcondition. So let's return to our skeleton, instantiate I with $\mathsf{Y} = \mathsf{n} - \mathsf{m}$, and check conditions (a) to (c) again.

(1) [83] -» (a - WRONG!) (2) [84] WHILE X <> 0 DO (3) [85] -» (c - WRONG!) (4) [86] Y ::=

[72] `I`

[73] `I/\~(X<>0)`

[74] `Y=n-m`

[75] `X=m/\Y=n`

[76] `True`

[77] `True/\X<>0`

[78] `True`

[79] `True`

[80] `True`

[81] `True/\X=0`

[82] `Y=n-m`

[83] `X=m/\Y=n`

[84] `Y=n-m`

[85] `Y=n-m/\X<>0`

[86] `Y-1=n-m`

Y - 1;; (5) [87] X ::= X - 1 (6) [88] END (7) [89] -» (b - OK) (8) [90]

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 invariant.

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 an invariant!

To do better, we need to generalize (8) 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 I in the skeleton above with Y - X = n - m.

(1) [91] -» (a - OK) (2) [92] WHILE X <> 0 DO (3) [93] -» (c - OK) (4) [94] Y ::= Y - 1;; (5) [95] X ::= X - 1 (6) [96] END (7) [97] -» (b - OK) (8) [98]

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.)

20.3.2 Exercise: Slow Assignment

Exercise: 2 starsM (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:

[99] Y ::= 0;; WHILE X <> 0 DO X ::= X - 1;; Y ::= Y + 1 END [100]

Write an informal decorated program showing that this procedure is correct.

[87] `Y=n-m`
[88] `Y=n-m`
[89] `Y=n-m/\X=0`
[90] `Y=n-m`
[91] `X=m/\Y=n`
[92] `Y-X=n-m`
[93] `Y-X=n-m/\X<>0`
[94] `(Y-1)-(X-1)=n-m`
[95] `Y-(X-1)=n-m`
[96] `Y-X=n-m`
[97] `Y-X=n-m/\X=0`
[98] `Y=n-m`
[99] `X=m`
[100] `Y=m`

□

20.3.3 Exercise: Slow Addition

Exercise: 3 stars, optional (add_slowly_decoration) The following program adds the variable X into the variable Z by repeatedly decrementing X and incrementing Z.

WHILE X <> 0 DO Z ::= Z + 1;; X ::= X - 1 END

Following the pattern of the subtract_slowly example above, pick a precondition and postcondition that give an appropriate specification of *add_slowly*; then (informally) decorate the program accordingly.

□

20.3.4 Example: Parity

Here is a cute little program for computing the parity of the value initially stored in X (due to Daniel Cristofani).

[101] WHILE 2 <= X DO X ::= X - 2 END [102]

The mathematical parity function used in the specification is defined in Coq as follows:

```
Fixpoint parity x :=
  match x with
  | 0 ⇒ 0
  | 1 ⇒ 1
  | S (S x') ⇒ parity x'
  end.
```

The postcondition does not hold at the beginning of the loop, since m = parity m does not hold for an arbitrary m, so we cannot use that as an invariant. To find an 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 parity X = parity m as an invariant we obtain the following decorated program:

[103] -» (a - OK) [104] WHILE 2 <= X DO [105] -» (c - OK) [106] X ::= X - 2 [107] END [108] -» (b - OK) [109]

---

[101] X=m
[102] X=paritym
[103] X=m
[104] parityX=paritym
[105] parityX=paritym/\2<=X
[106] parity(X-2)=paritym
[107] parityX=paritym
[108] parityX=paritym/\X<2
[109] X=paritym

With this invariant, conditions (a), (b), and (c) are all satisfied. For verifying (b), we observe that, when X < 2, we have parity X = X (we can easily see this in the definition of parity). For verifying (c), we observe that, when 2 ≤ X, we have parity X = parity (X-2).

Exercise: 3 stars, optional (parity_formal) Translate this proof to Coq. Refer to the *reduce_to_zero* example for ideas. You may find the following two lemmas useful:

```
Lemma parity_ge_2 : ∀ x,
  2 ≤ x →
  parity (x - 2) = parity x.
Proof.
  induction x; intro. reflexivity.
  destruct x. inversion H. inversion H1.
  simpl. rewrite ← minus_n_O. reflexivity.
Qed.

Lemma parity_lt_2 : ∀ x,
  ¬ 2 ≤ x →
  parity (x) = x.
Proof.
  intros. induction x. reflexivity. destruct x. reflexivity.
    exfalso. apply H. omega.
Qed.

Theorem parity_correct : ∀ m,
    {{ fun st ⇒ st X = m }}
  WHILE BLe (ANum 2) (AId X) DO
    X ::= AMinus (AId X) (ANum 2)
  END
    {{ fun st ⇒ st X = parity m }}.
Proof.
   Admitted.
```

□

20.3.5 Example: Finding Square Roots

The following program computes the square root of X by naive iteration:

[110] Z ::= 0;; WHILE (Z+1)*(Z+1) <= X DO Z ::= Z+1 END [111]

As above, we can try to use the postcondition as a candidate invariant, obtaining the following decorated program:

[110] X=m

[111] Z*Z<=m/\m<(Z+1)*(Z+1)

(1) [112] -» (a - second conjunct of (2) WRONG!) (2) [113] Z ::= 0;; (3) [114] WHILE (Z+1)*(Z+1) <= X DO (4) [115] -» (c - WRONG!) (5) [116] Z ::= Z+1 (6) [117] END (7) [118] -» (b - OK) (8) [119]

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), 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, but we didn't propagate this information from (1) into the loop invariant.

Also, looking at the second conjunct of (8), it seems quite hopeless as an invariant (why?); fortunately, we don't need it, since we can obtain it from the negation of the guard – the third conjunct in (7) – again under the assumption that X=m.

So we now try $X{=}m \wedge Z{\times}Z \leq m$ as the loop invariant:

[120] -» (a - OK) [121] Z ::= 0; [122] WHILE (Z+1)*(Z+1) <= X DO [123] -» (c - OK) [124] Z ::= Z+1 [125] END [126] -» (b - OK) [127]

This works, since conditions (a), (b), and (c) are now all trivially satisfied.

Very often, even if 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 add the fact that it doesn't change to the loop invariant.

20.3.6 Example: Squaring

Here is a program that squares X by repeated addition:

[128] Y ::= 0;; Z ::= 0;; WHILE Y <> X DO Z ::= Z + X;; Y ::= Y + 1 END [129]

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 is a good idea in such cases to add $X = m$ to the invariant. The other thing that we know is often useful in the invariant is the postcondition, so let's add that too, leading to the invariant candidate $Z = m \times m \wedge X = m$.

---

[112] `X=m`
[113] `0*0<=m/\m<1*1`
[114] `Z*Z<=m/\m<(Z+1)*(Z+1)`
[115] `Z*Z<=m/\(Z+1)*(Z+1)<=X`
[116] `(Z+1)*(Z+1)<=m/\m<(Z+2)*(Z+2)`
[117] `Z*Z<=m/\m<(Z+1)*(Z+1)`
[118] `Z*Z<=m/\m<(Z+1)*(Z+1)/\~((Z+1)*(Z+1)<=X)`
[119] `Z*Z<=m/\m<(Z+1)*(Z+1)`
[120] `X=m`
[121] `X=m/\0*0<=m`
[122] `X=m/\Z*Z<=m`
[123] `X=m/\Z*Z<=m/\(Z+1)*(Z+1)<=X`
[124] `X=m/\(Z+1)*(Z+1)<=m`
[125] `X=m/\Z*Z<=m`
[126] `X=m/\Z*Z<=m/\X<(Z+1)*(Z+1)`
[127] `Z*Z<=m/\m<(Z+1)*(Z+1)`
[128] `X=m`
[129] `Z=m*m`

[130] -» (a - WRONG) [131] Y ::= 0;; [132] Z ::= 0;; [133] WHILE Y <> X DO [134] -» (c - WRONG) [135] Z ::= Z + X;; [136] Y ::= Y + 1 [137] END [138] -» (b - OK) [139]

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 progesses in the loop, after the 1st iteration $Z = m$, after the 2nd iteration $Z = 2*m$, 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 invariant candidate: $Z = Y \times m \wedge X = m$.

[140] -» (a - OK) [141] Y ::= 0;; [142] Z ::= 0;; [143] WHILE Y <> X DO [144] -» (c - OK) [145] Z ::= Z + X; [146] Y ::= Y + 1 [147] END [148] -» (b - OK) [149]

This new 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 invariant. It is often the case that one has to replace parameters with variables – or with expressions involving both variables and parameters, like m - Y – when going from postconditions to invariants.

20.3.7 Exercise: Factorial

Exercise: 3 starsM (factorial) Recall that n! denotes the factorial of n (i.e., n! = 1*2*...*n). Here is an Imp program that calculates the factorial of the number initially stored in the variable X and puts it in the variable Y:

[150] Y ::= 1 ;; WHILE X <> 0 DO Y ::= Y * X ;; X ::= X - 1 END [151]

Fill in the blanks in following decorated program:

---

[130] X=m
[131] 0=m*m/\X=m
[132] 0=m*m/\X=m
[133] Z=m*m/\X=m
[134] Z=Y*m/\X=m/\Y<>X
[135] Z+X=m*m/\X=m
[136] Z=m*m/\X=m
[137] Z=m*m/\X=m
[138] Z=m*m/\X=m/\~(Y<>X)
[139] Z=m*m
[140] X=m
[141] 0=0*m/\X=m
[142] 0=Y*m/\X=m
[143] Z=Y*m/\X=m
[144] Z=Y*m/\X=m/\Y<>X
[145] Z+X=(Y+1)*m/\X=m
[146] Z=(Y+1)*m/\X=m
[147] Z=Y*m/\X=m
[148] Z=Y*m/\X=m/\~(Y<>X)
[149] Z=m*m
[150] X=m
[151] Y=m!

[152] -» [153] Y ::= 1;; [154] WHILE X <> 0 DO [155] -» [156] Y ::= Y * X;; [157] X ::= X - 1 [158] END [159] -» [160]

□

20.3.8 Exercise: Min

Exercise: 3 starsM (Min_Hoare) Fill in valid decorations for the following program. For the ⇒ steps in your annotations, you may rely (silently) on the following facts about min

Lemma lemma1 : forall x y, (x=0 \/ y=0) -> min x y = 0. Lemma lemma2 : forall x y, min (x-1) (y-1) = (min x y) - 1.

plus standard high-school algebra, as always.

[161] -» [162] X ::= a;; [163] Y ::= b;; [164] Z ::= 0;; [165] WHILE (X <> 0 /\ Y <> 0) DO [166] -» [167] X := X - 1;; [168] Y := Y - 1;; [169] Z := Z + 1 [170] END [171] -» [172]

□

Exercise: 3 starsM (two_loops) Here is a very 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 filling in the blanks in the following decorated program.

---

152 `X=m`
153
154
155
156
157
158
159
160 `Y=m!`
161 `True`
162
163
164
165
166
167
168
169
170
171
172 `Z=minab`

[173] -» [174] X ::= 0;; [175] Y ::= 0;; [176] Z ::= c;; [177] WHILE X <> a DO [178] -» [179] X ::= X + 1;; [180] Z ::= Z + 1 [181] END;; [182] -» [183] WHILE Y <> b DO [184] -» [185] Y ::= Y + 1;; [186] Z ::= Z + 1 [187] END [188] -» [189]

□

20.3.9 Exercise: Power Series

Exercise: 4 stars, optional (dpow2_down) Here is a program that computes the series: 1 + 2 + 2^2 + ... + 2^m = 2^(m+1) - 1

X ::= 0;; Y ::= 1;; Z ::= 1;; WHILE X <> m DO Z ::= 2 * Z;; Y ::= Y + Z;; X ::= X + 1 END

Write a decorated program for this.

□

20.4 Weakest Preconditions (Optional)

Some Hoare triples are more interesting than others. For example,

[190] X ::= Y + 1 [191]

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,

[192] X ::= Y + 1 [193]

---

[173] True
[174]
[175]
[176]
[177]
[178]
[179]
[180]
[181]
[182]
[183]
[184]
[185]
[186]
[187]
[188]
[189] Z=a+b+c
[190] False
[191] X<=5
[192] Y<=4/\Z=0
[193] X<=5

is useful: it tells us that, if we can somehow create a situation in which we know that $Y \le 4 \wedge Z = 0$, then running this command will produce a state satisfying the postcondition. However, this triple is still not as useful as it could be, because the $Z = 0$ clause in the precondition actually has nothing to do with the postcondition $X \le 5$. The *most* useful triple (for this command and postcondition) is this one:

[194] X ::= Y + 1 [195]

In other words, $Y \le 4$ is the *weakest* valid precondition of the command X ::= Y + 1 for the postcondition $X \le 5$.

In general, we say that "P is the weakest precondition of command c for postcondition Q" if {{P}} c {{Q}} and if, whenever P' is an assertion such that {{P'}} c {{Q}}, it is the case that P' *st* implies P *st* for all states *st*.

```
Definition is_wp P c Q :=
  {{P}} c {{Q}} ∧
  ∀ P', {{P'}} c {{Q}} → (P' -» P).
```

That is, P is the weakest precondition of c for Q if (a) P *is* a precondition for Q and c, and (b) P is the *weakest* (easiest to satisfy) assertion that guarantees that Q will hold after executing c.

Exercise: 1 star, optional (wp) What are the weakest preconditions of the following commands for the following postconditions?

1) [196] SKIP [197]
2) [198] X ::= Y + Z [199]
3) [200] X ::= Y [201]
4) [202] IFB X == 0 THEN Y ::= Z + 1 ELSE Y ::= W + 2 FI [203]
5) [204] X ::= 5 [205]
6) [206] WHILE True DO X ::= 0 END [207] ☐

Exercise: 3 stars, advanced, optional (is_wp_formal) Prove formally, using the definition of hoare_triple, that $Y \le 4$ is indeed the weakest precondition of X ::= Y + 1 with respect to postcondition $X \le 5$.

---

[194] Y<=4
[195] X<=5
[196] ?
[197] X=5
[198] ?
[199] X=5
[200] ?
[201] X=Y
[202] ?
[203] Y=5
[204] ?
[205] X=0
[206] ?
[207] X=0

```
Theorem is_wp_example :
  is_wp (fun st ⇒ st Y ≤ 4)
    (X ::= APlus (AId Y) (ANum 1)) (fun st ⇒ st 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 : ∀ 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 rule from the *himp_hoare* exercise in the Hoare chapter returns the weakest precondition. Module *Himp2*.

```
Import Himp.

Lemma hoare_havoc_weakest : ∀ (P Q : Assertion) (X : id),
  {{ P }} HAVOC X {{ Q }} →
  P -» havoc_pre X Q.
Proof.
   Admitted.
End Himp2.
```
☐

20.5 Formal Decorated Programs (Optional)

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 informal presentation style can actually be made completely formal and indeed that checking the validity of decorated programs can mostly be automated.

20.5.1 Syntax

The first thing we need to do is to formalize a variant of the syntax of commands with embedded assertions. We call the new commands *decorated commands*, or **dcom**s.

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, decorations are added corresponding to each postcondition. A separate type, **decorated**, is used to add the precondition for the entire program.

```
Inductive dcom : Type :=
  | DCSkip : Assertion → dcom
  | DCSeq : dcom → dcom → dcom
  | DCAsgn : id → aexp → Assertion → dcom
  | DCIf : bexp → Assertion → dcom → Assertion → dcom
              → Assertion→ dcom
  | DCWhile : bexp → Assertion → dcom → Assertion → dcom
  | DCPre : Assertion → dcom → dcom
  | DCPost : dcom → Assertion → dcom.

Inductive decorated : Type :=
  | Decorated : Assertion → dcom → decorated.

Notation "'SKIP' {{ P }}"
      := (DCSkip P)
      (at level 10) : dcom_scope.
Notation "l '::=' a {{ P }}"
      := (DCAsgn l a P)
      (at level 60, a at next level) : dcom_scope.
Notation "'WHILE' b 'DO' {{ Pbody }} d 'END' {{ Ppost }}"
      := (DCWhile b Pbody d Ppost)
      (at level 80, right associativity) : dcom_scope.
Notation "'IFB' b 'THEN' {{ P }} d 'ELSE' {{ P' }} d' 'FI' {{ Q }}"
      := (DCIf b P d P' d' Q)
      (at level 80, right associativity) : dcom_scope.
Notation "'->>' {{ P }} d"
      := (DCPre P d)
      (at level 90, right associativity) : dcom_scope.
Notation "d '->>' {{ P }}"
      := (DCPost d P)
      (at level 80, right associativity) : dcom_scope.
Notation " d ;; d' "
      := (DCSeq d d')
      (at level 80, right associativity) : dcom_scope.
Notation "{{ P }} d"
      := (Decorated P d)
      (at level 90) : dcom_scope.

Delimit Scope dcom_scope with dcom.
```

To avoid clashing with the existing `Notation` definitions for ordinary **com**mands, we introduce these notations in a special scope called *dcom_scope*, and we wrap examples with the declaration % **dcom** to signal that we want the notations to be interpreted in this scope.

Careful readers will note that we've defined two notations for the DCPre constructor, one with and one without a -». The "without" version is intended to be used to supply the initial precondition at the very top of the program.

```
Example dec_while : decorated := (
  {{ fun st ⇒ True }}
  WHILE (BNot (BEq (AId X) (ANum 0)))
  DO
    {{ fun st ⇒ True ∧ st X ≠ 0}}
    X ::= (AMinus (AId X) (ANum 1))
    {{ fun _ ⇒ True }}
  END
  {{ fun st ⇒ True ∧ st X = 0}} -»
  {{ fun st ⇒ st X = 0 }}
) % dcom.
```

It is easy to go from a **dcom** to a **com** by erasing all annotations.

```
Fixpoint extract (d:dcom) : com :=
  match d with
  | DCSkip _ ⇒ SKIP
  | DCSeq d1 d2 ⇒ (extract d1 ;; extract d2)
  | DCAsgn X a _ ⇒ X ::= a
  | DCIf b _ d1 _ d2 _ ⇒ IFB b THEN extract d1 ELSE extract d2 FI
  | DCWhile b _ d _ ⇒ WHILE b DO extract d END
  | DCPre _ d ⇒ extract d
  | DCPost d _ ⇒ extract d
  end.

Definition extract_dec (dec : decorated) : com :=
  match dec with
  | Decorated P d ⇒ extract d
  end.
```

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. But this would result in very verbose programs with a lot of repeated annotations: for example, a program like *SKIP*;*SKIP* would have to be annotated as

[208] ([209] SKIP [210]) ;; ([211] SKIP [212]) [213],

with pre- and post-conditions on each *SKIP*, plus identical pre- and post-conditions on the semicolon!

Instead, the rule we've followed is this:

- The *post*-condition expected by each **dcom** d is embedded in d.
- The *pre*-condition is supplied by the context.

In other words, the invariant of the representation is that a **dcom** d together with a precondition P determines a Hoare triple {{P}} (extract d) {{post d}}, where post is defined as follows:

```
Fixpoint post (d:dcom) : Assertion :=
  match d with
  | DCSkip P ⇒ P
  | DCSeq d1 d2 ⇒ post d2
  | DCAsgn X a Q ⇒ Q
  | DCIf _ _ d1 _ d2 Q ⇒ Q
  | DCWhile b Pbody c Ppost ⇒ Ppost
  | DCPre _ d ⇒ post d
  | DCPost c Q ⇒ Q
  end.
```

It is straightforward to extract the precondition and postcondition from a decorated program.

```
Definition pre_dec (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d ⇒ P
  end.
```

```
Definition post_dec (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d ⇒ post d
  end.
```

We can express what it means for a decorated program to be correct as follows:

```
Definition dec_correct (dec : decorated) :=
  {{pre_dec dec}} (extract_dec dec) {{post_dec dec}}.
```

[208] P
[209] P
[210] P
[211] P
[212] P
[213] P

To check whether this Hoare triple is *valid*, we need a way to extract the "proof obligations" from a decorated program. These obligations are often called *verification conditions*, because they are the facts that must be verified to see that the decorations are logically consistent and thus add up to a complete proof of correctness.

20.5.2 Extracting Verification Conditions

The function verification_conditions takes a **dcom** d together with a precondition P and returns a *proposition* that, if it can be proved, implies that the triple $\{\{P\}\}$ (extract d) $\{\{$post d$\}\}$ is valid.

It does this by walking over d and generating a big conjunction including all the "local checks" that we listed when we described the informal rules for decorated programs. (Strictly speaking, we need to massage the informal rules a little bit to add some uses of the rule of consequence, but the correspondence should be clear.)

```
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 ⇒
      ((fun st ⇒ P st ∧ bassn b st) ->> P1)
      ∧ ((fun st ⇒ P st ∧ ¬ (bassn b st)) ->> P2)
      ∧ (post d1 ->> Q) ∧ (post d2 ->> Q)
      ∧ verification_conditions P1 d1
      ∧ verification_conditions P2 d2
  | DCWhile b Pbody d Ppost ⇒

      (P ->> post d)
      ∧ ((fun st ⇒ post d st ∧ bassn b st) ->> Pbody)
      ∧ ((fun st ⇒ post d st ∧ ~(bassn b st)) ->> Ppost)
      ∧ verification_conditions Pbody d
  | DCPre P' d ⇒
      (P ->> P') ∧ verification_conditions P' d
  | DCPost d Q ⇒
      verification_conditions P d ∧ (post d ->> Q)
  end.
```

And now the key theorem, stating that verification_conditions does its job correctly. Not surprisingly, we need to use each of the Hoare Logic rules at some point in the proof.

```
Theorem verification_correct : ∀ d P,
  verification_conditions P d → {{P}} (extract d) {{post d}}.
Proof.
  induction d; intros P H; simpl in *.
  -
    eapply hoare_consequence_pre.
      apply hoare_skip.
      assumption.
  -
    inversion H as [H1 H2]. clear H.
    eapply hoare_seq.
      apply IHd2. apply H2.
      apply IHd1. apply H1.
  -
    eapply hoare_consequence_pre.
      apply hoare_asgn.
      assumption.
  -
    inversion H as [HPre1 [HPre2 [Hd1 [Hd2 [HThen HElse]]]]].
    clear H.
    apply IHd1 in HThen. clear IHd1.
    apply IHd2 in HElse. clear IHd2.
    apply hoare_if.
      + eapply hoare_consequence_post with (Q':=post d1); eauto.
         eapply hoare_consequence_pre; eauto.
      + eapply hoare_consequence_post with (Q':=post d2); eauto.
         eapply hoare_consequence_pre; eauto.
  -
    inversion H as [Hpre [Hbody1 [Hpost1 Hd]]]. clear H.
    eapply hoare_consequence_pre; eauto.
    eapply hoare_consequence_post; eauto.
    apply hoare_while.
    eapply hoare_consequence_pre; eauto.
  -
    inversion H as [HP Hd]; clear H.
    eapply hoare_consequence_pre. apply IHd. apply Hd. assumption.
  -
    inversion H as [Hd HQ]; clear H.
    eapply hoare_consequence_post. apply IHd. apply Hd. assumption.
Qed.
```

(If you expand the proof, you'll see that it uses an unfamiliar idiom: `simpl in *`. We have used ...in... variants of several tactics before, to apply them to values in the context rather than the goal. The syntax *tactic* in * extends this idea, applying *tactic* in the goal and every hypothesis in the context.)

20.5.3 Automation

Now that all the pieces are in place, we can verify an entire program.

```
Definition verification_conditions_dec (dec : decorated) : Prop :=
  match dec with
  | Decorated P d ⇒ verification_conditions P d
  end.

Lemma verification_correct_dec : ∀ dec,
  verification_conditions_dec dec → dec_correct dec.
Proof.
  intros [P d]. apply verification_correct.
Qed.
```

The propositions generated by verification_conditions are fairly big, and they contain many conjuncts that are essentially trivial.

```
Eval simpl in (verification_conditions_dec dec_while).
```

==> (((fun _: state => True) -» (fun _: state => True)) /\ ((fun st : state => True /\ bassn (BNot (BEq (AId X) (ANum 0))) st) -» (fun st : state => True /\ st X <> 0)) /\ ((fun st : state => True /\ ~ bassn (BNot (BEq (AId X) (ANum 0))) st) -» (fun st : state => True /\ st X = 0)) /\ (fun st : state => True /\ st X <> 0) -» (fun _: state => True) X |-> AMinus (AId X) (ANum 1)) /\ (fun st : state => True /\ st X = 0) -» (fun st : state => st X = 0)

In principle, we could work with such propositions using just the tactics we have so far, but we can make things much smoother with a bit of automation. We first define a custom *verify* tactic that uses split repeatedly to turn all the conjunctions into separate subgoals and then uses omega and eauto (described in chapter Auto) to deal with as many of them as possible.

```
Tactic Notation "verify" :=
  apply verification_correct;
  repeat split;
  simpl; unfold assert_implies;
  unfold bassn in *; unfold beval in *; unfold aeval in *;
  unfold assn_sub; intros;
  repeat rewrite t_update_eq;
  repeat (rewrite t_update_neq; [| (intro X; inversion X)]);
  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 beq_nat_true_iff in *;
  repeat rewrite beq_nat_false_iff in *;
  repeat rewrite leb_iff in *;
  repeat rewrite leb_iff_conv in *;
  try subst;
  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 omega.
```

What's left after *verify* does its thing is "just the interesting parts" of checking that the decorations are correct. For very simple examples *verify* immediately solves the goal (provided that the annotations are correct).

```
Theorem dec_while_correct :
  dec_correct dec_while.
Proof. verify. Qed.
```

Another example (formalizing a decorated program we've seen before):

```
Example subtract_slowly_dec (m:nat) (p:nat) : decorated := (
    {{ fun st ⇒ st X = m ∧ st Z = p }} ->»
    {{ fun st ⇒ st Z - st X = p - m }}
  WHILE BNot (BEq (AId X) (ANum 0))
  DO {{ fun st ⇒ st Z - st X = p - m ∧ st X ≠ 0 }} ->»
       {{ fun st ⇒ (st Z - 1) - (st X - 1) = p - m }}
     Z ::= AMinus (AId Z) (ANum 1)
       {{ fun st ⇒ st Z - (st X - 1) = p - m }} ;;
     X ::= AMinus (AId X) (ANum 1)
       {{ fun st ⇒ st Z - st X = p - m }}
  END
    {{ fun st ⇒ st Z - st X = p - m ∧ st X = 0 }} ->»
    {{ fun st ⇒ st Z = p - m }}
) % dcom.

Theorem subtract_slowly_dec_correct : ∀ m p,
  dec_correct (subtract_slowly_dec m p).
Proof. intros m p. verify. Qed.
```

20.5.4 Examples

In this section, we use the automation developed above to verify formal decorated programs corresponding to most of the informal ones we have seen.

Swapping Using Addition and Subtraction

```
Definition swap : com :=
  X ::= APlus (AId X) (AId Y);;
  Y ::= AMinus (AId X) (AId Y);;
  X ::= AMinus (AId X) (AId Y).

Definition swap_dec m n : decorated :=
  ({{ fun st ⇒ st X = m ∧ st Y = n}} -»
   {{ fun st ⇒ (st X + st Y) - ((st X + st Y) - st Y) = n
                  ∧ (st X + st Y) - st Y = m }}
  X ::= APlus (AId X) (AId Y)
   {{ fun st ⇒ st X - (st X - st Y) = n ∧ st X - st Y = m }};;
  Y ::= AMinus (AId X) (AId Y)
   {{ fun st ⇒ st X - st Y = n ∧ st Y = m }};;
  X ::= AMinus (AId X) (AId Y)
   {{ fun st ⇒ st X = n ∧ st Y = m}})%dcom.

Theorem swap_correct : ∀ m n,
  dec_correct (swap_dec m n).
Proof. intros; verify. Qed.
```

Simple Conditionals

```
Definition if_minus_plus_com :=
  IFB (BLe (AId X) (AId Y))
    THEN (Z ::= AMinus (AId Y) (AId X))
    ELSE (Y ::= APlus (AId X) (AId Z))
  FI.

Definition if_minus_plus_dec :=
  ({{fun st ⇒ True}}
  IFB (BLe (AId X) (AId Y)) THEN
      {{ fun st ⇒ True ∧ st X ≤ st Y }} -»
      {{ fun st ⇒ st Y = st X + (st Y - st X) }}
    Z ::= AMinus (AId Y) (AId X)
      {{ fun st ⇒ st Y = st X + st Z }}
  ELSE
      {{ fun st ⇒ True ∧ ~(st X ≤ st Y) }} -»
      {{ fun st ⇒ st X + st Z = st X + st Z }}
```

```
      Y ::= APlus (AId X) (AId Z)
        {{ fun st ⇒ st Y = st X + st Z }}
  FI
  {{fun st ⇒ st Y = st X + st Z}})%dcom.
```

```
Theorem if_minus_plus_correct :
  dec_correct if_minus_plus_dec.
Proof. intros; verify. Qed.
```

```
Definition if_minus_dec :=
  ( {{fun st ⇒ True}}
  IFB (BLe (AId X) (AId Y)) THEN
        {{fun st ⇒ True ∧ st X ≤ st Y }} ->>
        {{fun st ⇒ (st Y - st X) + st X = st Y
                  ∨ (st Y - st X) + st Y = st X}}
      Z ::= AMinus (AId Y) (AId X)
        {{fun st ⇒ st Z + st X = st Y ∨ st Z + st Y = st X}}
  ELSE
        {{fun st ⇒ True ∧ ~(st X ≤ st Y) }} ->>
        {{fun st ⇒ (st X - st Y) + st X = st Y
                  ∨ (st X - st Y) + st Y = st X}}
      Z ::= AMinus (AId X) (AId Y)
        {{fun st ⇒ st Z + st X = st Y ∨ st Z + st Y = st X}}
  FI
    {{fun st ⇒ st Z + st X = st Y ∨ st Z + st Y = st X}})%dcom.
```

```
Theorem if_minus_correct :
  dec_correct if_minus_dec.
Proof. verify. Qed.
```

Division

```
Definition div_mod_dec (a b : nat) : decorated := (
{{ fun st ⇒ True }} ->>
  {{ fun st ⇒ b × 0 + a = a }}
  X ::= ANum a
  {{ fun st ⇒ b × 0 + st X = a }};;
  Y ::= ANum 0
  {{ fun st ⇒ b × st Y + st X = a }};;
  WHILE (BLe (ANum b) (AId X)) DO
    {{ fun st ⇒ b × st Y + st X = a ∧ b ≤ st X }} ->>
    {{ fun st ⇒ b × (st Y + 1) + (st X - b) = a }}
    X ::= AMinus (AId X) (ANum b)
    {{ fun st ⇒ b × (st Y + 1) + st X = a }};;
    Y ::= APlus (AId Y) (ANum 1)
```

```
    {{ fun st ⇒ b × st Y + st X = a }}
  END
  {{ fun st ⇒ b × st Y + st X = a ∧ ~(b ≤ st X) }} -»
  {{ fun st ⇒ b × st Y + st X = a ∧ (st X < b) }}
)%dcom.
```

```
Theorem div_mod_dec_correct : ∀ a b,
  dec_correct (div_mod_dec a b).
Proof. intros a b. verify.
  rewrite mult_plus_distr_l. omega.
Qed.
```

Parity

```
Definition find_parity : com :=
  WHILE (BLe (ANum 2) (AId X)) DO
     X ::= AMinus (AId X) (ANum 2)
  END.
```

There are actually several ways to phrase the loop invariant for this program. Here is one natural one, which leads to a rather long proof:

```
Inductive ev : nat → Prop :=
  | ev_0 : ev O
  | ev_SS : ∀ n:nat, ev n → ev (S (S n)).
```

```
Definition find_parity_dec m : decorated :=
  ({{ fun st ⇒ st X = m}} -»
   {{ fun st ⇒ st X ≤ m ∧ ev (m - st X) }}
  WHILE (BLe (ANum 2) (AId X)) DO
     {{ fun st ⇒ (st X ≤ m ∧ ev (m - st X)) ∧ 2 ≤ st X }} -»
     {{ fun st ⇒ st X - 2 ≤ m ∧ (ev (m - (st X - 2))) }}
     X ::= AMinus (AId X) (ANum 2)
     {{ fun st ⇒ st X ≤ m ∧ ev (m - st X) }}
  END
   {{ fun st ⇒ (st X ≤ m ∧ ev (m - st X)) ∧ st X < 2 }} -»
   {{ fun st ⇒ st X=0 ↔ ev m }})%dcom.
```

```
Lemma l1 : ∀ m n p,
  p ≤ n →
  n ≤ m →
  m - (n - p) = m - n + p.
Proof. intros. omega. Qed.
```

```
Lemma l2 : ∀ m,
  ev m →
  ev (m + 2).
```

```
Proof. intros. rewrite plus_comm. simpl. constructor. assumption. Qed.

Lemma l3' : ∀ m,
  ev m →
  ¬ev (S m).
Proof. induction m; intros H1 H2. inversion H2. apply IHm.
       inversion H2; subst; assumption. assumption. Qed.

Lemma l3 : ∀ m,
  1 ≤ m →
  ev m →
  ev (m - 1) →
  False.
Proof. intros. apply l2 in H1.
       assert (G : m - 1 + 2 = S m). clear H0 H1. omega.
       rewrite G in H1. apply l3' in H0. apply H0. assumption. Qed.

Theorem find_parity_correct : ∀ m,
  dec_correct (find_parity_dec m).
Proof.
  intro m. verify;

    fold (leb 2 (st X)) in *;
    try rewrite leb_iff in *;
    try rewrite leb_iff_conv in *; eauto; try omega.
    -
      rewrite minus_diag. constructor.
    -
      rewrite l1; try assumption.
      apply l2; assumption.
    -
      rewrite ← minus_n_O in H2. assumption.
    -
      destruct (st X) as [| [| n]].
      +
        reflexivity.
      +
        apply l3 in H; try assumption. inversion H.
      +
        clear H0 H2.             omega.
Qed.
```

Here is a more intuitive way of writing the invariant:

```
Definition find_parity_dec' m : decorated :=
 ({{ fun st ⇒ st X = m}} -»
```

```
  {{ fun st ⇒ ev (st X) ↔ ev m }}
 WHILE (BLe (ANum 2) (AId X)) DO
     {{ fun st ⇒ (ev (st X) ↔ ev m) ∧ 2 ≤ st X }} -»
     {{ fun st ⇒ (ev (st X - 2) ↔ ev m) }}
     X ::= AMinus (AId X) (ANum 2)
     {{ fun st ⇒ (ev (st X) ↔ ev m) }}
 END
 {{ fun st ⇒ (ev (st X) ↔ ev m) ∧ ~(2 ≤ st X) }} -»
 {{ fun st ⇒ st X=0 ↔ ev m }})%dcom.

Lemma l4 : ∀ m,
  2 ≤ m →
  (ev (m - 2) ↔ ev m).
Proof.
  induction m; intros. split; intro; constructor.
  destruct m. inversion H. inversion H1. simpl in *.
  rewrite ← minus_n_O in *. split; intro.
    constructor. assumption.
    inversion H0. assumption.
Qed.

Theorem find_parity_correct' : ∀ m,
  dec_correct (find_parity_dec' m).
Proof.
  intros m. verify;

    fold (leb 2 (st X)) in *;
    try rewrite leb_iff in *;
    try rewrite leb_iff_conv in *; intuition; eauto; try omega.
  -
    rewrite l4 in H0; eauto.
  -
    rewrite l4; eauto.
  -
    apply H0. constructor.
  -
      destruct (st X) as [| [| n]].          +
        reflexivity.
      +
        inversion H.
      +
        clear H0 H H3.             omega.
Qed.
```

Here is the simplest invariant we've found for this program:

```
Definition parity_dec m : decorated :=
 ({{ fun st ⇒ st X = m}} -»
  {{ fun st ⇒ parity (st X) = parity m }}
  WHILE (BLe (ANum 2) (AId X)) DO
     {{ fun st ⇒ parity (st X) = parity m ∧ 2 ≤ st X }} -»
     {{ fun st ⇒ parity (st X - 2) = parity m }}
     X ::= AMinus (AId X) (ANum 2)
     {{ fun st ⇒ parity (st X) = parity m }}
  END
 {{ fun st ⇒ parity (st X) = parity m ∧ ~(2 ≤ st X) }} -»
 {{ fun st ⇒ st X = parity m }})%dcom.

Theorem parity_dec_correct : ∀ m,
  dec_correct (parity_dec m).
Proof.
  intros. verify;

    fold (leb 2 (st X)) in *;
    try rewrite leb_iff in *;
    try rewrite leb_iff_conv in *; eauto; try omega.
  -
    rewrite ← H. apply parity_ge_2. assumption.
  -
    rewrite ← H. symmetry. apply parity_lt_2. assumption.
Qed.
```

Square Roots

```
Definition sqrt_dec m : decorated := (
    {{ fun st ⇒ st X = m }} -»
    {{ fun st ⇒ st X = m ∧ 0*0 ≤ m }}
  Z ::= ANum 0
    {{ fun st ⇒ st X = m ∧ st Z×st Z ≤ m }};;
  WHILE BLe (AMult (APlus (AId Z) (ANum 1))
                   (APlus (AId Z) (ANum 1)))
            (AId X) DO
      {{ fun st ⇒ (st X = m ∧ st Z×st Z≤m)
                   ∧ (st Z + 1)*(st Z + 1) ≤ st X }} -»
      {{ fun st ⇒ st X = m ∧ (st Z+1)*(st Z+1)<=m }}
    Z ::= APlus (AId Z) (ANum 1)
      {{ fun st ⇒ st X = m ∧ st Z×st Z≤m }}
  END
    {{ fun st ⇒ (st X = m ∧ st Z×st Z≤m)
```

```
                     ∧ ~((st Z + 1)*(st Z + 1) ≤ st X) }} ->>
    {{ fun st ⇒ st Z×st Z≤m ∧ m<(st Z+1)*(st Z+1) }})%dcom.
```

```
Theorem sqrt_correct : ∀ m,
  dec_correct (sqrt_dec m).
Proof. intro m. verify. Qed.
```

Squaring

Again, there are several ways of annotating the squaring program. The simplest variant we've found, square_simpler_dec, is given last.

```
Definition square_dec (m : nat) : decorated := (
  {{ fun st ⇒ st X = m }}
  Y ::= AId X
  {{ fun st ⇒ st X = m ∧ st Y = m }};;
  Z ::= ANum 0
  {{ fun st ⇒ st X = m ∧ st Y = m ∧ st Z = 0}} ->>
  {{ fun st ⇒ st Z + st X × st Y = m × m }};;
  WHILE BNot (BEq (AId Y) (ANum 0)) DO
    {{ fun st ⇒ st Z + st X × st Y = m × m ∧ st Y ≠ 0 }} ->>
    {{ fun st ⇒ (st Z + st X) + st X × (st Y - 1) = m × m }}
    Z ::= APlus (AId Z) (AId X)
    {{ fun st ⇒ st Z + st X × (st Y - 1) = m × m }};;
    Y ::= AMinus (AId Y) (ANum 1)
    {{ fun st ⇒ st Z + st X × st Y = m × m }}
  END
  {{ fun st ⇒ st Z + st X × st Y = m × m ∧ st Y = 0 }} ->>
  {{ fun st ⇒ st Z = m × m }}
)%dcom.
```

```
Theorem square_dec_correct : ∀ m,
  dec_correct (square_dec m).
Proof.
  intro n. verify.
  -
    destruct (st Y) as [| y']. apply False_ind. apply H0.
    reflexivity.
    simpl. rewrite ← minus_n_O.
    assert (G : ∀ n m, n × S m = n + n × m). {
      clear. intros. induction n. reflexivity. simpl.
      rewrite IHn. omega. }
    rewrite ← H. rewrite G. rewrite plus_assoc. reflexivity.
Qed.
```

```
Definition square_dec' (n : nat) : decorated := (
```

```
    {{ fun st ⇒ True }}
    X ::= ANum n
    {{ fun st ⇒ st X = n }};;
    Y ::= AId X
    {{ fun st ⇒ st X = n ∧ st Y = n }};;
    Z ::= ANum 0
    {{ fun st ⇒ st X = n ∧ st Y = n ∧ st Z = 0 }} -»
    {{ fun st ⇒ st Z = st X × (st X - st Y)
                      ∧ st X = n ∧ st Y ≤ st X }};;
    WHILE BNot (BEq (AId Y) (ANum 0)) DO
      {{ fun st ⇒ (st Z = st X × (st X - st Y)
                      ∧ st X = n ∧ st Y ≤ st X)
                       ∧ st Y ≠ 0 }}
      Z ::= APlus (AId Z) (AId X)
      {{ fun st ⇒ st Z = st X × (st X - (st Y - 1))
                       ∧ st X = n ∧ st Y ≤ st X }};;
      Y ::= AMinus (AId Y) (ANum 1)
      {{ fun st ⇒ st Z = st X × (st X - st Y)
                       ∧ st X = n ∧ st Y ≤ st X }}
    END
    {{ fun st ⇒ (st Z = st X × (st X - st Y)
                    ∧ st X = n ∧ st Y ≤ st X)
                     ∧ st Y = 0 }} -»
    {{ fun st ⇒ st Z = n × n }}
)%dcom.

Theorem square_dec'_correct : ∀ n,
    dec_correct (square_dec' n).
Proof.
    intro n. verify.
    -
      rewrite minus_diag. omega.
    - subst.
      rewrite mult_minus_distr_l.
      repeat rewrite mult_minus_distr_l. rewrite mult_1_r.
      assert (G : ∀ n m p,
                       m ≤ n → p ≤ m → n - (m - p) = n - m + p).
        intros. omega.
      rewrite G. reflexivity. apply mult_le_compat_l. assumption.
      destruct (st Y). apply False_ind. apply H0. reflexivity.
        clear. rewrite mult_succ_r. rewrite plus_comm.
        apply le_plus_l.
    -
```

```
    rewrite ← minus_n_O. reflexivity.
Qed.

Definition square_simpler_dec (m : nat) : decorated := (
  {{ fun st ⇒ st X = m }} -»
  {{ fun st ⇒ 0 = 0*m ∧ st X = m }}
  Y ::= ANum 0
  {{ fun st ⇒ 0 = (st Y)*m ∧ st X = m }};;
  Z ::= ANum 0
  {{ fun st ⇒ st Z = (st Y)*m ∧ st X = m }}-»
  {{ fun st ⇒ st Z = (st Y)*m ∧ st X = m }};;
  WHILE BNot (BEq (AId Y) (AId X)) DO
    {{ fun st ⇒ (st Z = (st Y)*m ∧ st X = m)
        ∧ st Y ≠ st X }} -»
    {{ fun st ⇒ st Z + st X = ((st Y) + 1)*m ∧ st X = m }}
    Z ::= APlus (AId Z) (AId X)
    {{ fun st ⇒ st Z = ((st Y) + 1)*m ∧ st X = m }};;
    Y ::= APlus (AId Y) (ANum 1)
    {{ fun st ⇒ st Z = (st Y)*m ∧ st X = m }}
  END
  {{ fun st ⇒ (st Z = (st Y)*m ∧ st X = m) ∧ st Y = st X }} -»
  {{ fun st ⇒ st Z = m×m }}
)%dcom.

Theorem square_simpler_dec_correct : ∀ m,
  dec_correct (square_simpler_dec m).
Proof.
  intro m. verify.
  rewrite mult_plus_distr_r. simpl. rewrite ← plus_n_O.
  reflexivity.
Qed.
```

Two loops

```
Definition two_loops_dec (a b c : nat) : decorated :=
( {{ fun st ⇒ True }} -»
  {{ fun st ⇒ c = 0 + c ∧ 0 = 0 }}
  X ::= ANum 0
  {{ fun st ⇒ c = st X + c ∧ 0 = 0 }};;
  Y ::= ANum 0
  {{ fun st ⇒ c = st X + c ∧ st Y = 0 }};;
  Z ::= ANum c
  {{ fun st ⇒ st Z = st X + c ∧ st Y = 0 }};;
  WHILE BNot (BEq (AId X) (ANum a)) DO
```

```
      {{ fun st ⇒ (st Z = st X + c ∧ st Y = 0) ∧ st X ≠ a }} -»
      {{ fun st ⇒ st Z + 1 = st X + 1 + c ∧ st Y = 0 }}
      X ::= APlus (AId X) (ANum 1)
      {{ fun st ⇒ st Z + 1 = st X + c ∧ st Y = 0 }};;
      Z ::= APlus (AId Z) (ANum 1)
      {{ fun st ⇒ st Z = st X + c ∧ st Y = 0 }}
    END
    {{ fun st ⇒ (st Z = st X + c ∧ st Y = 0) ∧ st X = a }} -»
    {{ fun st ⇒ st Z = a + st Y + c }};;
    WHILE BNot (BEq (AId Y) (ANum b)) DO
      {{ fun st ⇒ st Z = a + st Y + c ∧ st Y ≠ b }} -»
      {{ fun st ⇒ st Z + 1 = a + st Y + 1 + c }}
      Y ::= APlus (AId Y) (ANum 1)
      {{ fun st ⇒ st Z + 1 = a + st Y + c }};;
      Z ::= APlus (AId Z) (ANum 1)
      {{ fun st ⇒ st Z = a + st Y + c }}
    END
    {{ fun st ⇒ (st Z = a + st Y + c) ∧ st Y = b }} -»
    {{ fun st ⇒ st Z = a + b + c }}
)%dcom.

Theorem two_loops_correct : ∀ a b c,
    dec_correct (two_loops_dec a b c).
Proof. intros a b c. verify. Qed.
```

Power Series

```
Fixpoint pow2 n :=
  match n with
  | 0 ⇒ 1
  | S n' ⇒ 2 × (pow2 n')
  end.

Definition dpow2_down n :=
( {{ fun st ⇒ True }} -»
  {{ fun st ⇒ 1 = (pow2 (0 + 1))-1 ∧ 1 = pow2 0 }}
  X ::= ANum 0
  {{ fun st ⇒ 1 = (pow2 (0 + 1))-1 ∧ 1 = pow2 (st X) }};;
  Y ::= ANum 1
  {{ fun st ⇒ st Y = (pow2 (st X + 1))-1 ∧ 1 = pow2 (st X) }};;
  Z ::= ANum 1
  {{ fun st ⇒ st Y = (pow2 (st X + 1))-1 ∧ st Z = pow2 (st X) }};;
  WHILE BNot (BEq (AId X) (ANum n)) DO
    {{ fun st ⇒ (st Y = (pow2 (st X + 1))-1 ∧ st Z = pow2 (st X))
```

```
                      ∧ st X ≠ n }} -»
    {{ fun st ⇒ st Y + 2 × st Z = (pow2 (st X + 2))-1
                      ∧ 2 × st Z = pow2 (st X + 1) }}
    Z ::= AMult (ANum 2) (AId Z)
    {{ fun st ⇒ st Y + st Z = (pow2 (st X + 2))-1
                      ∧ st Z = pow2 (st X + 1) }};;
    Y ::= APlus (AId Y) (AId Z)
    {{ fun st ⇒ st Y = (pow2 (st X + 2))-1
                      ∧ st Z = pow2 (st X + 1) }};;
    X ::= APlus (AId X) (ANum 1)
    {{ fun st ⇒ st Y = (pow2 (st X + 1))-1
                      ∧ st Z = pow2 (st X) }}
  END
  {{ fun st ⇒ (st Y = (pow2 (st X + 1))-1 ∧ st Z = pow2 (st X))
                  ∧ st X = n }} -»
  {{ fun st ⇒ st Y = pow2 (n+1) - 1 }}
)%dcom.

Lemma pow2_plus_1 : ∀ n,
  pow2 (n+1) = pow2 n + pow2 n.
Proof. induction n; simpl. reflexivity. omega. Qed.

Lemma pow2_le_1 : ∀ n, pow2 n ≥ 1.
Proof. induction n. simpl. constructor. simpl. omega. Qed.

Theorem dpow2_down_correct : ∀ n,
  dec_correct (dpow2_down n).
Proof.
  intro m. verify.
  -
    rewrite pow2_plus_1. rewrite ← H0. reflexivity.
  -
    rewrite ← plus_n_O.
    rewrite ← pow2_plus_1. remember (st X) as n.
    replace (pow2 (n + 1) - 1 + pow2 (n + 1))
       with (pow2 (n + 1) + pow2 (n + 1) - 1) by omega.
    rewrite ← pow2_plus_1.
    replace (n + 1 + 1) with (n + 2) by omega.
    reflexivity.
  -
    rewrite ← plus_n_O. rewrite ← pow2_plus_1.
    reflexivity.
  -
    replace (st X + 1 + 1) with (st X + 2) by omega.
    reflexivity.
```

```
Qed.
```

Further Exercises

Exercise: 3 stars, advanced (slow_assignment_dec) In the *slow_assignment* exercise above, we saw a roundabout way of assigning a number currently stored in X to the variable Y: start Y at 0, then decrement X until it hits 0, incrementing Y at each step. Write a formal version of this decorated program and prove it correct.

```
Example slow_assignment_dec (m:nat) : decorated
  . Admitted.

Theorem slow_assignment_dec_correct : ∀ m,
  dec_correct (slow_assignment_dec m).
Proof. Admitted.
```

☐

Exercise: 4 stars, advancedM (factorial_dec) Remember the factorial function we worked with before:

```
Fixpoint real_fact (n:nat) : nat :=
  match n with
  | O ⇒ 1
  | S n' ⇒ n × (real_fact n')
  end.
```

Following the pattern of subtract_slowly_dec, write a decorated program *factorial_dec* that implements the factorial function and prove it correct as *factorial_dec_correct*.

☐

Exercise: 4 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 Coq'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:

```
Lemma fib_eqn : ∀ n,
  n > 0 →
  fib n + fib (Init.Nat.pred n) = fib (n + 1).
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 <> n+1 DO T ::= Z; Z ::= Z + Y;; Y ::= T;; X ::= X + 1 END

Fill in the following definition of dfib and prove that it satisfies this specification:

[214] dfib [215]

```
Definition T : id := Id "T".

Definition dfib (n:nat) : decorated
. Admitted.

Theorem dfib_correct : ∀ n,
  dec_correct (dfib n).
   Admitted.
```
□

Exercise: 5 stars, advanced, optional (improve_dcom) The formal decorated programs defined in this section are intended to look as similar as possible to the informal ones defined earlier in the chapter. 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.

□

Exercise: 4 stars, advanced, optional (implement_dcom) Adapt the parser for Imp presented in chapter ImpParser to parse decorated commands (either ours or the ones you defined in the previous exercise).

□

$Date : 2016-12-2011:20:02-0500(Tue, 20Dec2016)$

[214] True
[215] Y=fibn

Chapter 21

Library Top.HoareAsLogic

21.1 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 Coq.

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.

```
Require Import Imp.
Require Import Hoare.
```

21.2 Definitions

```
Inductive hoare_proof : Assertion → com → Assertion → Type :=
  | H_Skip : ∀ P,
      hoare_proof P (SKIP) P
  | H_Asgn : ∀ Q V a,
      hoare_proof (assn_sub V a Q) (V ::= a) Q
  | H_Seq : ∀ P c Q d R,
      hoare_proof P c Q → hoare_proof Q d R → hoare_proof P (c;;d) R
  | H_If : ∀ P Q b c1 c2,
    hoare_proof (fun st ⇒ P st ∧ bassn b st) c1 Q →
    hoare_proof (fun st ⇒ P st ∧ ~(bassn b st)) c2 Q →
    hoare_proof P (IFB b THEN c1 ELSE c2 FI) Q
  | H_While : ∀ P b c,
```

```
      hoare_proof (fun st ⇒ P st ∧ bassn b st) c P →
      hoare_proof P (WHILE b DO c END) (fun st ⇒ P st ∧ ¬ (bassn b st))
  | H_Consequence : ∀ (P Q P' Q' : Assertion) c,
      hoare_proof P' c Q' →
      (∀ st, P st → P' st) →
      (∀ st, Q' st → Q st) →
      hoare_proof 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,
    hoare_proof P' c Q →
    (∀ st, P st → P' st) →
    hoare_proof P c Q.
Proof.
  intros. eapply H_Consequence.
    apply X. apply H. intros. apply H0. Qed.

Lemma H_Consequence_post : ∀ (P Q Q' : Assertion) c,
    hoare_proof P c Q' →
    (∀ st, Q' st → Q st) →
    hoare_proof P c Q.
Proof.
  intros. eapply H_Consequence.
    apply X. intros. apply H0. apply H. Qed.
```

As an example, let's construct a proof object representing a derivation for the hoare triple
[1] X::=X+1 ;; X::=X+2 [2].

We can use Coq's tactics to help us construct the proof object.

```
Example sample_proof :
  hoare_proof
    (assn_sub X (APlus (AId X) (ANum 1))
                (assn_sub X (APlus (AId X) (ANum 2))
                            (fun st ⇒ st X = 3) ))
    (X ::= APlus (AId X) (ANum 1);; (X ::= APlus (AId X) (ANum 2)))
    (fun st ⇒ st X = 3).
Proof.
  eapply H_Seq; apply H_Asgn.
Qed.
```

[1] assn_subX(X+1)(assn_subX(X+2)(X=3))

[2] X=3

21.3 Properties

Exercise: 2 stars (hoare_proof_sound) Prove that such proof objects represent true claims.

```
Theorem hoare_proof_sound : ∀ P c Q,
  hoare_proof P c Q → {{P}} c {{Q}}.
Proof.
   Admitted.
```

□

We can also use Coq's reasoning facilities to prove metatheorems about Hoare Logic. For example, here are the analogs of two theorems we saw in chapter Hoare – this time expressed in terms of the syntax of Hoare Logic derivations (provability) rather than directly in terms of the semantics of Hoare triples.

The first one says that, for every P and c, the assertion {{P}} c {{True}} is *provable* in Hoare Logic. Note that the proof is more complex than the semantic proof in Hoare: we actually need to perform an induction over the structure of the command c.

```
Theorem H_Post_True_deriv:
  ∀ c P, hoare_proof P c (fun _ ⇒ True).
Proof.
  intro c.
  induction c; intro P.
  -
    eapply H_Consequence.
    apply H_Skip.
    intros. apply H.
    intros. apply I.
  -
    eapply H_Consequence_pre.
    apply H_Asgn.
    intros. apply I.
  -
    eapply H_Consequence_pre.
    eapply H_Seq.
    apply (IHc1 (fun _ ⇒ True)).
    apply IHc2.
    intros. apply I.
  -
    apply H_Consequence_pre with (fun _ ⇒ True).
    apply H_If.
    apply IHc1.
    apply IHc2.
    intros. apply I.
```

```
  -
    eapply H_Consequence.
    eapply H_While.
    eapply IHc.
    intros; apply I.
    intros; apply I.
Qed.
```

Similarly, we can show that {{**False**}} c {{Q}} is provable for any c and Q.

```
Lemma False_and_P_imp: ∀ P Q,
  False ∧ P → Q.
Proof.
  intros P Q [CONTRA HP].
  destruct CONTRA.
Qed.

Tactic Notation "pre_false_helper" constr(CONSTR) :=
  eapply H_Consequence_pre;
    [eapply CONSTR | intros ? CONTRA; destruct CONTRA].

Theorem H_Pre_False_deriv:
  ∀ c Q, hoare_proof (fun _ ⇒ False) c Q.
Proof.
  intros c.
  induction c; intro Q.
  - pre_false_helper H_Skip.
  - pre_false_helper H_Asgn.
  - pre_false_helper H_Seq. apply IHc1. apply IHc2.
  -
    apply H_If; eapply H_Consequence_pre.
    apply IHc1. intro. eapply False_and_P_imp.
    apply IHc2. intro. eapply False_and_P_imp.
  -
    eapply H_Consequence_post.
    eapply H_While.
    eapply H_Consequence_pre.
      apply IHc.
      intro. eapply False_and_P_imp.
    intro. simpl. eapply False_and_P_imp.
Qed.
```

As a last step, we can show that the set of **hoare_proof** axioms is sufficient to prove any true fact about (partial) correctness. More precisely, any semantic Hoare triple that we can prove can also be proved from these axioms. Such a set of axioms is said to be *relatively complete*. Our proof is inspired by this one:

http://www.ps.uni-saarland.de/courses/sem-ws11/script/Hoare.html

To carry out the proof, we need to invent some intermediate assertions using a technical device known as *weakest preconditions*. 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 $P' \to P$. We can more directly define this as follows:

```
Definition wp (c:com) (Q:Assertion) : Assertion :=
  fun s ⇒ ∀ s', c / s \\ s' → Q s'.
```

Exercise: 1 star (wp_is_precondition) Lemma *wp_is_precondition*: ∀ *c Q*,

{{*wp c Q*}} *c* {{*Q*}}.

Admitted.

□

Exercise: 1 star (wp_is_weakest) Lemma *wp_is_weakest*: ∀ *c Q P'*,

{{*P'*}} *c* {{*Q*}} → ∀ *st*, *P' st* → *wp c Q st*.

Admitted.

The following utility lemma will also be useful.

```
Lemma bassn_eval_false : ∀ b st, ¬ bassn b st → beval st b = false.
Proof.
  intros b st H. unfold bassn in H. destruct (beval st b).
    exfalso. apply H. reflexivity.
    reflexivity.
Qed.
```

□

Exercise: 5 stars (hoare_proof_complete) Complete the proof of the theorem.

```
Theorem hoare_proof_complete: ∀ P c Q,
  {{P}} c {{Q}} → hoare_proof P c Q.
Proof.
  intros P c. generalize dependent P.
  induction c; intros P Q HT.
  -
    eapply H_Consequence.
     eapply H_Skip.
      intros. eassumption.
      intro st. apply HT. apply E_Skip.
  -
    eapply H_Consequence.
      eapply H_Asgn.
```

```
      intro st. apply HT. econstructor. reflexivity.
      intros; assumption.
  -
    apply H_Seq with (wp c2 Q).
     eapply IHc1.
       intros st st' E1 H. unfold wp. intros st'' E2.
         eapply HT. econstructor; eassumption. assumption.
     eapply IHc2. intros st st' E1 H. apply H; assumption.
Admitted.
```

□

Finally, we might hope that our axiomatic Hoare logic is *decidable*; that is, that there is an (terminating) algorithm (a *decision procedure*) that can determine whether or not a given Hoare triple is valid (derivable). But such a decision procedure cannot exist!

Consider the triple {{**True**}} c {{**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 {{**True**} *SKIP* {{*P*}} is valid if and only if ∀ s, *P* s is valid, where *P* is an arbitrary assertion of Coq's logic. But it is known that there can be no decision procedure for this logic.

Overall, this axiomatic style of presentation gives a clearer picture of what it means to "give a proof in Hoare logic." However, it is not entirely satisfactory from the point of view of writing down such proofs in practice: it is quite verbose. The section of chapter Hoare2 on formalizing decorated programs shows how we can do even better.

$Date : 2016 - 11 - 0611 : 48 : 34 - 0500(Sun, 06Nov2016)$

Chapter 22

Library Top.Smallstep

22.1 Smallstep: Small-step Operational Semantics

```
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Lists.List.
Import ListNotations.
Require Import Maps.
Require Import Imp.
```

The evaluators we have seen so far (for **aexp**s, **bexp**s, commands, ...) have been formulated in a "big-step" style: they specify how a given expression can be evaluated to its final value (or a command plus a store 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 natural way of talking about *concurrent* programming languages, where the semantics of a program – i.e., the essence of how it behaves – is not just which input states get mapped to which output states, but also includes the intermediate states that it passes through along the way, since these states can also be observed by concurrently executing code.

Another shortcoming of the big-step style is more technical, but 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 expressions 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 is really a bit of a hack.

A much more natural approach is simply to say that the behavior of an expression like 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 – are quite different. In particular, we want to allow the first (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 for 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. We replace the "big-step" eval relation with a "small-step" relation that specifies, for a given program, how the "atomic steps" of computation are performed.

22.2 A Toy Language

To save space in the discussion, let's go back to an incredibly simple language containing just constants and addition. (We use single letters – C and *P* (for Command 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 a1 a2 ⇒ evalF a1 + evalF a2
  end.
```

Here is the same evaluator, written in exactly the same style, but formulated as an inductively defined relation. Again, we use the notation $t \\ $ n for "t evaluates to n."

(E_Const) C n \\ n
t1 \\ n1 t2 \\ n2

(E_Plus) P t1 t2 \\ n1 + n2

```
Reserved Notation " t '\\' n " (at level 50, left associativity).

Inductive eval : tm → nat → Prop :=
  | E_Const : ∀ n,
      C n \\ n
  | E_Plus : ∀ t1 t2 n1 n2,
      t1 \\ n1 →
      t2 \\ n2 →
      P t1 t2 \\ (n1 + n2)

  where " t '\\' n " := (eval t n).

Module SimpleArith1.
```

Now, here is the corresponding *small-step* evaluation relation.

(ST_PlusConstConst) P (C n1) (C n2) ==> C (n1 + n2)

t1 ==> t1'

(ST_Plus1) P t1 t2 ==> P t1' t2

t2 ==> t2'

(ST_Plus2) P (C n1) t2 ==> P (C n1) t2'

```
Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ n1 t2 t2',
      t2 ==> t2' →
      P (C n1) t2 ==> P (C n1) t2'

  where " t '==>' t' " := (step t t').
```

Things to notice:

- We are defining just a single reduction step, in which 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 0) (C 3))
        (P (C 2) (C 4))
      ==>
      P
        (C (0 + 3))
        (P (C 2) (C 4)).
Proof.
  apply ST_Plus1. apply ST_PlusConstConst. Qed.
```

Exercise: 1 star (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 0) (C 3)))
      ==>
      P
        (C 0)
        (P
          (C 2)
          (C (0 + 3))).
Proof.
   Admitted.
```

☐

```
End SimpleArith1.
```

22.3 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* develops some of these ideas in a bit more detail; it may be useful if the treatment here is too dense.)

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: Type) := X→X→Prop.
```

Our main examples of such relations in this chapter will be the single-step reduction relation, ==>, and its multi-step variant, ==>* (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 ==> 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* ==> *t'* is provable). Formally, this is the same as saying that ==> is deterministic.

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_PlusConstConst, the result is immediate.
- The cases when both derivations end with ST_Plus1 or ST_Plus2 follow by the induction hypothesis.
- It cannot happen that one is ST_PlusConstConst and the other is ST_Plus1 or ST_Plus2, since this would imply that x has the form *P t1 t2* where both *t1* and *t2* are constants (by ST_PlusConstConst) *and* one of *t1* or *t2* has the form *P* _.
- Similarly, it cannot happen that one is ST_Plus1 and the other is ST_Plus2, 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. □

Formally:

```
Definition deterministic {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.

Module SimpleArith2.
Import SimpleArith1.

Theorem step_deterministic:
  deterministic step.
Proof.
  unfold deterministic. intros x y1 y2 Hy1 Hy2.
  generalize dependent y2.
  induction Hy1; intros y2 Hy2.
    - inversion Hy2.
      + reflexivity.
      + inversion H2.
```

```
    + inversion H2.
  - inversion Hy2.
    +
      rewrite ← H0 in Hy1. inversion Hy1.
    +
      rewrite ← (IHHy1 t1'0).
      reflexivity. assumption.
    +
      rewrite ← H in Hy1. inversion Hy1.
  - inversion Hy2.
    +
      rewrite ← H1 in Hy1. inversion Hy1.
    + inversion H2.
    +
      rewrite ← (IHHy1 t2'0).
      reflexivity. assumption.
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 for now, but it illustrates the power of Coq'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.

Let's see how a proof of the previous theorem can be simplified using this tactic...

```
Module SimpleArith3.
Import SimpleArith1.

Theorem step_deterministic_alt: deterministic step.
Proof.
  intros x y1 y2 Hy1 Hy2.
  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.
```

22.3.1 Values

Next, it will be useful to slightly reformulate the definition of single-step reduction by stating it in terms of "values."

It is useful to think of the ==> 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 ==> 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 the machine are always terms of the form `C n` for some `n`. We call such terms *values*.

```
Inductive value : tm → Prop :=
  | v_const : ∀ n, value (C n).
```

Having introduced the idea of values, we can use it in the definition of the ==> relation to write ST_Plus2 rule in a slightly more elegant way:

(ST_PlusConstConst) P (C n1) (C n2) ==> C (n1 + n2)
t1 ==> t1'

(ST_Plus1) P t1 t2 ==> P t1' t2
value v1 t2 ==> t2'

(ST_Plus2) P v1 t2 ==> P v1 t2' Again, the variable names here 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. We'll keep it for now, to maintain a close correspondence between the informal and Coq versions of the rules, but later on we'll drop it in informal rules for brevity.)

Here are the formal rules:

```
Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
          P (C n1) (C n2)
      ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
        t1 ==> t1' →
        P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ v1 t2 t2',
        value v1 →
        t2 ==> t2' →
        P v1 t2 ==> P v1 t2'

  where " t '==>' t' " := (step t t').
```

Exercise: 3 stars, recommended (redo_determinism) As a sanity check on this change, let's re-verify determinism.

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_PlusConstConst, the result is immediate.
- It cannot happen that one is ST_PlusConstConst and the other is ST_Plus1 or ST_Plus2, since this would imply that x has the form *P t1 t2* where both *t1* and *t2* are constants (by ST_PlusConstConst) *and* one of *t1* or *t2* has the form *P* _.

- Similarly, it cannot happen that one is ST_Plus1 and the other is ST_Plus2, 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).

- The cases when both derivations end with ST_Plus1 or ST_Plus2 follow by the induction hypothesis. □

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.
   □
```

22.3.2 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* ==> *t'*.

Proof: By induction on *t*.

- Suppose *t* = 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_PlusConstConst.
  - If *t1* is a value and *t2* can take a step, then so can *t*, by ST_Plus2.
  - If *t1* can take a step, then so can *t*, by ST_Plus1. □

Or, formally:

```
Theorem strong_progress : ∀ t,
  value t ∨ (∃ t', t ==> t').
Proof.
  induction t.
    - left. apply v_const.
    - right. inversion IHt1.
```

```
    + inversion IHt2.
      × inversion H. inversion H0.
        ∃ (C (n + n0)).
        apply ST_PlusConstConst.
      × inversion H0 as [t' H1].
        ∃ (P t1 t').
        apply ST_Plus2. apply H. apply H1.
    + inversion H as [t' H0].
        ∃ (P t' t2).
        apply ST_Plus1. apply H0. 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 just *progress*.)

The idea of "making progress" can be extended to tell us something interesting about values: in this language, values are exactly the terms that *cannot* 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:Type} (R:relation X) (t:X) : Prop :=
  ¬ ∃ 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, normal forms and values are actually the same thing.

```
Lemma value_is_nf : ∀ v,
  value v → normal_form step v.
Proof.
  unfold normal_form. intros v H. inversion H.
  intros contra. inversion contra. inversion H1.
Qed.

Lemma nf_is_value : ∀ t,
  normal_form step t → value t.
Proof.   unfold normal_form. intros t H.
  assert (G : value t ∨ ∃ t', t ==> t').
    { apply strong_progress. }
  inversion G.
    + apply H0.
    + exfalso. apply H. assumption. Qed.

Corollary nf_same_as_value : ∀ t,
```

normal_form step t ↔ 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 form of a term – 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 coincide! Indeed, we could easily have written the definitions so that they would *not* coincide.

Exercise: 3 stars, optional (value_not_same_as_normal_form1) We might, for example, mistakenly 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 Coq, make sure you can think of an example of such a term.)

```
Module Temp1.

Inductive value : tm → Prop :=
| v_const : ∀ n, value (C n)
| v_funny : ∀ t1 n2,
                 value (P t1 (C n2)).

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ 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, optional (value_not_same_as_normal_form2) Alternatively, we might mistakenly define **step** so that it permits something designated as a value to reduce further.

```
Module Temp2.

Inductive value : tm → Prop :=
| v_const : ∀ 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_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ 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, 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_const : ∀ n, value (C n).

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
```

```
    P t1 t2 ==> P t1' t2

  where " t '==>' t' " := (step t t').
```

(Note that ST_Plus2 is missing.)

```
Lemma value_not_same_as_normal_form :
  ∃ t, ¬ value t ∧ 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 and a conditional expression...

```
Inductive tm : Type :=
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm.

Inductive value : tm → Prop :=
  | v_true : value ttrue
  | v_false : value tfalse.

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_IfTrue : ∀ t1 t2,
      tif ttrue t1 t2 ==> t1
  | ST_IfFalse : ∀ t1 t2,
      tif tfalse t1 t2 ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      tif t1 t2 t3 ==> tif t1' t2 t3

  where " t '==>' t' " := (step t t').
```

Exercise: 1 starM (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 Coq.)

```
Definition bool_step_prop1 :=
```

```
  tfalse ==> tfalse.

Definition bool_step_prop2 :=
     tif
       ttrue
       (tif ttrue ttrue ttrue)
       (tif tfalse tfalse tfalse)
  ==>
     ttrue.

Definition bool_step_prop3 :=
     tif
       (tif ttrue ttrue ttrue)
       (tif ttrue ttrue ttrue)
       tfalse
   ==>
     tif
       ttrue
       (tif ttrue ttrue ttrue)
       tfalse.
```

□

Exercise: 3 stars, optional (progress_bool) Just as we proved a progress theorem for plus expressions, we can do so for boolean expressions, as well.

```
Theorem strong_progress : ∀ t,
  value t ∨ (∃ t', t ==> t').
Proof.
   Admitted.
```

□

Exercise: 2 stars, optional (step_deterministic)

```
Theorem step_deterministic :
  deterministic step.
Proof.
   Admitted.
```

□

```
Module Temp5.
```

Exercise: 2 stars (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 `ttrue` or `tfalse`) 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:

tif (tif ttrue ttrue ttrue) tfalse tfalse ==> tfalse.
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,
      tif ttrue t1 t2 ==> t1
  | ST_IfFalse : ∀ t1 t2,
      tif tfalse t1 t2 ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      tif t1 t2 t3 ==> tif t1' t2 t3


  where " t '==>' t' " := (step t t').

Definition bool_step_prop4 :=
         tif
            (tif ttrue ttrue ttrue)
            tfalse
            tfalse
     ==>
         tfalse.

Example bool_step_prop4_holds :
  bool_step_prop4.
Proof.
   Admitted.
```

□

Exercise: 3 stars, 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 Coq.

- Does a strong progress theorem hold? Write yes or no and briefly (1 sentence) explain your answer.

  Optional: prove your answer correct in Coq.

- In general, is there any way we could cause strong progress to fail if we took away one or more constructors from the original step relation? Write yes or no and briefly (1 sentence) explain your answer.

☐

`End` *Temp5*.
`End` *Temp4*.

22.4 Multi-Step Reduction

We've been working so far with the *single-step reduction* relation ==>, 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* ==>*, 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, let's take a little extra trouble and define it generically.

Given a relation R, we define a 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 and the Coq 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*, ..., *zn* such that

  R x z1 R z1 z2 ... R zn y.

Thus, if R describes a single-step of computation, then *z1...zn* is the sequence of intermediate steps of computation between x and y.

We write ==>* for the **multi step** relation on terms.

```
Notation " t '==>*' t' " := (multi step t t') (at level 40).
```

The relation **multi** R has several crucial properties.

First, it is obviously *reflexive* (that is, ∀ x, **multi** R x x). In the case of the ==>* (i.e., **multi step**) relation, the intuition is that a term can execute to itself by taking zero steps of execution.

Second, it contains R – that is, single-step executions 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 : ∀ (X:Type) (R:relation X) (x y : X),
       R x y → (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 :
  ∀ (X:Type) (R: relation X) (x y z : X),
      multi R x y →
      multi R y z →
      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==>*t2* and *t2==>*t3*, then *t1==>*t3*.

22.4.1 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_Plus1. apply ST_PlusConstConst.
  apply multi_step with
              (P (C (0 + 3))
                 (C (2 + 4))).
  apply ST_Plus2. apply v_const.
  apply ST_PlusConstConst.
  apply multi_R.
  apply ST_PlusConstConst. 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_Plus1. apply ST_PlusConstConst.
  eapply multi_step. apply ST_Plus2. apply v_const.
  apply ST_PlusConstConst.
  eapply multi_step. apply ST_PlusConstConst.
  apply multi_refl. Qed.
```

Exercise: 1 star, optional (test_multistep_2) `Lemma test_multistep_2:`

```
  C 3 ==>* C 3.
Proof.
   Admitted.
```

□

Exercise: 1 star, optional (test_multistep_3) `Lemma test_multistep_3:`

```
      P (C 0) (C 3)
   ==>*
      P (C 0) (C 3).
Proof.
   Admitted.
```

□

Exercise: 2 stars (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.
  □
```

22.4.2 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 step_normal_form := normal_form step.

Definition normal_form_of (t t' : tm) :=
  (t ==>* t' ∧ step_normal_form 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, optional (normal_forms_unique)

```
Theorem normal_forms_unique:
  deterministic normal_form_of.
Proof.
  unfold deterministic. unfold normal_form_of.
  intros x y1 y2 P1 P2.
  inversion P1 as [P11 P12]; clear P1.
  inversion P2 as [P21 P22]; clear P2.
  generalize dependent y2.
  Admitted.
  □
```

Indeed, something stronger is true for this language (though not for all languages): the reduction of *any* term t will eventually reach a normal form – i.e., normal_form_of is a *total* function. Formally, we say the **step** relation is *normalizing*.

```
Definition normalizing {X:Type} (R:relation X) :=
  ∀ t, ∃ t',
    (multi R) t t' ∧ normal_form R 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 left-hand child of a *P* node, and similarly when *t* appears as the right-hand child of a *P* node whose left-hand child is a value.

```
Lemma multistep_congr_1 : ∀ t1 t1' t2,
     t1 ==>* t1' →
     P t1 t2 ==>* P t1' t2.
Proof.
  intros t1 t1' t2 H. induction H.
    - apply multi_refl.
    - apply multi_step with (P y t2).
        apply ST_Plus1. apply H.
        apply IHmulti. Qed.
```

Exercise: 2 stars (multistep_congr_2) `Lemma` *multistep_congr_2* : ∀ *t1 t2 t2'*,

```
     value t1 →
     t2 ==>* t2' →
     P t1 t2 ==>* P t1 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* steps to *t'* and *t'* is a normal form.

Proof sketch: By induction on terms. There are two cases to consider:

- *t* = 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_const).

- *t* = *P t1 t2* for some *t1* and *t2*. By the IH, *t1* and *t2* have normal forms *t1'* and *t2'*. Recall that normal forms are values (by nf_same_as_value); we know that *t1'* = C *n1* and *t2'* = C *n2*, for some *n1* and *n2*. We can combine the ==>* derivations for *t1* and *t2* using *multi_congr_1* and *multi_congr_2* to prove that *P t1 t2* reduces in many steps to C (*n1* + *n2*).

  It is clear that our choice of *t'* = C (*n1* + *n2*) is a value, which is in turn a normal form. ☐

```
Theorem step_normalizing :
  normalizing step.
Proof.
```

```
unfold normalizing.
induction t.
  -
    ∃ (C n).
    split.
    + apply multi_refl.
    +

      rewrite nf_same_as_value. apply v_const.
  -
    destruct IHt1 as [t1' [H11 H12]].
    destruct IHt2 as [t2' [H21 H22]].
    rewrite nf_same_as_value in H12. rewrite nf_same_as_value in H22.
    inversion H12 as [n1 H]. inversion H22 as [n2 H'].
    rewrite ← H in H11.
    rewrite ← H' in H21.
    ∃ (C (n1 + n2)).
    split.
      +
        apply multi_trans with (P (C n1) t2).
        × apply multistep_congr_1. apply H11.
        × apply multi_trans with
            (P (C n1) (C n2)).
          { apply multistep_congr_2. apply v_const. apply H21. }
          { apply multi_R. apply ST_PlusConstConst. }
      +
        rewrite nf_same_as_value. apply v_const. Qed.
```

22.4.3 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.

Exercise: 3 stars (eval__multistep) `Theorem` *eval__multistep* : ∀ *t n*,
  t \\ *n* → *t* ==>* *C n*.

The key ideas in the proof can be seen in the following picture:

P t1 t2 ==> (by ST_Plus1) P t1' t2 ==> (by ST_Plus1) P t1" t2 ==> (by ST_Plus1) ... P (C n1) t2 ==> (by ST_Plus2) P (C n1) t2' ==> (by ST_Plus2) P (C n1) t2" ==> (by ST_Plus2) ... P (C n1) (C n2) ==> (by ST_PlusConstConst) C (n1 + n2)

That is, the multistep reduction of a term of the form *P t1 t2* proceeds in three phases:

- First, we use ST_Plus1 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_Plus2 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_PlusConstConst 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 ==>*: that it is reflexive, transitive, and includes ==>.

Proof.
Admitted.
□

Exercise: 3 stars, advanced (eval__multistep_inf) Write a detailed informal version of the proof of eval__multistep.
□

For the other direction, we need one lemma, which establishes a relation between single-step reduction and big-step evaluation.

Exercise: 3 stars (step__eval) Lemma *step__eval* : ∀ *t t' n*,
t ==> *t'* →
t' \\ *n* →
t \\ *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, once it is stated correctly.

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 (multistep__eval) Theorem *multistep__eval* : ∀ *t t'*,
normal_form_of t t' → ∃ *n*, *t'* = *C n* ∧ *t* \\ *n*.
Proof.
Admitted.
□

22.4.4 Additional Exercises

Exercise: 3 stars, 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 : ∀ t n,
  evalF t = n ↔ t \\ n.
Proof.
   Admitted.
```

□

Exercise: 4 starsM (combined_properties) We've considered arithmetic and conditional expressions separately. This exercise explores how the two interact.

```
Module Combined.

Inductive tm : Type :=
  | C : nat → tm
  | P : tm → tm → tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm.

Inductive value : tm → Prop :=
  | v_const : ∀ n, value (C n)
  | v_true : value ttrue
  | v_false : value tfalse.

Reserved Notation " t '==>' t' " (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀ n1 n2,
      P (C n1) (C n2) ==> C (n1 + n2)
  | ST_Plus1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      P t1 t2 ==> P t1' t2
  | ST_Plus2 : ∀ v1 t2 t2',
      value v1 →
      t2 ==> t2' →
      P v1 t2 ==> P v1 t2'
  | ST_IfTrue : ∀ t1 t2,
      tif ttrue t1 t2 ==> t1
  | ST_IfFalse : ∀ t1 t2,
      tif tfalse t1 t2 ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
```

```
    t1 ==> t1' →
    tif t1 t2 t3 ==> tif 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.

Prove or disprove these two properties for the combined language.

```
End Combined.
```

□

22.5 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 ==>a and ==>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 ==>b below (why?), though they might be if our language were a bit larger (why?).

```
Reserved Notation " t '/' st '==>a' t' "
                  (at level 40, st at level 39).

Inductive astep : state → aexp → aexp → Prop :=
  | AS_Id : ∀ st i,
      AId i / st ==>a ANum (st i)
  | AS_Plus : ∀ st n1 n2,
      APlus (ANum n1) (ANum n2) / st ==>a ANum (n1 + n2)
  | AS_Plus1 : ∀ st a1 a1' a2,
      a1 / st ==>a a1' →
      (APlus a1 a2) / st ==>a (APlus a1' a2)
  | AS_Plus2 : ∀ st v1 a2 a2',
      aval v1 →
      a2 / st ==>a a2' →
      (APlus v1 a2) / st ==>a (APlus v1 a2')
  | AS_Minus : ∀ st n1 n2,
```

```
      (AMinus (ANum n1) (ANum n2)) / st ==>a (ANum (minus n1 n2))
  | AS_Minus1 : ∀ st a1 a1' a2,
      a1 / st ==>a a1' →
      (AMinus a1 a2) / st ==>a (AMinus a1' a2)
  | AS_Minus2 : ∀ st v1 a2 a2',
      aval v1 →
      a2 / st ==>a a2' →
      (AMinus v1 a2) / st ==>a (AMinus v1 a2')
  | AS_Mult : ∀ st n1 n2,
      (AMult (ANum n1) (ANum n2)) / st ==>a (ANum (mult n1 n2))
  | AS_Mult1 : ∀ st a1 a1' a2,
      a1 / st ==>a a1' →
      (AMult a1 a2) / st ==>a (AMult a1' a2)
  | AS_Mult2 : ∀ st v1 a2 a2',
      aval v1 →
      a2 / st ==>a a2' →
      (AMult v1 a2) / st ==>a (AMult v1 a2')

    where " t '/' st '==>a' t' " := (astep st t t').

Reserved Notation " t '/' st '==>b' t' "
                  (at level 40, st at level 39).

Inductive bstep : state → bexp → bexp → Prop :=
| BS_Eq : ∀ st n1 n2,
    (BEq (ANum n1) (ANum n2)) / st ==>b
    (if (beq_nat n1 n2) then BTrue else BFalse)
| BS_Eq1 : ∀ st a1 a1' a2,
    a1 / st ==>a a1' →
    (BEq a1 a2) / st ==>b (BEq a1' a2)
| BS_Eq2 : ∀ st v1 a2 a2',
    aval v1 →
    a2 / st ==>a a2' →
    (BEq v1 a2) / st ==>b (BEq v1 a2')
| BS_LtEq : ∀ st n1 n2,
    (BLe (ANum n1) (ANum n2)) / st ==>b
             (if (leb n1 n2) then BTrue else BFalse)
| BS_LtEq1 : ∀ st a1 a1' a2,
    a1 / st ==>a a1' →
    (BLe a1 a2) / st ==>b (BLe a1' a2)
| BS_LtEq2 : ∀ st v1 a2 a2',
    aval v1 →
    a2 / st ==>a a2' →
    (BLe v1 a2) / st ==>b (BLe v1 a2')
```

```
| BS_NotTrue : ∀ st,
    (BNot BTrue) / st ==>b BFalse
| BS_NotFalse : ∀ st,
    (BNot BFalse) / st ==>b BTrue
| BS_NotStep : ∀ st b1 b1',
    b1 / st ==>b b1' →
    (BNot b1) / st ==>b (BNot b1')
| BS_AndTrueTrue : ∀ st,
    (BAnd BTrue BTrue) / st ==>b BTrue
| BS_AndTrueFalse : ∀ st,
    (BAnd BTrue BFalse) / st ==>b BFalse
| BS_AndFalse : ∀ st b2,
    (BAnd BFalse b2) / st ==>b BFalse
| BS_AndTrueStep : ∀ st b2 b2',
    b2 / st ==>b b2' →
    (BAnd BTrue b2) / st ==>b (BAnd BTrue b2')
| BS_AndStep : ∀ st b1 b1' b2,
    b1 / st ==>b b1' →
    (BAnd b1 b2) / st ==>b (BAnd b1' b2)

where " t '/' st '==>b' t' " := (bstep st t t').
```

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 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 needs to be saved somewhere while a single copy of the loop body is being reduced.)

```
Reserved Notation " t '/' st '==>' t' '/' st' "
                  (at level 40, st at level 39, t' at level 39).

Inductive cstep : (com × state) → (com × state) → Prop :=
  | CS_AssStep : ∀ st i a a',
      a / st ==>a a' →
```

```
      (i ::= a) / st ==> (i ::= a') / st
  | CS_Ass : ∀ st i n,
      (i ::= (ANum n)) / st ==> SKIP / (t_update st i n)
  | CS_SeqStep : ∀ st c1 c1' st' c2,
      c1 / st ==> c1' / st' →
      (c1 ;; c2) / st ==> (c1' ;; c2) / st'
  | CS_SeqFinish : ∀ st c2,
      (SKIP ;; c2) / st ==> c2 / st
  | CS_IfTrue : ∀ st c1 c2,
      IFB BTrue THEN c1 ELSE c2 FI / st ==> c1 / st
  | CS_IfFalse : ∀ st c1 c2,
      IFB BFalse THEN c1 ELSE c2 FI / st ==> c2 / st
  | CS_IfStep : ∀ st b b' c1 c2,
      b / st ==>b b' →
          IFB b THEN c1 ELSE c2 FI / st
      ==> (IFB b' THEN c1 ELSE c2 FI) / st
  | CS_While : ∀ st b c1,
          (WHILE b DO c1 END) / st
      ==> (IFB b THEN (c1;; (WHILE b DO c1 END)) ELSE SKIP FI) / st

  where " t '/' st '==>' t' '/' st' " := (cstep (t,st) (t',st')).
```

22.6 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.

```
Module CImp.

Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com

  | CPar : com → com → com.

Notation "'SKIP'" :=
  CSkip.
Notation "x '::=' a" :=
```

```
  (CAss x a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' b 'THEN' c1 'ELSE' c2 'FI'" :=
  (CIf b c1 c2) (at level 80, right associativity).
Notation "'PAR' c1 'WITH' c2 'END'" :=
  (CPar c1 c2) (at level 80, right associativity).

Inductive cstep : (com × state) → (com × state) → Prop :=

  | CS_AssStep : ∀ st i a a',
      a / st ==>a a' →
      (i ::= a) / st ==> (i ::= a') / st
  | CS_Ass : ∀ st i n,
      (i ::= (ANum n)) / st ==> SKIP / (t_update st i n)
  | CS_SeqStep : ∀ st c1 c1' st' c2,
      c1 / st ==> c1' / st' →
      (c1 ;; c2) / st ==> (c1' ;; c2) / st'
  | CS_SeqFinish : ∀ st c2,
      (SKIP ;; c2) / st ==> c2 / st
  | CS_IfTrue : ∀ st c1 c2,
      (IFB BTrue THEN c1 ELSE c2 FI) / st ==> c1 / st
  | CS_IfFalse : ∀ st c1 c2,
      (IFB BFalse THEN c1 ELSE c2 FI) / st ==> c2 / st
  | CS_IfStep : ∀ st b b' c1 c2,
      b /st ==>b b' →
          (IFB b THEN c1 ELSE c2 FI) / st
      ==> (IFB b' THEN c1 ELSE c2 FI) / st
  | CS_While : ∀ st b c1,
          (WHILE b DO c1 END) / st
      ==> (IFB b THEN (c1;; (WHILE b DO c1 END)) ELSE SKIP FI) / st

  | CS_Par1 : ∀ st c1 c1' c2 st',
      c1 / st ==> c1' / st' →
      (PAR c1 WITH c2 END) / st ==> (PAR c1' WITH c2 END) / st'
  | CS_Par2 : ∀ st c1 c2 c2' st',
      c2 / st ==> c2' / st' →
      (PAR c1 WITH c2 END) / st ==> (PAR c1 WITH c2' END) / st'
  | CS_ParDone : ∀ st,
      (PAR SKIP WITH SKIP END) / st ==> SKIP / st
  where " t '/' st '==>' t' '/' st' " := (cstep (t,st) (t',st')).
```

```
Definition cmultistep := multi cstep.

Notation " t '/' st '==>*' t' '/' st' " :=
   (multi cstep (t,st) (t',st'))
   (at level 40, st at level 39, t' at level 39).
```

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 :=
  PAR
    Y ::= ANum 1
  WITH
    WHILE BEq (AId Y) (ANum 0) DO
      X ::= APlus (AId X) (ANum 1)
    END
  END.
```

In particular, it can terminate with X set to 0:

```
Example par_loop_example_0:
  ∃ st',
       par_loop / empty_state ==>* SKIP / st'
    ∧ st' X = 0.
Proof.
  eapply ex_intro. split.
  unfold par_loop.
  eapply multi_step. apply CS_Par1.
    apply CS_Ass.
  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:

```
Example par_loop_example_2:
  ∃ st',
       par_loop / empty_state ==>* SKIP / st'
    ∧ st' X = 2.
Proof.
  eapply ex_intro. 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_AssStep. apply AS_Plus1. apply AS_Id.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_AssStep. apply AS_Plus.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_Ass.
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.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_AssStep. apply AS_Plus1. apply AS_Id.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_AssStep. apply AS_Plus.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_Ass.

eapply multi_step. apply CS_Par1. apply CS_Ass.
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, optional (par_body_n__Sn)

```
Lemma par_body_n__Sn : ∀ n st,
  st X = n ∧ st Y = 0 →
  par_loop / st ==>* par_loop / (t_update st X (S n)).
Proof.
  Admitted.
```

□

Exercise: 3 stars, optional (par_body_n) Lemma *par_body_n* : ∀ *n st*,
  st X = 0 ∧ *st Y* = 0 →
  ∃ *st'*,
    par_loop / *st* ==>* *par_loop* / *st'* ∧ *st' X* = *n* ∧ *st' Y* = 0.
Proof.
  Admitted.
  □
  ... the above loop can exit with X having any value whatsoever.

```
Theorem par_loop_any_X:
  ∀ n, ∃ st',
    par_loop / empty_state ==>* SKIP / st'
    ∧ st' X = n.
Proof.
  intros n.
  destruct (par_body_n n empty_state).
    split; unfold t_update; reflexivity.

  rename x into st.
  inversion H as [H' [HX HY]]; clear H.
  ∃ (t_update st Y 1). split.
  eapply multi_trans with (par_loop,st). apply H'.
  eapply multi_step. apply CS_Par1. apply CS_Ass.
  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. assumption. intro X; inversion X.
Qed.

End CImp.
```

22.7 A Small-Step Stack Machine

Our last example is a small-step semantics for the stack machine example from the Imp chapter.

```
Definition stack := list nat.
```

```
Definition prog := list sinstr.

Inductive stack_step : state → prog × stack → prog × stack → Prop :=
  | SS_Push : ∀ st stk n p',
    stack_step st (SPush n :: p', stk) (p', n :: stk)
  | SS_Load : ∀ st stk i p',
    stack_step st (SLoad i :: p', stk) (p', st i :: stk)
  | SS_Plus : ∀ st stk n m p',
    stack_step st (SPlus :: p', n::m::stk) (p', (m+n)::stk)
  | SS_Minus : ∀ st stk n m p',
    stack_step st (SMinus :: p', n::m::stk) (p', (m-n)::stk)
  | SS_Mult : ∀ st 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. We want now to prove *compile* correct with respect to the stack machine.

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.
```

□

$Date : 2016-12-2011 : 28 : 30-0500 (Tue, 20 Dec 2016)$

Chapter 23

Library Top.Auto

23.1 Auto: More Automation

```
Require Import Coq.omega.Omega.
Require Import Maps.
Require Import Imp.
```

Up to now, we've used the more manual part of Coq's tactic facilities. In this chapter, we'll learn more about some of Coq's powerful automation features: proof search via the `auto` tactic, automated forward reasoning via the `Ltac` hypothesis matching machinery, and deferred instantiation of existential variables using `eapply` and `eauto`. Using these features together with Ltac's scripting facilities will enable us to make our proofs startlingly short! Used properly, they can also make proofs more maintainable and robust to changes in underlying definitions. A deeper treatment of `auto` and `eauto` can be found in the `UseAuto` chapter.

There's another major category of automation we haven't discussed much yet, namely built-in decision procedures for specific kinds of problems: `omega` is one example, but there are others. This topic will be deferred for a while longer.

Our motivating example will be this proof, repeated with just a few small changes from the `Imp` chapter. We will simplify this proof in several stages.

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

Theorem ceval_deterministic: ∀ c st st1 st2,
    c / st \\ st1 →
    c / st \\ st2 →
    st1 = st2.
Proof.
  intros c st st1 st2 E1 E2;
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2.
  - 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.
```

23.2 The auto Tactic

Thus far, our proof scripts mostly apply relevant hypotheses or lemmas by name, and one at a time.

```
Example auto_example_1 : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  intros P Q R H1 H2 H3.
  apply H2. apply H1. assumption.
Qed.
```

The auto tactic frees us from this drudgery by *searching* for a sequence of applications that will prove the goal

```
Example auto_example_1' : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
```

```
  intros P Q R H1 H2 H3.
  auto.
Qed.
```

The `auto` tactic solves goals that are solvable by any combination of

- `intros` and
- `apply` (of hypotheses from the local context, by default).

Using `auto` is always "safe" in the sense that it will never fail and will never change the proof state: either it completely solves the current goal, or it does nothing.

Here is a more interesting example showing `auto`'s power:

```
Example auto_example_2 : ∀ P Q R S T U : Prop,
  (P → Q) →
  (P → R) →
  (T → R) →
  (S → T → U) →
  ((P→Q) → (P→S)) →
  T →
  P →
  U.
Proof. auto. Qed.
```

Proof search could, in principle, take an arbitrarily long time, so there are limits to how far `auto` will search by default.

```
Example auto_example_3 : ∀ (P Q R S T U: Prop),
  (P → Q) →
  (Q → R) →
  (R → S) →
  (S → T) →
  (T → U) →
  P →
  U.
Proof.
  auto.
  auto 6.
Qed.
```

When searching for potential proofs of the current goal, `auto` considers the hypotheses in the current context together with a *hint database* of other lemmas and constructors. Some common lemmas about equality and logical operators are installed in this hint database by default.

```
Example auto_example_4 : ∀ P Q R : Prop,
```

```
  Q →
  (Q → R) →
  P ∨ (Q ∧ R).
Proof. auto. Qed.
```

We can extend the hint database just for the purposes of one application of `auto` by writing `auto using` ....

```
Lemma le_antisym : ∀ n m: nat, (n ≤ m ∧ m ≤ n) → n = m.
Proof. intros. omega. Qed.

Example auto_example_6 : ∀ n m p : nat,
  (n ≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
Proof.
  intros.
  auto.   auto using le_antisym.
Qed.
```

Of course, in any given development there will probably be some specific constructors and lemmas that are used very often in proofs. We can add these to the global hint database by writing

Hint Resolve T.

at the top level, where `T` is a top-level theorem or a constructor of an inductively defined proposition (i.e., anything whose type is an implication). As a shorthand, we can write

Hint Constructors c.

to tell Coq to do a `Hint Resolve` for *all* of the constructors from the inductive definition of `c`.

It is also sometimes necessary to add

Hint Unfold d.

where `d` is a defined symbol, so that `auto` knows to expand uses of `d`, thus enabling further possibilities for applying lemmas that it knows about.

```
Hint Resolve le_antisym.

Example auto_example_6' : ∀ n m p : nat,
  (n≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
Proof.
  intros.
  auto. Qed.

Definition is_fortytwo x := x = 42.

Example auto_example_7: ∀ x, (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof.
```

```
  auto. Abort.

Hint Unfold is_fortytwo.

Example auto_example_7' : ∀ x, (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof. auto. Qed.
```

Now let's take a first pass over ceval_deterministic to simplify the proof script.

```
Theorem ceval_deterministic': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
       induction E1; intros st2 E2; inv E2; auto.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  -
    +
      rewrite H in H5. inversion H5.
  -
    +
      rewrite H in H5. inversion H5.
  -
    +
      rewrite H in H2. inversion H2.
  -
    rewrite H in H4. inversion H4.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
Qed.
```

When we are using a particular tactic many times in a proof, we can use a variant of the `Proof` command to make that tactic into a default within the proof. Saying `Proof with t` (where *t* is an arbitrary tactic) allows us to use *t1*... as a shorthand for *t1*;*t* within the proof. As an illustration, here is an alternate version of the previous proof, using `Proof with auto`.

```
Theorem ceval_deterministic'_alt: ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
```

```
    st1 = st2.
Proof with auto.
  intros c st st1 st2 E1 E2;
  generalize dependent st2;
  induction E1;
          intros st2 E2; inv E2...
  -
    assert (st' = st'0) as EQ1...
    subst st'0...
  -
    +
      rewrite H in H5. inversion H5.
  -
    +
      rewrite H in H5. inversion H5.
  -
    +
      rewrite H in H2. inversion H2.
  -
    rewrite H in H4. inversion H4.
  -
    assert (st' = st'0) as EQ1...
    subst st'0...
Qed.
```

23.3 Searching For Hypotheses

The proof has become simpler, but there is still an annoying amount of repetition. Let's start by tackling the contradiction cases. Each of them occurs in a situation where we have both

H1: beval st b = false

and

H2: beval st b = true

as hypotheses. The contradiction is evident, but demonstrating it is a little complicated: we have to locate the two hypotheses *H1* and *H2* and do a `rewrite` following by an `inversion`. We'd like to automate this process.

(In fact, Coq has a built-in tactic `congruence` that will do the job in this case. But we'll ignore the existence of this tactic for now, in order to demonstrate how to build forward search tactics by hand.)

As a first step, we can abstract out the piece of script in question by writing a little function in Coq's tactic programming language, Ltac.

```
Ltac rwinv H1 H2 := rewrite H1 in H2; inv H2.
```

```
Theorem ceval_deterministic'': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; auto.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  -
    +
      rwinv H H5.
  -
    +
      rwinv H H5.
  -
    +
      rwinv H H2.
  -
    rwinv H H4.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto. Qed.
```

That was is a bit better, but not much. We really want Coq to discover the relevant hypotheses for us. We can do this by using the `match goal` facility of Ltac.

```
Ltac find_rwinv :=
  match goal with
    H1: ?E = true,
    H2: ?E = false
    ⊢ _ ⇒ rwinv H1 H2
  end.
```

The `match goal` tactic looks for two distinct hypotheses that have the form of equalities, with the same arbitrary expression E on the left and with conflicting boolean values on the right. If such hypotheses are found, it binds *H1* and *H2* to their names and applies the *rwinv* tactic to *H1* and *H2*.

Adding this tactic to the ones that we invoke in each case of the induction handles all of the contradictory cases.

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; try find_rwinv; auto.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  -
    +
      assert (st' = st'0) as EQ1 by auto.
      subst st'0.
      auto. Qed.
```

Let's see about the remaining cases. Each of them involves applying a conditional hypothesis to extract an equality. Currently we have phrased these as assertions, so that we have to predict what the resulting equality will be (although we can then use `auto` to prove it). An alternative is to pick the relevant hypotheses to use and then rewrite with them, as follows:

```
Theorem ceval_deterministic'''': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; try find_rwinv; auto.
  -
    rewrite (IHE1_1 st'0 H1) in *. auto.
  -
    +
      rewrite (IHE1_1 st'0 H3) in *. auto. Qed.
```

Now we can automate the task of finding the relevant hypotheses to rewrite with.

```
Ltac find_eqn :=
  match goal with
    H1: ∀ x, ?P x → ?L = ?R,
    H2: ?P ?X
    ⊢ _ ⇒ rewrite (H1 X H2) in *
```

```
end.
```

The pattern ∀ x, ?*P* x → ?*L* = ?R matches any hypothesis of the form "for all x, *some property of* x implies *some equality*." The property of x is bound to the pattern variable *P*, and the left- and right-hand sides of the equality are bound to *L* and R. The name of this hypothesis is bound to *H1*. Then the pattern ?*P* ?X matches any hypothesis that provides evidence that *P* holds for some concrete X. If both patterns succeed, we apply the `rewrite` tactic (instantiating the quantified x with X and providing *H2* as the required evidence for *P* X) in all hypotheses and the goal.

One problem remains: in general, there may be several pairs of hypotheses that have the right general form, and it seems tricky to pick out the ones we actually need. A key trick is to realize that we can *try them all*! Here's how this works:

- each execution of `match goal` will keep trying to find a valid pair of hypotheses until the tactic on the RHS of the match succeeds; if there are no such pairs, it fails;
- `rewrite` will fail given a trivial equation of the form X = X;
- we can wrap the whole thing in a `repeat`, which will keep doing useful rewrites until only trivial ones are left.

```
Theorem ceval_deterministic''''': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; try find_rwinv;
    repeat find_eqn; auto.
Qed.
```

The big payoff in this approach is that our proof script should be robust in the face of modest changes to our language. For example, we can add a *REPEAT* command to the language.

```
Module Repeat.

Inductive com : Type :=
  | CSkip : com
  | CAsgn : id → aexp → com
  | CSeq : com → com → com
  | CIf : 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 "'SKIP'" :=
  CSkip.
Notation "c1 ; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "X '::=' a" :=
  (CAsgn X a) (at level 60).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'REPEAT' e1 'UNTIL' b2 'END'" :=
  (CRepeat e1 b2) (at level 80, right associativity).

Inductive ceval : state → com → state → Prop :=
  | E_Skip : ∀ st,
      ceval st SKIP st
  | E_Ass : ∀ st a1 n X,
      aeval st a1 = n →
      ceval st (X ::= a1) (t_update st X n)
  | E_Seq : ∀ c1 c2 st st' st'',
      ceval st c1 st' →
      ceval st' c2 st'' →
      ceval st (c1 ; c2) st''
  | E_IfTrue : ∀ st st' b1 c1 c2,
      beval st b1 = true →
      ceval st c1 st' →
      ceval st (IFB b1 THEN c1 ELSE c2 FI) st'
  | E_IfFalse : ∀ st st' b1 c1 c2,
      beval st b1 = false →
      ceval st c2 st' →
      ceval st (IFB b1 THEN c1 ELSE c2 FI) st'
  | E_WhileEnd : ∀ b1 st c1,
      beval st b1 = false →
      ceval st (WHILE b1 DO c1 END) st
  | E_WhileLoop : ∀ st st' st'' b1 c1,
      beval st b1 = true →
      ceval st c1 st' →
      ceval st' (WHILE b1 DO c1 END) st'' →
      ceval st (WHILE b1 DO c1 END) st''
  | E_RepeatEnd : ∀ st st' b1 c1,
      ceval st c1 st' →
      beval st' b1 = true →
      ceval st (CRepeat c1 b1) st'
```

```
  | E_RepeatLoop : ∀ st st' st'' b1 c1,
      ceval st c1 st' →
      beval st' b1 = false →
      ceval st' (CRepeat c1 b1) st'' →
      ceval st (CRepeat c1 b1) st''.
Notation "c1 '/' st '\\' st'" := (ceval st c1 st')
                                  (at level 40, st at level 39).
```

Our first attempt at the proof is disappointing: the E_RepeatEnd and E_RepeatLoop cases are not handled by our previous automation.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1;
    intros st2 E2; inv E2; try find_rwinv; repeat find_eqn; auto.
  -
    +
      find_rwinv.
  -
    +
      find_rwinv.
Qed.
```

To fix this, we just have to swap the invocations of *find_eqn* and *find_rwinv*.

```
Theorem ceval_deterministic': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1;
    intros st2 E2; inv E2; repeat find_eqn; try find_rwinv; auto.
Qed.
End Repeat.
```

These examples just give a flavor of what "hyper-automation" can achieve in Coq. The details of `match goal` are a bit tricky, and debugging scripts using it is, frankly, not very pleasant. But it is well worth adding at least simple uses to your proofs, both to avoid tedium and to "future proof" them.

eapply and eauto

To close the chapter, we'll introduce one more convenient feature of Coq: its ability to delay instantiation of quantifiers. To motivate this feature, recall this example from the Imp chapter:

```
Example ceval_example1:
    (X ::= ANum 2;;
     IFB BLe (AId X) (ANum 1)
       THEN Y ::= ANum 3
       ELSE Z ::= ANum 4
     FI)
   / empty_state
   \\ (t_update (t_update empty_state X 2) Z 4).
Proof.
  apply E_Seq with (t_update empty_state X 2).
  - apply E_Ass. reflexivity.
  - apply E_IfFalse. reflexivity. apply E_Ass. reflexivity.
Qed.
```

In the first step of the proof, we had to explicitly provide a longish expression to help Coq instantiate a "hidden" argument to the E_Seq constructor. This was needed because the definition of E_Seq...

E_Seq : forall c1 c2 st st' st", c1 / st \\ st' -> c2 / st' \\ st" -> (c1 ;; c2) / st \\ st"

is quantified over a variable, *st'*, that does not appear in its conclusion, so unifying its conclusion with the goal state doesn't help Coq find a suitable value for this variable. If we leave out the with, this step fails ("Error: Unable to find an instance for the variable *st'*").

What's silly about this error is that the appropriate value for *st'* will actually become obvious in the very next step, where we apply E_Ass. If Coq could just wait until we get to this step, there would be no need to give the value explicitly. This is exactly what the eapply tactic gives us:

```
Example ceval'_example1:
    (X ::= ANum 2;;
     IFB BLe (AId X) (ANum 1)
       THEN Y ::= ANum 3
       ELSE Z ::= ANum 4
     FI)
   / empty_state
   \\ (t_update (t_update empty_state X 2) Z 4).
Proof.
  eapply E_Seq.   - apply E_Ass.     reflexivity.   - apply E_IfFalse. reflexivity.
apply E_Ass. reflexivity.
Qed.
```

The tactic eapply *H* tactic behaves just like apply *H* except that, after it finishes unifying the goal state with the conclusion of H, it does not bother to check whether all the variables that were introduced in the process have been given concrete values during unification.

If you step through the proof above, you'll see that the goal state at position 1 mentions the *existential variable* ?*st'* in both of the generated subgoals. The next step (which gets us to position 2) replaces ?*st'* with a concrete value. This new value contains a new existential variable ?`n`, which is instantiated in its turn by the following `reflexivity` step, position 3. When we start working on the second subgoal (position 4), we observe that the occurrence of ?*st'* in this subgoal has been replaced by the value that it was given during the first subgoal.

Several of the tactics that we've seen so far, including ∃, `constructor`, and `auto`, have `e`... variants. For example, here's a proof using `eauto`:

```
Hint Constructors ceval.
Hint Transparent state.
Hint Transparent total_map.

Definition st12 := t_update (t_update empty_state X 1) Y 2.
Definition st21 := t_update (t_update empty_state X 2) Y 1.

Example auto_example_8 : ∃ s',
  (IFB (BLe (AId X) (AId Y))
    THEN (Z ::= AMinus (AId Y) (AId X))
    ELSE (Y ::= APlus (AId X) (AId Z))
  FI) / st21 \\ s'.
Proof. eauto. Qed.
```

The `eauto` tactic works just like `auto`, except that it uses `eapply` instead of `apply`.

$Date : 2016-10-1815:42:43-0400(Tue, 18Oct2016)$

Chapter 24

Library Top.Types

24.1 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 Coq!).

```
Require Import Coq.Arith.Arith.
Require Import Maps.
Require Import Imp.
Require Import Smallstep.

Hint Constructors multi.
```

24.2 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.

24.2.1 Syntax

Here is the syntax, informally:

t ::= true | false | if t then t else t | 0 | succ t | pred t | iszero t

And here it is formally:

```
Inductive tm : Type :=
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm
  | tzero : tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tiszero : tm → tm.
```

Values are true, false, and numeric values...

```
Inductive bvalue : tm → Prop :=
  | bv_true : bvalue ttrue
  | bv_false : bvalue tfalse.

Inductive nvalue : tm → Prop :=
  | nv_zero : nvalue tzero
  | nv_succ : ∀ t, nvalue t → nvalue (tsucc t).

Definition value (t:tm) := bvalue t ∨ nvalue t.

Hint Constructors bvalue nvalue.
Hint Unfold value.
Hint Unfold update.
```

24.2.2 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_PredZero) pred 0 ==> 0

numeric value v1

(ST_PredSucc) pred (succ v1) ==> v1

t1 ==> t1'

(ST_Pred) pred t1 ==> pred t1'

(ST_IszeroZero) iszero 0 ==> true

numeric value v1

(ST_IszeroSucc) iszero (succ v1) ==> false

t1 ==> t1'

(ST_Iszero) iszero t1 ==> iszero t1'

... and then formally:

```
Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_IfTrue : ∀ t1 t2,
      (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : ∀ t1 t2,
      (tif tfalse t1 t2) ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      (tif t1 t2 t3) ==> (tif t1' t2 t3)
  | ST_Succ : ∀ t1 t1',
      t1 ==> t1' →
      (tsucc t1) ==> (tsucc t1')
  | ST_PredZero :
      (tpred tzero) ==> tzero
  | ST_PredSucc : ∀ t1,
      nvalue t1 →
      (tpred (tsucc t1)) ==> t1
  | ST_Pred : ∀ t1 t1',
      t1 ==> t1' →
      (tpred t1) ==> (tpred t1')
  | ST_IszeroZero :
      (tiszero tzero) ==> ttrue
  | ST_IszeroSucc : ∀ t1,
       nvalue t1 →
      (tiszero (tsucc t1)) ==> tfalse
  | ST_Iszero : ∀ t1 t1',
      t1 ==> t1' →
      (tiszero t1) ==> (tiszero t1')

where "t1 '==>' t2" := (step t1 t2).

Hint Constructors step.
```

Notice that the **step** relation doesn't care about whether expressions make 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 (i.e., tsucc ttrue in the formal syntax) 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).

24.2.3 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 (because we have not included them 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.
```

Exercise: 2 stars (some_term_is_stuck)

```
Example some_term_is_stuck :
  ∃ t, stuck t.
Proof.
  Admitted.
```

☐

However, although values and normal forms are *not* the same in this language, the set of values is included in 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 (value_is_nf)

```
Lemma value_is_nf : ∀ t,
  value t → 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, 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.
□

24.2.4 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 :=
  | TBool : ty
  | TNat : ty.
```

In informal notation, the typing relation is often written $\vdash$ *t* \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_Zero) |- 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

```
Reserved Notation "'|-' t '\in' T" (at level 40).

Inductive has_type : tm → ty → Prop :=
  | T_True :
       ⊢ ttrue \in TBool
  | T_False :
       ⊢ tfalse \in TBool
  | T_If : ∀ t1 t2 t3 T,
```

```
      ⊢ t1 \in TBool →
      ⊢ t2 \in T →
      ⊢ t3 \in T →
      ⊢ tif t1 t2 t3 \in T
  | T_Zero :
      ⊢ tzero \in TNat
  | T_Succ : ∀ t1,
      ⊢ t1 \in TNat →
      ⊢ tsucc t1 \in TNat
  | T_Pred : ∀ t1,
      ⊢ t1 \in TNat →
      ⊢ tpred t1 \in TNat
  | T_Iszero : ∀ t1,
      ⊢ t1 \in TNat →
      ⊢ tiszero t1 \in TBool

where "'|-' t '\in' T" := (has_type t T).

Hint Constructors has_type.

Example has_type_1 :
  ⊢ tif tfalse tzero (tsucc tzero) \in TNat.
Proof.
  apply T_If.
    - apply T_False.
    - apply T_Zero.
    - apply T_Succ.
       + apply T_Zero.
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 has_type_not :
  ¬ (|- tif tfalse tzero ttrue \in TBool).
Proof.
  intros Contra. solve_by_inverts 2. Qed.
```

Exercise: 1 star, optional (succ_hastype_nat__hastype_nat) `Example` *succ_hastype_nat__hastype*

```
: ∀ t,
  ⊢ tsucc t \in TNat →
  ⊢ t \in TNat.
```

Proof.

Admitted.

□

Canonical forms

The following two lemmas capture the fundamental property that the definitions of boolean and numeric values agree with the typing relation.

```
Lemma bool_canonical : ∀ t,
  ⊢ t \in TBool → value t → bvalue t.
Proof.
  intros t HT HV.
  inversion HV; auto.
  induction H; inversion HT; auto.
Qed.
```

```
Lemma nat_canonical : ∀ t,
  ⊢ t \in TNat → value t → nvalue t.
Proof.
  intros t HT HV.
  inversion HV.
  inversion H; subst; inversion HT.
  auto.
Qed.
```

24.2.5 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 (finish_progress)

```
Theorem progress : ∀ t T,
  ⊢ t \in T →
  value t ∨ ∃ t', t ==> 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 with auto.
  intros t T HT.
  induction HT...
  -
    right. inversion IHHT1; clear IHHT1.
    +
    apply (bool_canonical t1 HT1) in H.
```

```
inversion H; subst; clear H.
  ∃ t2...
  ∃ t3...
+
  inversion H as [t1' H1].
  ∃ (tif t1' t2 t3)...
Admitted.
```
□

Exercise: 3 stars, advancedM (finish_progress_informal) Complete the corresponding informal proof:

Theorem: If ⊢ *t* \in T, then either *t* is a value or else *t* ==> *t'* for some *t'*.

Proof: By induction on a derivation of ⊢ *t* \in T.

- If the last rule in the derivation is T_If, then *t* = if *t1* then *t2* else *t3*, with ⊢ *t1* \in Bool, ⊢ *t2* \in T and ⊢ *t3* \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 ⊢ *t1* \in Bool we have that *t1* is a **bvalue** – i.e., it is either true or false. If *t1* = true, then *t* steps to *t2* by ST_IfTrue, while if *t1* = 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*.
-

□

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.

24.2.6 Type Preservation

The second critical property of typing is that, when a well-typed term takes a step, the result is also a well-typed term.

Exercise: 2 stars (finish_preservation) Theorem *preservation* : ∀ *t t' T*,
⊢ *t* \in *T* →
t ==> *t'* →
⊢ *t'* \in *T*.

Complete the formal proof of the preservation property. (Again, make sure you understand the informal proof fragment in the following exercise first.)

```
Proof with auto.
  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_If; try assumption.
        apply IHHT1; assumption.
   Admitted.
```

□

Exercise: 3 stars, advancedM (finish_preservation_informal) Complete the following informal proof:

Theorem: If ⊢ t \in T and t ==> t', then ⊢ t' \in T.

Proof: By induction on a derivation of ⊢ t \in T.

- If the last rule in the derivation is T_If, then t = `if` $t1$ `then` $t2$ `else` $t3$, with ⊢ $t1$ \in Bool, ⊢ $t2$ \in T and ⊢ $t3$ \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 ==> t' are ST_IfTrue, ST_IfFalse, or ST_If.

  - If the last rule was ST_IfTrue, then t' = $t2$. But we know that ⊢ $t2$ \in T, so we are done.
  - If the last rule was ST_IfFalse, then t' = $t3$. But we know that ⊢ $t3$ \in T, so we are done.
  - If the last rule was ST_If, then t' = `if` $t1'$ `then` $t2$ `else` $t3$, where $t1$ ==> $t1'$. We know ⊢ $t1$ \in Bool so, by the IH, ⊢ $t1'$ \in Bool. The T_If rule then gives us ⊢ `if` $t1'$ `then` $t2$ `else` $t3$ \in T, as required.

-

□

Exercise: 3 stars (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' : ∀ t t' T,
  ⊢ t \in T →
  t ==> t' →
  ⊢ t' \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.

24.2.7 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 : ∀ t t' T,
  ⊢ t \in T →
  t ==>* t' →
  ~(stuck t').
Proof.
  intros t t' T HT P. induction P; intros [R S].
  destruct (progress x T HT); auto.
  apply IHP. apply (preservation x y T HT H).
  unfold stuck. split; auto. Qed.
```

24.3 Aside: the *normalize* Tactic

When experimenting with definitions of programming languages in Coq, 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* ==>* *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**.

```
Module NormalizePlayground.
```

```
Import Smallstep.

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_Plus2.
      apply v_const.
      apply ST_PlusConstConst.
  apply multi_step with (C 10).
    apply ST_PlusConstConst.
  apply multi_refl.
Qed.
```

The proof repeatedly applies 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.
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 ⊢ ?x ⇒ idtac x end.

Tactic Notation "normalize" :=
  repeat (print_goal; eapply multi_step ;
            [ (eauto 10; fail) | (instantiate; 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''' : ∃ e',
  (P (C 3) (P (C 3) (C 4)))
  ==>* e'.
Proof.
  eapply ex_intro. normalize.
Qed.
```

Exercise: 1 star (normalize_ex) `Theorem normalize_ex : ∃ e',`

```
  (P (C 3) (P (C 2) (C 1)))
  ==>* e'.
Proof.
   Admitted.
```

☐

Exercise: 1 star, optional (normalize_ex') For comparison, prove it using `apply` instead of `eapply`.

```
Theorem normalize_ex' : ∃ e',
  (P (C 3) (P (C 2) (C 1)))
  ==>* e'.
Proof.
   Admitted.
```

☐

```
End NormalizePlayground.
Tactic Notation "print_goal" :=
  match goal with ⊢ ?x ⇒ idtac x end.
Tactic Notation "normalize" :=
  repeat (print_goal; eapply multi_step ;
            [ (eauto 10; fail) | (instantiate; simpl)]);
  apply multi_refl.
```

24.3.1 Additional Exercises

Exercise: 2 stars, recommendedM (subject_expansion) Having seen the subject reduction property, one might wonder whether the opposity property – subject *expansion* – also holds. That is, is it always the case that, if *t* ==> *t'* and ⊢ *t'* \in T, then ⊢ *t* \in T? If so, prove it. If not, give a counter-example. (You do not need to prove your counter-example in Coq, but feel free to do so.)

☐

Exercise: 2 starsM (variation1) Suppose, that we add this new rule to the typing relation:

| T_SuccBool : forall t, |- t \in TBool -> |- tsucc t \in TBool

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

□

Exercise: 2 starsM (variation2) Suppose, instead, that we add this new rule to the **step** relation:

| ST_Funny1 : forall t2 t3, (tif ttrue t2 t3) ==> 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, optional (variation3) Suppose instead that we add this rule:

| ST_Funny2 : forall t1 t2 t2' t3, t2 ==> t2' -> (tif t1 t2 t3) ==> (tif t1 t2' 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, optional (variation4) Suppose instead that we add this rule:

| ST_Funny3 : (tpred tfalse) ==> (tpred (tpred tfalse))

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, optional (variation5) Suppose instead that we add this rule:

| T_Funny4 : |- tzero \in TBool

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, optional (variation6) Suppose instead that we add this rule:

| T_Funny5 : |- tpred tzero \in TBool

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

Exercise: 3 stars, 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 starM (remove_predzero) The reduction rule *E_PredZero* is a bit counter-intuitive: we might feel that it makes more sense for the predecessor of zero to be undefined, rather than being defined to be zero. Can we achieve this simply by removing the rule from the definition of **step**? Would doing so create any problems elsewhere?

□

Exercise: 4 stars, advancedM (prog_pres_bigstep) Suppose our evaluation relation is defined in the big-step style. What are the appropriate analogs of the progress and preservation properties? (You do not need to prove them.)

□

$Date : 2016 - 12 - 2011 : 35 : 30 - 0500 (Tue, 20 Dec 2016)$

Chapter 25

Library Top.Stlc

25.1 Stlc: The Simply Typed Lambda-Calculus

```
Require Import Maps.
Require Import Smallstep.
Require Import Types.
```

25.2 The Simply Typed Lambda-Calculus

The simply typed lambda-calculus (STLC) is a tiny core calculus embodying the key concept of *functional abstraction*, which 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 which will take some work to deal with these.

25.2.1 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 construction of the language and 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. At the end of the 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 | \x:T1.t2 abstraction | t1 t2 application | true constant true | false constant false | if t1 then t2 else t3 conditional

The \ symbol in a function abstraction \x:*T1.t2* is generally written as a Greek letter "lambda" (hence the name of the calculus). The variable x is called the *parameter* to the function; the term *t2* is its *body*. The annotation :*T1* specifies the type of arguments that the function can be applied to.

Some examples:

- \x:Bool. x

  The identity function for booleans.

- (\x:Bool. x) true

  The identity function for booleans, applied to the boolean true.

- \x:Bool. if x then false else true

  The boolean "not" function.

- \x:Bool. true

  The constant function that takes every (boolean) argument to true.

- \x:Bool. \y:Bool. x

  A two-argument function that takes two booleans and returns the first one. (As in Coq, a two-argument function is really a one-argument function whose body is also a one-argument function.)

- (\x:Bool. \y:Bool. x) false true

  A two-argument function that takes two booleans and returns the first one, applied to the booleans false and true.

  As in Coq, application associates to the left – i.e., this expression is parsed as ((\x:Bool. \y:Bool. x) false) true.

- \f:Bool→Bool. f (f 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.

- (\f:Bool→Bool. f (f true)) (\x:Bool. 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 – all functions are "anonymous." We'll see in chapter MoreStlc that it is easy to add named functions to what we've got – 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 ::= Bool | T1 -> T2

For example:

- \x:Bool. false has type Bool→Bool
- \x:Bool. x has type Bool→Bool
- (\x:Bool. x) true has type Bool
- \x:Bool. \y:Bool. x has type Bool→Bool→Bool (i.e., Bool → (Bool→Bool))
- (\x:Bool. \y:Bool. x) false has type Bool→Bool
- (\x:Bool. \y:Bool. x) false true has type Bool

25.2.2 Syntax

We next formalize the syntax of the STLC.

```
Module STLC.
```

Types

```
Inductive ty : Type :=
  | TBool : ty
  | TArrow : ty → ty → ty.
```

Terms

```
Inductive tm : Type :=
  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm
  | ttrue : tm
```

  | $tfalse$: tm
  | tif : $tm \rightarrow tm \rightarrow tm \rightarrow tm$.

Note that an abstraction \x:$T.t$ (formally, tabs x T t) is always annotated with the type T of its parameter, in contrast to Coq (and other functional languages like ML, Haskell, etc.), which use type inference to fill in missing annotations. We're not considering type inference here.

Some examples...

```
Definition x := (Id "x").
Definition y := (Id "y").
Definition z := (Id "z").
Hint Unfold x.
Hint Unfold y.
Hint Unfold z.
```

idB = \x:Bool. x

```
Notation idB :=
  (tabs x TBool (tvar x)).
```

idBB = \x:Bool→Bool. x

```
Notation idBB :=
  (tabs x (TArrow TBool TBool) (tvar x)).
```

idBBBB = \x:(Bool→Bool) → (Bool→Bool). x

```
Notation idBBBB :=
  (tabs x (TArrow (TArrow TBool TBool)
                      (TArrow TBool TBool))
    (tvar x)).
```

k = \x:Bool. \y:Bool. x

```
Notation k := (tabs x TBool (tabs y TBool (tvar x))).
```

notB = \x:Bool. if x then false else true

```
Notation notB := (tabs x TBool (tif (tvar x) tfalse ttrue)).
```

(We write these as Notations rather than Definitions to make things easier for auto.)

25.2.3 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.

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 clearly 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 \x:T. *t1* is a value only when *t1* 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 \x:T. *t1* is always a value, no matter whether *t1* is one or not – in other words, we can say that reduction stops at abstractions.

Our usual way of evaluating expressions in Coq makes the first choice – for example,
Compute (fun x:bool => 3 + 4)
yields `fun` x:**bool** ⇒ 7.

Most real-world 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 T t,
      value (tabs x T t)
  | v_true :
      value ttrue
  | v_false :
      value tfalse.

Hint Constructors value.
```

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 is *closed* – that is, it contains no free variables.

(Conversely, a term with 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.

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*. In informal discussions, this is usually written [x:=s]*t* and pronounced "substitute x with s in *t*."

Here are some examples:

x:=true (if x then x 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 very important: substituting x with true in \x:Bool. x does *not* yield \x:Bool. true! The reason for this is that the x in the body of \x: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 <> y x:=s(\x:T11. t12) = \x:T11. t12 x:=s(\y:T11. t12) = \y:T11. x:=st12 if x <> y x:=s(t1 t2) = (x:=st1) (x:=st2) x:=strue = true x:=sfalse = false x:=s(if t1 then t2 else t3) = if x:=st1 then x:=st2 else x:=st3

... and formally:

```
Reserved Notation "'[' x ':=' s ']' t" (at level 20).

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' ⇒
      if beq_id x x' then s else t
  | tabs x' T t1 ⇒
      tabs x' T (if beq_id x x' then t1 else ([x:=s] t1))
```

```
  | tapp t1 t2 ⇒
      tapp ([x:=s] t1) ([x:=s] t2)
  | ttrue ⇒
      ttrue
  | tfalse ⇒
      tfalse
  | tif t1 t2 t3 ⇒
      tif ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
  end

where "'[' x ':=' s ']' t" := (subst x s t).
```

Technical note: Substitution 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. Since we are only interested here in defining the **step** relation on closed terms (i.e., terms like \x:Bool. x that include binders for all of the variables they mention), we can avoid this extra complexity here, but it must be dealt with when formalizing richer languages.

See, for example, *Aydemir* 2008 for further discussion of this issue.

Exercise: 3 stars (substi) The definition that we gave above uses Coq'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:id) : tm → tm → Prop :=
  | s_var1 :
      substi s x (tvar x) s


.

Hint Constructors substi.

Theorem substi_correct : ∀ s x t t',
  [x:=s]t = t' ↔ substi s x t t'.
Proof.
  Admitted.
```

☐

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

(\x:T.t12) v2 ==> x:=*v2*t12
is traditionally called “beta-reduction”.
value v2

(ST_AppAbs) (\x:T.t12) v2 ==> x:=*v2*t12
t1 ==> t1’

(ST_App1) t1 t2 ==> t1’ t2
value v1 t2 ==> t2’

(ST_App2) v1 t2 ==> v1 t2’ ... plus the usual rules for booleans:

(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)
Formally:

```
Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T t12 v2,
          value v2 →
          (tapp (tabs x T t12) v2) ==> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
          t1 ==> t1' →
          tapp t1 t2 ==> tapp t1' t2
  | ST_App2 : ∀ v1 t2 t2',
          value v1 →
          t2 ==> t2' →
          tapp v1 t2 ==> tapp v1 t2'
  | ST_IfTrue : ∀ t1 t2,
      (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : ∀ t1 t2,
      (tif tfalse t1 t2) ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      (tif t1 t2 t3) ==> (tif t1' t2 t3)

where "t1 '==>' t2" := (step t1 t2).

Hint Constructors step.
```

```
Notation multistep := (multi step).
Notation "t1 '==>*' t2" := (multistep t1 t2) (at level 40).
```

Examples

Example:
    (\x:Bool->Bool. x) (\x:Bool. x) ==>* \x:Bool. x
    i.e.,
    idBB idB ==>* idB

```
Lemma step_example1 :
  (tapp 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 :
  (tapp idBB (tapp 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) ttrue ==>* tfalse.

```
Lemma step_example3 :
  tapp (tapp idBB notB) ttrue ==>* tfalse.
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:
(\x:Bool -> Bool. x) ((\x:Bool. if x then false else true) true) ==>* false
i.e.,
idBB (notB ttrue) ==>* tfalse.

```
Lemma step_example4 :
  tapp idBB (tapp notB ttrue) ==>* tfalse.
Proof.
  eapply multi_step.
    apply ST_App2. auto.
    apply ST_AppAbs. auto. simpl.
  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 `Types` chapter to simplify these proofs.

```
Lemma step_example1' :
  (tapp idBB idB) ==>* idB.
Proof. normalize. Qed.

Lemma step_example2' :
  (tapp idBB (tapp idBB idB)) ==>* idB.
Proof. normalize. Qed.

Lemma step_example3' :
  tapp (tapp idBB notB) ttrue ==>* tfalse.
Proof. normalize. Qed.

Lemma step_example4' :
  tapp idBB (tapp notB ttrue) ==>* tfalse.
Proof. normalize. Qed.
```

Exercise: 2 stars (step_example3) Try to do this one both with and without *normalize*.

```
Lemma step_example5 :
       tapp (tapp idBBBB idBB) idB
  ==>* idB.
Proof.
   Admitted.

Lemma step_example5_with_normalize :
```

tapp (*tapp idBBBB idBB*) *idB*
==>* *idB*.

Proof.

Admitted.

□

25.2.4 Typing

Next we consider the typing relation of the STLC.

Contexts

Question: What is the type of the term "x y"?

Answer: It depends on the types of x and y!

I.e., in order to assign a type to a term, we need to know what assumptions we should make about the types of its free variables.

This leads us to a three-place *typing judgment*, informally written *Gamma* ⊢ *t* \in T, where *Gamma* is a "typing context" – a mapping from variables to their types.

Informally, we'll write *Gamma*, x:T for "extend the partial function *Gamma* to also map x to T." Formally, we use the function *extend* to add a binding to a partial map.

```
Definition context := partial_map ty.
```

Typing Relation

Gamma x = T

(T_Var) Gamma |- x \in T

Gamma , x:T11 |- t12 \in T12

(T_Abs) Gamma |- \x:T11.t12 \in T11->T12

Gamma |- t1 \in T11->T12 Gamma |- t2 \in T11

(T_App) Gamma |- t1 t2 \in T12

(T_True) Gamma |- true \in Bool

(T_False) Gamma |- false \in Bool

Gamma |- t1 \in Bool Gamma |- t2 \in T Gamma |- t3 \in T

(T_If) Gamma |- if t1 then t2 else t3 \in T

We can read the three-place relation *Gamma* ⊢ *t* \in T as: "to the term *t* we can assign the type T using as types for the free variables of *t* the ones specified in the context *Gamma*."

```
Reserved Notation "Gamma '|-' t '\in' T" (at level 40).
```

```
Inductive has_type : context → tm → ty → Prop :=
  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      Gamma ⊢ tvar x \in T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      update Gamma x T11 ⊢ t12 \in T12 →
      Gamma ⊢ tabs x T11 t12 \in TArrow T11 T12
  | T_App : ∀ T11 T12 Gamma t1 t2,
      Gamma ⊢ t1 \in TArrow T11 T12 →
      Gamma ⊢ t2 \in T11 →
      Gamma ⊢ tapp t1 t2 \in T12
  | T_True : ∀ Gamma,
       Gamma ⊢ ttrue \in TBool
  | T_False : ∀ Gamma,
       Gamma ⊢ tfalse \in TBool
  | T_If : ∀ t1 t2 t3 T Gamma,
       Gamma ⊢ t1 \in TBool →
       Gamma ⊢ t2 \in T →
       Gamma ⊢ t3 \in T →
       Gamma ⊢ tif t1 t2 t3 \in T

where "Gamma '|-' t '\in' T" := (has_type Gamma t T).

Hint Constructors has_type.
```

Examples

```
Example typing_example_1 :
  empty ⊢ tabs x TBool (tvar x) \in TArrow TBool TBool.
Proof.
  apply T_Abs. apply T_Var. reflexivity. Qed.
```

Note that since we added the **has_type** constructors to the hints database, auto can actually solve this one immediately.

```
Example typing_example_1' :
  empty ⊢ tabs x TBool (tvar x) \in TArrow TBool TBool.
Proof. auto. Qed.
```

Another example:
empty |- \x:A. \y:A->A. y (y x)) \in A -> (A->A) -> A.

```
Example typing_example_2 :
  empty ⊢
    (tabs x TBool
       (tabs y (TArrow TBool TBool)
```

```
            (tapp (tvar y) (tapp (tvar y) (tvar x))))) \in
    (TArrow TBool (TArrow (TArrow TBool TBool) TBool)).
Proof with auto using update_eq.
  apply T_Abs.
  apply T_Abs.
  eapply T_App. apply T_Var...
  eapply T_App. apply T_Var...
  apply T_Var...
Qed.
```

Exercise: 2 stars, optional (typing_example_2_full) Prove the same result without using auto, eauto, or eapply (or ...).

```
Example typing_example_2_full :
  empty ⊢
    (tabs x TBool
       (tabs y (TArrow TBool TBool)
          (tapp (tvar y) (tapp (tvar y) (tvar x))))) \in
    (TArrow TBool (TArrow (TArrow TBool TBool) TBool)).
Proof.
   Admitted.
```

□

Exercise: 2 stars (typing_example_3) Formally prove the following typing derivation holds:

empty |- \x:Bool->B. \y:Bool->Bool. \z:Bool. y (x z) \in T.

```
Example typing_example_3 :
  ∃ T,
    empty ⊢
      (tabs x (TArrow TBool TBool)
         (tabs y (TArrow TBool TBool)
            (tabs z TBool
               (tapp (tvar y) (tapp (tvar x) (tvar z)))))) \in
      T.
Proof with auto.
   Admitted.
```

□

We can also show that terms are *not* typable. For example, let's formally 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 : T.

```
Example typing_nonexample_1 :
  ¬ ∃ T,
```

```
    empty ⊢
      (tabs x TBool
          (tabs y TBool
             (tapp (tvar x) (tvar y)))) \in
      T.
Proof.
  intros Hc. inversion Hc.
  inversion H. subst. clear H.
  inversion H5. subst. clear H5.
  inversion H4. subst. clear H4.
  inversion H2. subst. clear H2.
  inversion H5. subst. clear H5.
  inversion H1. Qed.
```

Exercise: 3 stars, optional (typing_nonexample_3) Another nonexample:

~ (exists S, exists T, empty |- \x:S. x x \in T).

```
Example typing_nonexample_3 :
  ¬ (∃ S, ∃ T,
        empty ⊢
          (tabs x S
             (tapp (tvar x) (tvar x))) \in
          T).
Proof.
   Admitted.
   □
End STLC.
```

$Date : 2016-12-2012:03:19-0500(Tue, 20Dec2016)$

Chapter 26

Library Top.StlcProp

26.1 StlcProp: Properties of STLC

```
Require Import Maps.
Require Import Types.
Require Import Stlc.
Require Import Smallstep.
Module STLCProp.
Import STLC.
```

In this chapter, we develop the fundamental theory of the Simply Typed Lambda Calculus – in particular, the type safety theorem.

26.2 Canonical Forms

As we saw for the simple calculus 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 closed values) belonging to each type. For Bool, these are the boolean values ttrue and tfalse. For arrow types, the canonical forms are lambda-abstractions.

```
Lemma canonical_forms_bool : ∀ t,
  empty ⊢ t \in TBool →
  value t →
  (t = ttrue) ∨ (t = tfalse).
Proof.
  intros t HT HVal.
  inversion HVal; intros; subst; try inversion HT; auto.
Qed.
```

```
Lemma canonical_forms_fun : ∀ t T1 T2,
  empty ⊢ t \in (TArrow T1 T2) →
  value t →
```

```
  ∃ x u, t = tabs x T1 u.
Proof.
  intros t T1 T2 HT HVal.
  inversion HVal; intros; subst; try inversion HT; subst; auto.
  ∃ x0. ∃ t0. auto.
Qed.
```

26.3 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'll give the proof in English first, then the formal version.

```
Theorem progress : ∀ t T,
     empty ⊢ t \in T →
     value t ∨ ∃ t', t ==> t'.
```

Proof: By induction on the derivation of $\vdash t$ \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 ⊢ *t1* \in *T2* → T and ⊢ *t2* \in *T2* for some type *T2*. By the induction hypothesis, either *t1* is a value or it can take a reduction step.
  - If *t1* is a value, then consider *t2*, which by the other induction hypothesis 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_If, then *t* = if *t1* then *t2* else *t3*, where *t1* has type Bool. By the IH, *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_If).

```
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  induction Ht; subst Gamma...
  -

    inversion H.

  -

    right. destruct IHHt1...
    +
      destruct IHHt2...
      ×
        assert (∃ x0 t0, t1 = tabs x0 T11 t0).
        eapply canonical_forms_fun; eauto.
        destruct H1 as [x0 [t0 Heq]]. subst.
        ∃ ([x0:=t2]t0)...

      ×
        inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...

    +
      inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...

  -
    right. destruct IHHt1...

    +
      destruct (canonical_forms_bool t1); subst; eauto.

    +
      inversion H as [t1' Hstp]. ∃ (tif t1' t2 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' : ∀ t T,
     empty ⊢ t \in T →
     value t ∨ ∃ t', t ==> t'.
Proof.
  intros t.
  induction t; intros T Ht; auto.
   Admitted.
   □
```

26.4 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, 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) 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 substitition. This time, the tricky cases are the ones for variables and for function abstractions. In both, we discover that we need to take a term s that has been shown to be well-typed in some context *Gamma* and consider the same term s in a slightly different context *Gamma'*. For this we prove a...

- *context invariance* lemma, showing 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. And finally, for this, we need a careful definition of...

- the *free variables* of a term – i.e., those variables mentioned in a term and not in the scope of an enclosing function abstraction binding a variable of the same name.

To make Coq happy, we need to formalize the story in the opposite order...

26.4.1 Free Occurrences

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 \x:T→*U*. x y
- both x and y appear free in (\x:T→*U*. x y) x
- no variables appear free in \x:T→*U*. \y:T. x y

Formally:

Inductive *appears_free_in* : *id* → *tm* → Prop :=
  | *afi_var* : ∀ *x*,

```
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
      y ≠ x →
      appears_free_in x t12 →
      appears_free_in x (tabs y T11 t12)
  | afi_if1 : ∀ x t1 t2 t3,
      appears_free_in x t1 →
      appears_free_in x (tif t1 t2 t3)
  | afi_if2 : ∀ x t1 t2 t3,
      appears_free_in x t2 →
      appears_free_in x (tif t1 t2 t3)
  | afi_if3 : ∀ x t1 t2 t3,
      appears_free_in x t3 →
      appears_free_in x (tif t1 t2 t3).
Hint Constructors appears_free_in.
```

The *free variables* of a term are just the variables that appear free in it. A term with no free variables is said to be *closed*.

```
Definition closed (t:tm) :=
  ∀ x, ¬ appears_free_in x t.
```

An *open* term is one that is not closed (or not known to be closed).

Exercise: 1 starM (afi) 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.

☐

26.4.2 Substitution

To prove that substitution preserves typing, we first need a technical lemma connecting free variables and typing contexts: 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 : ∀ x t T Gamma,
    appears_free_in x t →
    Gamma ⊢ t \in T →
    ∃ T', Gamma x = 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:*T11.t12* and x appears free in *t12*, 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 *t12* is well typed under (*Gamma*, y:*T11*), so the IH allows us to conclude that x is assigned some type by the extended context (*Gamma*, y:*T11*). To conclude that *Gamma* assigns a type to x, we appeal to lemma update_neq, noting that x and y are different variables.

```
Proof.
  intros x t T Gamma H H0. generalize dependent Gamma.
  generalize dependent T.
  induction H;
         intros; try solve [inversion H0; eauto].
  -
    inversion H1; subst.
    apply IHappears_free_in in H7.
    rewrite update_neq in H7; assumption.
Qed.
```

Next, we'll need the fact that any term t that is well typed in the empty context is closed (it has no free variables).

Exercise: 2 stars, optional (typable_empty__closed) Corollary *typable_empty__closed* : ∀ t T,

```
    empty ⊢ t \in T →
    closed t.
Proof.
   Admitted.
```

☐

Sometimes, when we have a proof *Gamma* ⊢ t : T, we will need to replace *Gamma* by a different context *Gamma'*. When is it safe to do this? Intuitively, it must at least be the case that *Gamma'* assigns the same types as *Gamma* to all the variables that appear free in *t*. In fact, this is the only condition that is needed.

```
Lemma context_invariance : ∀ Gamma Gamma' t T,
     Gamma ⊢ t \in T →
     (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
     Gamma' ⊢ t \in T.
```

Proof: By induction on the derivation of *Gamma* ⊢ *t* \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'* ⊢ *t* \in T by T_Var.

- If the last rule was T_Abs, then *t* = \y:*T11*. *t12*, with T = *T11* → *T12* and *Gamma*, y:*T11* ⊢ *t12* \in *T12*. The induction hypothesis is that, for any context *Gamma''*, if *Gamma*, y:*T11* and *Gamma''* assign the same types to all the free variables in *t12*, then *t12* has type *T12* under *Gamma''*. Let *Gamma'* be a context which agrees with *Gamma* on the free variables in *t*; we must show *Gamma'* ⊢ \y:*T11*. *t12* \in *T11* → *T12*.

  By T_Abs, it suffices to show that *Gamma'*, y:*T11* ⊢ *t12* \in *T12*. By the IH (setting *Gamma''* = *Gamma'*, y:*T11*), it suffices to show that *Gamma*, y:*T11* and *Gamma'*, y:*T11* agree on all the variables that appear free in *t12*.

  Any variable occurring free in *t12* must be either y or some other variable. *Gamma*, y:*T11* and *Gamma'*, y:*T11* clearly agree on y. Otherwise, note that any variable other than y that occurs free in *t12* also occurs free in *t* = \y:*T11*. *t12*, and by assumption *Gamma* and *Gamma'* agree on all such variables; hence so do *Gamma*, y:*T11* and *Gamma'*, y:*T11*.

- If the last rule was T_App, then *t* = *t1 t2*, with *Gamma* ⊢ *t1* \in *T2* → T and *Gamma* ⊢ *t2* \in *T2*. One induction hypothesis states that for all contexts *Gamma'*, if *Gamma'* agrees with *Gamma* on the free variables in *t1*, then *t1* has type *T2* → T under *Gamma'*; there is a similar IH for *t2*. We must show that *t1 t2* also has type T under *Gamma'*, given the assumption that *Gamma'* agrees with *Gamma* 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 *Gamma'* as under *Gamma*. 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.

```
Proof with eauto.
  intros.
  generalize dependent Gamma'.
  induction H; intros; auto.
  -
    apply T_Var. rewrite ← H0...
  -
```

```
    apply T_Abs.
    apply IHhas_type. intros x1 Hafi.
    unfold update. unfold t_update. destruct (beq_id x0 x1) eqn: Hx0x1...
    rewrite beq_id_false_iff in Hx0x1. auto.
  -
    apply T_App with T11...
Qed.
```

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: If *Gamma*,x:U ⊢ t \in T and ⊢ v \in U, then *Gamma* ⊢ [x:=v]t \in T.

```
Lemma substitution_preserves_typing : ∀ Gamma x U t v T,
     update Gamma x U ⊢ t \in T →
     empty ⊢ v \in U →
     Gamma ⊢ [x:=v]t \in T.
```

One technical subtlety in the statement of the lemma is that we assign v the type U in the *empty* context – in other words, we assume v is closed. This assumption considerably simplifies the T_Abs case of the proof (compared to assuming *Gamma* ⊢ v \in U, which would be the other reasonable assumption at this point) because the context invariance lemma then tells us that v has type U in any context at all – we don't have to worry about free variables in v clashing with the variable being introduced into the context by T_Abs.

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 *Gamma*,x:U ⊢ t \in T and ⊢ v \in U, then *Gamma* ⊢ [x:=v]t \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 *Gamma*, x:U ⊢ x \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 context invariance: if a closed term has type T in the empty context, it has that type in any context.

- If t is some variable y that is not equal to x, then we need only note that y has the same type under *Gamma*, x: U as under *Gamma*.

- If t is an abstraction \y: *T11. t12*, then the IH tells us, for all *Gamma'* and *T'*, that if *Gamma'*,x: $U \vdash$ *t12* \in *T'* and $\vdash v$ \in U, then *Gamma'* $\vdash$ [x:=v]*t12* \in *T'*.

  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 *Gamma* $\vdash t$ \in T. But we know *Gamma*,x: $U \vdash t$: T, and, since y does not appear free in \y: *T11. t12*, the context invariance lemma yields *Gamma* $\vdash t$ \in T.

  Second, suppose x $\neq$ y. We know *Gamma*,x: U,y: *T11* $\vdash$ *t12* \in *T12* by inversion of the typing relation, from which *Gamma*,y: *T11*,x: $U \vdash$ *t12* \in *T12* follows by the context invariance lemma, so the IH applies, giving us *Gamma*,y: *T11* $\vdash$ [x:=v]*t12* \in *T12*. By T_Abs, *Gamma* $\vdash$ \y: *T11.* [x:=v]*t12* \in *T11*$\rightarrow$*T12*, and by the definition of substitution (noting that x $\neq$ y), *Gamma* $\vdash$ \y: *T11.* [x:=v]*t12* \in *T11*$\rightarrow$*T12* 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.

Technical note: This proof is a rare case where an induction on terms, rather than typing derivations, yields a simpler argument. The reason for this is that the assumption update *Gamma* x $U \vdash t$ \in T is not completely generic, in the sense that one of the "slots" in the typing relation – namely the context – is not just a variable, and this means that Coq's native induction tactic does not give us the induction hypothesis that we want. It is possible to work around this, but the needed generalization is a little tricky. The term t, on the other hand, is completely generic.

```
Proof with eauto.
  intros Gamma x U t v T Ht Ht'.
  generalize dependent Gamma. generalize dependent T.
  induction t; intros T Gamma H;

    inversion H; subst; simpl...
  -
    rename i into y. destruct (beq_idP x y) as [Hxy|Hxy].
    +
      subst.
      rewrite update_eq in H2.
      inversion H2; subst.
```

```
      eapply context_invariance. eassumption.
      apply typable_empty__closed in Ht'. unfold closed in Ht'.
      intros. apply (Ht' x0) in H0. inversion H0.
    +
      apply T_Var. rewrite update_neq in H2...
  -
    rename i into y. rename t into T. apply T_Abs.
    destruct (beq_idP x y) as [Hxy | Hxy].
    +
      subst. rewrite update_shadow in H5. apply H5.
    +
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP y z) as [Hyz | Hyz]; subst; trivial.
      rewrite <- beq_id_false_iff in Hxy.
      rewrite Hxy...
Qed.
```

26.4.3 Main Theorem

We now have the tools we need to prove preservation: if a closed 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 ⊢ *t* \in T.

- We can immediately rule out T_Var, T_Abs, T_True, and T_False as the 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*. There are three cases 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 IH *t1'* has the same type as *t1*, and hence *t1' t2* has the same type as *t1 t2*.
    - The ST_App2 case is similar.
    - If *t1 t2* takes a step by ST_AppAbs, then *t1* = \x:*T11.t12* and *t1 t2* steps to [x:=*t2*]*t12*; the desired result now follows from the fact that substitution preserves types.

- If the last rule in the derivation is T_If, then t = `if` *t1* `then` *t2* `else` *t3*, and there are again three cases depending on how t steps.

  - If t steps to *t2* or *t3*, 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 induction hypothesis.

```
Proof with eauto.
  remember (@empty ty) as Gamma.
  intros t t' T HT. generalize dependent t'.
  induction HT;
       intros t' HE; subst Gamma; subst;
       try solve [inversion HE; subst; auto].
  -
    inversion HE; subst...
    +
      apply substitution_preserves_typing with T11...
      inversion HT1...
Qed.
```

Exercise: 2 stars, recommendedM (subject_expansion_stlc) An exercise in the `Types` chapter asked about the *subject expansion* property for the simple language of arithmetic and boolean expressions. Does this property hold for STLC? That is, is it always the case that, if *t* ==> *t'* and **has_type** *t'* T, then empty ⊢ *t* \in T? If so, prove it. If not, give a counter-example not involving conditionals.

☐

26.5 Type Soundness

Exercise: 2 stars, 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 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.
□

26.6 Uniqueness of Types

Exercise: 3 starsM (types_unique) Another nice property of the STLC is that types are unique: a given term (in a given context) has at most one type. Formalize this statement and prove it.

□

26.7 Additional Exercises

Exercise: 1 starM (progress_preservation_statement) Without peeking at their statements above, write down the progress and preservation theorems for the simply typed lambda-calculus (as Coq theorems).

□

Exercise: 2 starsM (stlc_variation1) Suppose we add a new term *zap* with the following reduction rule

---

(ST_Zap) t ==> zap
and the following typing rule:

(T_Zap) Gamma |- zap : 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

□

Exercise: 2 starsM (stlc_variation2) Suppose instead that we add a new term foo with the following reduction rules:

(ST_Foo1) (\x:A. x) ==> foo

(ST_Foo2) foo ==> 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 starsM (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

□

Exercise: 2 stars, optional (stlc_variation4) Suppose instead that we add the following new rule to the reduction relation:

(ST_FunnyIfTrue) (if true then t1 else t2) ==> 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, optional (stlc_variation5) Suppose instead that we add the following new rule to the typing relation:

Gamma |- t1 \in Bool->Bool->Bool Gamma |- t2 \in Bool

(T_FunnyApp) Gamma |- t1 t2 \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, optional (stlc_variation6) Suppose instead that we add the following new rule to the typing relation:

Gamma |- t1 \in Bool Gamma |- t2 \in Bool

(T_FunnyApp') Gamma |- t1 t2 \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, optional (stlc_variation7) Suppose we add the following new rule to the typing relation of the STLC:

(T_FunnyAbs) |- \x:Bool.t \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*.

26.7.1 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 :=
  | TArrow : ty → ty → ty
  | TNat : ty.
```

To terms, we add natural number constants, along with successor, predecessor, multiplication, and zero-testing.

```
Inductive tm : Type :=
  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm
  | tnat : nat → tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tmult : tm → tm → tm
  | tif0 : tm → tm → tm → tm.
```

Exercise: 4 starsM (stlc_arith) Finish formalizing the definition and properties of the STLC extended with arithmetic. Specifically:

- Copy the core definitions and theorems for STLC that we went through above (from the definition of values through the Preservation theorem, inclusive), and paste it into the file at this point. Do not copy examples, exercises, etc. (In particular, make sure you don't copy any of the □ comments at the end of exercises, to avoid confusing the autograder.)
- Extend the definitions of the `subst` operation and the **step** relation to include appropriate clauses for the arithmetic operators.
- Extend the proofs of all the properties (up to preservation) of the original STLC to deal with the new syntactic forms. Make sure Coq accepts the whole file.

□

```
End STLCArith.
```

$Date : 2016-12-2012:03:19-0500(Tue, 20Dec2016)$

Chapter 27

Library Top.MoreStlc

27.1 MoreStlc: More on the Simply Typed Lambda-Calculus

```
Require Import Maps.
Require Import Types.
Require Import Smallstep.
Require Import Stlc.
```

27.2 Simple Extensions to STLC

The simply typed lambda-calculus has 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.

27.2.1 Numbers

As we saw in exercise *stlc_arith* 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.

27.2.2 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 Coq), 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 ::= Terms | ... (other terms same as before) | let x=t in t let-binding

Reduction:

t1 ==> t1'

(ST_Let1) let x=t1 in t2 ==> let x=t1' in t2

(ST_LetValue) let x=v1 in t2 ==> `x:=`*v1* t2

Typing:

Gamma |- t1 : T1 Gamma, x:T1 |- t2 : T2

(T_Let) Gamma |- let x=t1 in t2 : T2

27.2.3 Pairs

Our functional programming examples in Coq 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 Coq, 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:

\x : Nat*Nat. let sum = x.fst + x.snd in let diff = x.fst - x.snd 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.fst* for the first projection from *t* and *t.snd* for the second projection – plus one new type constructor, $T1 \times T2$, called the *product* of *T1* and *T2*.

Syntax:

t ::= Terms | (t,t) pair | t.fst first projection | t.snd second projection | ...

v ::= Values | (v,v) pair value | ...

T ::= Types | T * T product type | ...

For reduction, we need several new rules specifying how pairs and projection behave.

t1 ==> t1'

(ST_Pair1) (t1,t2) ==> (t1',t2)
t2 ==> t2'

(ST_Pair2) (v1,t2) ==> (v1,t2')
t1 ==> t1'

(ST_Fst1) t1.fst ==> t1'.fst

(ST_FstPair) (v1,v2).fst ==> v1
t1 ==> t1'

(ST_Snd1) t1.snd ==> t1'.snd

(ST_SndPair) (v1,v2).snd ==> 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 |- t1 : T1 Gamma |- t2 : T2

(T_Pair) Gamma |- (t1,t2) : T1*T2
Gamma |- t1 : T11*T12

(T_Fst) Gamma |- t1.fst : T11
Gamma |- t1 : T11*T12

(T_Snd) Gamma |- t1.snd : T12

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 *t1* has a product type $T11 \times T12$ (i.e., if it will reduce to a pair), then the types of the projections from this pair are *T11* and *T12*.

27.2.4 Unit

Another handy base type, found especially in languages in the ML family, 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 ::= Terms | unit unit value | ...

v ::= Values | unit unit | ...

T ::= Types | Unit Unit type | ...

Typing:

(T_Unit) Gamma |- unit : 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.

27.2.5 Sums

Many programs need to deal with values that can take two distinct forms. For example, we might identify employees in an accounting application 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

inl : Nat -> Nat + Bool inr : Bool -> Nat + 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*.

One important usage of sums is signaling errors:

div : Nat -> Nat -> (Nat + Unit) = div = \x:Nat. \y:Nat. if iszero y then inr unit else inl ... The type Nat + *Unit* above is in fact isomorphic to **option** nat in Coq – 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 Coq's `match`) to destruct them. For example, the following procedure converts a Nat+Bool into a Nat:

getNat = \x:Nat+Bool. case x of inl n => n | inr b => if b then 1 else 0 More formally...

Syntax:

t ::= Terms | inl T t tagging (left) | inr T t tagging (right) | case t of case inl x => t | inr x => t | ...

v ::= Values | inl T v tagged value (left) | inr T v tagged value (right) | ...

T ::= Types | T + T sum type | ...

Reduction:

t1 ==> t1'

(ST_Inl) inl T t1 ==> inl T t1'

t1 ==> t1'

(ST_Inr) inr T t1 ==> inr T t1'

t0 ==> t0'

(ST_Case) case t0 of inl x1 => t1 | inr x2 => t2 ==> case t0' of inl x1 => t1 | inr x2 => t2

(ST_CaseInl) case (inl T v0) of inl x1 => t1 | inr x2 => t2 ==> *x1*:=*v0*t1

(ST_CaseInr) case (inr T v0) of inl x1 => t1 | inr x2 => t2 ==> *x2*:=*v0*t2

Typing:

Gamma |- t1 : T1

(T_Inl) Gamma |- inl T2 t1 : T1 + T2

Gamma |- t1 : T2

(T_Inr) Gamma |- inr T1 t1 : T1 + T2

Gamma |- t0 : T1+T2 Gamma , x1:T1 |- t1 : T Gamma , x2:T2 |- t2 : T

(T_Case) Gamma |- case t0 of inl x1 => t1 | inr x2 => t2 : T

We use the type annotation in *inl* and *inr* to make the typing relation simpler, similarly to what 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 *inl* 5 : Nat + Nat and *inl* 5 : Nat + 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.

27.2.6 Lists

The typing features we have seen can be classified into *base types* like Bool, and *type constructors* like → and × 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 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 Coq. We use *lcase* 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. lcase x of nil -> 0 | a::x' -> lcase x' of nil -> a | b::x" -> a+b

Syntax:

t ::= Terms | nil T | cons t t | lcase t of nil -> t | x::x -> t | ...

v ::= Values | nil T nil value | cons v v cons value | ...

T ::= Types | List T list of Ts | ...

Reduction:

t1 ==> t1'

(ST_Cons1) cons t1 t2 ==> cons t1' t2

t2 ==> t2'

(ST_Cons2) cons v1 t2 ==> cons v1 t2'

t1 ==> t1'

(ST_Lcase1) (lcase t1 of nil -> t2 | xh::xt -> t3) ==> (lcase t1' of nil -> t2 | xh::xt -> t3)

(ST_LcaseNil) (lcase nil T of nil -> t2 | xh::xt -> t3) ==> t2

(ST_LcaseCons) (lcase (cons vh vt) of nil -> t2 | xh::xt -> t3) ==> $xh{:=}vh{,}xt{:=}vt$t3

Typing:

(T_Nil) Gamma |- nil T : List T
Gamma |- t1 : T Gamma |- t2 : List T

(T_Cons) Gamma |- cons t1 t2: List T
Gamma |- t1 : List T1 Gamma |- t2 : T Gamma , h:T1, t:List T1 |- t3 : T

(T_Lcase) Gamma |- (lcase t1 of nil -> t2 | h::t -> t3) : T

27.2.7 General Recursion

Another facility found in most programming languages (including Coq) is the ability to define recursive functions. For example, we might like to be able to define the factorial function like this:

fact = \x:Nat. if x=0 then 1 else x * (fact (pred x)))

Note that the right-hand side of this binder mentions the variable being bound – something that is not allowed by our formalization of `let` above.

Directly formalizing this "recursive definition" mechanism is possible, but it requires a bit of extra effort: in particular, we'd have to pass around an "environment" of recursive function definitions in the definition of the **step** relation.

Here is another way of presenting recursive functions that is equally powerful (though not quite as convenient for the programmer) and more straightforward to formalize: instead of writing recursive definitions, we 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 Nat→Nat.
- Use all of this as the right-hand side of an ordinary `let`-binding for fact.

The intuition 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 (Nat→Nat)->(Nat→Nat) is a function f of type Nat→Nat such that F f behaves the same as f.)

Syntax:

t ::= Terms | fix t fixed-point operator | ...

Reduction:

t1 ==> t1'

(ST_Fix1) fix t1 ==> fix t1'

(ST_FixAbs) fix (\xf:T1.t2) ==> *xf*:=`fix` (*xf*:*T1.t2*) t2

Typing:

Gamma |- t1 : T1->T1

(T_Fix) Gamma |- fix t1 : 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

==> *ST_FixAbs* + ST_App1

(\x. if x=0 then 1 else x * (fix F (pred x))) 3

==> ST_AppAbs

if 3=0 then 1 else 3 * (fix F (pred 3))

==> ST_If0_Nonzero

3 * (fix F (pred 3))

==> *ST_FixAbs* + ST_Mult2

3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 3))

==> ST_PredNat + ST_Mult2 + ST_App2

3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 2)

==> ST_AppAbs + 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

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

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

One important point to note is that, unlike `Fixpoint` definitions in Coq, there is nothing to prevent functions defined using `fix` from diverging.

Exercise: 1 star, optional (halve_fix) Translate this informal recursive definition into one using `fix`:

halve = \x:Nat. if x=0 then 0 else if (pred x)=0 then 0 else 1 + (halve (pred (pred x))))

☐

Exercise: 1 star, 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).

☐

The ability to form the fixed point of a function of type T→T for any 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 T, we can define the term

fix (\x: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 (\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))

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):

evenodd = fix (\eo: (Nat->Bool * Nat->Bool). let e = \n:Nat. if n=0 then true else eo.snd (pred n) in let o = \n:Nat. if n=0 then false else eo.fst (pred n) in (e,o))

even = evenodd.fst odd = evenodd.snd

27.2.8 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 ::= Terms | {i1=t1, ..., in=tn} record | t.i projection | ...
v ::= Values | {i1=v1, ..., in=vn} record value | ...
T ::= Types | {i1:T1, ..., in:Tn} 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 ==> ti'

(ST_Rcd) {i1=v1, ..., im=vm, in=ti, ...} ==> {i1=v1, ..., im=vm, in=ti', ...}
t1 ==> t1'

(ST_Proj1) t1.i ==> t1'.i

(ST_ProjRcd) {..., i=vi, ...}.i ==> 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 only be one field called i, and that all the other fields must contain values.

The typing rules are also simple:
Gamma |- t1 : T1 ... Gamma |- tn : Tn

(T_Rcd) Gamma |- {i1=t1, ..., in=tn} : {i1:T1, ..., in:Tn}
Gamma |- t : {..., i:Ti, ...}

(T_Proj) Gamma |- t.i : 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**.

First, observe that we can encode arbitrary-size *tuples* using nested pairs and the **unit** value. To avoid overloading the pair notation (*t1*,*t2*), 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.

{} —-> unit {t1, t2, ..., tn} —-> (t1, trest) where {t2, ..., tn} —-> trest

Similarly, we can encode tuple types using nested product types:

{} —-> Unit {T1, T2, ..., Tn} —-> T1 * TRest where {T2, ..., Tn} —-> 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 —-> t.fst t.(n+1) —-> (t.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} —-> {5,6} {a=5, c=7} —-> {5,unit,7} {c=7, a=5} —-> {5,unit,7} {c=5, b=3} —-> {unit,3,5} {f=8,c=5,a=7} —-> {7,unit,5,unit,unit,8} {f=8,c=5} —-> {unit,unit,5,unit,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:Nat, b:Nat} —-> {Nat,Nat} {c:Nat, a:Nat} —-> {Nat,Unit,Nat} {f:Nat,c:Nat} —-> {Unit,Unit,Nat,Unit,Unit,Nat}

Finally, record projection is encoded as a tuple projection from the appropriate position: t.l —-> t.(position of l)

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 <l1:*T1*,l2:*T2*,...*ln*:*Tn*> where l1,l2,... 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.

27.3 Exercise: Formalizing the Extensions

Exercise: 5 stars (STLC_extensions) In this exercise, 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, in two passes, rather than trying to work through the file from start to finish 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.

```
Module STLCExtended.
```

Syntax

```
Inductive ty : Type :=
  | TArrow : ty → ty → ty
  | TNat : ty
  | TUnit : ty
  | TProd : ty → ty → ty
  | TSum : ty → ty → ty
  | TList : ty → ty.

Inductive tm : Type :=

  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm

  | tnat : nat → tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tmult : tm → tm → tm
  | tif0 : tm → tm → tm → tm

  | tpair : tm → tm → tm
  | tfst : tm → tm
  | tsnd : tm → tm

  | tunit : tm

  | tlet : id → tm → tm → tm


  | tinl : ty → tm → tm
  | tinr : ty → tm → tm
```

| *tcase* : *tm* → *id* → *tm* → *id* → *tm* → *tm*

| *tnil* : *ty* → *tm*
| *tcons* : *tm* → *tm* → *tm*
| *tlcase* : *tm* → *tm* → *id* → *id* → *tm* → *tm*

| *tfix* : *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 ...

Substitution

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒
      if beq_id x y then s else t
  | tabs y T t1 ⇒
      tabs y T (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒
      tapp (subst x s t1) (subst x s t2)
  | tnat n ⇒
      tnat n
  | tsucc t1 ⇒
      tsucc (subst x s t1)
  | tpred t1 ⇒
      tpred (subst x s t1)
  | tmult t1 t2 ⇒
      tmult (subst x s t1) (subst x s t2)
  | tif0 t1 t2 t3 ⇒
      tif0 (subst x s t1) (subst x s t2) (subst x s t3)

  | tunit ⇒ tunit

  | tinl T t1 ⇒
      tinl T (subst x s t1)
  | tinr T t1 ⇒
      tinr T (subst x s t1)
```

```
  | tcase t0 y1 t1 y2 t2 ⇒
      tcase (subst x s t0)
          y1 (if beq_id x y1 then t1 else (subst x s t1))
          y2 (if beq_id x y2 then t2 else (subst x s t2))
  | tnil T ⇒
      tnil T
  | tcons t1 t2 ⇒
      tcons (subst x s t1) (subst x s t2)
  | tlcase t1 t2 y1 y2 t3 ⇒
      tlcase (subst x s t1) (subst x s t2) y1 y2
        (if beq_id x y1 then
           t3
         else if beq_id x y2 then t3
              else (subst x s t3))

  | _ ⇒ t
  end.
Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Reduction

Next we define the values of our language.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T11 t12,
      value (tabs x T11 t12)

  | v_nat : ∀ n1,
      value (tnat n1)

  | v_pair : ∀ v1 v2,
      value v1 →
      value v2 →
      value (tpair v1 v2)

  | v_unit : value tunit

  | v_inl : ∀ v T,
      value v →
      value (tinl T v)
  | v_inr : ∀ v T,
      value v →
      value (tinr T v)
```

```
  | v_lnil : ∀ T, value (tnil T)
  | v_lcons : ∀ v1 vl,
      value v1 →
      value vl →
      value (tcons v1 vl)
  .

Hint Constructors value.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T11 t12 v2,
          value v2 →
          (tapp (tabs x T11 t12) v2) ==> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
          t1 ==> t1' →
          (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
          value v1 →
          t2 ==> t2' →
          (tapp v1 t2) ==> (tapp v1 t2')

  | ST_Succ1 : ∀ t1 t1',
        t1 ==> t1' →
        (tsucc t1) ==> (tsucc t1')
  | ST_SuccNat : ∀ n1,
        (tsucc (tnat n1)) ==> (tnat (S n1))
  | ST_Pred : ∀ t1 t1',
        t1 ==> t1' →
        (tpred t1) ==> (tpred t1')
  | ST_PredNat : ∀ n1,
        (tpred (tnat n1)) ==> (tnat (pred n1))
  | ST_Mult1 : ∀ t1 t1' t2,
        t1 ==> t1' →
        (tmult t1 t2) ==> (tmult t1' t2)
  | ST_Mult2 : ∀ v1 t2 t2',
        value v1 →
        t2 ==> t2' →
        (tmult v1 t2) ==> (tmult v1 t2')
  | ST_MultNats : ∀ n1 n2,
        (tmult (tnat n1) (tnat n2)) ==> (tnat (mult n1 n2))
  | ST_If01 : ∀ t1 t1' t2 t3,
        t1 ==> t1' →
```

```
      (tif0 t1 t2 t3) ==> (tif0 t1' t2 t3)
  | ST_If0Zero : ∀ t2 t3,
      (tif0 (tnat 0) t2 t3) ==> t2
  | ST_If0Nonzero : ∀ n t2 t3,
      (tif0 (tnat (S n)) t2 t3) ==> t3

  | ST_Inl : ∀ t1 t1' T,
       t1 ==> t1' →
       (tinl T t1) ==> (tinl T t1')
  | ST_Inr : ∀ t1 t1' T,
       t1 ==> t1' →
       (tinr T t1) ==> (tinr T t1')
  | ST_Case : ∀ t0 t0' x1 t1 x2 t2,
       t0 ==> t0' →
       (tcase t0 x1 t1 x2 t2) ==> (tcase t0' x1 t1 x2 t2)
  | ST_CaseInl : ∀ v0 x1 t1 x2 t2 T,
       value v0 →
       (tcase (tinl T v0) x1 t1 x2 t2) ==> [x1:=v0]t1
  | ST_CaseInr : ∀ v0 x1 t1 x2 t2 T,
       value v0 →
       (tcase (tinr T v0) x1 t1 x2 t2) ==> [x2:=v0]t2

  | ST_Cons1 : ∀ t1 t1' t2,
      t1 ==> t1' →
      (tcons t1 t2) ==> (tcons t1' t2)
  | ST_Cons2 : ∀ v1 t2 t2',
      value v1 →
      t2 ==> t2' →
      (tcons v1 t2) ==> (tcons v1 t2')
  | ST_Lcase1 : ∀ t1 t1' t2 x1 x2 t3,
      t1 ==> t1' →
      (tlcase t1 t2 x1 x2 t3) ==> (tlcase t1' t2 x1 x2 t3)
  | ST_LcaseNil : ∀ T t2 x1 x2 t3,
      (tlcase (tnil T) t2 x1 x2 t3) ==> t2
  | ST_LcaseCons : ∀ v1 vl t2 x1 x2 t3,
      value v1 →
      value vl →
      (tlcase (tcons v1 vl) t2 x1 x2 t3) ==> (subst x2 vl (subst x1 v1 t3))
```

```
where "t1 '==>' t2" := (step t1 t2).
```

```
Notation multistep := (multi step).
Notation "t1 '==>*' t2" := (multistep t1 t2) (at level 40).
```

```
Hint Constructors step.
```

Typing

```
Definition context := partial_map ty.
```

Next we define the typing rules. These are nearly direct transcriptions of the inference rules shown above.

```
Reserved Notation "Gamma '|-' t '\in' T" (at level 40).
```

```
Inductive has_type : context → tm → ty → Prop :=

  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      Gamma ⊢ (tvar x) \in T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      (update Gamma x T11) ⊢ t12 \in T12 →
      Gamma ⊢ (tabs x T11 t12) \in (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma t1 t2,
      Gamma ⊢ t1 \in (TArrow T1 T2) →
      Gamma ⊢ t2 \in T1 →
      Gamma ⊢ (tapp t1 t2) \in T2

  | T_Nat : ∀ Gamma n1,
      Gamma ⊢ (tnat n1) \in TNat
  | T_Succ : ∀ Gamma t1,
      Gamma ⊢ t1 \in TNat →
      Gamma ⊢ (tsucc t1) \in TNat
  | T_Pred : ∀ Gamma t1,
      Gamma ⊢ t1 \in TNat →
      Gamma ⊢ (tpred t1) \in TNat
  | T_Mult : ∀ Gamma t1 t2,
      Gamma ⊢ t1 \in TNat →
      Gamma ⊢ t2 \in TNat →
      Gamma ⊢ (tmult t1 t2) \in TNat
  | T_If0 : ∀ Gamma t1 t2 t3 T1,
      Gamma ⊢ t1 \in TNat →
```

```
      Gamma ⊢ t2 \in T1 →
      Gamma ⊢ t3 \in T1 →
      Gamma ⊢ (tif0 t1 t2 t3) \in T1



  | T_Unit : ∀ Gamma,
      Gamma ⊢ tunit \in TUnit



  | T_Inl : ∀ Gamma t1 T1 T2,
      Gamma ⊢ t1 \in T1 →
      Gamma ⊢ (tinl T2 t1) \in (TSum T1 T2)
  | T_Inr : ∀ Gamma t2 T1 T2,
      Gamma ⊢ t2 \in T2 →
      Gamma ⊢ (tinr T1 t2) \in (TSum T1 T2)
  | T_Case : ∀ Gamma t0 x1 T1 t1 x2 T2 t2 T,
      Gamma ⊢ t0 \in (TSum T1 T2) →
      (update Gamma x1 T1) ⊢ t1 \in T →
      (update Gamma x2 T2) ⊢ t2 \in T →
      Gamma ⊢ (tcase t0 x1 t1 x2 t2) \in T

  | T_Nil : ∀ Gamma T,
      Gamma ⊢ (tnil T) \in (TList T)
  | T_Cons : ∀ Gamma t1 t2 T1,
      Gamma ⊢ t1 \in T1 →
      Gamma ⊢ t2 \in (TList T1) →
      Gamma ⊢ (tcons t1 t2) \in (TList T1)
  | T_Lcase : ∀ Gamma t1 T1 t2 x1 x2 t3 T2,
      Gamma ⊢ t1 \in (TList T1) →
      Gamma ⊢ t2 \in T2 →
      (update (update Gamma x2 (TList T1)) x1 T1) ⊢ t3 \in T2 →
      Gamma ⊢ (tlcase t1 t2 x1 x2 t3) \in T2



where "Gamma '|-' t '\in' T" := (has_type Gamma t T).

Hint Constructors has_type.
```

27.3.1 Examples

This section presents formalized versions of the examples from above (plus several more). The ones 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:

```
Notation x := (Id "x").
Notation y := (Id "y").
Notation a := (Id "a").
Notation f := (Id "f").
Notation g := (Id "g").
Notation l := (Id "l").
Notation k := (Id "k").
Notation i1 := (Id "i1").
Notation i2 := (Id "i2").
Notation processSum := (Id "processSum").
Notation n := (Id "n").
Notation eq := (Id "eq").
Notation m := (Id "m").
Notation evenodd := (Id "evenodd").
Notation even := (Id "even").
Notation odd := (Id "odd").
Notation eo := (Id "eo").
```

Next, a bit of Coq 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* ⊢ (`tapp` *e1 e1*) `\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 _ (tapp _ _) _) ⇒
  eapply T_App; auto.
Hint Extern 2 (has_type _ (tlcase _ _ _ _ _) _) ⇒
  eapply T_Lcase; auto.
```

```
Hint Extern 2 (_ = _) ⇒ compute; reflexivity.
```

Numbers

```
Module Numtest.

Definition test :=
  tif0
    (tpred
      (tsucc
        (tpred
          (tmult
            (tnat 2)
            (tnat 0)))))
    (tnat 5)
    (tnat 6).
```

Remove the comment braces once you've implemented enough of the definitions that you think this should work.

```
End Numtest.
```

Products

```
Module Prodtest.

Definition test :=
  tsnd
    (tfst
      (tpair
        (tpair
          (tnat 5)
          (tnat 6))
        (tnat 7))).

End Prodtest.
```

`let`

```
Module LetTest.

Definition test :=
  tlet
    x
    (tpred (tnat 6))
    (tsucc (tvar x)).
```

End *LetTest*.

Sums

Module *Sumtest1*.

```
Definition test :=
  tcase (tinl TNat (tnat 5))
    x (tvar x)
    y (tvar y).
```

End *Sumtest1*.

Module *Sumtest2*.

```
Definition test :=
  tlet
    processSum
    (tabs x (TSum TNat TNat)
      (tcase (tvar x)
          n (tvar n)
          n (tif0 (tvar n) (tnat 1) (tnat 0))))
    (tpair
      (tapp (tvar processSum) (tinl TNat (tnat 5)))
      (tapp (tvar processSum) (tinr TNat (tnat 5)))).
```

End *Sumtest2*.

Lists

Module *ListTest*.

```
Definition test :=
  tlet l
    (tcons (tnat 5) (tcons (tnat 6) (tnil TNat)))
    (tlcase (tvar l)
       (tnat 0)
       x y (tmult (tvar x) (tvar x))).
```

End *ListTest*.

`fix`

Module *FixTest1*.

Definition *fact* :=

```
  tfix
    (tabs f (TArrow TNat TNat)
      (tabs a TNat
        (tif0
            (tvar a)
            (tnat 1)
            (tmult
                (tvar a)
                (tapp (tvar f) (tpred (tvar a))))))).
```

(Warning: you may be able to typecheck fact but still have some rules wrong!)

```
End FixTest1.

Module FixTest2.

Definition map :=
  tabs g (TArrow TNat TNat)
    (tfix
      (tabs f (TArrow (TList TNat) (TList TNat))
        (tabs l (TList TNat)
          (tlcase (tvar l)
            (tnil TNat)
            a l (tcons (tapp (tvar g) (tvar a))
                              (tapp (tvar f) (tvar l))))))).

End FixTest2.

Module FixTest3.

Definition equal :=
  tfix
    (tabs eq (TArrow TNat (TArrow TNat TNat))
      (tabs m TNat
        (tabs n TNat
          (tif0 (tvar m)
            (tif0 (tvar n) (tnat 1) (tnat 0))
            (tif0 (tvar n)
              (tnat 0)
              (tapp (tapp (tvar eq)
                              (tpred (tvar m)))
                        (tpred (tvar n)))))))).

End FixTest3.

Module FixTest4.
```

```
Definition eotest :=
  tlet evenodd
    (tfix
      (tabs eo (TProd (TArrow TNat TNat) (TArrow TNat TNat))
        (tpair
          (tabs n TNat
            (tif0 (tvar n)
              (tnat 1)
              (tapp (tsnd (tvar eo)) (tpred (tvar n)))))
          (tabs n TNat
            (tif0 (tvar n)
              (tnat 0)
              (tapp (tfst (tvar eo)) (tpred (tvar n))))))))
  (tlet even (tfst (tvar evenodd))
  (tlet odd (tsnd (tvar evenodd))
  (tpair
    (tapp (tvar even) (tnat 3))
    (tapp (tvar even) (tnat 4))))).

End FixTest4.

End Examples.
```

27.3.2 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

```
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.
      ∃ (subst x t2 t12)...
    ×
      inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
  +
    inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...
-
  left...
-
  right.
  destruct IHHt...
  +
    inversion H; subst; try solve_by_invert.
    ∃ (tnat (S n1))...
  +
    inversion H as [t1' Hstp].
    ∃ (tsucc t1')...
-
  right.
  destruct IHHt...
  +
    inversion H; subst; try solve_by_invert.
    ∃ (tnat (pred n1))...
  +
    inversion H as [t1' Hstp].
    ∃ (tpred t1')...
-
  right.
  destruct IHHt1...
  +
    destruct IHHt2...
```

```
    *
      inversion H; subst; try solve_by_invert.
      inversion H0; subst; try solve_by_invert.
      exists (tnat (mult n1 n0))...
    *
      inversion H0 as [t2' Hstp].
      exists (tmult t1 t2')...
  +
    inversion H as [t1' Hstp].
    exists (tmult t1' t2)...
-
  right.
  destruct IHHt1...
  +
    inversion H; subst; try solve_by_invert.
    destruct n1 as [|n1'].
    *
      exists t2...
    *
      exists t3...
  +
    inversion H as [t1' H0].
    exists (tif0 t1' t2 t3)...
-
  left...
-
  destruct IHHt...
  +
    right. inversion H as [t1' Hstp]...
-
  destruct IHHt...
  +
    right. inversion H as [t1' Hstp]...
-
  right.
  destruct IHHt1...
  +
    inversion H; subst; try solve_by_invert.
    *
      exists ([x1:=v]t1)...
    *
      exists ([x2:=v]t2)...
```

```
      +
        inversion H as [t0' Hstp].
        ∃ (tcase t0' x1 t1 x2 t2)...
  -
    left...
  -
    destruct IHHt1...
    +
      destruct IHHt2...
      ×
        right. inversion H0 as [t2' Hstp].
        ∃ (tcons t1 t2')...
    +
      right. inversion H as [t1' Hstp].
      ∃ (tcons t1' t2)...
  -
    right.
    destruct IHHt1...
    +
      inversion H; subst; try solve_by_invert.
      ×
        ∃ t2...
      ×
        ∃ ([x2:=vl]([x1:=v1]t3))...
    +
      inversion H as [t1' Hstp].
      ∃ (tlcase t1' t2 x1 x2 t3)...
Qed.
```

Context Invariance

```
Inductive appears_free_in : id → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
        y ≠ x →
        appears_free_in x t12 →
        appears_free_in x (tabs y T11 t12)
```

```
| afi_succ : ∀ x t,
    appears_free_in x t →
    appears_free_in x (tsucc t)
| afi_pred : ∀ x t,
    appears_free_in x t →
    appears_free_in x (tpred t)
| afi_mult1 : ∀ x t1 t2,
    appears_free_in x t1 →
    appears_free_in x (tmult t1 t2)
| afi_mult2 : ∀ x t1 t2,
    appears_free_in x t2 →
    appears_free_in x (tmult t1 t2)
| afi_if01 : ∀ x t1 t2 t3,
    appears_free_in x t1 →
    appears_free_in x (tif0 t1 t2 t3)
| afi_if02 : ∀ x t1 t2 t3,
    appears_free_in x t2 →
    appears_free_in x (tif0 t1 t2 t3)
| afi_if03 : ∀ x t1 t2 t3,
    appears_free_in x t3 →
    appears_free_in x (tif0 t1 t2 t3)
```

```
| afi_inl : ∀ x t T,
      appears_free_in x t →
      appears_free_in x (tinl T t)
| afi_inr : ∀ x t T,
      appears_free_in x t →
      appears_free_in x (tinr T t)
| afi_case0 : ∀ x t0 x1 t1 x2 t2,
      appears_free_in x t0 →
      appears_free_in x (tcase t0 x1 t1 x2 t2)
| afi_case1 : ∀ x t0 x1 t1 x2 t2,
      x1 ≠ x →
      appears_free_in x t1 →
      appears_free_in x (tcase t0 x1 t1 x2 t2)
| afi_case2 : ∀ x t0 x1 t1 x2 t2,
      x2 ≠ x →
      appears_free_in x t2 →
```

```
        appears_free_in x (tcase t0 x1 t1 x2 t2)

  | afi_cons1 : ∀ x t1 t2,
      appears_free_in x t1 →
      appears_free_in x (tcons t1 t2)
  | afi_cons2 : ∀ x t1 t2,
      appears_free_in x t2 →
      appears_free_in x (tcons t1 t2)
  | afi_lcase1 : ∀ x t1 t2 y1 y2 t3,
      appears_free_in x t1 →
      appears_free_in x (tlcase t1 t2 y1 y2 t3)
  | afi_lcase2 : ∀ x t1 t2 y1 y2 t3,
      appears_free_in x t2 →
      appears_free_in x (tlcase t1 t2 y1 y2 t3)
  | afi_lcase3 : ∀ x t1 t2 y1 y2 t3,
      y1 ≠ x →
      y2 ≠ x →
      appears_free_in x t3 →
      appears_free_in x (tlcase t1 t2 y1 y2 t3)


.
Hint Constructors appears_free_in.

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 with eauto.
  intros. generalize dependent Gamma'.
  induction H;
    intros Gamma' Heqv...
  -
    apply T_Var... rewrite ← Heqv...
  -
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold update, t_update.
    destruct (beq_idP x y)...
  -
    apply T_Mult...
  -
    apply T_If0...
  -
```

```
    eapply T_Case...
    + apply IHhas_type2. intros y Hafi.
      unfold update, t_update.
      destruct (beq_idP x1 y)...
    + apply IHhas_type3. intros y Hafi.
      unfold update, t_update.
      destruct (beq_idP x2 y)...
  -
    apply T_Cons...
  -
    eapply T_Lcase... apply IHhas_type3. intros y Hafi.
    unfold update, t_update.
    destruct (beq_idP x1 y)...
    destruct (beq_idP x2 y)...
Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   Gamma ⊢ t \in T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  induction Htyp; inversion Hafi; subst...
  -
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx...
  -
    destruct IHHtyp2 as [T' Hctx]... ∃ T'.
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx...
  -
    destruct IHHtyp3 as [T' Hctx]... ∃ T'.
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx...
  -
    clear Htyp1 IHHtyp1 Htyp2 IHHtyp2.
    destruct IHHtyp3 as [T' Hctx]... ∃ T'.
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx...
    rewrite false_beq_id in Hctx...
Qed.
```

Substitution

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     (update Gamma x U) ⊢ t \in S →
     empty ⊢ v \in U →
     Gamma ⊢ ([x:=v]t) \in S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  induction t;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  -
    simpl. rename i into y.
    unfold update, t_update in H1.
    destruct (beq_idP x y).
    +

      subst.
      inversion H1; subst. clear H1.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra)
        as [T' HT']...
      inversion HT'.
    +

      apply T_Var...
  -
    rename i into y. rename t into T11.
    apply T_Abs...
    destruct (beq_idP x y) as [Hxy|Hxy].
    +

      eapply context_invariance...
      subst.
      intros x Hafi. unfold update, t_update.
      destruct (beq_id y x)...
    +

      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP y z) as [Hyz|Hyz]...
      subst.
```

```
    rewrite false_beq_id...
-
  rename i into x1. rename i0 into x2.
  eapply T_Case...
  +
    destruct (beq_idP x x1) as [Hxx1|Hxx1].
    ×
      eapply context_invariance...
      subst.
      intros z Hafi. unfold update, t_update.
      destruct (beq_id x1 z)...
    ×
      apply IHt2. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP x1 z) as [Hx1z|Hx1z]...
      subst. rewrite false_beq_id...
  +
    destruct (beq_idP x x2) as [Hxx2|Hxx2].
    ×
      eapply context_invariance...
      subst.
      intros z Hafi. unfold update, t_update.
      destruct (beq_id x2 z)...
    ×
      apply IHt3. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP x2 z)...
      subst. rewrite false_beq_id...
-
  rename i into y1. rename i0 into y2.
  eapply T_Lcase...
  destruct (beq_idP x y1).
  +
    simpl.
    eapply context_invariance...
    subst.
    intros z Hafi. unfold update, t_update.
    destruct (beq_idP y1 z)...
  +
    destruct (beq_idP x y2).
    ×
      eapply context_invariance...
```

```
        subst.
        intros z Hafi. unfold update, t_update.
        destruct (beq_idP y2 z)...
      ×
        apply IHt3. eapply context_invariance...
        intros z Hafi. unfold update, t_update.
        destruct (beq_idP y1 z)...
        subst. rewrite false_beq_id...
        destruct (beq_idP y2 z)...
        subst. rewrite false_beq_id...
Qed.
```

Preservation

```
Theorem preservation : ∀ t t' T,
     empty ⊢ t \in T →
     t ==> t' →
     empty ⊢ t' \in T.
Proof with eauto.
  intros t t' T HT.
  remember empty as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  induction HT;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  -

    inversion HE; subst...
    +

      apply substitution_preserves_typing with T1...
      inversion HT1...
  -
    inversion HT1; subst.
    eapply substitution_preserves_typing...
  -
    inversion HT1; subst.
    eapply substitution_preserves_typing...
  -
    +
      inversion HT1; subst.
      apply substitution_preserves_typing with (TList T1)...
      apply substitution_preserves_typing with T1...
Qed.
```

`End` *STLCExtended*.

□

Chapter 28

Library Top.Sub

28.1 Sub: Subtyping

```
Require Import Maps.
Require Import Types.
Require Import Smallstep.
```

28.2 Concepts

We now turn to the study of *subtyping*, a key feature needed to support the object-oriented programming style.

28.2.1 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 lamdba-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 one-field record to an argument that actually provides two 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.

28.2.2 Subtyping and Object-Oriented Languages

Subtyping plays a fundamental role in many programming languages – in particular, it is closely related to the notion of *subclassing* in object-oriented languages.

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 *a1..an* roughly consists of projecting out the m field of *o* and applying it to *a1..an*.

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 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, if you are interested.) Instead, we'll study the core concepts behind the subclass / subinterface relation in the simplified setting of the STLC.

28.2.3 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 |- t : S S <: T

(T_Sub) Gamma |- t : T

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 is a record with two fields (e.g., $\mathsf{S} = \{\mathsf{x{:}A{\to}A},\ \mathsf{y{:}B{\to}B}\}$), but choose to forget about one of the fields ($\mathsf{T} = \{\mathsf{y{:}B{\to}B}\}$) so that we can pass t to a function that requires just a single-field record.

28.2.4 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 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.

S1 <: T1 S2 <: T2

(S_Prod) S1 * S2 <: T1 * T2

Arrows

The subtyping rule for arrows is a little less intuitive. Suppose we have functions f and g with these types:

f : C -> Student g : (C->Person) -> D

That is, f is a function that yields a record of type $\mathsf{Student}$, 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) S1 -> S2 <: S1 -> T2

We can generalize this to allow the arguments of the two arrow types to be in the subtype relation as well:

T1 <: S1 S2 <: T2

(S_Arrow) S1 -> S2 <: T1 -> 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 : Person -> C g : (Student -> C) -> 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 Person → C is a subtype of Student → 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,

{name:String, age:Nat, gpa:Nat} <: {name:String, age:Nat}

{name:String, age:Nat} <: {name:String} {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:Student} <: {x: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,

{name:String,age:Nat} <: {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 {*age*:Nat,*name*:String} <: {*name*:String}.

Second, subtyping can be applied inside the components of a compound record type:

S1 <: T1 ... 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:Student, x:Nat} <: {y:Person}.

Third, subtyping can reorder fields. For example, we want {*name*:String, *gpa*:Nat, *age*:Nat} <: Person. (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:

- A subclass may 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).
- 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).

Exercise: 2 stars, recommendedM (arrow_sub_wrong) Suppose we had incorrectly defined subtyping as covariant on both the right and the left of arrow types:
S1 <: T1 S2 <: T2

(S_Arrow_wrong) S1 -> S2 <: T1 -> T2
Give a concrete example of functions `f` and `g` with the following types...
f : Student -> Nat g : (Person -> Nat) -> 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.)
☐

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 <: 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*,
- adding the rule of subsumption

  Gamma |- t : S S <: T

  - ———————— (T_Sub) Gamma |- t : T

  to the typing relation, and
- defining a subtype relation as follows:

  S <: U U <: T

  - ————— (S_Trans) S <: T
    - —— (S_Refl)

T <: T

- ——- (S_Top)

S <: Top

S1 <: T1 S2 <: T2

- —————- (S_Prod) S1 * S2 <: T1 * T2

T1 <: S1 S2 <: T2

- —————- (S_Arrow)

S1 -> S2 <: T1 -> T2

n > m

- ———————— (S_RcdWidth)

{i1:T1...in:Tn} <: {i1:T1...im:Tm}

S1 <: T1 ... Sn <: Tn

- ———————— (S_RcdDepth)

{i1:S1...in:Sn} <: {i1:T1...in:Tn}

{i1:S1...in:Sn} is a permutation of {j1:T1...jn:Tn}

- ———————————— (S_RcdPerm) {i1:S1...in:Sn} <: {j1:T1...jn:Tn}

28.2.5 Exercises

Exercise: 1 star, 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→S <: T→S
- *Top*→U <: S→*Top*
- (C→C) → (A×B) <: (C→C) → (*Top*×B)
- T→T→U <: S→S→V
- (T→T)->U <: (S→S)->V
- ((T→S)->T)->U <: ((S→T)->S)->V
- S×V <: T×U

☐

Exercise: 2 starsM (subtype_order) The following types happen to form a linear order with respect to subtyping:

- Top
- $\mathit{Top} \to \mathsf{Student}$
- $\mathsf{Student} \to \mathsf{Person}$
- $\mathsf{Student} \to \mathit{Top}$
- $\mathsf{Person} \to \mathsf{Student}$

Write these types in order from the most specific to the most general.
Where does the type $\mathit{Top}{\to}\mathit{Top}{\to}\mathsf{Student}$ fit into this order?
□

Exercise: 1 starM (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
□

Exercise: 1 starM (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 *S0*, *S1*, etc., such that all the *Si*'s are different and each $\mathsf{S(i{+}1)}$ is a subtype of *Si*.

- There is an infinite *ascending* chain of distinct types in the subtype relation—that is, an infinite sequence of types *S0*, *S1*, etc., such that all the *Si*'s are different and each S(i+1) is a supertype of *Si*.

☐

Exercise: 2 starsM (proper_subtypes) Is the following statement true or false? Briefly explain your answer. (Here TBase 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 = TBool \/ exists n, T = TBase n) -> exists S, S <: T /\ S <> T

☐

Exercise: 2 starsM (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.)

  empty |- (\p:T*Top. p.fst) ((\z:A.z), unit) : A->A

- What is the *largest* type T that makes the same assertion true?

☐

Exercise: 2 starsM (small_large_2)

- What is the *smallest* type T that makes the following assertion true?

  empty |- (\p:(A->A * B->B). p) ((\z:A.z), (\z:B.z)) : T

- What is the *largest* type T that makes the same assertion true?

☐

Exercise: 2 stars, optional (small_large_3)

- What is the *smallest* type T that makes the following assertion true?

  a:A |- (\p:(A*T). (p.snd) (p.fst)) (a , \z:A.z) : A

- What is the *largest* type T that makes the same assertion true?

☐

Exercise: 2 starsM (small_large_4)

- What is the *smallest* type T that makes the following assertion true?

  exists S, empty |- (\p:(A*T). (p.snd) (p.fst)) : S

- What is the *largest* type T that makes the same assertion true?

☐

Exercise: 2 starsM (smallest_1) What is the *smallest* type T that makes the following assertion true?

exists S, exists t, empty |- (\x:T. x x) t : S]] ☐

Exercise: 2 starsM (smallest_2) What is the *smallest* type T that makes the following assertion true?

empty |- (\x:Top. x) ((\z:A.z) , (\z:B.z)) : T]] ☐

Exercise: 3 stars, optional (count_supertypes) How many supertypes does the record type {x:A, y:C→C} have? That is, how many different types T are there such that {x:A, y:C→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 starsM (pair_permutation) The subtyping rule for product types

S1 <: T1 S2 <: T2

(S_Prod) 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.

☐

28.3 Formal Definitions

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.

28.3.1 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 :=
  | TTop : ty
  | TBool : ty
  | TBase : id → ty
  | TArrow : ty → ty → ty
  | TUnit : ty
.

Inductive tm : Type :=
  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm
  | tunit : tm
.
```

Substitution

The definition of substitution remains exactly the same as for the pure STLC.

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒
      if beq_id x y then s else t
  | tabs y T t1 ⇒
      tabs y T (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒
      tapp (subst x s t1) (subst x s t2)
  | ttrue ⇒
      ttrue
  | tfalse ⇒
      tfalse
  | tif t1 t2 t3 ⇒
      tif (subst x s t1) (subst x s t2) (subst x s t3)
```

```
  | tunit ⇒
      tunit
  end.
Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Reduction

Likewise the definitions of the **value** property and the **step** relation.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_true :
      value ttrue
  | v_false :
      value tfalse
  | v_unit :
      value tunit
.
Hint Constructors value.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T t12 v2,
         value v2 →
         (tapp (tabs x T t12) v2) ==> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
         t1 ==> t1' →
         (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
         value v1 →
         t2 ==> t2' →
         (tapp v1 t2) ==> (tapp v1 t2')
  | ST_IfTrue : ∀ t1 t2,
      (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : ∀ t1 t2,
      (tif tfalse t1 t2) ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      (tif t1 t2 t3) ==> (tif t1' t2 t3)
where "t1 '==>' t2" := (step t1 t2).

Hint Constructors step.
```

28.3.2 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 <: TTop
  | S_Arrow : ∀ S1 S2 T1 T2,
      T1 <: S1 →
      S2 <: T2 →
      (TArrow S1 S2) <: (TArrow T1 T2)
where "T '<:' U" := (subtype T U).
```

Note that we don't need any special rules for base types (TBool and TBase): they are automatically subtypes of themselves (by S_Refl) and *Top* (by S_Top), and that's all we want.

```
Hint Constructors subtype.

Module Examples.

Notation x := (Id "x").
Notation y := (Id "y").
Notation z := (Id "z").

Notation A := (TBase (Id "A")).
Notation B := (TBase (Id "B")).
Notation C := (TBase (Id "C")).

Notation String := (TBase (Id "String")).
Notation Float := (TBase (Id "Float")).
Notation Integer := (TBase (Id "Integer")).

Example subtyping_example_0 :
  (TArrow C TBool) <: (TArrow C TTop).
Proof. auto. Qed.
```

Exercise: 2 stars, optional (subtyping_judgements) (Wait to do this exercise after you have added 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 Coq. To get full benefit from the exercises, make sure you also understand how to prove them on paper!

Exercise: 1 star, optional (subtyping_example_1)

```
Example subtyping_example_1 :
  (TArrow TTop Student) <: (TArrow (TArrow C C) Person).
Proof with eauto.
  Admitted.
```

□

Exercise: 1 star, optional (subtyping_example_2)

```
Example subtyping_example_2 :
  (TArrow TTop Person) <: (TArrow Person TTop).
Proof with eauto.
  Admitted.
```

□

```
End Examples.
```

28.3.3 Typing

The only change to the typing relation is the addition of the rule of subsumption, T_Sub.

```
Definition context := partial_map ty.
```

```
Reserved Notation "Gamma '|-' t '\in' T" (at level 40).
```

```
Inductive has_type : context → tm → ty → Prop :=

  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      Gamma ⊢ (tvar x) \in T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      (update Gamma x T11) ⊢ t12 \in T12 →
      Gamma ⊢ (tabs x T11 t12) \in (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma t1 t2,
      Gamma ⊢ t1 \in (TArrow T1 T2) →
      Gamma ⊢ t2 \in T1 →
      Gamma ⊢ (tapp t1 t2) \in T2
  | T_True : ∀ Gamma,
       Gamma ⊢ ttrue \in TBool
  | T_False : ∀ Gamma,
       Gamma ⊢ tfalse \in TBool
  | T_If : ∀ t1 t2 t3 T Gamma,
       Gamma ⊢ t1 \in TBool →
       Gamma ⊢ t2 \in T →
       Gamma ⊢ t3 \in T →
       Gamma ⊢ (tif t1 t2 t3) \in T
  | T_Unit : ∀ Gamma,
      Gamma ⊢ tunit \in TUnit

  | T_Sub : ∀ Gamma t S T,
      Gamma ⊢ t \in S →
      S <: T →
      Gamma ⊢ t \in T

where "Gamma '|-' t '\in' T" := (has_type Gamma t T).
```

```
Hint Constructors has_type.
```

The following hints help `auto` and `eauto` construct typing derivations. (See chapter UseAuto for more on hints.)

```
Hint Extern 2 (has_type _ (tapp _ _) _) ⇒
  eapply T_App; auto.
Hint Extern 2 (_ = _) ⇒ compute; reflexivity.
```

```
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 Coq and prove it.

Exercise: 1 star, optional (typing_example_0) □

Exercise: 2 stars, optional (typing_example_1) □

Exercise: 2 stars, optional (typing_example_2) □

End *Examples2*.

28.4 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 References), 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.

28.4.1 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, optional (sub_inversion_Bool) Lemma *sub_inversion_Bool* : ∀ *U*,
U <: *TBool* →
U = *TBool*.
Proof with auto.
intros *U Hs*.
remember TBool as *V*.
Admitted.

Exercise: 3 stars, optional (sub_inversion_arrow) Lemma *sub_inversion_arrow* : ∀ *U V1 V2*,
U <: (*TArrow V1 V2*) →
∃ *U1*, ∃ *U2*,
U = (*TArrow U1 U2*) ∧ (*V1* <: *U1*) ∧ (*U2* <: *V2*).

```
Proof with eauto.
  intros U V1 V2 Hs.
  remember (TArrow V1 V2) as V.
  generalize dependent V2. generalize dependent V1.
   Admitted.
```
□

28.4.2 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*→*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* ⊢ *t1* : *T11*→*T12* is subsumption, then there is some *sub*-derivation whose subject is also *t1*, 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, optional (canonical_forms_of_arrow_types)

```
Lemma canonical_forms_of_arrow_t
: ∀ Gamma s T1 T2,
  Gamma ⊢ s \in (TArrow T1 T2) →
  value s →
  ∃ x, ∃ S1, ∃ s2,
     s = tabs x S1 s2.
Proof with eauto.
   Admitted.
```
□

Similarly, the canonical forms of type Bool are the constants true and false.

```
Lemma canonical_forms_of_Bool : ∀ Gamma s,
  Gamma ⊢ s \in TBool →
  value s →
  (s = ttrue ∨ s = tfalse).
Proof with eauto.
  intros Gamma s Hty Hv.
  remember TBool as T.
```

```
  induction Hty; try solve_by_invert...
  -
    subst. apply sub_inversion_Bool in H. subst...
Qed.
```

28.4.3 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 empty ⊢ *t* : T then *t* is a value or *t* ==> *t'* for some term *t'*.

Proof: Let *t* and T be given, with empty ⊢ *t* : 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 *t1 t2* and types *T1* and *T2* such that *t* = *t1 t2*, T = *T2*, empty ⊢ *t1* : *T1* → *T2*, and empty ⊢ *t2* : *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:

  - Suppose *t1* ==> *t1'* for some term *t1'*. Then *t1 t2* ==> *t1' t2* by ST_App1.
  - Suppose *t1* is a value and *t2* ==> *t2'* for some term *t2'*. Then *t1 t2* ==> *t1 t2'* by rule ST_App2 because *t1* is a value.
  - Finally, suppose *t1* and *t2* are both values. By the canonical forms lemma for arrow types, we know that *t1* has the form \x:*S1*.*s2* for some x, *S1*, and *s2*. But then (\x:*S1*.*s2*) *t2* ==> [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* = `if` *t1* `then` *t2* `else` *t3*, with empty ⊢ *t1* : Bool and with empty ⊢ *t2* : T and empty ⊢ *t3* : 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* = true or *t1* = 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 S such that S <: T and empty ⊢ *t* : S. The desired result is exactly the induction hypothesis for the typing subderivation.

```
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)...
      ×
        inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    +
      inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...
  -
    right.
    destruct IHHt1.
    + eauto.
    + assert (t1 = ttrue ∨ t1 = tfalse)
        by (eapply canonical_forms_of_Bool; eauto).
      inversion H0; subst...
    + inversion H. rename x into t1'. eauto.
Qed.
```

28.4.4 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* ⊢ \x:*S1.t2* : T whose subject is an abstraction, then there must be some subderivation giving a type to the body *t2*.

Lemma: If *Gamma* ⊢ \x:*S1.t2* : T, then there is a type *S2* such that *Gamma*, x:*S1* ⊢ *t2* : *S2* and *S1* → *S2* <: T.

(Notice that the lemma does *not* say, "then T itself is an arrow type" – this is tempting, but false!)

Proof: Let *Gamma*, x, *S1*, *t2* and T be given as described. Proceed by induction on the derivation of *Gamma* ⊢ \x:*S1.t2* : T. Cases T_Var, 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* → *T12* and *Gamma*, x:*S1* ⊢ *t2* : *T12*. Picking *T12* for *S2* gives us what we need: *S1* → *T12* <: *S1* → *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* ⊢ \x:*S1.t2* : S. The IH for the typing subderivation tell us that there is some type *S2* with *S1* → *S2* <: S and *Gamma*, x:*S1* ⊢ *t2* : *S2*. Picking type *S2* gives us what we need, since *S1* → *S2* <: T then follows by S_Trans.

```
Lemma typing_inversion_abs : ∀ Gamma x S1 t2 T,
      Gamma ⊢ (tabs x S1 t2) \in T →
      (∃ S2, (TArrow S1 S2) <: T
                 ∧ (update Gamma x S1) ⊢ t2 \in S2).
Proof with eauto.
  intros Gamma x S1 t2 T H.
  remember (tabs x S1 t2) as t.
  induction H;
    inversion Heqt; subst; intros; try solve_by_invert.
  -
    ∃ T12...
  -
    destruct IHhas_type as [S2 [Hsub Hty]]...
  Qed.
```

Similarly...

```
Lemma typing_inversion_var : ∀ Gamma x T,
  Gamma ⊢ (tvar x) \in T →
  ∃ S,
    Gamma x = Some S ∧ S <: T.
Proof with eauto.
  intros Gamma x T Hty.
  remember (tvar x) as t.
  induction Hty; intros;
```

```
    inversion Heqt; subst; try solve_by_invert.
  -
    ∃ T...
  -
    destruct IHHty as [U [Hctx HsubU]]... Qed.
Lemma typing_inversion_app : ∀ Gamma t1 t2 T2,
  Gamma ⊢ (tapp t1 t2) \in T2 →
  ∃ T1,
    Gamma ⊢ t1 \in (TArrow T1 T2) ∧
    Gamma ⊢ t2 \in T1.
Proof with eauto.
  intros Gamma t1 t2 T2 Hty.
  remember (tapp t1 t2) as t.
  induction Hty; intros;
    inversion Heqt; subst; try solve_by_invert.
  -
    ∃ T1...
  -
    destruct IHHty as [U1 [Hty1 Hty2]]...
Qed.
Lemma typing_inversion_true : ∀ Gamma T,
  Gamma ⊢ ttrue \in T →
  TBool <: T.
Proof with eauto.
  intros Gamma T Htyp. remember ttrue as tu.
  induction Htyp;
    inversion Heqtu; subst; intros...
Qed.
Lemma typing_inversion_false : ∀ Gamma T,
  Gamma ⊢ tfalse \in T →
  TBool <: T.
Proof with eauto.
  intros Gamma T Htyp. remember tfalse as tu.
  induction Htyp;
    inversion Heqtu; subst; intros...
Qed.
Lemma typing_inversion_if : ∀ Gamma t1 t2 t3 T,
  Gamma ⊢ (tif t1 t2 t3) \in T →
  Gamma ⊢ t1 \in TBool
  ∧ Gamma ⊢ t2 \in T
  ∧ Gamma ⊢ t3 \in T.
```

```
Proof with eauto.
  intros Gamma t1 t2 t3 T Hty.
  remember (tif t1 t2 t3) as t.
  induction Hty; intros;
    inversion Heqt; subst; try solve_by_invert.
  -
    auto.
  -
    destruct (IHHty H0) as [H1 [H2 H3]]...
Qed.

Lemma typing_inversion_unit : ∀ Gamma T,
  Gamma ⊢ tunit \in T →
    TUnit <: T.
Proof with eauto.
  intros Gamma T Htyp. remember tunit 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 ⊢ (tabs x S1 s2) \in (TArrow T1 T2) →
     T1 <: S1
  ∧ (update empty x S1) ⊢ s2 \in T2.
Proof with eauto.
  intros x S1 s2 T1 T2 Hty.
  apply typing_inversion_abs in Hty.
  inversion Hty as [S2 [Hsub Hty1]].
  apply sub_inversion_arrow in Hsub.
  inversion Hsub as [U1 [U2 [Heq [Hsub1 Hsub2]]]].
  inversion Heq; subst... Qed.
```

28.4.5 Context Invariance

The context invariance lemma follows the same pattern as in the pure STLC.

```
Inductive appears_free_in : id → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
```

```
        appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
          y ≠ x →
          appears_free_in x t12 →
          appears_free_in x (tabs y T11 t12)
  | afi_if1 : ∀ x t1 t2 t3,
        appears_free_in x t1 →
        appears_free_in x (tif t1 t2 t3)
  | afi_if2 : ∀ x t1 t2 t3,
        appears_free_in x t2 →
        appears_free_in x (tif t1 t2 t3)
  | afi_if3 : ∀ x t1 t2 t3,
        appears_free_in x t3 →
        appears_free_in x (tif t1 t2 t3)
.

Hint Constructors appears_free_in.

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 with eauto.
  intros. generalize dependent Gamma'.
  induction H;
    intros Gamma' Heqv...
  -
    apply T_Var... rewrite ← Heqv...
  -
    apply T_Abs... apply IHhas_type. intros x0 Hafi.
    unfold update, t_update. destruct (beq_idP x x0)...
  -
    apply T_If...
Qed.

Lemma free_in_context : ∀ x t T Gamma,
    appears_free_in x t →
    Gamma ⊢ t \in T →
    ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  induction Htyp;
      subst; inversion Hafi; subst...
  -
    destruct (IHHtyp H4) as [T Hctx]. ∃ T.
```

```
    unfold update, t_update in Hctx.
    rewrite ← beq_id_false_iff in H2.
    rewrite H2 in Hctx... Qed.
```

28.4.6 Substitution

The *substitution lemma* is proved along the same lines as for the pure STLC. The only significant change is that there are several places where, instead of the built-in `inversion` tactic, we need to use the inversion lemmas that we proved above to extract structural information from assumptions about the well-typedness of subterms.

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     (update Gamma x U) ⊢ t \in S →
     empty ⊢ v \in U →
     Gamma ⊢ ([x:=v]t) \in S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent S. generalize dependent Gamma.
  induction t; intros; simpl.
  -
    rename i into y.
    destruct (typing_inversion_var _ _ _ Htypt)
        as [T [Hctx Hsub]].
    unfold update, t_update in Hctx.
    destruct (beq_idP x y) as [Hxy|Hxy]; eauto;
    subst.
    inversion Hctx; subst. clear Hctx.
    apply context_invariance with empty...
    intros x Hcontra.
    destruct (free_in_context _ _ S empty Hcontra)
        as [T' HT']...
    inversion HT'.
  -
    destruct (typing_inversion_app _ _ _ _ Htypt)
        as [T1 [Htypt1 Htypt2]].
    eapply T_App...
  -
    rename i into y. rename t into T1.
    destruct (typing_inversion_abs _ _ _ _ _ Htypt)
      as [T2 [Hsub Htypt2]].
    apply T_Sub with (TArrow T1 T2)... apply T_Abs...
    destruct (beq_idP x y) as [Hxy|Hxy].
    +
```

```
      eapply context_invariance...
      subst.
      intros x Hafi. unfold update, t_update.
      destruct (beq_id y x)...
    +
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP y z)...
      subst.
      rewrite ← beq_id_false_iff in Hxy. rewrite Hxy...
  -
      assert (TBool <: S)
        by apply (typing_inversion_true _ _ Htypt)...
  -
      assert (TBool <: S)
        by apply (typing_inversion_false _ _ Htypt)...
  -
    assert ((update Gamma x U) ⊢ t1 \in TBool
            ∧ (update Gamma x U) ⊢ t2 \in S
            ∧ (update Gamma x U) ⊢ t3 \in S)
      by apply (typing_inversion_if _ _ _ _ _ Htypt).
    inversion H as [H1 [H2 H3]].
    apply IHt1 in H1. apply IHt2 in H2. apply IHt3 in H3.
    auto.
  -
    assert (TUnit <: S)
      by apply (typing_inversion_unit _ _ Htypt)...
Qed.
```

28.4.7 Preservation

The proof of preservation now proceeds pretty much as in earlier chapters, using the substitution lemma at the appropriate point and again using inversion lemmas from above to extract structural information from typing assumptions.

Theorem (Preservation): If *t*, *t'* are terms and T is a type such that empty ⊢ *t* : T and *t* ==> *t'*, then empty ⊢ *t'* : T.

Proof: Let *t* and T be given such that empty ⊢ *t* : T. We proceed by induction on the structure of this typing derivation, leaving *t'* general. The cases 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*, empty ⊢ *t1* : *T1* → *T2*, and empty ⊢

t2 : *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* = \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* ⊢ *s2* : *T2*. It then follows by the substitution lemma (substitution_preserves_typing) that empty ⊢ [x:=*t2*] *t12* : *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* = `if` *t1* `then` *t2* `else` *t3*, with empty ⊢ *t1* : Bool and with empty ⊢ *t2* : T and empty ⊢ *t3* : T. Moreover, by the induction hypothesis, if *t1* steps to *t1'* then empty ⊢ *t1'* : Bool. There are three cases to consider, depending on which rule was used to show *t* ==> *t'*.
  - If *t* ==> *t'* by rule ST_If, then *t'* = `if` *t1'* `then` *t2* `else` *t3* with *t1* ==> *t1'*. By the induction hypothesis, empty ⊢ *t1'* : Bool, and so empty ⊢ *t'* : T by T_If.
  - If *t* ==> *t'* by rule ST_IfTrue or ST_IfFalse, then either *t'* = *t2* or *t'* = *t3*, and empty ⊢ *t'* : 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 empty ⊢ *t* : S. The result is immediate by the induction hypothesis for the typing subderivation and an application of T_Sub. ☐

```
Theorem preservation : ∀ t t' T,
     empty ⊢ t \in T →
     t ==> t' →
     empty ⊢ t' \in T.
Proof with eauto.
  intros t t' T HT.
  remember empty as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  induction HT;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  -
    inversion HE; subst...
    +
      destruct (abs_arrow _ _ _ _ _ HT1) as [HA1 HA2].
      apply substitution_preserves_typing with T...
Qed.
```

28.4.8 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:Nat, b:Nat} —-> {Nat,Nat} i.e., (Nat,(Nat,Top)) {c:Nat, a:Nat} —-> {Nat,Top,Nat} i.e., (Nat,(Top,(Nat,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.

28.4.9 Exercises

Exercise: 2 starsM (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:

  Gamma |- t : S1->S2 S1 <: T1 T1 <: S1 S2 <: T2

  - ———————————————- (T_Funny1) Gamma |- t : T1->T2

- Suppose we add the following reduction rule:

  - ————————- (ST_Funny21)

  unit ==> (\x:Top. x)

- Suppose we add the following subtyping rule:

  - ——————— (S_Funny3)

  Unit <: Top->Top

- Suppose we add the following subtyping rule:

  - ——————— (S_Funny4)

Top->Top <: Unit

- Suppose we add the following reduction rule:

  - ——————— (ST_Funny5)

  (unit t) ==> (t unit)

- Suppose we add the same reduction rule *and* a new typing rule:

  - ——————— (ST_Funny5)

  (unit t) ==> (t unit)

  - ———————— (T_Funny6)

  empty |- unit : Top->Top

- Suppose we *change* the arrow subtyping rule to:

  S1 <: T1 S2 <: T2

  - ——————- (S_Arrow')

  S1->S2 <: T1->T2

☐

28.5 Exercise: Adding Products

Exercise: 4 stars (products) Adding pairs, projections, and product types to the system we have defined is a relatively straightforward matter. Carry out this extension:

- Below, we've added constructors for pairs, first and second projections, and product types to the definitions of **ty** and **tm**.

- Copy the definitions of the substitution function and value relation from above and extend them as in chapter *MoreSTLC* to include products.

- Similarly, copy and extend the operational semantics with the same reduction rules as in chapter *MoreSTLC*.

- (Copy and) extend the subtyping relation with this rule:

  S1 <: T1 S2 <: T2

  - ——————— (Sub_Prod)

S1 * S2 <: T1 * T2

- Extend the typing relation with the same rules for pairs and projections as in chapter *MoreSTLC*.
- 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.)

```
Module ProductExtension.

Inductive ty : Type :=
  | TTop : ty
  | TBool : ty
  | TBase : id → ty
  | TArrow : ty → ty → ty
  | TUnit : ty
  | TProd : ty → ty → ty.

Inductive tm : Type :=
  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm
  | tunit : tm
  | tpair : tm → tm → tm
  | tfst : tm → tm
  | tsnd : tm → tm.

Theorem progress : ∀ t T,
      empty ⊢ t \in T →
      value t ∨ ∃ t', t ==> t'.
Proof.
  Admitted.

Theorem preservation : ∀ t t' T,
      empty ⊢ t \in T →
      t ==> t' →
      empty ⊢ t' \in T.
Proof.
  Admitted.

End ProductExtension.
```

□

Date : 2016 − 12 − 2013 : 03 : 18 − 0500(*Tue*, 20*Dec*2016)

Chapter 29

Library Top.Typechecking

29.1 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, tell us how to *check* 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.

```
Require Import Coq.Bool.Bool.
Require Import Maps.
Require Import Smallstep.
Require Import Stlc.

Module STLCChecker.
Import STLC.
```

29.2 Comparing Types

First, we need a function to compare two types for equality...

```
Fixpoint beq_ty (T1 T2:ty) : bool :=
  match T1,T2 with
  | TBool, TBool ⇒
      true
  | TArrow T11 T12, TArrow T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
  | _,_ ⇒
      false
```

```
  end.
```

... and we need to establish the usual two-way connection between the boolean result returned by beq_ty and the logical proposition that its inputs are equal.

```
Lemma beq_ty_refl : ∀ T1,
  beq_ty T1 T1 = true.
Proof.
  intros T1. induction T1; simpl.
    reflexivity.
    rewrite IHT1_1. rewrite IHT1_2. reflexivity. Qed.

Lemma beq_ty__eq : ∀ T1 T2,
  beq_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.
```

29.3 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 tapp 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 TArrow *T11 T12* for some *T1* and *T2*).

```
Fixpoint type_check (Gamma:context) (t:tm) : option ty :=
  match t with
  | tvar x ⇒
      Gamma x
  | tabs x T11 t12 ⇒
      match type_check (update Gamma x T11) t12 with
      | Some T12 ⇒ Some (TArrow T11 T12)
      | _ ⇒ None
      end
  | tapp t1 t2 ⇒
      match type_check Gamma t1, type_check Gamma t2 with
      | Some (TArrow T11 T12),Some T2 ⇒
          if beq_ty T11 T2 then Some T12 else None
      | _,_ ⇒ None
```

```
    end
  | ttrue ⇒
      Some TBool
  | tfalse ⇒
      Some TBool
  | tif guard t f ⇒
      match type_check Gamma guard with
      | Some TBool ⇒
          match type_check Gamma t, type_check Gamma f with
          | Some T1, Some T2 ⇒
              if beq_ty T1 T2 then Some T1 else None
          | _,_ ⇒ None
          end
      | _ ⇒ None
      end
  end.
```

29.4 Properties

To verify that this 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 : ∀ Gamma t T,
  type_check Gamma t = Some T → has_type Gamma t T.
Proof with eauto.
  intros Gamma t. generalize dependent Gamma.
  induction t; intros Gamma T Htc; inversion Htc.
  - eauto.
  -
    remember (type_check Gamma t1) as TO1.
    remember (type_check Gamma t2) as TO2.
    destruct TO1 as [T1|]; try solve_by_invert;
    destruct T1 as [|T11 T12]; try solve_by_invert.
    destruct TO2 as [T2|]; try solve_by_invert.
    destruct (beq_ty T11 T2) eqn: Heqb;
    try solve_by_invert.
    apply beq_ty__eq in Heqb.
    inversion H0; subst...
  -
    rename i into y. rename t into T1.
    remember (update Gamma y T1) as G'.
    remember (type_check G' t0) as TO2.
```

```
    destruct TO2; try solve_by_invert.
    inversion H0; subst...
  - eauto.
  - eauto.
  -
    remember (type_check Gamma t1) as TOc.
    remember (type_check Gamma t2) as TO1.
    remember (type_check Gamma t3) as TO2.
    destruct TOc as [Tc|]; try solve_by_invert.
    destruct Tc; try solve_by_invert.
    destruct TO1 as [T1|]; try solve_by_invert.
    destruct TO2 as [T2|]; try solve_by_invert.
    destruct (beq_ty T1 T2) eqn:Heqb;
    try solve_by_invert.
    apply beq_ty__eq in Heqb.
    inversion H0. subst. subst...
Qed.

Theorem type_checking_complete : ∀ Gamma t T,
  has_type Gamma t T → type_check Gamma t = Some T.
Proof with auto.
  intros Gamma t T Hty.
  induction Hty; simpl.
  - eauto.
  - rewrite IHHty...
  -
    rewrite IHHty1. rewrite IHHty2.
    rewrite (beq_ty_refl T11)...
  - eauto.
  - eauto.
  - rewrite IHHty1. rewrite IHHty2.
    rewrite IHHty3. rewrite (beq_ty_refl T)...
Qed.

End STLCChecker.
```

29.5 Exercises

Exercise: 5 stars (typechecker_extensions) 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.
Require Import MoreStlc.
```

```
Import STLCExtended.

Fixpoint beq_ty (T1 T2: ty) : bool :=
  match T1,T2 with
  | TNat, TNat ⇒
      true
  | TUnit, TUnit ⇒
      true
  | TArrow T11 T12, TArrow T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
  | TProd T11 T12, TProd T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
  | TSum T11 T12, TSum T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
  | TList T11, TList T21 ⇒
      beq_ty T11 T21
  | _,_ ⇒
      false
  end.

Lemma beq_ty_refl : ∀ T1,
  beq_ty T1 T1 = true.
Proof.
  intros T1.
  induction T1; simpl;
    try reflexivity;
    try (rewrite IHT1_1; rewrite IHT1_2; reflexivity);
    try (rewrite IHT1; reflexivity). Qed.

Lemma beq_ty__eq : ∀ T1 T2,
  beq_ty T1 T2 = true → 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.

Fixpoint type_check (Gamma:context) (t:tm) : option ty :=
  match t with
  | tvar x ⇒
      Gamma x
  | tabs x T11 t12 ⇒
```

```
    match type_check (update Gamma x T11) t12 with
    | Some T12 ⇒ Some (TArrow T11 T12)
    | _ ⇒ None
    end
| tapp t1 t2 ⇒
    match type_check Gamma t1, type_check Gamma t2 with
    | Some (TArrow T11 T12),Some T2 ⇒
        if beq_ty T11 T2 then Some T12 else None
    | _,_ ⇒ None
    end
| tnat _ ⇒
    Some TNat
| tsucc t1 ⇒
    match type_check Gamma t1 with
    | Some TNat ⇒ Some TNat
    | _ ⇒ None
    end
| tpred t1 ⇒
    match type_check Gamma t1 with
    | Some TNat ⇒ Some TNat
    | _ ⇒ None
    end
| tmult t1 t2 ⇒
    match type_check Gamma t1, type_check Gamma t2 with
    | Some TNat, Some TNat ⇒ Some TNat
    | _,_ ⇒ None
    end
| tif0 guard t f ⇒
    match type_check Gamma guard with
    | Some TNat ⇒
        match type_check Gamma t, type_check Gamma f with
        | Some T1, Some T2 ⇒
            if beq_ty T1 T2 then Some T1 else None
        | _,_ ⇒ None
        end
    | _ ⇒ None
    end

| tlcase t0 t1 x21 x22 t2 ⇒
    match type_check Gamma t0 with
    | Some (TList T) ⇒
        match type_check Gamma t1,
```

```
              type_check (update (update Gamma x22 (TList T)) x21 T) t2 with
        | Some T1', Some T2' ⇒
            if beq_ty T1' T2' then Some T1' else None
        | _,_ ⇒ None
        end
    | _ ⇒ None
    end

  | _ ⇒ None
  end.
Ltac invert_typecheck Gamma t T :=
  remember (type_check Gamma t) as TO;
  destruct TO as [T|];
  try solve_by_invert; try (inversion H0; eauto); try (subst; eauto).
Ltac fully_invert_typecheck Gamma t T T1 T2 :=
  let TX := fresh T in
  remember (type_check Gamma t) as TO;
  destruct TO as [TX|]; try solve_by_invert;
  destruct TX as [T1 T2| | | T1 T2| T1 T2| T1];
  try solve_by_invert; try (inversion H0; eauto); try (subst; eauto).
Ltac case_equality S T :=
  destruct (beq_ty S T) eqn: Heqb;
  inversion H0; apply beq_ty__eq in Heqb; subst; subst; eauto.
Theorem type_checking_sound : ∀ Gamma t T,
  type_check Gamma t = Some T → has_type Gamma t T.
Proof with eauto.
  intros Gamma t. generalize dependent Gamma.
  induction t; intros Gamma T Htc; inversion Htc.
  - eauto.
  -
    fully_invert_typecheck Gamma t1 T1 T11 T12.
    invert_typecheck Gamma t2 T2.
    case_equality T11 T2.
  -
    rename i into x. rename t into T1.
    remember (update Gamma x T1) 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.
  -
    fully_invert_typecheck Gamma t1 T1 T11 T12.
    fully_invert_typecheck Gamma t2 T2 T21 T12.
  -
    fully_invert_typecheck Gamma t1 T1 T11 T12.
    invert_typecheck Gamma t2 T2.
    invert_typecheck Gamma t3 T3.
    case_equality T2 T3.
  -
    rename i into x31. rename i0 into x32.
    fully_invert_typecheck Gamma t1 T1 T11 T12.
    invert_typecheck Gamma t2 T2.
    remember (update (update Gamma x32 (TList T11)) x31 T11) as Gamma'2.
    invert_typecheck Gamma'2 t3 T3.
    case_equality T2 T3.
Qed.

Theorem type_checking_complete : ∀ Gamma t T,
  has_type Gamma t T → type_check Gamma t = Some T.
Proof.
  intros Gamma t T Hty.
  induction Hty; simpl;
    try (rewrite IHHty);
    try (rewrite IHHty1);
    try (rewrite IHHty2);
    try (rewrite IHHty3);
    try (rewrite (beq_ty_refl T));
    try (rewrite (beq_ty_refl T1));
    try (rewrite (beq_ty_refl T2));
    eauto.
  Admitted. End TypecheckerExtensions.
```

☐

Exercise: 5 stars, optional (stlc_step_function) 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 TypecheckerExtensions.

End StepFunction.
```

□

Exercise: 5 stars, optional (stlc_impl) Using the Imp parser described in the ImpParser chapter as a guide, build a parser for extended Stlc programs. Combine it with the type-checking and stepping functions from above to yield a complete typechecker and interpreter for this language.

Module *StlcImpl*.
Import *StepFunction*.

End *StlcImpl*.

□

$Date : 2016-12-0122:35:27-0500(Thu, 01Dec2016)$

Chapter 30

Library Top.Records

30.1 Records: Adding Records to STLC

```
Require Import Maps.
Require Import Imp.
Require Import Smallstep.
Require Import Stlc.
```

30.2 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, it would be carried out in the parser, which we are eliding here), and anyway it is not very efficient. 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 ::= Terms: | {i1=t1, ..., in=tn} record | t.i projection | ...

v ::= Values: | {i1=v1, ..., in=vn} record value | ...

T ::= Types: | {i1:T1, ..., in:Tn} record type | ...

Reduction:

ti ==> ti' (ST_Rcd)

{i1=v1, ..., im=vm, in=tn, ...} ==> {i1=v1, ..., im=vm, in=tn', ...}

t1 ==> t1'

(ST_Proj1) t1.i ==> t1'.i

(ST_ProjRcd) {..., i=vi, ...}.i ==> vi

Typing:

Gamma |- t1 : T1 ... Gamma |- tn : Tn

(T_Rcd) Gamma |- {i1=t1, ..., in=tn} : {i1:T1, ..., in:Tn}
Gamma |- t : {..., i:Ti, ...}

(T_Proj) Gamma |- t.i : Ti

30.3 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 (id × X).

Inductive ty : Type :=
  | TBase : id → ty
  | TArrow : ty → ty → ty
  | TRcd : (alist ty) → ty.
```

Unfortunately, we encounter here a limitation in Coq: 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 Coq, 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 Coq 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 Coq **list** type, we can essentially incorporate its constructors ("nil" and "cons") in the syntax of our types.

```
Inductive ty : Type :=
  | TBase : id → ty
  | TArrow : ty → ty → ty
  | TRNil : ty
  | TRCons : id → ty → ty → ty.
```

Similarly, at the level of terms, we have constructors trnil, for the empty record, and trcons, which adds a single field to the front of a list of fields.

```
Inductive tm : Type :=
```

```
  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm

  | tproj : tm → id → tm
  | trnil : tm
  | trcons : id → tm → tm → tm.
```

Some examples...

```
Notation a := (Id "a").
Notation f := (Id "f").
Notation g := (Id "g").
Notation l := (Id "l").
Notation A := (TBase (Id "A")).
Notation B := (TBase (Id "B")).
Notation k := (Id "k").
Notation i1 := (Id "i1").
Notation i2 := (Id "i2").
```

{ i1:A }

{ i1:A→B, i2:A }

Well-Formedness

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 := TRCons X 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 TRNil and TRCons at the outermost level.

```
Inductive record_ty : ty → Prop :=
  | RTnil :
        record_ty TRNil
  | RTcons : ∀ i T1 T2,
        record_ty (TRCons i T1 T2).
```

With this, we can define well-formed types.

```
Inductive well_formed_ty : ty → Prop :=
  | wfTBase : ∀ i,
        well_formed_ty (TBase i)
```

```
  | wfTArrow : ∀ T1 T2,
        well_formed_ty T1 →
        well_formed_ty T2 →
        well_formed_ty (TArrow T1 T2)
  | wfTRNil :
        well_formed_ty TRNil
  | wfTRCons : ∀ i T1 T2,
        well_formed_ty T1 →
        well_formed_ty T2 →
        record_ty T2 →
        well_formed_ty (TRCons i T1 T2).
Hint Constructors record_ty well_formed_ty.
```

Note that **record_ty** and **record_tm** are not recursive – they just check 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 TRCons) is a record.

Of course, we should also be concerned about ill-formed terms, not just types; but type-checking can rules 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 trcons.

```
Inductive record_tm : tm → Prop :=
  | rtnil :
        record_tm trnil
  | rtcons : ∀ i t1 t2,
        record_tm (trcons i t1 t2).
Hint Constructors record_tm.
```

Substitution

Substitution extends easily.

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒ if beq_id x y then s else t
  | tabs y T t1 ⇒ tabs y T
                      (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)
  | tproj t1 i ⇒ tproj (subst x s t1) i
  | trnil ⇒ trnil
  | trcons i t1 tr1 ⇒ trcons i (subst x s t1) (subst x s tr1)
  end.
Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Reduction

A record is a value if all of its fields are.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T11 t12,
      value (tabs x T11 t12)
  | v_rnil : value trnil
  | v_rcons : ∀ i v1 vr,
      value v1 →
      value vr →
      value (trcons i v1 vr).
Hint Constructors value.
```

To define reduction, we'll need a utility function for extracting one field from record term:

```
Fixpoint tlookup (i:id) (tr:tm) : option tm :=
  match tr with
  | trcons i' t tr' ⇒ if beq_id 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 "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T11 t12 v2,
         value v2 →
         (tapp (tabs x T11 t12) v2) ==> ([x:=v2]t12)
  | ST_App1 : ∀ t1 t1' t2,
         t1 ==> t1' →
         (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
         value v1 →
         t2 ==> t2' →
         (tapp v1 t2) ==> (tapp v1 t2')
  | ST_Proj1 : ∀ t1 t1' i,
        t1 ==> t1' →
        (tproj t1 i) ==> (tproj t1' i)
  | ST_ProjRcd : ∀ tr i vi,
        value tr →
        tlookup i tr = Some vi →
        (tproj tr i) ==> vi
  | ST_Rcd_Head : ∀ i t1 t1' tr2,
        t1 ==> t1' →
```

```
        (trcons i t1 tr2) ==> (trcons i t1' tr2)
  | ST_Rcd_Tail : ∀ i v1 tr2 tr2',
        value v1 →
        tr2 ==> tr2' →
        (trcons i v1 tr2) ==> (trcons i v1 tr2')

where "t1 '==>' t2" := (step t1 t2).

Notation multistep := (multi step).
Notation "t1 '==>*' t2" := (multistep t1 t2) (at level 40).

Hint Constructors step.
```

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:id) (Tr:ty) : option ty :=
  match Tr with
  | TRCons i' T Tr' ⇒
      if beq_id i i' then Some T else Tlookup i Tr'
  | _ ⇒ None
  end.

Definition context := partial_map ty.

Reserved Notation "Gamma '|-' t '\in' T" (at level 40).

Inductive has_type : context → tm → ty → Prop :=
  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      well_formed_ty T →
      Gamma ⊢ (tvar x) \in T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      well_formed_ty T11 →
```

```
      (update Gamma x T11) ⊢ t12 \in T12 →
      Gamma ⊢ (tabs x T11 t12) \in (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma t1 t2,
      Gamma ⊢ t1 \in (TArrow T1 T2) →
      Gamma ⊢ t2 \in T1 →
      Gamma ⊢ (tapp t1 t2) \in T2

  | T_Proj : ∀ Gamma i t Ti Tr,
      Gamma ⊢ t \in Tr →
      Tlookup i Tr = Some Ti →
      Gamma ⊢ (tproj t i) \in Ti
  | T_RNil : ∀ Gamma,
      Gamma ⊢ trnil \in TRNil
  | T_RCons : ∀ Gamma i t T tr Tr,
      Gamma ⊢ t \in T →
      Gamma ⊢ tr \in Tr →
      record_ty Tr →
      record_tm tr →
      Gamma ⊢ (trcons i t tr) \in (TRCons i T Tr)

where "Gamma '|-' t '\in' T" := (has_type Gamma t T).

Hint Constructors has_type.
```

30.3.1 Examples

Exercise: 2 stars (examples) Finish the proofs below. Feel free to use Coq'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 ⊢
    (tapp (tabs a (TRCons i1 (TArrow A A)
                      (TRCons i2 (TArrow B B)
                       TRNil))
              (tproj (tvar a) i2))
            (trcons i1 (tabs a A (tvar a))
            (trcons i2 (tabs a B (tvar a))
             trnil))) \in
    (TArrow B B).
Proof.
   Admitted.
```

```
Example typing_nonexample :
  ¬ ∃ T,
      (update empty a (TRCons i2 (TArrow A A)
                                    TRNil)) ⊢
                (trcons i1 (tabs a B (tvar a)) (tvar a)) \in
                T.
Proof.
   Admitted.

Example typing_nonexample_2 : ∀ y,
  ¬ ∃ T,
    (update empty y A) ⊢
           (tapp (tabs a (TRCons i1 A TRNil)
                     (tproj (tvar a) i1))
                   (trcons i1 (tvar y) (trcons i2 (tvar y) trnil))) \in
           T.
Proof.
   Admitted.
```

30.3.2 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 (beq_id i i0)...
    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 empty $\vdash v$: T and Tlookup i T returns Some *Ti*, then tlookup i v returns Some *ti* for some term *ti* such that empty $\vdash$ *ti* \in *Ti*.

Proof: By induction on the typing derivation *Htyp*. Since Tlookup i T $=$ 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 $t =$ trcons *i0* t *tr* and T $=$ TRCons *i0* T *Tr* for some *i0*, t, *tr*, T and *Tr*.

This leaves two possiblities to consider - either *i0* $=$ i or not.

- If i $=$ *i0*, then since Tlookup i (TRCons *i0* T *Tr*) $=$ Some *Ti* we have T $=$ *Ti*. It follows that t itself satisfies the theorem.
- On the other hand, suppose i $\neq$ *i0*. Then

  Tlookup i T = Tlookup i Tr

  and

  tlookup i t = tlookup i tr,

  so the result follows from the induction hypothesis. □

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 ty) as Gamma.
induction Htyp; subst; try solve_by_invert...
-
  simpl in Hget. simpl. destruct (beq_id i i0).
  +
    simpl. inversion 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 ty) 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]t12)...
      ×

        destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    +

      destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
```

```
-
  right. destruct IHHt...
  +
    destruct (lookup_field_in_value _ _ _ _ H0 Ht H)
      as [ti [Hlkup _]].
    ∃ ti...
  +
    destruct H0 as [t' Hstp]. ∃ (tproj t' i)...
-
  left...
-
  destruct IHHt1...
  +
    destruct IHHt2; try reflexivity.
    ×
      left...
    ×
      right. destruct H2 as [tr' Hstp].
      ∃ (trcons i t tr')...
  +
    right. destruct H1 as [t' Hstp].
    ∃ (trcons i t' tr)... Qed.
```

Context Invariance

```
Inductive appears_free_in : id → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
        y ≠ x →
        appears_free_in x t12 →
```

```
          appears_free_in x (tabs y T11 t12)
  | afi_proj : ∀ x t i,
      appears_free_in x t →
      appears_free_in x (tproj t i)
  | afi_rhead : ∀ x i ti tr,
       appears_free_in x ti →
       appears_free_in x (trcons i ti tr)
  | afi_rtail : ∀ x i ti tr,
       appears_free_in x tr →
       appears_free_in x (trcons i ti tr).

Hint Constructors appears_free_in.

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 with eauto.
  intros. generalize dependent Gamma'.
  induction H;
    intros Gamma' Heqv...
  -
    apply T_Var... rewrite ← Heqv...
  -
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold update, t_update. destruct (beq_idP x y)...
  -
    apply T_App with T1...
  -
    apply T_RCons... Qed.

Lemma free_in_context : ∀ x t T Gamma,
    appears_free_in x t →
    Gamma ⊢ t \in T →
    ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  induction Htyp; inversion Hafi; subst...
  -
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx...
Qed.
```

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
      (update Gamma x U) ⊢ t \in S →
      empty ⊢ v \in U →
      Gamma ⊢ ([x:=v]t) \in S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  induction t;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  -
    simpl. rename i into y.
    unfold update, t_update in H0.
    destruct (beq_idP x y) as [Hxy|Hxy].
    +

      subst.
      inversion H0; subst. clear H0.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra)
        as [T' HT']...
      inversion HT'.
    +

      apply T_Var...
  -
    rename i into y. rename t into T11.
    apply T_Abs...
    destruct (beq_idP x y) as [Hxy|Hxy].
    +

      eapply context_invariance...
      subst.
      intros x Hafi. unfold update, t_update.
      destruct (beq_id y x)...
    +

      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP y z)...
      subst. rewrite false_beq_id...
```

```
  -
    apply T_RCons... inversion H7; subst; simpl...
Qed.

Theorem preservation : ∀ t t' T,
     empty ⊢ t \in T →
     t ==> t' →
     empty ⊢ t' \in T.
Proof with eauto.
  intros t t' T HT.
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  induction HT;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  -

    inversion HE; subst...
    +

      apply substitution_preserves_typing with T1...
      inversion HT1...
  -

    destruct (lookup_field_in_value _ _ _ _ H2 HT H)
      as [vi [Hget Htyp]].
    rewrite H4 in Hget. inversion Hget. subst...
  -

    apply T_RCons... eapply step_preserves_record_tm...
Qed.
```

□

```
End STLCExtendedRecords.
```

$Date : 2016-11-2916 : 09 : 40-0500(Tue, 29Nov2016)$

Chapter 31

Library Top.References

31.1 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.

However, 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.

```
Require Import Coq.Arith.Arith.
Require Import Coq.omega.Omega.
Require Import Coq.Lists.List.
Import ListNotations.
Require Import Maps.
Require Import Smallstep.
```

31.2 Definitions

Pretty much every programming language provides some form of assignment operation that changes the contents of a previously allocated piece of storage. (Coq'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.

31.3 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 need to 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 ::= Nat | Unit | T -> T | Ref T

```
Inductive ty : Type :=
  | TNat : ty
  | TUnit : ty
  | TArrow : ty → ty → ty
  | TRef : ty → ty.
```

Terms

Besides variables, abstractions, applications, natural-number-related terms, and **unit**, we need four more sorts of terms in order to handle mutable references:

t ::= ... Terms | ref t allocation | !t dereference | t := t assignment | l location

```
Inductive tm : Type :=

  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm
  | tnat : nat → tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tmult : tm → tm → tm
  | tif0 : tm → tm → tm → tm

  | tunit : tm
  | tref : tm → tm
  | tderef : tm → tm
  | tassign : tm → tm → tm
  | tloc : nat → tm.
```

Intuitively:

- *ref t* (formally, tref *t*) allocates a new reference cell with the value *t* and reduces to the location of the newly allocated cell;
- !*t* (formally, tderef *t*) reduces to the contents of the cell referenced by *t*;
- *t1* := *t2* (formally, tassign *t1 t2*) assigns *t2* to the cell referenced by *t1*; and
- l (formally, tloc 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 will look like this:

Gamma |- t1 : T1

(T_Ref) Gamma |- ref t1 : Ref T1

Gamma |- t1 : Ref T11

(T_Deref) Gamma |- !t1 : T11

Gamma |- t1 : Ref T11 Gamma |- t2 : T11

(T_Assign) Gamma |- t1 := t2 : 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 and numbers, we have two new types of values: the unit value, and locations.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_nat : ∀ n,
      value (tnat n)
  | v_unit :
      value tunit
  | v_loc : ∀ l,
      value (tloc l).

Hint Constructors value.
```

Extending substitution to handle the new syntax of terms is straightforward.

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' ⇒
      if beq_id x x' then s else t
  | tapp t1 t2 ⇒
      tapp (subst x s t1) (subst x s t2)
  | tabs x' T t1 ⇒
      if beq_id x x' then t else tabs x' T (subst x s t1)
  | tnat n ⇒
```

```
      t
  | tsucc t1 ⇒
      tsucc (subst x s t1)
  | tpred t1 ⇒
      tpred (subst x s t1)
  | tmult t1 t2 ⇒
      tmult (subst x s t1) (subst x s t2)
  | tif0 t1 t2 t3 ⇒
      tif0 (subst x s t1) (subst x s t2) (subst x s t3)
  | tunit ⇒
      t
  | tref t1 ⇒
      tref (subst x s t1)
  | tderef t1 ⇒
      tderef (subst x s t1)
  | tassign t1 t2 ⇒
      tassign (subst x s t1) (subst x s t2)
  | tloc _ ⇒
      t
  end.
Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

31.4 Pragmatics

31.4.1 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 tseq t1 t2 :=
```

tapp (*tabs* (*Id* "x") *TUnit t2*) *t1*.

31.4.2 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!

31.4.3 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 `let`s are executed simply by binding these functions to the names *incc* and *decc*, rather than by actually incrementing or decrementing `c`. Later, each caddll 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.

31.4.4 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 // Note that 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 (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.

□

31.4.5 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 an (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 simply 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 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, recommended (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?

☐

31.4.6 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 occasionally 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. 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).

31.4.7 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 ML and Java) 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: 1 star (type_safety_violation) Show how this can lead to a violation of type safety.

☐

31.5 Operational Semantics

31.5.1 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 reduci 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 Coq 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.

31.5.2 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 tref *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 will 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 tunit.
```

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
    | O ⇒ 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. omega.
Qed.

Lemma lookup_replace_neq : ∀ l1 l2 t st,
  l1 ≠ l2 →
  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.
```

31.5.3 Reduction

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 ==> t'* to *t / st ==> 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) (\x:T.t12) v2 / st ==> x:=*v2*t12 / st

t1 / st ==> t1' / st'

(ST_App1) t1 t2 / st ==> t1' t2 / st'
value v1 t2 / st ==> t2' / st'

(ST_App2) v1 t2 / st ==> 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 ==> t1' / st'

(ST_Deref) !t1 / st ==> !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) !(loc l) / st ==> lookup l 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 ==> t1' / st'

(ST_Assign1) t1 := t2 / st ==> t1' := t2 / st'
t2 / st ==> t2' / st'

(ST_Assign2) v1 := t2 / st ==> v1 := t2' / st'

Once we have finished with *t1* and *t2*, we have an expression of the form l:=*v2*, which we execute by updating the store to make location l contain *v2*:

l < |st|

(ST_Assign) loc l := v2 / st ==> unit / l:=*v2*st

The notation [l:=*v2*]*st* means "the store that maps l to *v2* 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 reduct an expression of the form *ref t1*, we first reduce *t1* until it becomes a value:

t1 / st ==> t1' / st'

(ST_Ref) ref t1 / st ==> 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 *v1*.

(ST_RefValue) ref v1 / st ==> loc |st| / st,v1

The value resulting from this step is the newly allocated location itself. (Formally, *st*,*v1* means *st* ++ *v1*::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 "t1 '/' st1 '==>' t2 '/' st2"
  (at level 40, st1 at level 39, t2 at level 39).

Import ListNotations.

Inductive step : tm × store → tm × store → Prop :=
  | ST_AppAbs : ∀ x T t12 v2 st,
         value v2 →
         tapp (tabs x T t12) v2 / st ==> [x:=v2]t12 / st
  | ST_App1 : ∀ t1 t1' t2 st st',
         t1 / st ==> t1' / st' →
         tapp t1 t2 / st ==> tapp t1' t2 / st'
  | ST_App2 : ∀ v1 t2 t2' st st',
         value v1 →
         t2 / st ==> t2' / st' →
         tapp v1 t2 / st ==> tapp v1 t2'/ st'
  | ST_SuccNat : ∀ n st,
         tsucc (tnat n) / st ==> tnat (S n) / st
  | ST_Succ : ∀ t1 t1' st st',
         t1 / st ==> t1' / st' →
         tsucc t1 / st ==> tsucc t1' / st'
```

```
| ST_PredNat : ∀ n st,
        tpred (tnat n) / st ==> tnat (pred n) / st
| ST_Pred : ∀ t1 t1' st st',
        t1 / st ==> t1' / st' →
        tpred t1 / st ==> tpred t1' / st'
| ST_MultNats : ∀ n1 n2 st,
        tmult (tnat n1) (tnat n2) / st ==> tnat (mult n1 n2) / st
| ST_Mult1 : ∀ t1 t2 t1' st st',
        t1 / st ==> t1' / st' →
        tmult t1 t2 / st ==> tmult t1' t2 / st'
| ST_Mult2 : ∀ v1 t2 t2' st st',
        value v1 →
        t2 / st ==> t2' / st' →
        tmult v1 t2 / st ==> tmult v1 t2' / st'
| ST_If0 : ∀ t1 t1' t2 t3 st st',
        t1 / st ==> t1' / st' →
        tif0 t1 t2 t3 / st ==> tif0 t1' t2 t3 / st'
| ST_If0_Zero : ∀ t2 t3 st,
        tif0 (tnat 0) t2 t3 / st ==> t2 / st
| ST_If0_Nonzero : ∀ n t2 t3 st,
        tif0 (tnat (S n)) t2 t3 / st ==> t3 / st
| ST_RefValue : ∀ v1 st,
        value v1 →
        tref v1 / st ==> tloc (length st) / (st ++ v1::nil)
| ST_Ref : ∀ t1 t1' st st',
        t1 / st ==> t1' / st' →
        tref t1 / st ==> tref t1' / st'
| ST_DerefLoc : ∀ st l,
        l < length st →
        tderef (tloc l) / st ==> store_lookup l st / st
| ST_Deref : ∀ t1 t1' st st',
        t1 / st ==> t1' / st' →
        tderef t1 / st ==> tderef t1' / st'
| ST_Assign : ∀ v2 l st,
        value v2 →
        l < length st →
        tassign (tloc l) v2 / st ==> tunit / replace l v2 st
| ST_Assign1 : ∀ t1 t1' t2 st st',
        t1 / st ==> t1' / st' →
        tassign t1 t2 / st ==> tassign t1' t2 / st'
| ST_Assign2 : ∀ v1 t2 t2' st st',
        value v1 →
```

t2 / *st* ==> *t2'* / *st'* →
tassign v1 t2 / *st* ==> *tassign v1 t2'* / *st'*

`where` "t1 '/' st1 '==>' t2 '/' st2" := (*step* (*t1*,*st1*) (*t2*,*st2*)).

One slightly ugly point should be noted here: In the ST_RefValue rule, we extend the state by writing *st* ++ *v1*::nil rather than the more natural *st* ++ [*v1*]. 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*.

`Definition` *multistep* := (*multi step*).
`Notation` "t1 '/' st '==>*' t2 '/' st'" :=
(*multistep* (*t1*,*st*) (*t2*,*st'*))
(`at level` 40, *st* `at level` 39, *t2* `at level` 39).

31.6 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*.

31.6.1 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**, \x:*Unit.x*], the result is \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*→*Unit*. This observation leads us immediately to a first attempt at a typing rule for locations:

Gamma |- lookup l st : T1

Gamma |- loc l : 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 *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: $Gamma; st \vdash t : \mathsf{T}$. Our rule for typing references now has the form

Gamma; st |- lookup l st : T1

Gamma; st |- loc l : 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:

\x:Nat. (!(*loc* 1)) x, \x:Nat. (!(*loc* 0)) x

Exercise: 2 stars (cyclic_store) Can you find a term whose reduction will create this particular cyclic store? ☐

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 TUnit.
```

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 [$Unit$, $Unit \rightarrow Unit$], then we can immediately infer that !(loc 1) has type $Unit \rightarrow Unit$. More generally, the typing rule for locations can be reformulated in terms of store typings like this:

l < |ST|

Gamma; ST |- loc l : Ref (lookup 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.

31.6.2 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 < |ST|

(T_Loc) Gamma; ST |- loc l : Ref (lookup l ST)

Gamma; ST |- t1 : T1

(T_Ref) Gamma; ST |- ref t1 : Ref T1

Gamma; ST |- t1 : Ref T11

(T_Deref) Gamma; ST |- !t1 : T11

Gamma; ST |- t1 : Ref T11 Gamma; ST |- t2 : T11

(T_Assign) Gamma; ST |- t1 := t2 : Unit

```
Reserved Notation "Gamma ';' ST '|-' t '\in' T" (at level 40).

Inductive has_type : context → store_ty → tm → ty → Prop :=
  | T_Var : ∀ Gamma ST x T,
      Gamma x = Some T →
      Gamma; ST ⊢ (tvar x) \in T
  | T_Abs : ∀ Gamma ST x T11 T12 t12,
      (update Gamma x T11); ST ⊢ t12 \in T12 →
      Gamma; ST ⊢ (tabs x T11 t12) \in (TArrow T11 T12)
```

```
  | T_App : ∀ T1 T2 Gamma ST t1 t2,
      Gamma; ST ⊢ t1 \in (TArrow T1 T2) →
      Gamma; ST ⊢ t2 \in T1 →
      Gamma; ST ⊢ (tapp t1 t2) \in T2
  | T_Nat : ∀ Gamma ST n,
      Gamma; ST ⊢ (tnat n) \in TNat
  | T_Succ : ∀ Gamma ST t1,
      Gamma; ST ⊢ t1 \in TNat →
      Gamma; ST ⊢ (tsucc t1) \in TNat
  | T_Pred : ∀ Gamma ST t1,
      Gamma; ST ⊢ t1 \in TNat →
      Gamma; ST ⊢ (tpred t1) \in TNat
  | T_Mult : ∀ Gamma ST t1 t2,
      Gamma; ST ⊢ t1 \in TNat →
      Gamma; ST ⊢ t2 \in TNat →
      Gamma; ST ⊢ (tmult t1 t2) \in TNat
  | T_If0 : ∀ Gamma ST t1 t2 t3 T,
      Gamma; ST ⊢ t1 \in TNat →
      Gamma; ST ⊢ t2 \in T →
      Gamma; ST ⊢ t3 \in T →
      Gamma; ST ⊢ (tif0 t1 t2 t3) \in T
  | T_Unit : ∀ Gamma ST,
      Gamma; ST ⊢ tunit \in TUnit
  | T_Loc : ∀ Gamma ST l,
      l < length ST →
      Gamma; ST ⊢ (tloc l) \in (TRef (store_Tlookup l ST))
  | T_Ref : ∀ Gamma ST t1 T1,
      Gamma; ST ⊢ t1 \in T1 →
      Gamma; ST ⊢ (tref t1) \in (TRef T1)
  | T_Deref : ∀ Gamma ST t1 T11,
      Gamma; ST ⊢ t1 \in (TRef T11) →
      Gamma; ST ⊢ (tderef t1) \in T11
  | T_Assign : ∀ Gamma ST t1 t2 T11,
      Gamma; ST ⊢ t1 \in (TRef T11) →
      Gamma; ST ⊢ t2 \in T11 →
      Gamma; ST ⊢ (tassign t1 t2) \in TUnit

where "Gamma ';' ST '|-' t '\in' T" := (has_type Gamma ST t T).

Hint Constructors has_type.
```

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 $Gamma \vdash t : \mathsf{T}$, then we can replace the free variables in t with values of the types listed in *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.

31.7 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.

31.7.1 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_wrong1 : ∀ ST T t st t' st',
  empty; ST ⊢ t \in T →
  t / st ==> t' / st' →
  empty; ST ⊢ t' \in T.
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) :=
  length ST = length st ∧
```

($\forall$ *l*, *l* < *length st* $\rightarrow$
    empty; *ST* $\vdash$ (*store_lookup l st*) \in (*store_Tlookup l ST*)).

Informally, we will write $ST \vdash st$ 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: 2 stars (store_not_unique) Can you find a store *st*, and two different store typings *ST1* and *ST2* such that both $ST1 \vdash st$ and $ST2 \vdash st$?

☐

We can now state something closer to the desired preservation property:

```
Theorem preservation_wrong2 : ∀ ST T t st t' st',
  empty; ST ⊢ t \in T →
  t / st ==> t' / st' →
  store_well_typed ST st →
  empty; ST ⊢ t' \in T.
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*.

31.7.2 Extending Store Typings

Evidently, since the store can increase in size during reduction, we need to allow the store typing to grow 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 → store_ty → Prop :=
  | extends_nil : ∀ ST',
      extends ST' nil
  | extends_cons : ∀ x ST' ST,
      extends ST' ST →
      extends (x::ST') (x::ST).

Hint Constructors extends.
```

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 : ∀ l ST ST',
```

```
  l < length ST ->
  extends ST' ST ->
  store_Tlookup l ST' = store_Tlookup l ST.
Proof with auto.
  intros l ST ST' Hlen H.
  generalize dependent ST'. generalize dependent l.
  induction ST as [|a ST2]; intros l Hlen ST' HST'.
  - inversion Hlen.
  - unfold store_Tlookup in *.
    destruct ST'.
    + inversion HST'.
    +
      inversion HST'; subst.
      destruct l as [|l'].
      × auto.
      × simpl. apply IHST2...
        simpl in Hlen; omega.
Qed.
```

Next, if *ST'* extends *ST*, the length of *ST'* is at least that of *ST*.

```
Lemma length_extends : ∀ l ST ST',
  l < length ST ->
  extends ST' ST ->
  l < length ST'.
Proof with eauto.
  intros. generalize dependent l. induction H0; intros l Hlen.
    inversion Hlen.
    simpl in *.
    destruct l; try omega.
      apply lt_n_S. apply IHextends. omega.
Qed.
```

Finally, *ST* ++ T extends *ST*, and **extends** is reflexive.

```
Lemma extends_app : ∀ ST T,
  extends (ST ++ T) ST.
Proof with auto.
  induction ST; intros T...
  simpl...
Qed.
```

```
Lemma extends_refl : ∀ ST,
  extends ST ST.
Proof.
  induction ST; auto.
```

```
Qed.
```

31.7.3 Preservation, Finally

We can now give the final, correct statement of the type preservation property:

```
Definition preservation_theorem := ∀ 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').
```

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*::nil, where *T1* is the type of the value *v1* in the extended store *st* ++ *v1*::nil, but stating this explicitly would complicate the statement of the theorem without actually 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.

31.7.4 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.

```
Inductive appears_free_in : id → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
      y ≠ x →
      appears_free_in x t12 →
      appears_free_in x (tabs y T11 t12)
  | afi_succ : ∀ x t1,
```

```
      appears_free_in x t1 →
      appears_free_in x (tsucc t1)
  | afi_pred : ∀ x t1,
      appears_free_in x t1 →
      appears_free_in x (tpred t1)
  | afi_mult1 : ∀ x t1 t2,
      appears_free_in x t1 →
      appears_free_in x (tmult t1 t2)
  | afi_mult2 : ∀ x t1 t2,
      appears_free_in x t2 →
      appears_free_in x (tmult t1 t2)
  | afi_if0_1 : ∀ x t1 t2 t3,
      appears_free_in x t1 →
      appears_free_in x (tif0 t1 t2 t3)
  | afi_if0_2 : ∀ x t1 t2 t3,
      appears_free_in x t2 →
      appears_free_in x (tif0 t1 t2 t3)
  | afi_if0_3 : ∀ x t1 t2 t3,
      appears_free_in x t3 →
      appears_free_in x (tif0 t1 t2 t3)
  | afi_ref : ∀ x t1,
      appears_free_in x t1 → appears_free_in x (tref t1)
  | afi_deref : ∀ x t1,
      appears_free_in x t1 → appears_free_in x (tderef t1)
  | afi_assign1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tassign t1 t2)
  | afi_assign2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tassign t1 t2).

Hint Constructors appears_free_in.

Lemma free_in_context : ∀ x t T Gamma ST,
   appears_free_in x t →
   Gamma; ST ⊢ t \in T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros. generalize dependent Gamma. generalize dependent T.
  induction H;
        intros; (try solve [ inversion H0; subst; eauto ]).
  -
    inversion H1; subst.
    apply IHappears_free_in in H8.
    rewrite update_neq in H8; assumption.
Qed.
```

```
Lemma context_invariance : ∀ Gamma Gamma' ST t T,
  Gamma; ST ⊢ t \in T →
  (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
  Gamma'; ST ⊢ t \in T.
Proof with eauto.
  intros.
  generalize dependent Gamma'.
  induction H; intros...
  -
    apply T_Var. symmetry. rewrite ← H...
  -
    apply T_Abs. apply IHhas_type; intros.
    unfold update, t_update.
    destruct (beq_idP x x0)...
  -
    eapply T_App.
      apply IHhas_type1...
      apply IHhas_type2...
  -
    eapply T_Mult.
      apply IHhas_type1...
      apply IHhas_type2...
  -
    eapply T_If0.
      apply IHhas_type1...
      apply IHhas_type2...
      apply IHhas_type3...
  -
    eapply T_Assign.
      apply IHhas_type1...
      apply IHhas_type2...
Qed.
Lemma substitution_preserves_typing : ∀ Gamma ST x s S t T,
  empty; ST ⊢ s \in S →
  (update Gamma x S); ST ⊢ t \in T →
  Gamma; ST ⊢ ([x:=s]t) \in T.
Proof with eauto.
  intros Gamma ST x s S t T Hs Ht.
  generalize dependent Gamma. generalize dependent T.
  induction t; intros T Gamma H;
    inversion H; subst; simpl...
  -
```

```
    rename i into y.
    destruct (beq_idP x y).
    +
      subst.
      rewrite update_eq in H3.
      inversion H3; subst.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ _ _ _ Hcontra Hs)
        as [T' HT'].
      inversion HT'.
    +
      apply T_Var.
      rewrite update_neq in H3...
  - subst.
    rename i into y.
    destruct (beq_idP x y).
    +
      subst.
      apply T_Abs. eapply context_invariance...
      intros. rewrite update_shadow. reflexivity.
    +
      apply T_Abs. apply IHt.
      eapply context_invariance...
      intros. unfold update, t_update.
      destruct (beq_idP y x0)...
      subst.
      rewrite false_beq_id...
Qed.
```

31.7.5 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 : ∀ ST st l t,
  l < length st →
  store_well_typed ST st →
  empty; ST ⊢ t \in (store_Tlookup l ST) →
  store_well_typed ST (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 (beq_nat l' l) eqn: Heqll'.
  -
    apply beq_nat_true in Heqll'; subst.
    rewrite lookup_replace_eq...
  -
    apply beq_nat_false in Heqll'.
    rewrite lookup_replace_neq...
    rewrite length_replace in Hl'.
    apply H0...
Qed.
```

31.7.6 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.

(The lemma is called store_weakening because it resembles the "weakening" lemmas found in proof theory, which show that adding a new assumption to some logical theory does not decrease the set of provable theorems.)

```
Lemma store_weakening : ∀ Gamma ST ST' t T,
  extends ST' ST →
  Gamma; ST ⊢ t \in T →
  Gamma; ST' ⊢ t \in T.
Proof with eauto.
  intros. induction H0; eauto.
  -
    erewrite ← extends_lookup...
    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 : ∀ 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 *.
  inversion H as [Hlen Hmatch]; clear H.
  rewrite app_length, plus_comm. simpl.
  rewrite app_length, plus_comm. simpl.
  split...
  -
    intros l Hl.
    unfold store_lookup, store_Tlookup.
    apply le_lt_eq_dec in Hl; inversion Hl as [Hlt | Heq].
    +
      apply lt_S_n in Hlt.
      rewrite !app_nth1...
      × apply store_weakening with ST. apply extends_app.
        apply Hmatch...
      × rewrite Hlen...
    +
      inversion Heq.
      rewrite app_nth2; try omega.
      rewrite ← Hlen.
      rewrite minus_diag. simpl.
      apply store_weakening with ST...
      { apply extends_app. }
        rewrite app_nth2; try omega.
      rewrite minus_diag. simpl. trivial.
Qed.
```

31.7.7 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 : ∀ A (l:list A) x d,
  nth (length l) (l ++ x::nil) d = x.
Proof.
  induction l; intros; [ auto | simpl; rewrite IHl; auto ].
Qed.
```

And here, at last, is the preservation theorem and proof:

```
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 with eauto using store_weakening, extends_refl.
  remember (@empty ty) 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...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    eapply IHHt2 in H5...
    inversion H5 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    +
      eapply IHHt in H0...
      inversion H0 as [ST' [Hext [Hty Hsty]]].
      ∃ ST'...
  -
    +
      eapply IHHt in H0...
      inversion H0 as [ST' [Hext [Hty Hsty]]].
      ∃ ST'...
  -
    eapply IHHt1 in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    eapply IHHt2 in H5...
    inversion H5 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    +
      eapply IHHt1 in H0...
```

```
      inversion H0 as [ST' [Hext [Hty Hsty]]].
      ∃ ST'... split...
  -
    ∃ (ST ++ T1::nil).
    inversion HST; subst.
    split.
      apply extends_app.
    split.
      replace (TRef T1)
        with (TRef (store_Tlookup (length st) (ST ++ T1::nil))).
      apply T_Loc.
      rewrite ← H. rewrite app_length, plus_comm. simpl. omega.
      unfold store_Tlookup. rewrite ← H. rewrite nth_eq_last.
      reflexivity.
      apply store_well_typed_app; assumption.
  -
    eapply IHHt in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    ∃ ST. split; try split...
    inversion HST as [_ Hsty].
    replace T11 with (store_Tlookup l ST).
    apply Hsty...
    inversion Ht; subst...
  -
    eapply IHHt in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    ∃ ST. split; try split...
    eapply assign_pres_store_typing...
    inversion Ht1; subst...
  -
    eapply IHHt1 in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    eapply IHHt2 in H5...
    inversion H5 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
Qed.
```

Exercise: 3 stars (preservation_informal) Write a careful informal proof of the preservation theorem, concentrating on the T_App, T_Deref, T_Assign, and T_Ref cases.
□

31.7.8 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 : ∀ ST t T st,
  empty; ST ⊢ t \in T →
  store_well_typed ST st →
  (value t ∨ ∃ t', ∃ st', t / st ==> t' / st').
Proof with eauto.
  intros ST t T st Ht HST. remember (@empty ty) 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]...
      ×
        inversion Ht2p as [t2' [st' Hstep]].
        ∃ (tapp (tabs x T t) t2'). ∃ st'...
    +
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tapp t1' t2). ∃ st'...
  -
    right. destruct IHHt as [Ht1p | Ht1p]...
    +
      inversion Ht1p; subst; try solve [ inversion Ht ].
      ×
        ∃ (tnat (S n)). ∃ st...
    +
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tsucc t1'). ∃ st'...
  -
    right. destruct IHHt as [Ht1p | Ht1p]...
    +
      inversion Ht1p; subst; try solve [inversion Ht ].
      ×
        ∃ (tnat (pred n)). ∃ st...
    +
```

```
    inversion Ht1p as [t1' [st' Hstep]].
    ∃ (tpred 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].
      ∃ (tnat (mult n n0)). ∃ st...
    ×
      inversion Ht2p as [t2' [st' Hstep]].
      ∃ (tmult (tnat n) t2'). ∃ st'...
  +
    inversion Ht1p as [t1' [st' Hstep]].
    ∃ (tmult t1' t2). ∃ st'...
-
  right. destruct IHHt1 as [Ht1p | Ht1p]...
  +
    inversion Ht1p; subst; try solve [inversion Ht1].
    destruct n.
    × ∃ t2. ∃ st...
    × ∃ t3. ∃ st...
  +
    inversion Ht1p as [t1' [st' Hstep]].
    ∃ (tif0 t1' t2 t3). ∃ st'...
-
  right. destruct IHHt as [Ht1p | Ht1p]...
  +
    inversion Ht1p as [t1' [st' Hstep]].
    ∃ (tref 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...
  +
    inversion Ht1p as [t1' [st' Hstep]].
    ∃ (tderef 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 H5...
      ×
        inversion Ht2p as [t2' [st' Hstep]].
        ∃ (tassign t1 t2'). ∃ st'...
    +
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tassign t1' t2). ∃ st'...
Qed.
```

31.8 References and Nontermination

An important fact about the STLC (proved in chapter Norm) is that it is 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* (\x:*Unit.unit*) creates a reference to a cell of type *Unit* → *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 \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 Coq:

```
Module ExampleVariables.

Definition x := Id "x".
Definition y := Id "y".
Definition r := Id "r".
Definition s := Id "s".

End ExampleVariables.

Module RefsAndNontermination.
Import ExampleVariables.
```

```
Definition loop_fun :=
  tabs x TUnit (tapp (tderef (tvar r)) tunit).

Definition loop :=
  tapp
    (tabs r (TRef (TArrow TUnit TUnit))
      (tseq (tassign (tvar r) loop_fun)
              (tapp (tderef (tvar r)) tunit)))
    (tref (tabs x TUnit tunit)).
```

This term is well typed:

```
Lemma loop_typeable : ∃ T, empty; nil ⊢ 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.
    unfold update, t_update. simpl. reflexivity. auto.
  eapply T_Assign.
    eapply T_Var. unfold update, t_update. simpl. 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 ==>+. This is just like the reflexive step closure of single-step reduction (which we're been writing ==>*), except that it is not reflexive: *t* ==>+ *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 →
                step_closure R y z →
                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 ⊢ ?x ⇒ idtac x end.
Ltac reduce :=
    repeat (print_goal; eapply multi_step ;
            [ (eauto 10; fail) | (instantiate; 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 ==>*
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil.
Proof.
  unfold loop.
  reduce.
Qed.
```

Finally, we show that the latter expression reduces in two steps to itself!

```
Lemma loop_fun_step_self :
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil ==>+
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) 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 (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 (TArrow TNat TNat).
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.

□

31.9 Additional Exercises

Exercise: 5 stars, optional (garabage_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*.

$Date : 2016 - 10 - 1111 : 45 : 39 - 0400(Tue, 11Oct2016)$

Chapter 32

Library Top.RecordSub

32.1 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.

```
Require Import Maps.
Require Import Smallstep.
Require Import MoreStlc.
```

32.2 Core Definitions

Syntax

```
Inductive ty : Type :=

  | TTop : ty
  | TBase : id → ty
  | TArrow : ty → ty → ty

  | TRNil : ty
  | TRCons : id → ty → ty → ty.

Inductive tm : Type :=

  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm
  | tproj : tm → id → tm
```

```
  | trnil : tm
  | trcons : id → tm → tm → tm.
```

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.

An interesting technical question is whether the basic properties of the system – progress **and** preservation *– remain* true if *we* drop *these conditions.* I *believe they* do, **and** I *would encourage motivated readers to* try *to check this* by *dropping the conditions from the definitions of typing* **and** *subtyping* **and** *adjusting the proofs* in *the rest of the chapter accordingly. This is* not a trivial *exercise* (**or** *I'd have* done *it*!), *but it should* not *involve changing the basic structure of the proofs.* If *someone does* do *it, please* let *me know. –BCP* 5/16.

```
Inductive record_ty : ty → Prop :=
  | RTnil :
        record_ty TRNil
  | RTcons : ∀ i T1 T2,
        record_ty (TRCons i T1 T2).

Inductive record_tm : tm → Prop :=
  | rtnil :
        record_tm trnil
  | rtcons : ∀ i t1 t2,
        record_tm (trcons i t1 t2).

Inductive well_formed_ty : ty → Prop :=
  | wfTTop :
        well_formed_ty TTop
  | wfTBase : ∀ i,
        well_formed_ty (TBase i)
  | wfTArrow : ∀ T1 T2,
        well_formed_ty T1 →
        well_formed_ty T2 →
        well_formed_ty (TArrow T1 T2)
  | wfTRNil :
        well_formed_ty TRNil
  | wfTRCons : ∀ i T1 T2,
        well_formed_ty T1 →
        well_formed_ty T2 →
        record_ty T2 →
        well_formed_ty (TRCons i T1 T2).
```

```
Hint Constructors record_ty record_tm well_formed_ty.
```

Substitution

Substitution and reduction are as before.

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒ if beq_id x y then s else t
  | tabs y T t1 ⇒ tabs y T (if beq_id x y then t1
                             else (subst x s t1))
  | tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)
  | tproj t1 i ⇒ tproj (subst x s t1) i
  | trnil ⇒ trnil
  | trcons i t1 tr2 ⇒ trcons i (subst x s t1) (subst x s tr2)
  end.
Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Reduction

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_rnil : value trnil
  | v_rcons : ∀ i v vr,
      value v →
      value vr →
      value (trcons i v vr).
Hint Constructors value.
Fixpoint Tlookup (i:id) (Tr:ty) : option ty :=
  match Tr with
  | TRCons i' T Tr' ⇒
      if beq_id i i' then Some T else Tlookup i Tr'
  | _ ⇒ None
  end.
Fixpoint tlookup (i:id) (tr:tm) : option tm :=
  match tr with
  | trcons i' t tr' ⇒
      if beq_id i i' then Some t else tlookup i tr'
  | _ ⇒ None
  end.
Reserved Notation "t1 '==>' t2" (at level 40).
```

```
Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T t12 v2,
          value v2 →
          (tapp (tabs x T t12) v2) ==> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
          t1 ==> t1' →
          (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
          value v1 →
          t2 ==> t2' →
          (tapp v1 t2) ==> (tapp v1 t2')
  | ST_Proj1 : ∀ tr tr' i,
        tr ==> tr' →
        (tproj tr i) ==> (tproj tr' i)
  | ST_ProjRcd : ∀ tr i vi,
        value tr →
        tlookup i tr = Some vi →
       (tproj tr i) ==> vi
  | ST_Rcd_Head : ∀ i t1 t1' tr2,
        t1 ==> t1' →
        (trcons i t1 tr2) ==> (trcons i t1' tr2)
  | ST_Rcd_Tail : ∀ i v1 tr2 tr2',
        value v1 →
        tr2 ==> tr2' →
        (trcons i v1 tr2) ==> (trcons i v1 tr2')

where "t1 '==>' t2" := (step t1 t2).

Hint Constructors step.
```

32.3 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.

32.3.1 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 : ∀ T,
    well_formed_ty T →
    T <: T
  | S_Trans : ∀ S U T,
    S <: U →
    U <: T →
    S <: T
  | S_Top : ∀ S,
    well_formed_ty S →
    S <: TTop
  | S_Arrow : ∀ S1 S2 T1 T2,
    T1 <: S1 →
    S2 <: T2 →
    TArrow S1 S2 <: TArrow T1 T2

  | S_RcdWidth : ∀ i T1 T2,
    well_formed_ty (TRCons i T1 T2) →
    TRCons i T1 T2 <: TRNil
  | S_RcdDepth : ∀ i S1 T1 Sr2 Tr2,
    S1 <: T1 →
    Sr2 <: Tr2 →
    record_ty Sr2 →
    record_ty Tr2 →
    TRCons i S1 Sr2 <: TRCons i T1 Tr2
  | S_RcdPerm : ∀ i1 i2 T1 T2 Tr3,
    well_formed_ty (TRCons i1 T1 (TRCons i2 T2 Tr3)) →
    i1 ≠ i2 →
        TRCons i1 T1 (TRCons i2 T2 Tr3)
    <: TRCons i2 T2 (TRCons i1 T1 Tr3)

where "T '<:' U" := (subtype T U).

Hint Constructors subtype.
```

32.3.2 Examples

```
Module Examples.

Notation x := (Id "x").
Notation y := (Id "y").
Notation z := (Id "z").
Notation j := (Id "j").
```

```
Notation k := (Id "k").
Notation i := (Id "i").
Notation A := (TBase (Id "A")).
Notation B := (TBase (Id "B")).
Notation C := (TBase (Id "C")).

Definition TRcd_j :=
  (TRCons j (TArrow B B) TRNil). Definition TRcd_kj :=
  TRCons k (TArrow A A) TRcd_j.

Example subtyping_example_0 :
  subtype (TArrow C TRcd_kj)
          (TArrow C TRNil).
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 Coq. To get full benefit, make sure you also understand how to prove them on paper!

Exercise: 2 stars (subtyping_example_1)

```
Example subtyping_example_1 :
  subtype TRcd_kj TRcd_j.
Proof with eauto.
   Admitted.
```

☐

Exercise: 1 star (subtyping_example_2)

```
Example subtyping_example_2 :
  subtype (TArrow TTop TRcd_kj)
          (TArrow (TArrow C C) TRcd_j).
Proof with eauto.
   Admitted.
```

☐

Exercise: 1 star (subtyping_example_3)

```
Example subtyping_example_3 :
  subtype (TArrow TRNil (TRCons j A TRNil))
          (TArrow (TRCons k B TRNil) TRNil).
Proof with eauto.
   Admitted.
```

☐

Exercise: 2 stars (subtyping_example_4)

```
Example subtyping_example_4 :
  subtype (TRCons x A (TRCons y B (TRCons z C TRNil)))
```

```
        (TRCons z C (TRCons y B (TRCons x A TRNil))).
Proof with eauto.
  Admitted.
  □
End Examples.
```

32.3.3 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 IHHsub1; destruct IHHsub2)...
  -
    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 (beq_id i i0)... 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 (beq_id i k)...
    +
      inversion Hget. subst. ∃ S1...
  -
    ∃ Ti. split.
    +
      unfold Tlookup. unfold Tlookup in Hget.
      destruct (beq_idP i i1)...
      ×
        destruct (beq_idP i i2)...
        destruct H0.
        subst...
    +
      inversion H. subst. inversion H5. subst... Qed.
```

Exercise: 3 stars (rcd_types_match_informal) Write a careful informal proof of the rcd_types_match lemma.

□

Inversion Lemmas

Exercise: 3 stars, optional (sub_inversion_arrow) Lemma *sub_inversion_arrow* : ∀ *U V1 V2*,

```
      subtype U (TArrow V1 V2) →
      ∃ U1, ∃ U2,
        (U=(TArrow U1 U2)) ∧ (subtype V1 U1) ∧ (subtype U2 V2).
Proof with eauto.
```

```
intros U V1 V2 Hs.
remember (TArrow V1 V2) as V.
generalize dependent V2. generalize dependent V1.
 Admitted.
```
□

32.4 Typing

```
Definition context := partial_map ty.

Reserved Notation "Gamma '|-' t '\in' T" (at level 40).

Inductive has_type : context → tm → ty → Prop :=
  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      well_formed_ty T →
      Gamma ⊢ tvar x \in T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      well_formed_ty T11 →
      update Gamma x T11 ⊢ t12 \in T12 →
      Gamma ⊢ tabs x T11 t12 \in TArrow T11 T12
  | T_App : ∀ T1 T2 Gamma t1 t2,
      Gamma ⊢ t1 \in TArrow T1 T2 →
      Gamma ⊢ t2 \in T1 →
      Gamma ⊢ tapp t1 t2 \in T2
  | T_Proj : ∀ Gamma i t T Ti,
      Gamma ⊢ t \in T →
      Tlookup i T = Some Ti →
      Gamma ⊢ tproj t i \in Ti

  | T_Sub : ∀ Gamma t S T,
      Gamma ⊢ t \in S →
      subtype S T →
      Gamma ⊢ t \in T

  | T_RNil : ∀ Gamma,
      Gamma ⊢ trnil \in TRNil
  | T_RCons : ∀ Gamma i t T tr Tr,
      Gamma ⊢ t \in T →
      Gamma ⊢ tr \in Tr →
      record_ty Tr →
      record_tm tr →
      Gamma ⊢ trcons i t tr \in TRCons i T Tr
```

where "Gamma '|-' t '\in' T" := (*has_type Gamma t T*).

Hint Constructors *has_type*.

32.4.1 Typing Examples

Module *Examples2*.
Import *Examples*.

Exercise: 1 star (typing_example_0) Definition *trcd_kj* :=
(*trcons k* (*tabs z A* (*tvar z*))
(*trcons j* (*tabs z B* (*tvar z*))
trnil)).

Example *typing_example_0* :
has_type empty
(*trcons k* (*tabs z A* (*tvar z*))
(*trcons j* (*tabs z B* (*tvar z*))
trnil))
TRcd_kj.

Proof.
Admitted.
□

Exercise: 2 stars (typing_example_1) Example *typing_example_1* :
has_type empty
(*tapp* (*tabs x TRcd_j* (*tproj* (*tvar x*) *j*)))
(*trcd_kj*))
(*TArrow B B*).

Proof with eauto.
Admitted.
□

Exercise: 2 stars, optional (typing_example_2) Example *typing_example_2* :
has_type empty
(*tapp* (*tabs z* (*TArrow* (*TArrow C C*) *TRcd_j*)
(*tproj* (*tapp* (*tvar z*)
(*tabs x C* (*tvar x*)))
j))
(*tabs z* (*TArrow C C*) *trcd_kj*))
(*TArrow B B*).

Proof with eauto.

Admitted.
□

End *Examples2*.

32.4.2 Properties of Typing

Well-Formedness

```
Lemma has_type__wf : ∀ Gamma t T,
  has_type Gamma t T → well_formed_ty T.
Proof with eauto.
  intros Gamma t T Htyp.
  induction Htyp...
  -
    inversion IHHtyp1...
  -
    eapply wf_rcd_lookup...
  -
    apply subtype__wf in H.
    destruct H...
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; eauto.
Qed.
```

Field Lookup

```
Lemma lookup_field_in_value : ∀ v T i Ti,
  value v →
  has_type empty v T →
  Tlookup i T = Some Ti →
  ∃ vi, tlookup i v = Some vi ∧ has_type empty vi Ti.
Proof with eauto.
  remember empty as Gamma.
  intros t T i Ti Hval Htyp. revert Ti HeqGamma Hval.
  induction Htyp; intros; subst; try solve_by_invert.
  -
```

```
    apply (rcd_types_match S) in H0...
    destruct H0 as [Si [HgetSi Hsub]].
    destruct (IHHtyp Si) as [vi [Hget Htyvi]]...
  -
    simpl in H0. simpl. simpl in H1.
    destruct (beq_id i i0).
    +
      inversion H1. subst. ∃ t...
    +
      destruct (IHHtyp2 Ti) as [vi [get Htyvi]]...
      inversion Hval... Qed.
```

Progress

Exercise: 3 stars (canonical_forms_of_arrow_types)

```
Lemma canonical_forms_of_arrow_types
: ∀ Gamma s T1 T2,
     has_type Gamma s (TArrow T1 T2) →
     value s →
     ∃ x, ∃ S1, ∃ s2,
        s = tabs x S1 s2.
Proof with eauto.
   Admitted.
```

□

```
Theorem progress : ∀ t T,
     has_type empty t 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*]. ∃ (*tapp t1 t2'*)...

+

`destruct` *H* `as` [*t1' Hstp*]. ∃ (*tapp t1' t2*)...

-

`right`. `destruct` *IHHt*...

+

`destruct` (*lookup_field_in_value t T i Ti*)
`as` [*t'* [*Hget Ht'*]]...

+

`destruct` *H0* `as` [*t' Hstp*]. ∃ (*tproj t' i*)...

-

`destruct` *IHHt1*...

+

`destruct` *IHHt2*...

×

`right`. `destruct` *H2* `as` [*tr' Hstp*].
∃ (*trcons i t tr'*)...

+

`right`. `destruct` *H1* `as` [*t' Hstp*].
∃ (*trcons i t' tr*)... `Qed`.

Theorem : For any term *t* and type T, if empty ⊢ *t* : T then *t* is a value or *t* ==> *t'* for some term *t'*.

Proof: Let *t* and T be given such that empty ⊢ *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*, empty ⊢ *t1* : *T1* → *T2* and empty ⊢ *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* ==> *t1'* for some term *t1'*. Then *t1 t2* ==> *t1' t2* by ST_App1.
  - Otherwise *t1* is a value.
    - Suppose *t2* ==> *t2'* for some term *t2'*. Then *t1 t2* ==> *t1 t2'* by rule ST_App2 because *t1* is a value.
    - Otherwise, *t2* is a value. By Lemma *canonical_forms_for_arrow_types*, *t1* = \x:*S1.s2* for some x, *S1*, and *s2*. But then (\x:*S1.s2*) *t2* ==> [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 $t = tr.i$, empty ⊢ *tr* : *Tr*, and Tlookup i *Tr* = Some T.

  By the IH, either *tr* is a value or it steps. If *tr* ==> *tr'* for some term *tr'*, then *tr.i* ==> *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* = Some *ti*. It follows that *tr.i* ==> *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 empty ⊢ *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_tm** *tr*, **record_tm** *Tr*, empty ⊢ *t1* : *T1* and empty ⊢ *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* ==> *t1'* for some term *t1'*. Then {i=*t1*, *tr*} ==> {i=*t1'*, *tr*} by rule ST_Rcd_Head.
  - Otherwise *t1* is a value.
    - Suppose *tr* ==> *tr'* for some term *tr'*. Then {i=*t1*, *tr*} ==> {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 Lemmas

```
Lemma typing_inversion_var : ∀ Gamma x T,
  has_type Gamma (tvar x) T →
  ∃ S,
    Gamma x = Some S ∧ subtype S T.
Proof with eauto.
  intros Gamma x T Hty.
  remember (tvar x) as t.
  induction Hty; intros;
    inversion Heqt; subst; try solve_by_invert.
  -
    ∃ T...
  -
    destruct IHHty as [U [Hctx HsubU]]... Qed.
Lemma typing_inversion_app : ∀ Gamma t1 t2 T2,
```

```
  has_type Gamma (tapp t1 t2) T2 →
  ∃ T1,
    has_type Gamma t1 (TArrow T1 T2) ∧
    has_type Gamma t2 T1.
Proof with eauto.
  intros Gamma t1 t2 T2 Hty.
  remember (tapp t1 t2) as t.
  induction Hty; intros;
    inversion Heqt; subst; try solve_by_invert.
  -
    ∃ T1...
  -
    destruct IHHty as [U1 [Hty1 Hty2]]...
    assert (Hwf := has_type__wf _ _ _ Hty2).
    ∃ U1... Qed.

Lemma typing_inversion_abs : ∀ Gamma x S1 t2 T,
     has_type Gamma (tabs x S1 t2) T →
     (∃ S2, subtype (TArrow S1 S2) T
               ∧ has_type (update Gamma x S1) t2 S2).
Proof with eauto.
  intros Gamma x S1 t2 T H.
  remember (tabs x S1 t2) as t.
  induction H;
    inversion Heqt; subst; intros; try solve_by_invert.
  -
    assert (Hwf := has_type__wf _ _ _ H0).
    ∃ T12...
  -
    destruct IHhas_type as [S2 [Hsub Hty]]...
    Qed.

Lemma typing_inversion_proj : ∀ Gamma i t1 Ti,
  has_type Gamma (tproj t1 i) Ti →
  ∃ T, ∃ Si,
    Tlookup i T = Some Si ∧ subtype Si Ti ∧ has_type Gamma t1 T.
Proof with eauto.
  intros Gamma i t1 Ti H.
  remember (tproj t1 i) as t.
  induction H;
    inversion Heqt; subst; intros; try solve_by_invert.
  -
    assert (well_formed_ty Ti) as Hwf.
    {
```

```
      apply (wf_rcd_lookup i T Ti)...
      apply has_type__wf in H... }
    ∃ T. ∃ Ti...
  -
    destruct IHhas_type as [U [Ui [Hget [Hsub Hty]]]]...
    ∃ U. ∃ Ui... Qed.

Lemma typing_inversion_rcons : ∀ Gamma i ti tr T,
  has_type Gamma (trcons i ti tr) T →
  ∃ Si, ∃ Sr,
    subtype (TRCons i Si Sr) T ∧ has_type Gamma ti Si ∧
    record_tm tr ∧ has_type Gamma tr Sr.
Proof with eauto.
  intros Gamma i ti tr T Hty.
  remember (trcons i ti tr) as t.
  induction Hty;
    inversion Heqt; subst...
  -
    apply IHHty in H0.
    destruct H0 as [Ri [Rr [HsubRS [HtypRi HtypRr]]]].
    ∃ Ri. ∃ Rr...
  -
    assert (well_formed_ty (TRCons i T Tr)) as Hwf.
    {
      apply has_type__wf in Hty1.
      apply has_type__wf in Hty2... }
    ∃ T. ∃ Tr... Qed.

Lemma abs_arrow : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
     subtype T1 S1
  ∧ has_type (update empty x S1) s2 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.
```

Context Invariance

```
Inductive appears_free_in : id → tm → Prop :=
  | afi_var : ∀ x,
```

```
        appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
        appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
        appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
          y ≠ x →
          appears_free_in x t12 →
          appears_free_in x (tabs y T11 t12)
  | afi_proj : ∀ x t i,
        appears_free_in x t →
        appears_free_in x (tproj t i)
  | afi_rhead : ∀ x i t tr,
        appears_free_in x t →
        appears_free_in x (trcons i t tr)
  | afi_rtail : ∀ x i t tr,
        appears_free_in x tr →
        appears_free_in x (trcons i t tr).

Hint Constructors appears_free_in.

Lemma context_invariance : ∀ Gamma Gamma' t S,
      has_type Gamma t S →
      (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
      has_type Gamma' t S.
Proof with eauto.
  intros. generalize dependent Gamma'.
  induction H;
    intros Gamma' Heqv...
  -
    apply T_Var... rewrite ← Heqv...
  -
    apply T_Abs... apply IHhas_type. intros x0 Hafi.
    unfold update, t_update. destruct (beq_idP x x0)...
  -
    apply T_App with T1...
  -
    apply T_RCons... Qed.

Lemma free_in_context : ∀ x t T Gamma,
    appears_free_in x t →
    has_type Gamma t T →
    ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
```

```
  induction Htyp; subst; inversion Hafi; subst...
  -
    destruct (IHHtyp H5) as [T Hctx]. ∃ T.
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx... Qed.
```

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (update Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent S. generalize dependent Gamma.
  induction t; intros; simpl.
  -
    rename i into y.
    destruct (typing_inversion_var _ _ _ Htypt) as [T [Hctx Hsub]].
    unfold update, t_update in Hctx.
    destruct (beq_idP x y)...
    +
      subst.
      inversion Hctx; subst. clear Hctx.
      apply context_invariance with empty...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
      inversion HT'.
    +
      destruct (subtype__wf _ _ Hsub)...
  -
    destruct (typing_inversion_app _ _ _ _ Htypt)
      as [T1 [Htypt1 Htypt2]].
    eapply T_App...
  -
    rename i into y. rename t into T1.
    destruct (typing_inversion_abs _ _ _ _ _ Htypt)
      as [T2 [Hsub Htypt2]].
    destruct (subtype__wf _ _ Hsub) as [Hwf1 Hwf2].
    inversion Hwf2. subst.
    apply T_Sub with (TArrow T1 T2)... apply T_Abs...
    destruct (beq_idP x y).
    +
```

```
      eapply context_invariance...
      subst.
      intros x Hafi. unfold update, t_update.
      destruct (beq_id y x)...
    +
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP y z)...
      subst. rewrite false_beq_id...
  -
    destruct (typing_inversion_proj _ _ _ _ Htypt)
      as [T [Ti [Hget [Hsub Htypt1]]]]...
  -
    eapply context_invariance...
    intros y Hcontra. inversion Hcontra.
  -
    destruct (typing_inversion_rcons _ _ _ _ _ Htypt) as
      [Ti [Tr [Hsub [HtypTi [Hrcdt2 HtypTr]]]]].
    apply T_Sub with (TRCons i Ti Tr)...
    apply T_RCons...
    +
      apply subtype__wf in Hsub. destruct Hsub. inversion H0...
    +
      inversion Hrcdt2; subst; simpl... Qed.
Theorem preservation : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember empty as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  induction HT;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  -
    inversion HE; subst...
    +
      destruct (abs_arrow _ _ _ _ _ HT1) as [HA1 HA2].
      apply substitution_preserves_typing with T...
  -
    destruct (lookup_field_in_value _ _ _ _ H2 HT H)
      as [vi [Hget Hty]].
```

```
rewrite H4 in Hget. inversion Hget. subst...
-
eauto using step_preserves_record_tm. Qed.
```

Theorem: If t, t' are terms and T is a type such that empty $\vdash t$: T and t ==> t', then empty $\vdash t'$: T.

Proof: Let t and T be given such that 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 = $T2$, empty $\vdash t1 : T1 \rightarrow T2$ and 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 $t1$ = \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 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 = $tr.i$, empty $\vdash tr : Tr$, and Tlookup i Tr = Some T.

  The IH for the typing derivation gives us that, for any term tr', if tr ==> tr' then 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 $tr.i$ steps by ST_ProjRcd. Then tr is a value and there is some term vi such that tlookup i tr = Some vi and t' = vi. But by lemma lookup_field_in_value, 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 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_tm** tr, **record_tm** Tr, empty $\vdash t1 : T1$ and 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.

$Date : 2016 - 10 - 1909 : 26 : 05 - 0400(Wed, 19Oct2016)$

Chapter 33

Library Top.Norm

33.1 Norm: Normalization of STLC

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 starsM (norm_fail) Where do we fail if we attempt to prove normalization by a straightforward induction on the size of a well-typed term?

□

Exercise: 5 stars, recommended (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. □

33.2 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

```
Require Import Coq.Lists.List.
Import ListNotations.
Require Import Maps.
Require Import Smallstep.
Hint Constructors multi.

Inductive ty : Type :=
  | TBool : ty
  | TArrow : ty → ty → ty
  | TProd : ty → ty → ty
.

Inductive tm : Type :=

  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm

  | tpair : tm → tm → tm
  | tfst : tm → tm
  | tsnd : tm → tm

  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm.
```

Substitution

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
```

```
  | tvar y ⇒ if beq_id x y then s else t
  | tabs y T t1 ⇒
      tabs y T (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)
  | tpair t1 t2 ⇒ tpair (subst x s t1) (subst x s t2)
  | tfst t1 ⇒ tfst (subst x s t1)
  | tsnd t1 ⇒ tsnd (subst x s t1)
  | ttrue ⇒ ttrue
  | tfalse ⇒ tfalse
  | tif t0 t1 t2 ⇒
      tif (subst x s t0) (subst x s t1) (subst x s t2)
  end.
```

```
Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Reduction

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T11 t12,
      value (tabs x T11 t12)
  | v_pair : ∀ v1 v2,
      value v1 →
      value v2 →
      value (tpair v1 v2)
  | v_true : value ttrue
  | v_false : value tfalse
.
```

```
Hint Constructors value.
```

```
Reserved Notation "t1 '==>' t2" (at level 40).
```

```
Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T11 t12 v2,
         value v2 →
         (tapp (tabs x T11 t12) v2) ==> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
         t1 ==> t1' →
         (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
         value v1 →
         t2 ==> t2' →
         (tapp v1 t2) ==> (tapp v1 t2')

  | ST_Pair1 : ∀ t1 t1' t2,
```

```
        t1 ==> t1' →
        (tpair t1 t2) ==> (tpair t1' t2)
  | ST_Pair2 : ∀ v1 t2 t2',
        value v1 →
        t2 ==> t2' →
        (tpair v1 t2) ==> (tpair v1 t2')
  | ST_Fst : ∀ t1 t1',
        t1 ==> t1' →
        (tfst t1) ==> (tfst t1')
  | ST_FstPair : ∀ v1 v2,
        value v1 →
        value v2 →
        (tfst (tpair v1 v2)) ==> v1
  | ST_Snd : ∀ t1 t1',
        t1 ==> t1' →
        (tsnd t1) ==> (tsnd t1')
  | ST_SndPair : ∀ v1 v2,
        value v1 →
        value v2 →
        (tsnd (tpair v1 v2)) ==> v2

  | ST_IfTrue : ∀ t1 t2,
        (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : ∀ t1 t2,
        (tif tfalse t1 t2) ==> t2
  | ST_If : ∀ t0 t0' t1 t2,
        t0 ==> t0' →
        (tif t0 t1 t2) ==> (tif t0' t1 t2)

where "t1 '==>' t2" := (step t1 t2).

Notation multistep := (multi step).
Notation "t1 '==>*' t2" := (multistep t1 t2) (at level 40).

Hint Constructors step.

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.
```

```
Inductive has_type : context → tm → ty → Prop :=

  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      has_type Gamma (tvar x) T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      has_type (update Gamma x T11) t12 T12 →
      has_type Gamma (tabs x T11 t12) (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma t1 t2,
      has_type Gamma t1 (TArrow T1 T2) →
      has_type Gamma t2 T1 →
      has_type Gamma (tapp t1 t2) T2

  | T_Pair : ∀ Gamma t1 t2 T1 T2,
      has_type Gamma t1 T1 →
      has_type Gamma t2 T2 →
      has_type Gamma (tpair t1 t2) (TProd T1 T2)
  | T_Fst : ∀ Gamma t T1 T2,
      has_type Gamma t (TProd T1 T2) →
      has_type Gamma (tfst t) T1
  | T_Snd : ∀ Gamma t T1 T2,
      has_type Gamma t (TProd T1 T2) →
      has_type Gamma (tsnd t) T2

  | T_True : ∀ Gamma,
      has_type Gamma ttrue TBool
  | T_False : ∀ Gamma,
      has_type Gamma tfalse TBool
  | T_If : ∀ Gamma t0 t1 t2 T,
      has_type Gamma t0 TBool →
      has_type Gamma t1 T →
      has_type Gamma t2 T →
      has_type Gamma (tif t0 t1 t2) T
.

Hint Constructors has_type.

Hint Extern 2 (has_type _ (tapp _ _) _) ⇒ eapply T_App; auto.
Hint Extern 2 (_ = _) ⇒ compute; reflexivity.
```

Context Invariance

```
Inductive appears_free_in : id → tm → Prop :=
  | afi_var : ∀ x,
```

```
        appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
        appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
        appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
          y ≠ x →
          appears_free_in x t12 →
          appears_free_in x (tabs y T11 t12)

  | afi_pair1 : ∀ x t1 t2,
        appears_free_in x t1 →
        appears_free_in x (tpair t1 t2)
  | afi_pair2 : ∀ x t1 t2,
        appears_free_in x t2 →
        appears_free_in x (tpair t1 t2)
  | afi_fst : ∀ x t,
        appears_free_in x t →
        appears_free_in x (tfst t)
  | afi_snd : ∀ x t,
        appears_free_in x t →
        appears_free_in x (tsnd t)

  | afi_if0 : ∀ x t0 t1 t2,
        appears_free_in x t0 →
        appears_free_in x (tif t0 t1 t2)
  | afi_if1 : ∀ x t0 t1 t2,
        appears_free_in x t1 →
        appears_free_in x (tif t0 t1 t2)
  | afi_if2 : ∀ x t0 t1 t2,
        appears_free_in x t2 →
        appears_free_in x (tif t0 t1 t2)
.

Hint Constructors appears_free_in.

Definition closed (t:tm) :=
  ∀ x, ¬ appears_free_in x t.

Lemma context_invariance : ∀ Gamma Gamma' t S,
      has_type Gamma t S →
      (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
      has_type Gamma' t S.
Proof with eauto.
  intros. generalize dependent Gamma'.
```

```
  induction H;
    intros Gamma' Heqv...
  -
    apply T_Var... rewrite ← Heqv...
  -
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold update, t_update. destruct (beq_idP x y)...
  -
    apply T_Pair...
  -
    eapply T_If...
Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  induction Htyp; inversion Hafi; subst...
  -
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx...
Qed.

Corollary typable_empty__closed : ∀ t T,
    has_type empty t T →
    closed t.
Proof.
  intros. unfold closed. intros x H1.
  destruct (free_in_context _ _ _ _ H1 H) as [T' C].
  inversion C. Qed.
```

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (update Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  induction t;
```

```
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  -
    simpl. rename i into y.
    unfold update, t_update in H1.
    destruct (beq_idP x y).
    +

      subst.
      inversion H1; subst. clear H1.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
      inversion HT'.
    +

      apply T_Var...
  -
    rename i into y. rename t into T11.
    apply T_Abs...
    destruct (beq_idP x y).
    +

      eapply context_invariance...
      subst.
      intros x Hafi. unfold update, t_update.
      destruct (beq_id y x)...
    +

      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_idP y z)...
      subst. rewrite false_beq_id...
Qed.

Theorem preservation : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  induction HT;
```

```
    intros t' HeqGamma HE; subst; inversion HE; subst...
  -

    inversion HE; subst...
    +

      apply substitution_preserves_typing with T1...
      inversion HT1...
  -
    inversion HT...
  -
    inversion HT...
Qed.
```

Determinism

```
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...
   - inversion H3.
   - exfalso; apply value__normal in H...
   - inversion E1.
   - f_equal...
   - exfalso; apply value__normal in H1...
   - exfalso; apply value__normal in H3...
   - exfalso; apply value__normal in H...
   - f_equal...
   - f_equal...
   - exfalso; apply value__normal in H1...
   - exfalso; apply value__normal in H...
   - f_equal...
   - f_equal...
   - exfalso.
     inversion E1; subst.
     + apply value__normal in H0...
     + apply value__normal in H1...
   - exfalso.
     inversion H2; subst.
     + apply value__normal in H...
```

```
      + apply value__normal in H0...
    - f_equal...
    - exfalso.
      inversion E1; subst.
      + apply value__normal in H0...
      + apply value__normal in H1...
    - exfalso.
      inversion H2; subst.
      + apply value__normal in H...
      + apply value__normal in H0...
    -
        inversion H3.
    -
        inversion H3.
    - inversion E1.
    - inversion E1.
    - f_equal...
Qed.
```

33.3 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 halts (t:tm) : Prop := ∃ t', t ==>* t' ∧ value t'.
```

A trivial fact:

```
Lemma value_halts : ∀ v, value v → halts v.
Proof.
  intros v H. unfold halts.
  ∃ 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 T, a set *R_T* of closed terms of type T. We will specify these sets using a relation R and write R 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 R for the base language:

- R **bool** t iff t is a closed term of type **bool** and t halts in a value
- R ($T1 \to T2$) t iff t is a closed term of type $T1 \to T2$ 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→A *preserve* property P, all terms of type (A→A)->(A→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 Coq, 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, has_type empty t TBool -> halts t -> R TBool t | R_arrow : forall T1 T2 t, has_type empty t (TArrow T1 T2) -> halts t -> (forall s, R T1 s -> R T2 (tapp t s)) -> R (TArrow T1 T2) t.

Unfortunately, Coq 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 → R TS (tapp 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 Coq's logical soundness. Even though the relation we want in this case might be perfectly innocent, Coq 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) {struct T} : Prop :=
  has_type empty t T ∧ halts t ∧
  (match T with
```

```
| TBool ⇒ True
| TArrow T1 T2 ⇒ (∀ s, R T1 s → R T2 (tapp t s))


| TProd 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 : ∀ {T} {t}, R T t → halts t.
Proof.
  intros. destruct T; unfold R in H; inversion H; inversion H1; assumption.
Qed.

Lemma R_typable_empty : ∀ {T} {t}, R T t → has_type empty t T.
Proof.
  intros. destruct T; unfold R in H; inversion H; inversion H1; 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.

33.3.1 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 determinstic. 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]].
  inversion STM; subst.
   exfalso. apply value__normal in V. unfold normal_form in V. apply V. ∃ t'. auto.
   rewrite (step_deterministic _ _ _ ST H). ∃ t''. 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 : ∀ T t t', (t ==> t') → R T t → 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 : ∀ T t t',
  (t ==>* t') → R T t → 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' : ∀ T t t',
  has_type empty t T → (t ==> t') → R T t' → R T t.
Proof.
   Admitted.

Lemma multistep_preserves_R' : ∀ T t t',
  has_type empty t T → (t ==>* t') → R T t' → 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.
```

33.3.2 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 tabs x *T1 t2* belongs to *R_(T1→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*,..*xn*:*Tn* ⊢ *t* : T and *v1*,...,*vn* are values such that R *T1 v1*, R *T2 v2*, ..., R *Tn vn*, then R T ([*x1*:=*v1*][*x2*:=*v2*]...[*xn*:=*vn*]*t*).

The proof will proceed by induction on the typing derivation *x1*:*T1*,..*xn*:*Tn* ⊢ *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 Coq 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*,...*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 ...,y:**bool**,...,y:nat,... and a corresponding term substitution written as ...[y:=(*tbool* true)]...[y:=(tnat 3)]...*t*. Since environments are extended from left to right, the binding y:nat hides the binding y:**bool**; since substitutions are performed right to left, we do the substitution y:=(tnat 3) first, so that the substitution y:=(*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 (id × tm).

Fixpoint msubst (ss:env) (t:tm) {struct ss} : 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 (id × 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 assigments

```
Fixpoint lookup {X:Set} (k : id) (l : list (id × X)) {struct l}
                  : option X :=
  match l with
    | nil ⇒ None
    | (j,x) :: l' ⇒
      if beq_id j k then Some x else lookup k l'
  end.

Fixpoint drop {X:Set} (n:id) (nxs:list (id × X)) {struct nxs}
               : list (id × X) :=
  match nxs with
    | nil ⇒ nil
    | ((n',x)::nxs') ⇒
        if beq_id n' n then drop n nxs'
        else (n',x)::(drop n nxs')
  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 : ∀ t x,
      ¬ appears_free_in x t →
      ∀ t', [x:=t']t = t.
Proof with eauto.
   Admitted.

Lemma subst_closed: ∀ t,
      closed t →
      ∀ x t', [x:=t']t = t.
Proof.
  intros. apply vacuous_substitution. apply H. Qed.

Lemma subst_not_afi : ∀ t x v,
     closed v → ¬ appears_free_in x ([x:=v]t).
Proof with eauto.   unfold closed, not.
  induction t; intros x v P A; simpl in A.
    -
     destruct (beq_idP x i)...
     inversion A; subst. auto.
    -
     inversion A; subst...
    -
     destruct (beq_idP x i)...
     + inversion A; subst...
     + inversion A; subst...
    -
     inversion A; subst...
    -
     inversion A; subst...
    -
     inversion A; subst...
    -
     inversion A.
    -
     inversion A.
    -
     inversion A; subst...
Qed.

Lemma duplicate_subst : ∀ t' x t v,
```

```
  closed v → [x:=t]([x:=v]t') = [x:=v]t'.
Proof.
  intros. eapply vacuous_substitution. apply subst_not_afi. auto.
Qed.
```

```
Lemma swap_subst : ∀ t x x1 v v1,
    x ≠ x1 →
    closed v → closed v1 →
    [x1:=v1]([x:=v]t) = [x:=v]([x1:=v1]t).
Proof with eauto.
 induction t; intros; simpl.
  -
   destruct (beq_idP x i); destruct (beq_idP x1 i).
   + subst. exfalso...
   + subst. simpl. rewrite ← beq_id_refl. apply subst_closed...
   + subst. simpl. rewrite ← beq_id_refl. rewrite subst_closed...
   + simpl. rewrite false_beq_id... rewrite false_beq_id...
   Admitted.
```

Properties of Multi-Substitutions

```
Lemma msubst_closed: ∀ t, closed t → ∀ 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) {struct env} :=
  match env with
  | nil ⇒ True
  | (x,t)::env' ⇒ closed t ∧ closed_env env'
  end.
```

Next come a series of lemmas charcterizing how `msubst` of closed terms distributes over `subst` and over each term form

```
Lemma subst_msubst: ∀ env x v t, closed v → closed_env env →
    msubst env ([x:=v]t) = [x:=v](msubst (drop x env) t).
Proof.
  induction env0; intros; auto.
  destruct a. simpl.
  inversion H0. fold closed_env in H2.
  destruct (beq_idP i x).
```

```
  - subst. rewrite duplicate_subst; auto.
  - simpl. rewrite swap_subst; eauto.
Qed.

Lemma msubst_var: ∀ ss x, closed_env ss →
    msubst ss (tvar x) =
    match lookup x ss with
    | Some t ⇒ t
    | None ⇒ tvar x
  end.
Proof.
  induction ss; intros.
    reflexivity.
    destruct a.
     simpl. destruct (beq_id i x).
      apply msubst_closed. inversion H; auto.
      apply IHss. inversion H; auto.
Qed.

Lemma msubst_abs: ∀ ss x T t,
  msubst ss (tabs x T t) = tabs x T (msubst (drop x ss) t).
Proof.
  induction ss; intros.
    reflexivity.
    destruct a.
      simpl. destruct (beq_id i x); simpl; auto.
Qed.

Lemma msubst_app : ∀ ss t1 t2, msubst ss (tapp t1 t2) = tapp (msubst ss t1) (msubst ss
t2).
Proof.
 induction ss; intros.
   reflexivity.
   destruct a.
    simpl. rewrite ← IHss. 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 : ∀ (c : tass) (x:id),
    lookup x c = (mupdate empty c) x.
```

```
Proof.
  induction c; intros.
    auto.
    destruct a. unfold lookup, mupdate, update, t_update. destruct (beq_id i x); auto.
Qed.

Lemma mupdate_drop : ∀ (c: tass) Gamma x x',
      mupdate Gamma (drop x c) x'
    = if beq_id x x' then Gamma x' else mupdate Gamma c x'.
Proof.
  induction c; intros.
  - destruct (beq_idP x x'); auto.
  - destruct a. simpl.
    destruct (beq_idP i x).
    + subst. rewrite IHc.
      unfold update, t_update. destruct (beq_idP x x'); auto.
    + simpl. unfold update, t_update. destruct (beq_idP i x'); auto.
      subst. rewrite false_beq_id; congruence.
Qed.
```

Properties of Instantiations

These are strightforward.

```
Lemma instantiation_domains_match: ∀ {c} {e},
    instantiation c e →
    ∀ {x} {T},
      lookup x c = Some T → ∃ t, lookup x e = Some t.
Proof.
  intros c e V. induction V; intros x0 T0 C.
    solve_by_invert.
    simpl in *.
    destruct (beq_id x x0); eauto.
Qed.

Lemma instantiation_env_closed : ∀ c e,
  instantiation c e → closed_env e.
Proof.
  intros c e V; induction V; intros.
    econstructor.
    unfold closed_env. fold closed_env.
    split. eapply typable_empty__closed. eapply R_typable_empty. eauto.
        auto.
Qed.

Lemma instantiation_R : ∀ c e,
```

```
    instantiation c e →
    ∀ x t T,
      lookup x c = Some T →
      lookup x e = Some t → R T t.
Proof.
  intros c e V. induction V; intros x' t' T' G E.
    solve_by_invert.
    unfold lookup in *. destruct (beq_id x x').
      inversion G; inversion E; subst. auto.
      eauto.
Qed.
```

```
Lemma instantiation_drop : ∀ c env,
    instantiation c env →
    ∀ x, instantiation (drop x c) (drop x env).
Proof.
  intros c e V. induction V.
    intros. simpl. constructor.
    intros. unfold drop. destruct (beq_id 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 : ∀ v t t',
  value v → (t ==>* t') → (tapp v t) ==>* (tapp v t').
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 : ∀ c e,
     instantiation c e →
     ∀ Gamma t S, has_type (mupdate Gamma c) t S →
     has_type Gamma (msubst e t) S.
Proof.
```

```
  induction 1; intros.
    simpl in H. simpl. auto.
    simpl in H2. simpl.
    apply IHinstantiation.
    eapply substitution_preserves_typing; eauto.
    apply (R_typable_empty H0).
Qed.
```

And at long last, the main lemma.

```
Lemma msubst_R : ∀ c env t T,
    has_type (mupdate empty c) t T →
    instantiation c env →
    R T (msubst env t).
Proof.
  intros c env0 t T HT V.
  generalize dependent env0.
  remember (mupdate empty c) as Gamma.
  assert (∀ x, Gamma x = lookup x c).
    intros. rewrite HeqGamma. rewrite mupdate_lookup. auto.
  clear HeqGamma.
  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: has_type empty (tabs x T11 (msubst (drop x env0) t12)) (TArrow T11
T12)).
    { 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 (beq_idP x x0).
      + auto.
      + rewrite H.
        clear - c n. induction c.
        simpl. rewrite false_beq_id; auto.
        simpl. destruct a. unfold update, t_update.
        destruct (beq_id i 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) t12).
       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,T11)::c)).
          intros. unfold update, t_update, lookup. destruct (beq_id 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

```
Theorem normalization : ∀ t T, has_type empty t T → 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.
```

$Date : 2016-10-1909:26:05-0400(Wed, 19Oct2016)$

Chapter 34

Library Top.LibTactics

34.1 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 Coq. It intends to overcome a number of limitations of the standard set of tactics, and thereby to help user to write shorter and more robust scripts.

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

The 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, hidding the fact that the underlying implementation is based on binary propositions.

- Convenient support for automation: tactic followed with the symbol "~" or "*" will call automation on the generated subgoals. 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 mecanism. 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 *rapply*,

```
Set Implicit Arguments.
Require Import List.
Remove Hints Bool.trans_eq_bool.
```

34.2 Tools for Programming with Ltac

34.2.1 Identity Continuation

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

34.2.2 Untyped Arguments for Tactics

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

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

34.2.3 Optional Arguments for Tactics

ltac_no_arg is a constant that can be used to simulate optional arguments in tactic definitions. Use *mytactic* ltac_no_arg on the tactic invokation, 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.
```

34.2.4 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*.

34.2.5 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*. If 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.
```

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.
```

34.2.6 List of Arguments for Tactics

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

```
Notation "'»'" :=
  (@nil Boxer)
  (at level 0)
  : ltac_scope.
Notation "'»' v1" :=
  ((boxer v1)::nil)
  (at level 0, v1 at level 0)
  : ltac_scope.
Notation "'»' v1 v2" :=
  ((boxer v1)::(boxer v2)::nil)
  (at level 0, v1 at level 0, v2 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   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.
```

34.2.7 Databases of Lemmas

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

Then, to map *mykey* to *myvalue*, write the hint: `Hint Extern` 1 (*Register mydatabase mykey*) ⇒ *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 ].
```

34.2.8 On-the-Fly Removal of Hypotheses

In a list of arguments » *H1 H2* .. *HN* passed to a tactic such as *lets* or *applys* or *forwards* or *specializes*, the term rm, an identity function, can be placed in front of the name of an hypothesis to be deleted.

```
Definition 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.
```

34.2.9 Numbers as Arguments

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

```
Require BinPos Coq.ZArith.BinInt.

Definition ltac_nat_from_int (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 nat_from_number N :=
  match type of N with
  | nat => constr:(N)
  | BinInt.Z => let N' := constr:(ltac_nat_from_int N) in eval compute in N'
  end.
```

ltac_pattern E `at` K is the same as `pattern` E `at` K except that K is a Coq natural 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 nat_from_number 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
  end.

Tactic Notation "ltac_pattern" constr(E) "at" constr(K) "in" hyp(H) :=
  match nat_from_number 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
  end.
```

34.2.10 Testing Tactics

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

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

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

```
Lemma dup_lemma : ∀ P, P → P → P.
Proof using. auto. Qed.

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

Tactic Notation "dup" constr(N) :=
  dup_tactic N.
Tactic Notation "dup" :=
  dup 2.
```

34.2.11 Check No Evar in Goal

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

Ltac check_noevar_hyp H :=
  let T := type of H in check_noevar T.
Ltac check_noevar_goal :=
  match goal with ⊢ ?G ⇒ check_noevar G end.
```

34.2.12 Helper Function for Introducing Evars

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

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

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

34.2.13 Tagging of Hypotheses

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

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

34.2.14 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 _ ⊢ _ ⇒ 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.
```

34.2.15 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* ... *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
  | ?P _ _ _ _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ ⇒ constr:(P)
  | ?P _ _ ⇒ constr:(P)
```

```
| ?P _ ⇒ constr:(P)
| ?P ⇒ constr:(P)
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,X))
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X ⇒ constr:((X1 X2 X3 X4 X5,X))
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X ⇒ constr:((X1 X2 X3 X4,X))
  | ?X1 ?X2 ?X3 ?X4 ?X ⇒ constr:((X1 X2 X3,X))
  | ?X1 ?X2 ?X3 ?X ⇒ constr:((X1 X2,X))
  | ?X1 ?X2 ?X ⇒ constr:((X1,X))
  | ?X1 ?X ⇒ constr:((X1,X))
  end.
```

34.2.16 Action at Occurence and Action Not at Occurence

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

```
Tactic Notation "ltac_action_at" constr(K) "of" constr(E) "do" tactic(Tac) :=
  let p := fresh 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 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.
```

34.2.17 An Alias for `eq`

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

```
Definition eq' := @eq.

Hint Unfold eq'.

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

34.3 Common Tactics for Simplifying Goals Like `intuition`

```
Ltac jauto_set_hyps :=
  repeat match goal with H: ?T ⊢ _ ⇒
    match T with
    | _ ∧ _ ⇒ destruct H
    | ∃ a, _ ⇒ destruct H
    | _ ⇒ generalize H; clear H
    end
  end.

Ltac jauto_set_goal :=
  repeat match goal with
  | ⊢ ∃ a, _ ⇒ esplit
  | ⊢ _ ∧ _ ⇒ split
  end.

Ltac jauto_set :=
  intros; jauto_set_hyps;
  intros; jauto_set_goal;
  unfold not in *.
```

34.4 Backward and Forward Chaining

34.4.1 Application

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

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

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

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

```
Tactic Notation "rapply_0" constr(t) :=
  refine (@t).
Tactic Notation "rapply_1" constr(t) :=
  refine (@t _).
Tactic Notation "rapply_2" constr(t) :=
  refine (@t _ _).
Tactic Notation "rapply_3" constr(t) :=
  refine (@t _ _ _).
Tactic Notation "rapply_4" constr(t) :=
  refine (@t _ _ _ _).
Tactic Notation "rapply_5" constr(t) :=
  refine (@t _ _ _ _ _).
Tactic Notation "rapply_6" constr(t) :=
  refine (@t _ _ _ _ _ _).
Tactic Notation "rapply_7" constr(t) :=
  refine (@t _ _ _ _ _ _ _).
Tactic Notation "rapply_8" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _).
Tactic Notation "rapply_9" constr(t) :=
```

```
  refine (@t _ _ _ _ _ _ _ _ _).
Tactic Notation "rapply_10" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _ _ _).
```

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

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

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

```
Tactic Notation "applys_to" hyp(H) constr(E) :=
  let H' := fresh 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')
         | lets_base H (E _ _ _ _ _ _ _ _ _ H') ]
  ); clear H'.
```

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

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

constructors calls `constructor` or `econstructor`.

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

34.4.2 Assertions

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

```
Tactic Notation "asserts" simple_intropattern(I) ":" constr(T) :=
  let H := fresh 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 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 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.
```

34.4.3 Instantiation and Forward-Chaining

The instantiation tactics are used to instantiate a lemma *E* (whose type is a product) on some arguments. The type of *E* is made of implications and universal quantifications, e.g. ∀ x, *P* x → ∀ y z, *Q* x y z → 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* (» *E1* .. *EN*) where *tactic* is the name of the tactic (possibly with some arguments) and *Ei* are the arguments. Moreover, some tactics accept the syntax *tactic E1* .. *EN* as short for *tactic* (» *E1* .. *EN*) for values of *N* up to 5.

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

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

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

```
  subst x; cont t''.
Ltac app_arg t P v cont :=
  let H := fresh "TEMP" in
  assert (H : P); [ apply v | cont(t H); try clear H ].
Ltac build_app_alls t final :=
  let rec go t :=
    match type of t with
    | ?P → ?Q ⇒ app_assert t P go
    | ∀ _:?A, _ ⇒ app_evar t A go
    | _ ⇒ final t
    end in
  go t.
Ltac boxerlist_next_type vs :=
  match vs with
  | nil ⇒ constr:(ltac_wild)
  | (boxer ltac_wild)::?vs' ⇒ boxerlist_next_type vs'
  | (boxer ltac_wilds)::_ ⇒ constr:(ltac_wild)
  | (@boxer ?T _)::_ ⇒ constr:(T)
  end.

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

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

newer version : support for typeclasses

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

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

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

Ltac unfold_head_until_product T :=
  eval hnf in T.

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

```
  end.

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

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

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

  build_app args ltac:(fun R ⇒ lets_base I R).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E) :=
  lets_build I E.
Tactic Notation "lets" ":" constr(E) :=
  let H := fresh in lets H: E.
Tactic Notation "lets" ":" constr(E0)
 constr(A1) :=
  lets: (» E0 A1).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) :=
  lets: (» E0 A1 A2).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets: (» E0 A1 A2 A3).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: (» E0 A1 A2 A3 A4).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: (» E0 A1 A2 A3 A4 A5).

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

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

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

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

forwards H: (» *E0 E1* .. *EN*) is short for *forwards H*: (» *E0 E1* .. *EN* ___). The arguments *Ei* 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).

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).

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 "applys" constr(E0) constr(A1) :=
  applys (» E0 A1).
```

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

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

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

Tactic Notation "fapplys" constr(E0) :=
  match type of E0 with
  | list Boxer ⇒ fapplys_build E0
  | _ ⇒ fapplys_build (» E0)
  end.
Tactic Notation "fapplys" constr(E0) constr(A1) :=
  fapplys (» E0 A1).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) :=
  fapplys (» E0 A1 A2).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) constr(A3) :=
  fapplys (» E0 A1 A2 A3).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  fapplys (» E0 A1 A2 A3 A4).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  fapplys (» E0 A1 A2 A3 A4 A5).
```

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

```
Ltac specializes_build H Ei :=
  let H' := fresh "TEMP" in rename H into H';
  let args := list_boxer_of Ei in
```

```
  let args := constr:((boxer H')::args) in
  let args := args_unfold_head_if_not_product args in
  build_app args ltac:(fun R ⇒ lets H: R);
  clear H'.

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

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

specializes_vars H is equivalent to *specializes* H __ .. __ with as many double underscore as the number of dependent arguments visible from the type of H. Note that no unfolding is currently being performed (this behavior might change in the future). The current implementation is restricted to the case where H is an existing hypothesis – TODO: generalize.

```
Ltac specializes_var_base H :=
  match type of H with
  | ?P → ?Q ⇒ fail 1
  | ∀ _:_, _ ⇒ specializes H __
  end.

Ltac specializes_vars_base H :=
  repeat (specializes_var_base H).

Tactic Notation "specializes_var" hyp(H) :=
  specializes_var_base H.

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

34.4.4 Experimental Tactics for Application

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

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

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

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

```
Tactic Notation "sapply" constr(H) :=
  first [ apply H | eapply H | rapply H | applys H
        | hnf; apply H | hnf; eapply H | hnf; applys H
        | fapply H ].
```

34.4.5 Adding Assumptions

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

```
Tactic Notation "lets_simpl" ident(H) ":" constr(E) :=
  lets H: E; try simpl in H.
```

```
Tactic Notation "lets_simpl" ":" constr(T) :=
  let H := fresh 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 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).
```

34.4.6 Application of Tautologies

logic E, where E is a fact, is equivalent to `assert` H:E; [`tauto` | `eapply` H; `clear` H]. *It is useful* `for` *instance to prove* a *conjunction* [A ∧ 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).
```

34.4.7 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 ∀ n m, P n n m, because x and y might be equal but not convertible.

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

```
Section equatesLemma.
Variables (A0 A1 : Type).
Variables (A2 : ∀ (x1 : A1), Type).
Variables (A3 : ∀ (x1 : A1) (x2 : A2 x1), Type).
Variables (A4 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), Type).
Variables (A5 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), Type).
Variables (A6 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), Type).

Lemma equates_0 : ∀ (P Q:Prop),
  P → P = Q → Q.
Proof. intros. subst. auto. Qed.

Lemma equates_1 :
  ∀ (P:A0→Prop) x1 y1,
  P y1 → x1 = y1 → P x1.
Proof. intros. subst. auto. Qed.

Lemma equates_2 :
  ∀ y1 (P:A0→∀(x1:A1),Prop) x1 x2,
  P y1 x2 → x1 = y1 → P x1 x2.
Proof. intros. subst. auto. Qed.

Lemma equates_3 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1),Prop) x1 x2 x3,
  P y1 x2 x3 → x1 = y1 → P x1 x2 x3.
Proof. intros. subst. auto. Qed.

Lemma equates_4 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2),Prop) x1 x2 x3 x4,
  P y1 x2 x3 x4 → x1 = y1 → P x1 x2 x3 x4.
Proof. intros. subst. auto. Qed.

Lemma equates_5 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2)(x4:A4 x3),Prop) x1 x2 x3 x4 x5,
  P y1 x2 x3 x4 x5 → x1 = y1 → P x1 x2 x3 x4 x5.
Proof. intros. subst. auto. Qed.

Lemma equates_6 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2)(x4:A4 x3)(x5:A5 x4),Prop)
```

```
  x1 x2 x3 x4 x5 x6,
  P y1 x2 x3 x4 x5 x6 → x1 = y1 → P x1 x2 x3 x4 x5 x6.
Proof. intros. subst. auto. Qed.

End equatesLemma.

Ltac equates_lemma n :=
  match nat_from_number n with
  | 0 ⇒ constr:(equates_0)
  | 1 ⇒ constr:(equates_1)
  | 2 ⇒ constr:(equates_2)
  | 3 ⇒ constr:(equates_3)
  | 4 ⇒ constr:(equates_4)
  | 5 ⇒ constr:(equates_5)
  | 6 ⇒ constr:(equates_6)
  end.

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

Ltac equates_several E cont :=
  let all_pos := match type of E with
    | List.list Boxer ⇒ constr:(E)
    | _ ⇒ constr:((boxer E)::nil)
    end in
  let rec go pos :=
     match pos with
     | nil ⇒ cont tt
     | (boxer ?n)::?pos' ⇒ equates_one n; [ instantiate; go pos' | ]
     end in
  go all_pos.

Tactic Notation "equates" constr(E) :=
  equates_several E ltac:(fun _ ⇒ idtac).
Tactic Notation "equates" constr(n1) constr(n2) :=
  equates (» n1 n2).
Tactic Notation "equates" constr(n1) constr(n2) constr(n3) :=
  equates (» n1 n2 n3).
Tactic Notation "equates" constr(n1) constr(n2) constr(n3) constr(n4) :=
  equates (» n1 n2 n3 n4).
```

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

```
Tactic Notation "applys_eq" constr(H) constr(E) :=
  equates_several E ltac:(fun _ ⇒ sapply H).
Tactic Notation "applys_eq" constr(H) constr(n1) constr(n2) :=
```

applys_eq H (» *n1 n2*).
Tactic Notation "applys_eq" constr(*H*) constr(*n1*) constr(*n2*) constr(*n3*) :=
applys_eq H (» *n1 n2 n3*).
Tactic Notation "applys_eq" constr(*H*) constr(*n1*) constr(*n2*) constr(*n3*) constr(*n4*) :=
applys_eq H (» *n1 n2 n3 n4*).

34.4.8 Absurd Goals

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

```
Tactic Notation "false_goal" :=
  elimtype False.
```

false_post is the underlying tactic used to prove goals of the form **False**. In the default implementation, it proves the goal if the context contains **False** or an hypothesis of the form C *x1* .. *xN* = *D y1* .. *yM*, or if the congruence tactic finds a proof of $x \neq x$ for some x.

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

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

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

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

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

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

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

Tactic Notation "false" constr(E) :=
  false_then E ltac:(fun _ ⇒ idtac).
Tactic Notation "false" constr(E) constr(E1) :=
  false (» E E1).
Tactic Notation "false" constr(E) constr(E1) constr(E2) :=
  false (» E E1 E2).
```

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

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

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

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

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

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

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

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

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

Tactic Notation "tryfalse_invert" :=
  try false_invert.
```

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

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

34.5 Introduction and Generalization

34.5.1 Introduction

introv is used to name only non-dependent hypothesis.

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

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

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

  Ltac introv_noarg_not_optimized :=
    intro; match goal with H:_⊢_ ⇒ revert H end; introv_rec.

Ltac introv_arg H :=
  hnf; match goal with
  | ⊢ ?P → ?Q ⇒ intros H
  | ⊢ ∀ _, _ ⇒ 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 *introz* to a goal of the form $\forall$ x, P x $\rightarrow \neg Q$ will introduce x and P x and Q, and will leave **False** in the goal.

Tactic Notation "intros_all" :=
repeat intro.

intros_hnf introduces an hypothesis and sets in head normal form

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

34.5.2 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.
```

34.5.3 Naming

sets X: E is the same as `set` (X := E) `in` *, that is, it replaces all occurences 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 occurences of term E with a fresh variable X and the equality $\mathsf{X} = E$ as extra hypothesis to the current conclusion. In other words a conclusion C will be turned into $(\mathsf{X} = E) \rightarrow \mathsf{C}$. *gen_eq*: E and *gen_eq*: E `as` X are also accepted.

```
Tactic Notation "gen_eq" ident(X) ":" constr(E) :=
  let EQ := fresh 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).
```

34.6 Rewriting

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

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

Tactic Notation "rewrites" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M ).
Tactic Notation "rewrites" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in H).
Tactic Notation "rewrites" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in *).
```

```
Tactic Notation "rewrites" "<-" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M ).
Tactic Notation "rewrites" "<-" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in H).
Tactic Notation "rewrites" "<-" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in *).
```

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

```
Ltac asserts_rewrite_tactic E action :=
  let EQ := fresh 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 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 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 occurence of $T1$ in the current goal. Syntaxes *rewrites* $\leftarrow$ E `at` K and *rewrites* E `at` K `in` H are also available.

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

34.6.1 Replace

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

```
Tactic Notation "replaces" constr(E) "with" constr(F) :=
  let T := fresh in assert (T: E = F); [ | replace E with F; clear T ].
```

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

replaces E at K with F replaces the K-th occurence of E with F in the current goal. Syntax *replaces* E at K with F in H is also available.

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

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

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

Tactic Notation "changes" constr($E1$) "with" constr($E2$) :=
  asserts_rewrite ($E1$ = $E2$); [reflexivity |].

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

34.6.3 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.
```

34.6.4 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 overriden in LibReflect.

```
Ltac apply_to_head_of E cont :=
  let go E :=
    let P := get_head E in cont P in
  match E with
  | ∀ _,_ ⇒ intros; apply_to_head_of E cont
  | ?A = ?B ⇒ first [ go A | go B ]
  | ?A ⇒ go A
  end.
```

```
Ltac unfolds_base :=
  match goal with ⊢ ?G ⇒
   apply_to_head_of G 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.
```

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.
```

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

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

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

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

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

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

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

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

34.6.6 Reduction

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

34.6.7 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 ⊢ _ ⇒
            first [ subst x | subst y ] end).
```

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

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

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

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

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

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

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

```
Ltac intro_subst_hyp := fail.
```

subst_hyp H substitutes the equality contained in H.

```
Ltac subst_hyp_base H :=
  match type of H with
  | (_,_,_,_,_) = (_,_,_,_,_) ⇒ 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.
```

34.6.8 Tactics to Work with Proof Irrelevance

```
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 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).
```

34.6.9 Proving Equalities

Note: current implementation only supports up to arity 5

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

```
Ltac fequal_base :=
  let go := f_equal; [ fequal_base | ] in
  match goal with
  | ⊢ (_,_,_) = (_,_,_) ⇒ go
  | ⊢ (_,_,_,_) = (_,_,_,_) ⇒ go
  | ⊢ (_,_,_,_,_) = (_,_,_,_,_) ⇒ go
  | ⊢ (_,_,_,_,_,_) = (_,_,_,_,_,_) ⇒ go
```

| ⊢ _ ⇒ f_equal
end.

```
Tactic Notation "fequal" :=
  fequal_base.
```

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 .. xN* = `f` *y1 .. yN* and produces some subgoals of the form *xi* = *yi*).

```
Ltac fequal_post :=
  first [ reflexivity | congruence | apply proof_irrelevance | idtac ].
Tactic Notation "fequals" :=
  fequal; fequal_post.
```

fequals_rec calls *fequals* recursively. It is equivalent to `repeat` (`progress` *fequals*).

```
Tactic Notation "fequals_rec" :=
  repeat (progress fequals).
```

34.7 Inversion

34.7.1 Basic Inversion

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

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

invert keep H `as` *X1 .. XN* is the same as `inversion` *H* `as` ... except that only hypotheses which are not variable need to be named explicitely, 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 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).
```

34.7.2 Inversion with Substitution

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

```
Axiom inj_pair2 :
  ∀ (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 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 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; 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 in rename H into H'; tac H'; clear H'.
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1 I2).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3) :=
```

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

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

```
Ltac inverts_as_tactic H :=
  let rec go tt :=
    match goal with
    | ⊢ (ltac_Mark → _) ⇒ intros _
    | ⊢ (?x = ?y → _) ⇒ let H := fresh "TEMP" in intro H;
                          first [ subst x | subst y ];
                          go tt
    | ⊢ (existT ?P ?p ?x = existT ?P ?p ?y → _) ⇒
          let H := fresh in intro H;
          generalize (@inj_pair2 _ P p x y H);
          clear H; go tt
    | ⊢ (∀ _, _) ⇒
        intro; let H := get_last_hyp tt in mark_to_generalize H; go tt
    end in
  pose ltac_mark; inversion H;
  generalize ltac_mark; gen_until_mark;
  go tt; gen_to_generalize; unfolds ltac_to_generalize;
  unfold eq' in *.

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

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

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

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

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

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

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

34.7.3 Injection with Substitution

Underlying implementation of *injects*

```
Ltac injects_tactic H :=
  let rec go _ :=
    match goal with
    | ⊢ (ltac_Mark → _) ⇒ intros _
    | ⊢ (?x = ?y → _) ⇒ let H := fresh 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* .. *aN* = C *b1* .. *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.
```

34.7.4 Inversion and Injection with Substitution –rough implementation

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

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; 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 `intros` and `clear` *H* 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.
```

34.7.5 Case Analysis

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

```
Tactic Notation "cases" constr(E) "as" ident(H) :=
  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 is to be defined later as a tactic to clean up goals. By defaults, it looks for obvious contradictions. Currently, this tactic is extended in LibReflect to clean up boolean propositions.

```
Ltac case_if_post := 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 and `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 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.
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 in case_if as Eq.
Tactic Notation "case_if" "in" hyp(H) :=
  let Eq := fresh 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 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.
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 in cases_if as Eq.
Tactic Notation "cases_if" "in" hyp(H) :=
  let Eq := fresh 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 and `if` statement.

```
Ltac destruct_if_post := tryfalse.

Tactic Notation "destruct_if"
 "as" simple_intropattern(Eq1) simple_intropattern(Eq2) :=
  match goal with
  | ⊢ context [if ?B then _ else _] ⇒ destruct B as [Eq1|Eq2]
  | K: context [if ?B then _ else _] ⊢ _ ⇒ 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 _] ⇒
    destruct B as [Eq1|Eq2] end;
  destruct_if_post.

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

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

BROKEN since v8.5beta2.

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

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

Ltac destruct_head_match_core cont :=
  match goal with
  | ⊢ ?T1 = ?T2 ⇒ first [ let E := find_head_match T1 in cont E
```

```
                                   | let E := find_head_match T2 in cont E ]
  | ⊢ ?T1 ⇒ let E := find_head_match T1 in cont E
  end;
  destruct_if_post.
```

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

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

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

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

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

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

```
Ltac cases_if_on' E Eq :=
  match type of E with
  | {_}+{_} ⇒ destruct E as [Eq|Eq]; try subst_hyp Eq
  | _ ⇒ let X := fresh in
        sets_eq X Eq: E;
        destruct X
  end; case_if_post.
```

```
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 in cases_if' as Eq.
```

34.8 Induction

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

```
Require Import Coq.Program.Equality.
```

```
Ltac inductions_post :=
```

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

induction_wf IH: *E* X is used to apply the well-founded induction principle, for a given well-founded relation. It applies to a goal *PX* where *PX* is a proposition on X. First, it sets up the goal in the form (fun a ⇒ *P* a) X, using pattern X, and then it applies the well-founded induction principle instantiated on *E*, where *E* is a term of type *well_founded* R, and R is a binary relation. Syntaxes *induction_wf*: *E* X and *induction_wf E* X.

```
Tactic Notation "induction_wf" ident(IH) ":" constr(E) ident(X) :=
  pattern X; apply (well_founded_ind E); clear X; intros X IH.
Tactic Notation "induction_wf" ":" constr(E) ident(X) :=
  let IH := fresh "IH" in induction_wf IH: E X.
Tactic Notation "induction_wf" ":" constr(E) ident(X) :=
  induction_wf: E X.
```

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

```
Require Import Compare_dec Omega.

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

```
  ∃ n, n1 < n ∧ n2 < n.
Proof using.
  intros. destruct (lt_dec n1 n2).
  ∃ (S n2). omega.
  ∃ (S n1). omega.
Qed.
```

```
Ltac induct_height_step x :=
  match goal with
  | H: ∃ _, _ ⊢ _ ⇒
      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
  | _ ⇒ ∃ (S x); eauto
 end.
```

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

34.9 Coinduction

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

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

```
Tactic Notation "cofixs" ident(IH) :=
  cofix IH;
  match type of IH with ?P ⇒ 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 _ ⊢ _ ⇒ 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* `omega` to obtain the same behavior as `omega` but with an auxiliary lemma being generated.

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

34.10 Decidable Equality

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

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

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

34.11 Equivalence

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

```
Lemma iff_intro_swap : ∀ (P Q : Prop),
  (Q → P) → (P → Q) → (P ↔ Q).
Proof using. intuition. Qed.
```

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

```
Tactic Notation "iff" "<-" simple_intropattern(H1) simple_intropattern(H2) :=
  apply iff_intro_swap; [ intros H1 | intros H2 ].
Tactic Notation "iff" "<-" simple_intropattern(H) :=
  iff ← H H.
Tactic Notation "iff" "<-" :=
  let H := fresh "H" in iff ← H.
```

34.12 N-ary Conjunctions and Disjunctions

N-ary Conjunctions Splitting in Goals
Underlying implementation of *splits*.

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

```
  end.

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

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

  end.

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

splits applies to a goal of the form (*T1* ∧ .. ∧ *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 := nat_from_number N in
  splits_tactic N.
```

splits_all will recursively split any conjunction, unfolding definitions when necessary. Warning: this tactic will loop on goals of the form *well_founded* R. Todo: fix this

```
Ltac splits_all_base := repeat split.
```

```
Tactic Notation "splits_all" :=
  splits_all_base.
```

N-ary Conjunctions Deconstruction
Underlying implementation of *destructs*.

```
Ltac destructs_conjunction_tactic N T :=
  match N with
  | 2 ⇒ destruct T as [? ?]
  | 3 ⇒ destruct T as [? [? ?]]
   | 4 ⇒ destruct T as [? [? [? ?]]]
   | 5 ⇒ destruct T as [? [? [? [? ?]]]]
   | 6 ⇒ destruct T as [? [? [? [? [? ?]]]]]
   | 7 ⇒ destruct T as [? [? [? [? [? [? ?]]]]]]
   end.
```

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

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

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

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

Proving goals which 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.
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* ∨ ... ∨ *PK* ∨ ... ∨ *PN* and leaves the goal *PK*. It only able to unfold the head definition (if there is one), but for more complex unfolding one should use the tactic *branch K of N*.

```
Tactic Notation "branch" constr(K) :=
  let K := nat_from_number 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* ∨ ... ∨ *PK* ∨ ... ∨ *PN* and leaves the goal *PK*.

```
Tactic Notation "branch" constr(K) "of" constr(N) :=
  let N := nat_from_number N in
  let K := nat_from_number 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 := nat_from_number N in
  destructs_disjunction_tactic N T.
```

N-ary Existentials

```
Ltac get_term_existential_arity T :=
  match T with
  | ∃ x1 x2 x3 x4 x5 x6 x7 x8, _ ⇒ constr:(8)
  | ∃ x1 x2 x3 x4 x5 x6 x7, _ ⇒ constr:(7)
  | ∃ x1 x2 x3 x4 x5 x6, _ ⇒ constr:(6)
  | ∃ x1 x2 x3 x4 x5, _ ⇒ constr:(5)
  | ∃ x1 x2 x3 x4, _ ⇒ constr:(4)
  | ∃ x1 x2 x3, _ ⇒ constr:(3)
  | ∃ x1 x2, _ ⇒ constr:(2)
  | ∃ x1, _ ⇒ constr:(1)
  | _ → ?T' ⇒ get_term_existential_arity T'
  | _ ⇒ let P := get_head T in
        let T' := eval unfold P in T in
        match T' with
        | T ⇒ fail 1
        | _ ⇒ get_term_existential_arity T'
        end
  end.

Ltac get_goal_existential_arity :=
  match goal with ⊢ ?T ⇒ get_term_existential_arity T end.
```

∃ *T1* ... *TN* is a shorthand for ∃ *T1*; ...; ∃ *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. ∃ *T1* .. *TN* ___ is equivalent to ∃ *T1* .. *TN* __ __ __ with as many __ as possible.

```
Tactic Notation "exists_original" constr(T1) :=
  ∃ T1.
Tactic Notation "exists" constr(T1) :=
  match T1 with
  | ltac_wild ⇒ esplit
  | ltac_wilds ⇒ repeat esplit
  | _ ⇒ ∃ T1
  end.
Tactic Notation "exists" constr(T1) constr(T2) :=
  ∃ T1; ∃ T2.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) :=
  ∃ T1; ∃ T2; ∃ T3.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
 constr(T5) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
 constr(T5) constr(T6) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5; ∃ T6.

Tactic Notation "exists___" constr(N) :=
  let rec aux N :=
    match N with
    | 0 ⇒ idtac
    | S ?N' ⇒ esplit; aux N'
    end in
  let N := nat_from_number N in aux N.

Tactic Notation "exists___" :=
  let N := get_goal_existential_arity in
  exists___ N.

Tactic Notation "exists" :=
  exists___.

Tactic Notation "exists_all" := exists___.
```

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: _ ∧ _ ⊢ _ ⇒ destruct H
```

```
  | H: ∃ a, _ ⊢ _ ⇒ destruct H
  end.
Ltac unpack_from H :=
  first [ progress (unpack_core)
        | destruct H; unpack_core ].
Tactic Notation "unpack" :=
  unpack_core.
Tactic Notation "unpack" constr(H) :=
  unpack_from H.
```

34.13 Tactics to Prove Typeclass Instances

typeclass is an automation tactic specialized for finding typeclass instances.

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

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

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

34.14 Tactics to Invoke Automation

34.14.1 Definitions for Parsing Compatibility

```
Tactic Notation "f_equal" :=
  f_equal.
Tactic Notation "constructor" :=
  constructor.
Tactic Notation "simple" :=
  simpl.
Tactic Notation "split" :=
  split.
Tactic Notation "right" :=
  right.
Tactic Notation "left" :=
  left.
```

34.14.2 *hint* to Add Hints Local to a Lemma

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

34.14.3 *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].

34.14.4 Definitions of Automation Tactics

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

auto_tilde is the tactic which will be called each time a symbol ¬ 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 × is used after a tactic.

Ltac *auto_star_default* := try solve [*jauto*].
Ltac *auto_star* := *auto_star_default*.

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

```
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; lets: E2; auto_tilde.
Tactic Notation "autos" "~" constr(E1) constr(E2) constr(E3) :=
  lets: E1; lets: E2; lets: E3; auto_tilde.
```

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

```
Tactic Notation "autos" "*" :=
  auto_star.
Tactic Notation "autos" "*" constr(E1) :=
  lets: E1; auto_star.
Tactic Notation "autos" "*" constr(E1) constr(E2) :=
  lets: E1; lets: E2; auto_star.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) :=
  lets: E1; lets: E2; lets: E3; auto_star.
```

auto_false is a version of `auto` able to spot some contradictions. There is an ad-hoc support for goals in ↔: 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 ⊢ _ ↔ _ ⇒ split end;
    solve [ cont tt | intros_all; false; cont tt ] ].

Tactic Notation "auto_false" :=
   auto_false_base ltac:(fun tt ⇒ auto).
Tactic Notation "auto_false" "~" :=
   auto_false_base ltac:(fun tt ⇒ auto_tilde).
Tactic Notation "auto_false" "*" :=
   auto_false_base ltac:(fun tt ⇒ auto_star).
```

34.14.5 Parsing for Light Automation

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

- *cuts* and *asserts* only call `auto` on their first subgoal,

- `apply¬` relies on *sapply* rather than `apply`,
- *tryfalse¬* 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¬` will not be available.

```
Tactic Notation "equates" "~" constr(E) :=
  equates E; auto_tilde.
Tactic Notation "equates" "~" constr(n1) constr(n2) :=
  equates n1 n2; auto_tilde.
Tactic Notation "equates" "~" constr(n1) constr(n2) constr(n3) :=
  equates n1 n2 n3; auto_tilde.
Tactic Notation "equates" "~" constr(n1) constr(n2) constr(n3) constr(n4) :=
  equates n1 n2 n3 n4; auto_tilde.

Tactic Notation "applys_eq" "~" constr(H) constr(E) :=
  applys_eq H E; auto_tilde.
Tactic Notation "applys_eq" "~" constr(H) constr(n1) constr(n2) :=
  applys_eq H n1 n2; auto_tilde.
Tactic Notation "applys_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) :=
  applys_eq H n1 n2 n3; auto_tilde.
Tactic Notation "applys_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) constr(n4)
:=
  applys_eq H n1 n2 n3 n4; auto_tilde.

Tactic Notation "apply" "~" constr(H) :=
  sapply H; auto_tilde.

Tactic Notation "destruct" "~" constr(H) :=
  destruct H; auto_tilde.
Tactic Notation "destruct" "~" constr(H) "as" simple_intropattern(I) :=
  destruct H as I; auto_tilde.
Tactic Notation "f_equal" "~" :=
  f_equal; auto_tilde.
Tactic Notation "induction" "~" constr(H) :=
  induction H; auto_tilde.
Tactic Notation "inversion" "~" constr(H) :=
  inversion H; auto_tilde.
Tactic Notation "split" "~" :=
  split; auto_tilde.
Tactic Notation "subst" "~" :=
  subst; auto_tilde.
Tactic Notation "right" "~" :=
  right; auto_tilde.
Tactic Notation "left" "~" :=
```

```
  left; auto_tilde.
Tactic Notation "constructor" "~" :=
  constructor; auto_tilde.
Tactic Notation "constructors" "~" :=
  constructors; auto_tilde.

Tactic Notation "false" "~" :=
  false; auto_tilde.
Tactic Notation "false" "~" constr(E) :=
  false_then E ltac:(fun _ => auto_tilde).
Tactic Notation "false" "~" constr(E0) constr(E1) :=
  false~ (>> E0 E1).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) :=
  false~ (>> E0 E1 E2).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) constr(E3) :=
  false~ (>> E0 E1 E2 E3).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) constr(E3) constr(E4)
:=
  false~ (>> E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "~" :=
  try solve [ false~ ].

Tactic Notation "asserts" "~" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_tilde | idtac ].
Tactic Notation "asserts" "~" ":" constr(E) :=
  let H := fresh "H" in asserts~ H: E.
Tactic Notation "cuts" "~" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_tilde | idtac ].
Tactic Notation "cuts" "~" ":" constr(E) :=
  cuts: E; [ auto_tilde | idtac ].

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

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

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

Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E) :=
  forwards I: E; auto_tilde.
Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E0)
 constr(A1) :=
  forwards I: E0 A1; auto_tilde.
Tactic Notation "forwards" "~" 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 "applys" "~" constr(H) :=
  sapply H; auto_tilde. Tactic Notation "applys" "~" constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) :=
  applys E0 A1 A2; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applys E0 A1 A2 A3; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  applys E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  applys 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 "substs" "~" :=
  substs; auto_tilde.
Tactic Notation "intro_hyp" "~" hyp(H) :=
  subst_hyp H; auto_tilde.
Tactic Notation "intro_subst" "~" :=
  intro_subst; auto_tilde.
Tactic Notation "subst_eq" "~" constr(E) :=
  subst_eq E; auto_tilde.

Tactic Notation "rewrite" "~" constr(E) :=
  rewrite E; auto_tilde.
Tactic Notation "rewrite" "~" "<-" constr(E) :=
  rewrite ← E; auto_tilde.
Tactic Notation "rewrite" "~" 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 "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 "destruct_head_match" "~" :=
  destruct_head_match; 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 "splits" "~" :=
  splits; auto_tilde.
Tactic Notation "splits" "~" constr(N) :=
  splits N; auto_tilde.
Tactic Notation "splits_all" "~" :=
  splits_all; 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" "~" 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___" "~" :=
  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*.

34.14.6 Parsing for Strong Automation

Any tactic followed by the symbol × will have `auto`× called on all of its subgoals. The exceptions to these rules are the same as for light automation.

Exception: use *subs*× instead of `subst`× if you import the library *Coq.Classes.Equivalence*.

Tactic Notation "equates" "*" constr(*E*) :=
   equates E; *auto_star*.
Tactic Notation "equates" "*" constr(*n1*) constr(*n2*) :=
  equates n1 n2; *auto_star*.
Tactic Notation "equates" "*" constr(*n1*) constr(*n2*) constr(*n3*) :=
  equates n1 n2 n3; *auto_star*.
Tactic Notation "equates" "*" constr(*n1*) constr(*n2*) constr(*n3*) constr(*n4*) :=
  equates n1 n2 n3 n4; *auto_star*.

Tactic Notation "applys_eq" "*" constr(*H*) constr(*E*) :=
  applys_eq H E; *auto_star*.
Tactic Notation "applys_eq" "*" constr(*H*) constr(*n1*) constr(*n2*) :=
  applys_eq H n1 n2; *auto_star*.
Tactic Notation "applys_eq" "*" constr(*H*) constr(*n1*) constr(*n2*) constr(*n3*) :=
  applys_eq H n1 n2 n3; *auto_star*.
Tactic Notation "applys_eq" "*" constr(*H*) constr(*n1*) constr(*n2*) constr(*n3*) constr(*n4*)
:=
  applys_eq H n1 n2 n3 n4; *auto_star*.

Tactic Notation "apply" "*" constr(*H*) :=
  sapply H; *auto_star*.

Tactic Notation "destruct" "*" constr(*H*) :=
  destruct *H*; *auto_star*.
Tactic Notation "destruct" "*" constr(*H*) "as" *simple_intropattern*(*I*) :=
  destruct *H* as *I*; *auto_star*.

```
Tactic Notation "f_equal" "*" :=
  f_equal; auto_star.
Tactic Notation "induction" "*" constr(H) :=
  induction H; auto_star.
Tactic Notation "inversion" "*" constr(H) :=
  inversion H; auto_star.
Tactic Notation "split" "*" :=
  split; auto_star.
Tactic Notation "subs" "*" :=
  subst; auto_star.
Tactic Notation "subst" "*" :=
  subst; auto_star.
Tactic Notation "right" "*" :=
  right; auto_star.
Tactic Notation "left" "*" :=
  left; auto_star.
Tactic Notation "constructor" "*" :=
  constructor; auto_star.
Tactic Notation "constructors" "*" :=
  constructors; auto_star.

Tactic Notation "false" "*" :=
  false; auto_star.
Tactic Notation "false" "*" constr(E) :=
  false_then E ltac:(fun _ ⇒ auto_star).
Tactic Notation "false" "*" constr(E0) constr(E1) :=
  false× (» E0 E1).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) :=
  false× (» E0 E1 E2).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) constr(E3) :=
  false× (» E0 E1 E2 E3).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) constr(E3) constr(E4)
:=
  false× (» E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "*" :=
  try solve [ false× ].

Tactic Notation "asserts" "*" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_star | idtac ].
Tactic Notation "asserts" "*" ":" constr(E) :=
  let H := fresh "H" in asserts× H: E.
Tactic Notation "cuts" "*" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_star | idtac ].
Tactic Notation "cuts" "*" ":" constr(E) :=
```

```
  cuts: E; [ auto_star | idtac ].
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E) :=
  lets I: E; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) :=
  lets I: E0 A1; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) :=
  lets I: E0 A1 A2; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets I: E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets I: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "lets" "*" ":" constr(E) :=
  lets: E; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) :=
  lets: E0 A1; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) :=
  lets: E0 A1 A2; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets: E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E) :=
  forwards I: E; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) :=
  forwards I: E0 A1; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) :=
```

```
  forwards I: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  forwards I: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards I: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "forwards" "*" ":" constr(E) :=
  forwards: E; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) :=
  forwards: E0 A1; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) :=
  forwards: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  forwards: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "applys" "*" constr(H) :=
  sapply H; auto_star. Tactic Notation "applys" "*" constr(E0) constr(A1) :=
  applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) :=
  applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) :=
  applys E0 A1 A2; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applys E0 A1 A2 A3; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  applys E0 A1 A2 A3 A4; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  applys E0 A1 A2 A3 A4 A5; auto_star.
```

```
Tactic Notation "specializes" "*" hyp(H) :=
  specializes H; auto_star.
Tactic Notation "specializes" "~" hyp(H) constr(A1) :=
  specializes H A1; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H A1 A2 A3; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  specializes H A1 A2 A3 A4; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_star.

Tactic Notation "fapply" "*" constr(E) :=
  fapply E; auto_star.
Tactic Notation "sapply" "*" constr(E) :=
  sapply E; auto_star.

Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ ⇒ auto_star).

Tactic Notation "intros_all" "*" :=
  intros_all; auto_star.

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

Tactic Notation "simple" "*" :=
  simpl; auto_star.
Tactic Notation "simple" "*" "in" hyp(H) :=
  simpl in H; auto_star.
Tactic Notation "simpls" "*" :=
  simpls; auto_star.
Tactic Notation "hnfs" "*" :=
  hnfs; auto_star.
```

```
Tactic Notation "hnfs" "*" "in" hyp(H) :=
  hnf in H; auto_star.
Tactic Notation "substs" "*" :=
  substs; auto_star.
Tactic Notation "intro_hyp" "*" hyp(H) :=
  subst_hyp H; auto_star.
Tactic Notation "intro_subst" "*" :=
  intro_subst; auto_star.
Tactic Notation "subst_eq" "*" constr(E) :=
  subst_eq E; auto_star.

Tactic Notation "rewrite" "*" constr(E) :=
  rewrite E; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) :=
  rewrite ← E; auto_star.
Tactic Notation "rewrite" "*" constr(E) "in" hyp(H) :=
  rewrite E in H; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  rewrite ← E in H; auto_star.

Tactic Notation "rewrites" "*" constr(E) :=
  rewrites E; auto_star.
Tactic Notation "rewrites" "*" constr(E) "in" hyp(H):=
  rewrites E in H; auto_star.
Tactic Notation "rewrites" "*" constr(E) "in" "*":=
  rewrites E in *; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) :=
  rewrites ← E; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" hyp(H):=
  rewrites ← E in H; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" "*":=
  rewrites ← E in *; auto_star.

Tactic Notation "rewrite_all" "*" constr(E) :=
  rewrite_all E; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) :=
  rewrite_all ← E; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) "in" ident(H) :=
  rewrite_all E in H; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" ident(H) :=
  rewrite_all ← E in H; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) "in" "*" :=
  rewrite_all E in *; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" "*" :=
  rewrite_all ← E in *; auto_star.
```

```
Tactic Notation "asserts_rewrite" "*" constr(E) :=
  asserts_rewrite E; auto_star.
Tactic Notation "asserts_rewrite" "*" "<-" constr(E) :=
  asserts_rewrite ← E; auto_star.
Tactic Notation "asserts_rewrite" "*" constr(E) "in" hyp(H) :=
  asserts_rewrite E; auto_star.
Tactic Notation "asserts_rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  asserts_rewrite ← E; auto_star.
Tactic Notation "asserts_rewrite" "*" constr(E) "in" "*" :=
  asserts_rewrite E in *; auto_tilde.
Tactic Notation "asserts_rewrite" "*" "<-" constr(E) "in" "*" :=
  asserts_rewrite ← E in *; auto_tilde.

Tactic Notation "cuts_rewrite" "*" constr(E) :=
  cuts_rewrite E; auto_star.
Tactic Notation "cuts_rewrite" "*" "<-" constr(E) :=
  cuts_rewrite ← E; auto_star.
Tactic Notation "cuts_rewrite" "*" constr(E) "in" hyp(H) :=
  cuts_rewrite E in H; auto_star.
Tactic Notation "cuts_rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  cuts_rewrite ← E in H; auto_star.

Tactic Notation "erewrite" "*" constr(E) :=
  erewrite E; auto_star.

Tactic Notation "fequal" "*" :=
  fequal; auto_star.
Tactic Notation "fequals" "*" :=
  fequals; auto_star.
Tactic Notation "pi_rewrite" "*" constr(E) :=
  pi_rewrite E; auto_star.
Tactic Notation "pi_rewrite" "*" constr(E) "in" hyp(H) :=
  pi_rewrite E in H; auto_star.

Tactic Notation "invert" "*" hyp(H) :=
  invert H; auto_star.
Tactic Notation "inverts" "*" hyp(H) :=
  inverts H; auto_star.
Tactic Notation "inverts" "*" hyp(E) "as" :=
  inverts E as; auto_star.
Tactic Notation "injects" "*" hyp(H) :=
  injects H; auto_star.
Tactic Notation "inversions" "*" hyp(H) :=
  inversions H; auto_star.

Tactic Notation "cases" "*" constr(E) "as" ident(H) :=
```

```
  cases E as H; auto_star.
Tactic Notation "cases" "*" constr(E) :=
  cases E; auto_star.
Tactic Notation "case_if" "*" :=
  case_if; auto_star.
Tactic Notation "case_ifs" "*" :=
  case_ifs; auto_star.
Tactic Notation "case_if" "*" "in" hyp(H) :=
  case_if in H; auto_star.
Tactic Notation "cases_if" "*" :=
  cases_if; auto_star.
Tactic Notation "cases_if" "*" "in" hyp(H) :=
  cases_if in H; auto_star.
 Tactic Notation "destruct_if" "*" :=
  destruct_if; auto_star.
Tactic Notation "destruct_if" "*" "in" hyp(H) :=
  destruct_if in H; auto_star.
Tactic Notation "destruct_head_match" "*" :=
  destruct_head_match; auto_star.

Tactic Notation "cases'" "*" constr(E) "as" ident(H) :=
  cases' E as H; auto_star.
Tactic Notation "cases'" "*" constr(E) :=
  cases' E; auto_star.
Tactic Notation "cases_if'" "*" "as" ident(H) :=
  cases_if' as H; auto_star.
Tactic Notation "cases_if'" "*" :=
  cases_if'; auto_star.

Tactic Notation "decides_equality" "*" :=
  decides_equality; auto_star.

Tactic Notation "iff" "*" :=
  iff; auto_star.
Tactic Notation "iff" "*" simple_intropattern(I) :=
  iff I; auto_star.
Tactic Notation "splits" "*" :=
  splits; auto_star.
Tactic Notation "splits" "*" constr(N) :=
  splits N; auto_star.
Tactic Notation "splits_all" "*" :=
  splits_all; auto_star.

Tactic Notation "destructs" "*" constr(T) :=
  destructs T; auto_star.
```

```
Tactic Notation "destructs" "*" constr(N) constr(T) :=
  destructs N T; auto_star.
Tactic Notation "branch" "*" constr(N) :=
  branch N; auto_star.
Tactic Notation "branch" "*" constr(K) "of" constr(N) :=
  branch K of N; auto_star.
Tactic Notation "branches" "*" constr(T) :=
  branches T; auto_star.
Tactic Notation "branches" "*" constr(N) constr(T) :=
  branches N T; auto_star.
Tactic Notation "exists" "*" :=
  ∃; auto_star.
Tactic Notation "exists___" "*" :=
  exists___; auto_star.
Tactic Notation "exists" "*" constr(T1) :=
  ∃ T1; 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.
```

34.15 Tactics to Sort Out the Proof Context

34.15.1 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.
```

34.15.2 Sorting Hypotheses

sort sorts out hypotheses from the context by moving all the propositions (hypotheses of type Prop) to the bottom of the context.

```
Ltac sort_tactic :=
  try match goal with H: ?T ⊢ _ ⇒
  match type of T with Prop ⇒
    generalizes H; (try sort_tactic); intro
  end end.

Tactic Notation "sort" :=
  sort_tactic.
```

34.15.3 Clearing Hypotheses

clears X1 ... XN is a variation on `clear` which clears the variables *X1..XN* as well as all the hypotheses which depend on them. Contrary to `clear`, it never fails.

```
Tactic Notation "clears" ident(X1) :=
  let rec doit _ :=
  match goal with
  | H:context[X1] ⊢ _ ⇒ clear H; try (doit tt)
  | _ ⇒ clear X1
  end in doit tt.
Tactic Notation "clears" ident(X1) ident(X2) :=
  clears X1; clears X2.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) :=
  clears X1; clears X2; clears X3.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4) :=
  clears X1; clears X2; clears X3; clears X4.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
 ident(X5) :=
  clears X1; clears X2; clears X3; clears X4; clears X5.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
 ident(X5) ident(X6) :=
```

clears X1; *clears X2*; *clears X3*; *clears X4*; *clears X5*; *clears X6*.

clears (without any argument) clears all the unused variables from the context. In other words, it removes any variable which is not a proposition (i.e., not of type Prop) and which does not appear in another hypothesis nor in the goal.

```
Ltac clears_tactic :=
  match goal with H: ?T ⊢ _ ⇒
  match type of T with
  | Prop ⇒ generalizes H; (try clears_tactic); intro
  | ?TT ⇒ clear H; (try clears_tactic)
  | ?TT ⇒ generalizes H; (try clears_tactic); intro
  end end.
```

```
Tactic Notation "clears" :=
  clears_tactic.
```

clears_all clears all the hypotheses from the context that can be cleared. It leaves only the hypotheses that are mentioned in the goal.

```
Ltac clears_or_generalizes_all_core :=
  repeat match goal with H: _ ⊢ _ ⇒
           first [ clear H | generalizes H] end.
```

```
Tactic Notation "clears_all" :=
  generalize ltac_mark;
  clears_or_generalizes_all_core;
  intro_until_mark.
```

clears_but 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 nat_from_number N with
  | 0 ⇒ idtac
  | S ?p ⇒ clears_last; clears_last_base p
  end.
Tactic Notation "clears_last" constr(N) :=
  clears_last_base N.
```

34.16 Tactics for Development Purposes

34.16.1 Skipping Subgoals

DEPRECATED: the new "admit" tactics now works fine.

The *skip* tactic can be used at any time to admit the current goal. Using *skip* is much more efficient than using the *Focus* top-level command to reach a particular subgoal.

There are two possible implementations of *skip*. The first one relies on the use of an existential variable. The second one relies on an axiom of type **False**. Remark that the builtin tactic *admit* is not applicable if the current goal contains uninstantiated variables.

The advantage of the first technique is that a proof using *skip* must end with *Admitted*, since `Qed` will be rejected with the message "*uninstantiated existential variables*". It is thereafter clear that the development is incomplete.

The advantage of the second technique is exactly the converse: one may conclude the proof using `Qed`, and thus one saves the pain from renaming `Qed` into *Admitted* and vice-versa all the time. Note however, that it is still necessary to instantiate all the existential variables introduced by other tactics in order for `Qed` to be accepted.

The two implementation are provided, so that you can select the one that suits you best. By default *skip'* uses the first implementation, and *skip* uses the second implementation.

```
Ltac skip_with_existential :=
  match goal with ⊢ ?G ⇒
    let H := fresh in evar(H:G); eexact H end.

Parameter skip_axiom : False.
Ltac skip_with_axiom :=
  elimtype False; apply skip_axiom.

Tactic Notation "skip" :=
   skip_with_axiom.
Tactic Notation "skip'" :=
   skip_with_existential.
```

demo is like *admit* but it documents the fact that admit is intended `Tactic Notation "demo" :=
  skip.`

skip H: T adds an assumption named *H* of type T to the current context, blindly assuming that it is true. *skip*: T and *skip H_asserts*: T and *skip_asserts*: T are other possible syntax. Note that H may be an intro pattern. The syntax *skip H1 .. HN*: T can be used when T is a conjunction of *N* items.

```
Tactic Notation "skip" simple_intropattern(I) ":" constr(T) :=
  asserts I: T; [ skip | ].
Tactic Notation "skip" ":" constr(T) :=
  let H := fresh in skip H: T.
Tactic Notation "skip" "~" ":" constr(T) :=
  skip: T; auto_tilde.
Tactic Notation "skip" "*" ":" constr(T) :=
  skip: T; auto_star.

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

skip [*I1* [*I2* [*I3* [*I4* *I5*]]]]: *T*.

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

Tactic Notation "skip_asserts" *simple_intropattern*(*I*) ":" constr(*T*) :=
skip *I*: *T*.

Tactic Notation "skip_asserts" ":" constr(*T*) :=
skip: *T*.

skip_cuts T simply replaces the current goal with T.

Tactic Notation "skip_cuts" constr(*T*) :=
cuts: *T*; [*skip* |].

skip_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 *skip_goal* is also accepted.

Tactic Notation "skip_goal" *ident*(*H*) :=
match goal with $\vdash$?*G* $\Rightarrow$ *skip* *H*: *G* end.

Tactic Notation "skip_goal" :=
let *IH* := fresh "IH" in *skip_goal* *IH*.

skip_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 "skip_rewrite" constr(*T*) :=
let *M* := fresh in *skip_asserts* *M*: *T*; rewrite *M*; clear *M*.

skip_rewrite T in *H* is similar as *rewrite_skip*, except that it rewrites in hypothesis *H*.

Tactic Notation "skip_rewrite" constr(*T*) "in" *hyp*(*H*) :=
let *M* := fresh in *skip_asserts* *M*: *T*; rewrite *M* in *H*; clear *M*.

skip_rewrites_all T is similar as *rewrite_skip*, except that it rewrites everywhere (goal and all hypotheses).

Tactic Notation "skip_rewrite_all" constr(*T*) :=
let *M* := fresh in *skip_asserts* *M*: *T*; *rewrite_all* *M*; clear *M*.

skip_induction *E* applies to any goal. It simply assumes the current goal to be true (the assumption is named "IH" by default), and call destruct *E* instead of induction *E*. It is useful to try and set up a proof by induction first, and fix the applications of the induction hypotheses during a second pass on the Proof using.

Tactic Notation "skip_induction" constr(*E*) :=
let *IH* := fresh "IH" in *skip_goal* *IH*; destruct *E*.

Tactic Notation "skip_induction" constr(*E*) "as" *simple_intropattern*(*I*) :=
let *IH* := fresh "IH" in *skip_goal* *IH*; destruct *E* as *I*.

34.17 Compatibility with Standard Library

The module Program contains definitions that conflict with the current module. If you import Program, either directly or indirectly (e.g., through Setoid or ZArith), you will need to import the compability definitions through the top-level command: `Import` LibTacticsCompatibility.

```
Module LibTacticsCompatibility.
  Tactic Notation "apply" "*" constr(H) :=
    sapply H; auto_star.
  Tactic Notation "subst" "*" :=
    subst; auto_star.
End LibTacticsCompatibility.
```

```
Open Scope nat_scope.
```

$Date : 2017-01-3119 : 12 : 59-0500(Tue, 31Jan2017)$

Chapter 35

Library Top.UseTactics

35.1 UseTactics: Tactic Library for Coq: A Gentle Introduction

Coq 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 Coq. These tactics are defined in the *LibTactics.v* file.

```
Require Import Coq.Arith.Arith.

Require Import Maps.
Require Import Imp.
Require Import Types.
Require Import Smallstep.
Require 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 Coq 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 http://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.

35.2 Tactics for Introduction and Case Analysis

This section presents the following tactics:

- *introv*, for naming hypotheses more efficiently,
- *inverts*, for improving the `inversion` tactic,
- *cases*, for performing a case analysis without losing information,
- *cases_if*, for automating case analysis on the argument of `if`.

35.2.1 The Tactic *introv*

```
Module IntrovExamples.
  Require Import Stlc.
  Import Imp.
  Import STLC.
```

The tactic *introv* 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 name where 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: ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  introv E1 E2. Abort.
```

When there is no hypothesis to be named, one can call *introv* without any argument.

```
Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
  (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
Proof.
  introv. Abort.
```

The tactic *introv* also applies to statements in which ∀ and → are interleaved.

```
Theorem ceval_deterministic': ∀ c st st1,
  (c / st \\ st1) → ∀ st2, (c / st \\ st2) → st1 = st2.
Proof.
  introv E1 E2. Abort.
```

Like the arguments of `intros`, the arguments of *introv* can be structured patterns.

```
Theorem exists_impl: ∀ X (P : X → Prop) (Q : Prop) (R : Prop),
      (∀ x, P x → Q) →
      ((∃ x, P x) → Q).
Proof.
  introv [x H2]. eauto.
Qed.
```

Remark: the tactic *introv* works even when definitions need to be unfolded in order to reveal hypotheses.

```
End IntrovExamples.
```

35.2.2 The Tactic *inverts*

```
Module InvertsExamples.
  Require Import Stlc.
  Require Import Equiv.
  Require Import Imp.
  Import STLC.
```

The `inversion` tactic of Coq 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 intented 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: ∀ c,
  cequiv (SKIP;; c) 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: ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ 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': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ 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.
```

A more involved example appears next. In particular, this example shows that the name of the hypothesis being inverted can be reused.

```
Example typing_nonexample_1 :
  ¬ ∃ T,
      has_type empty
        (tabs x TBool
            (tabs y TBool
               (tapp (tvar x) (tvar y))))
        T.
Proof.
  dup 3.

  - intros C. destruct C.
  inversion H. subst. clear H.
  inversion H5. subst. clear H5.
  inversion H4. subst. clear H4.
  inversion H2. subst. clear H2.
  inversion H5. subst. clear H5.
  inversion H1.

  - intros C. destruct C.
  inverts H as H1.
  inverts H1 as H2.
  inverts H2 as H3.
  inverts H3 as H4.
  inverts H4.

  - intros C. destruct C.
  inverts H as H.
  inverts H as H.
  inverts H as H.
  inverts H as H.
  inverts H.
Qed.

End InvertsExamples.
```

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.

35.3 Tactics for N-ary Connectives

Because Coq encodes conjunctions and disjunctions using binary constructors $\wedge$ and $\vee$, working with a conjunction or a disjunction of N facts can sometimes be quite cumbursome. For this reason, "LibTactics" provides tactics offering direct support for n-ary conjunctions and disjunctions. It also provides direct support for n-ary existententials.

This section presents the following tactics:

- *splits* for decomposing n-ary conjunctions,
- *branch* for decomposing n-ary disjunctions,
- $\exists$ for proving n-ary existentials.

```
Module NaryExamples.
  Require Import References.
  Require Import Smallstep.
  Import STLCRef.
```

35.3.1 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 : ∀ n m,
  n > 0 ∧ n < m ∧ m < n+10 ∧ m ≠ 3.
Proof.
  intros. splits.
Abort.
```

35.3.2 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 : ∀ n m,
  n < m ∨ n = m ∨ 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.
```

35.3.3 The Tactic ∃

The library "LibTactics" introduces a notation for n-ary existentials. For example, one can write ∃ x y z, *H* instead of ∃ x, ∃ y, ∃ z, *H*. Similarly, the library provides a n-ary tactic ∃ a b c, which is a shorthand for ∃ a; ∃ b; ∃ c. The following example illustrates both the notation and the tactic for dealing with n-ary existentials.

```
Theorem progress : ∀ ST t T st,
  has_type empty ST t T →
  store_well_typed ST st →
  value t ∨ ∃ t' st', t / st ==> t' / st'.
Proof with eauto.
  intros ST t T st Ht HST. remember (@empty ty) 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]...
      inversion Ht2p as [t2' [st' Hstep]].
      ∃ (tapp (tabs x T t) t2') st'...
Abort.
```

Remark: a similar facility for n-ary existentials is provided by the module *Coq.Program.Syntax* from the standard library. (*Coq.Program.Syntax* supports existentials up to arity 4; LibTactics supports them up to arity 10.

```
End NaryExamples.
```

35.4 Tactics for Working with Equality

One of the major weakness of Coq 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,

- *cuts_rewrite*, which is similar except that its subgoals are swapped,
- *substs* for improving the `subst` tactic,
- *fequals* for improving the `f_equal` tactic,
- *applys_eq* for proving P `x` `y` using an hypothesis P `x` `z`, automatically producing an equality `y` = `z` as subgoal.

`Module` *EqualityExamples*.

35.4.1 The Tactics *asserts_rewrite* and *cuts_rewrite*

The tactic *asserts_rewrite* ($E1$ = $E2$) replaces $E1$ with $E2$ in the goal, and produces the goal $E1$ = $E2$.

```
Theorem mult_0_plus : ∀ n m : nat,
  (0 + n) × m = n × 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.
```

The tactic *cuts_rewrite* ($E1$ = $E2$) is like *asserts_rewrite* ($E1$ = $E2$), except that the equality $E1$ = $E2$ appears as first subgoal.

```
Theorem mult_0_plus' : ∀ n m : nat,
  (0 + n) × m = n × m.
Proof.
  intros n m.
  cuts_rewrite (0 + n = n).
    reflexivity.    reflexivity. Qed.
```

More generally, the tactics *asserts_rewrite* and *cuts_rewrite* can be provided a lemma as argument. For example, one can write *asserts_rewrite* (∀ `a` `b`, `a`*(`S` `b`) = `a`×`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'' : ∀ u v w x y z: nat,
  (u + v) × (S (w × x + y)) = z.
Proof.
  intros. asserts_rewrite (∀ a b, a*(S b) = a×b+a).
Abort.
```

35.4.2 The Tactic *substs*

The tactic *substs* is similar to `subst` except that it does not fail when the goal contains "circular equalities", such as $x = f\ x$.

```
Lemma demo_substs : ∀ x y (f:nat→nat),
  x = f x → y = x → y = f x.
Proof.
  intros. substs.  assumption.
Qed.
```

35.4.3 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 : ∀ (a b c d e : nat) (f : nat→nat→nat→nat→nat),
  a = 1 → b = e → e = 2 →
  f a b c d = f 1 2 c 4.
Proof.
  intros. fequals.
Abort.
```

35.4.4 The Tactic *applys_eq*

The tactic *applys_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 *applys_eq*, we can call *applys_eq* H 1, which proves the goal and leaves only the subgoal y = z. The value 1 given as argument to *applys_eq* indicates that we want an equality to be introduced for the first argument of P x y counting from the right. The three following examples illustrate the behavior of a call to *applys_eq* H 1, a call to *applys_eq* H 2, and a call to *applys_eq* H 1 2.

```
Axiom big_expression_using : nat→nat.

Lemma demo_applys_eq_1 : ∀ (P:nat→nat→Prop) x y z,
  P x (big_expression_using z) →
  P x (big_expression_using y).
Proof.
  introv H. dup.

  assert (Eq: big_expression_using y = big_expression_using z).
    admit.  rewrite Eq. apply H.
```

applys_eq H 1.
  admit. Abort.

If the mismatch was on the first argument of *P* instead of the second, we would have written *applys_eq H* 2. Recall that the occurences are counted from the right.

```
Lemma demo_applys_eq_2 : ∀ (P:nat→nat→Prop) x y z,
  P (big_expression_using z) x →
  P (big_expression_using y) x.
Proof.
  introv H. applys_eq H 2.
Abort.
```

When we have a mismatch on two arguments, we want to produce two equalities. To achieve this, we may call *applys_eq H* 1 2. More generally, the tactic *applys_eq* expects a lemma and a sequence of natural numbers as arguments.

```
Lemma demo_applys_eq_3 : ∀ (P:nat→nat→Prop) x1 x2 y1 y2,
  P (big_expression_using x2) (big_expression_using y2) →
  P (big_expression_using x1) (big_expression_using y1).
Proof.
  introv H. applys_eq H 1 2.
Abort.

End EqualityExamples.
```

35.5 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 `dependent generalize`,
- *skip* for skipping a subgoal even if it contains existential variables,
- *sort* for re-ordering the proof context by moving moving all propositions at the bottom.

35.5.1 The Tactic *unfolds*

```
Module UnfoldsExample.
  Require 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 : ∀ b st,
  beval st b = true → (bassn b) st.
Proof.
  intros b st Hbe. dup.

  unfold bassn. 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.
```

35.5.2 The Tactics false and *tryfalse*

The tactic false can be used to replace any goal with **False**. In short, it is a shorthand for *exfalso*. Moreover, false proves the goal if it contains an absurd assumption, such as **False** or $0 = \mathsf{S}\ \mathsf{n}$, or if it contains contradictory assumptions, such as $\mathsf{x} = \mathsf{true}$ and $\mathsf{x} = \mathsf{false}$.

```
Lemma demo_false :
  ∀ n, S n = 1 → 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 :
  (∀ n, n < 0 → False) → (3 < 0) → 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 :
  ∀ n, S n = 1 → n = 0.
Proof.
  intros. destruct n; tryfalse. reflexivity.
Qed.
```

35.5.3 The Tactic *gen*

The tactic *gen* is a shortand for `generalize dependent` that accepts several arguments at once. An invokation of this tactic takes the form *gen* `x y z`.

```
Module GenExample.
  Require Import Stlc.
  Import STLC.

Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (update Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof.
  dup.

  intros Gamma x U v t S Htypt Htypv.
  generalize dependent S. generalize dependent Gamma.
  induction t; intros; simpl.
  admit. admit. admit. admit. admit. admit.

  introv Htypt Htypv. gen S Gamma.
  induction t; intros; simpl.
  admit. admit. admit. admit. admit. admit.
Abort.

End GenExample.
```

35.5.4 The Tactics *skip*, *skip_rewrite* and *skip_goal*

Temporarily admitting a given subgoal is very useful when constructing proofs. It gives the ability to focus first on the most interesting cases of a proof. The tactic *skip* is like *admit* except that it also works when the proof includes existential variables. Recall that existential variables are those whose name starts with a question mark, (e.g., ?24), and which are typically introduced by `eapply`.

```
Module SkipExample.
  Require Import Stlc.
  Import STLC.

Notation " t '/' st '==>a*' t' " := (multi (astep st) t t')
                                    (at level 40, st at level 39).

Example astep_example1 :
  (APlus (ANum 3) (AMult (ANum 3) (ANum 4))) / empty_state ==>a× (ANum 15).
Proof.
  eapply multi_step. skip.   eapply multi_step. skip. skip.
Abort.
```

The tactic *skip* 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: *skip* H: P is simply a shorthand for `assert` $(H{:}P)$. *skip*.

```
Theorem demo_skipH : True.
Proof.
  skip H: (∀ n m : nat, (0 + n) × m = n × m).
Abort.
```

The tactic *skip_rewrite* ($E1$ = $E2$) replaces $E1$ with $E2$ in the goal, without checking that $E1$ is actually equal to $E2$.

```
Theorem mult_0_plus : ∀ n m : nat,
  (0 + n) × m = n × m.
Proof.
  dup.

  intros n m.
  assert (H: 0 + n = n). skip. rewrite → H.
  reflexivity.

  intros n m.
  skip_rewrite (0 + n = n).
  reflexivity.
Qed.
```

Remark: the tactic *skip_rewrite* can in fact be given a lemma statement as argument, in the same way as *asserts_rewrite*.

The tactic *skip_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 invokations of the induction hypothesis.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  skip_goal.
  introv 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.
```

35.5.5 The Tactic *sort*

```
Module SortExamples.
  Require 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: ∀ c st st1 st2,
   c / st \\ st1 →
   c / st \\ st2 →
   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.
```

35.6 Tactics for Advanced Lemma Instantiation

This last section describes a mechanism for instantiating a lemma by providing some of its arguments and leaving other implicit. Variables whose instantiation is not provided are turned into existentential variables, and facts whose instantiation is not provided are turned into subgoals.

Remark: this instantion 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 Coq 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 & $Hctx$ & $Hsub$).

35.6.1 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 prettly 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* I: $E0$ $E1$ .. EN, for building an hypothesis named I 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.
  Require Import Sub.

Axiom typing_inversion_var : ∀ (G:context) (x:id) (T:ty),
  has_type G (tvar x) T →
  ∃ S, G x = Some S ∧ subtype S T.
```

First, assume we have an assumption H with the type of the form **has_type** G (tvar 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 : ∀ (G:context) (x:id) (T:ty),
  has_type G (tvar x) T → 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 (tvar 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* : ∀ (*G*:context) (*x*:*id*) (*T*:*ty*), *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* (tvar 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* : ∀ (*x*:*id*), *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 writting *lets* *K*: *H* __ 3.

Lemma *demo_lets_underscore* :
  $(\forall\ n\ m,\ n \le m \to n < m{+}1) \to$ *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.*

35.6.2 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 instantion

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.

35.6.3 Example of Instantiations

```
Module ExamplesInstantiations.
  Require Import Sub.
```

The following proof shows several examples where *lets* is used instead of `destruct`, as well as examples where *applys* is used instead of `apply`. The proof also contains some holes that you need to fill in as an exercise.

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (update Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent S. generalize dependent Gamma.
  (induction t); intros; simpl.
  -
```

```
  rename i into y.
 lets (T&Hctx&Hsub): typing_inversion_var Htypt.
  unfold update, t_update in Hctx.
  destruct (beq_idP x y)...
  +
    subst.
    inversion Hctx; subst. clear Hctx.
    apply context_invariance with empty...
    intros x Hcontra.
      lets [T' HT']: free_in_context S (@empty ty) Hcontra...
      inversion HT'.
-

  admit.
-
  rename i into y. rename t into T1.
 lets (T2&Hsub&Htypt2): typing_inversion_abs Htypt.
 applys T_Sub (TArrow T1 T2)...
   apply T_Abs...
  destruct (beq_idP x y).
  +
    eapply context_invariance...
    subst.
    intros x Hafi. unfold update, t_update.
    destruct (beq_idP y x)...
  +
    apply IHt. eapply context_invariance...
    intros z Hafi. unfold update, t_update.
    destruct (beq_idP y z)...
    subst. rewrite false_beq_id...
-
  lets: typing_inversion_true Htypt...
-
  lets: typing_inversion_false Htypt...
-
  lets (Htyp1&Htyp2&Htyp3): typing_inversion_if Htypt...
-
```

lets: *typing_inversion_unit Htypt*...

Admitted.

`End` *ExamplesInstantiations*.

35.7 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.
- *cases* and *cases_if* help with case analysis.
- *splits*, *branch* and ∃ to deal with n-ary constructs.
- *asserts_rewrite*, *cuts_rewrite*, *substs* and *fequals* help working with equalities.
- *lets*, *forwards*, *specializes* and *applys* provide means of very conveniently instantiating lemmas.
- *applys_eq* can save the need to perform manual rewriting steps before being able to apply lemma.
- *skip*, *skip_rewrite* and *skip_goal* give the flexibility to choose which subgoals to try and discharge first.

Making use of these tactics can boost one's productivity in Coq proofs.

If you are interested in using *LibTactics.v* in your own developments, make sure you get the lastest version from: http://www.chargueraud.org/softs/tlc/.

$Date : 2017-01-3019:42:52-0500(Mon, 30Jan2017)$

Chapter 36

Library Top.UseAuto

36.1 UseAuto: Theory and Practice of Automation in Coq Proofs

In a machine-checked proof, every single detail has to be justified. This can result in huge proof scripts. Fortunately, Coq 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 working of automation.

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 fragment of the logic of Coq.

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.

```
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Import ListNotations.
```

```
Require Import Maps.
Require Import Smallstep.
Require Import Stlc.
Require Import LibTactics.
```

36.2 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.

36.2.1 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 Coq. 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.

36.2.2 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* : $\forall$ (P Q : *nat*→Prop),
  ($\forall$ n, Q n $\rightarrow$ P n) $\rightarrow$
  ($\forall$ n, Q n) $\rightarrow$
  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 : $\forall$ n : nat, Q n → P n) ($H0$: $\forall$ 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).

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 direction 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* : $\forall$ (P Q : *nat*→Prop),
  ($\forall$ n m, Q m $\rightarrow$ P n) $\rightarrow$
  Q 1 $\rightarrow$ P 2.
Proof. auto. eauto. Qed.

36.2.3 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* : $\forall$ (P : *nat*→Prop) (F : Prop),
  ($\forall$ n, P n) $\rightarrow$ 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' : 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' : 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 : *nat*→Prop) (F : 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 Coq proof search mechanisms.

Lemma *solving_conj_hyp_forall* : $\forall$ (P Q : *nat*→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 : *nat*→Prop) (F : Prop),
  $(\forall\ n, P\ n) \wedge (\forall\ n, Q\ n) \rightarrow P\ 2$.
Proof. *jauto*. Qed.

36.2.4 Disjunctions

The tactics auto and eauto can handle disjunctions that occur in the goal.

Lemma *solving_disj_goal* : $\forall$ (F F' : Prop),
  $F \rightarrow F \vee F'$.

```
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 $F \lor F'$ entails $F' \lor F$.

```
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.

36.2.5 Existentials

The tactics `eauto`, *iauto*, and *jauto* can prove goals whose conclusion is an existential. For example, if the goal is ∃ 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 ∃ x, *P*.

```
Lemma solving_exists_hyp : ∀ (f g : nat→Prop),
  (∀ x, f x → g x) →
  (∃ a, f a) →
  (∃ a, g a).
Proof.
  auto. eauto. iauto.   jauto. Qed.
```

36.2.6 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 $P \to$ **False** but that the unfolding of this definition is not performed automatically. Consider the following example.

Lemma *negation_study_1* : $\forall$ (P : *nat*$\to$Prop),
  P 0 $\to$ ($\forall$ x, $\neg$ P x) $\to$ *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 : *nat*$\to$Prop),
  P 0 $\to$ ($\forall$ x, $\neg$ P x) $\to$ *False*.
Proof. *jauto*. Qed.

We will come back later on to the behavior of proof search with respect to the unfolding of definitions.

36.2.7 Equalities

Coq'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 : *nat*$\to$Prop),
  ($\forall$ x, f x = g x) $\to$ 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.

36.3 How Proof Search Works

36.3.1 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 and error. This application produces subgoals. There are two possible cases. If the sugboals 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 ∧ True ∧ True ∧ True ∧ True ∧ 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 sugoals. 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 $F \wedge F$'. 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.
```

36.3.2 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 command that one can use for tracing all the steps that proof-search considers. To view such a trace, one should write `debug eauto`. (For some reason, the command `debug auto` does not exist, so we have to use the command `debug eauto` instead.)

```
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. eauto. Qed.
```

The output message produced by `debug eauto` is as follows.

depth=5 depth=4 apply H2 depth=3 apply H2 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* : $\forall$ (P : *nat*$\rightarrow$`Prop`),
  (P 0) $\rightarrow$
  ($\forall$ k, P (k+1) $\rightarrow$ P k) $\rightarrow$
  ($\forall$ k, P (k-1) $\rightarrow$ P k) $\rightarrow$
  (P 2).
`Proof`. `intros` *P H1 H3 H2*. `eauto`. `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.

In 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
        - -$>$ 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*.

36.3.3 Adding Hints

By default, `auto` (and `eauto`) only tries to apply the hypotheses that appear in the proof context. There are two possibilities for telling `auto` to exploit a lemma that have been proved previously: either adding the lemma as an assumption just before calling `auto`, or adding the lemma as a hint, so that it can be used by every calls to `auto`.

The first possibility is useful to have `auto` exploit a lemma that only serves at this particular point. To add the lemma as hypothesis, one can type `generalize` *mylemma*; `intros`, or simply *lets*: *mylemma* (the latter requires *LibTactics.v*).

The second possibility is useful for lemmas that need to be exploited several times. The syntax for adding a lemma as a hint is `Hint Resolve` *mylemma*. For example, the lemma asserting than any number is less than or equal to itself, $\forall$ x, x $\leq$ x, called *Le.le_refl* in the Coq standard library, can be added as a hint as follows.

`Hint Resolve` *Le.le_refl*.

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.

36.3.4 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 Coq 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*$\times$ I: *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 *applys* *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 the succeed, they save a lot of time, and when they fail to prove the goal, they fail very quickly.".

36.4 Examples of Use of 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.

36.4.1 Determinism

```
Module DeterministicImp.
  Require Import Imp.
```

Recall the original proof of the determinism lemma for the IMP language, shown below.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  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': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  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,
- stop using *ceval_cases* to label subcases.

```
Theorem ceval_deterministic'': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  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 invokation 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 Coq 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,
  c / st \\ st1 →
  c / st \\ 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 only four lines, including no more than 10 tactics.

```
Theorem ceval_deterministic'''': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ 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.
```

36.4.2 Preservation for STLC

```
Module PreservationProgressStlc.
  Require 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 : ∀ t t' T,
  has_type empty t T →
  t ==> t' →
  has_type empty t' T.
Proof with eauto.
  remember (@empty ty) as Gamma.
  intros t t' T HT. generalize dependent t'.
  (induction HT); intros t' HE; subst Gamma.
  -
    inversion HE.
  -
    inversion HE.
  -
    inversion HE; subst...
    +
      apply substitution_preserves_typing with T11...
      inversion HT1...
  -
    inversion HE.
  -
    inversion HE.
  -
    inversion HE; subst...
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*× and *applys*× to call auto× after a call to *inverts* or to *applys*. The solution is three lines long.

```
Theorem preservation' : ∀ t t' T,
  has_type empty t T →
  t ==> t' →
  has_type empty t' T.
Proof.
   admit.
Admitted.
```

36.4.3 Progress for STLC

Consider the proof of the progress theorem.

```
Theorem progress : ∀ t T,
  has_type empty t T →
  value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  (induction Ht); subst Gamma...
  -
    inversion H.
  -
    right. destruct IHHt1...
    +
      destruct IHHt2...
      ×
        inversion H; subst; try solve_by_invert.
        ∃ ([x0:=t2]t)...
      ×
       destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    +
      destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
  -
    right. destruct IHHt1...
    destruct t1; try solve_by_invert...
    inversion H. ∃ (tif x0 t2 t3)...
Qed.
```

Exercise: optimize the above proof. Hint: make use of destruct× and *inverts*×. The solution consists of 10 short lines.

```
Theorem progress' : ∀ t T,
```

```
  has_type empty t T →
  value t ∨ ∃ t', t ==> t'.
Proof.
    admit.
Admitted.

End PreservationProgressStlc.
```

36.4.4 BigStep and SmallStep

```
Module Semantics.
Require Import Smallstep.
```

Consider the proof relating a small-step reduction judgment to a big-step reduction judgment.

```
Theorem multistep__eval : ∀ t v,
  normal_form_of 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_Const.
  -
    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 is 3 lines long.

```
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 is 2 lines long.

```
Theorem multistep__eval' : ∀ t v,
```

$normal\_form\_of\ t\ v \rightarrow \exists\ n,\ v = C\ n \land t \setminus\setminus n.$

```
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 looses information when applied to a property whose arguments are not reduced to variables, such as t ==>* (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 shown below.

```
Require Import Program.
```

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 is 5 lines long.

```
Theorem multistep__eval'' : ∀ t v,
  normal_form_of t v → ∃ n, v = C n ∧ t \\ n.
Proof.
    admit.
Admitted.

End Semantics.
```

36.4.5 Preservation for STLCRef

```
Module PreservationProgressReferences.
  Require Import Coq.omega.Omega.
  Require Import References.
  Import STLCRef.
  Hint Resolve store_weakening extends_refl.
```

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',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     has_type empty ST' t' T ∧
     store_well_typed ST' st').
Proof.
```

remember (@*empty ty*) `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*. `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 1.
`rewrite` *app_length*. `simpl`. `omega`.
`unfold` *store_Tlookup*. `rewrite` $\leftarrow$ *H*. `rewrite`$\times$ *app_nth2*.
`rewrite` *minus_diag*. `simpl`. `reflexivity`.
`apply`$\times$ *store_well_typed_app*.

- *forwards**: *IHHt*.

-
+

$\exists$ *ST*. *splits*$\times$.

lets [_ *Hsty*]: *HST*.
applys_eq× *Hsty* 1.
inverts× *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 occuring in *snoc* l x. 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 (*snoc* l x) d = x, with a premise asserting n = length l.

```
Lemma nth_eq_last' : ∀ (A : Type) (l : list A) (x d : A) (n : nat),
  n = length l → nth n (l ++ x::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 : ∀ (st:store) (ST : store_ty) T1,
  length ST = length st →
  TRef T1 = TRef (store_Tlookup (length st) (ST ++ T1::nil)).
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' : ∀ ST t t' T st st',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
```

```
    (extends ST' ST ∧
     has_type empty ST' t' T ∧
     store_well_typed ST' st').
Proof.
  remember (@empty ty) as Gamma. introv Ht. gen t'.
  induction Ht; introv HST Hstep; subst Gamma; inverts Hstep; eauto.
  - ∃ ST. inverts Ht1. splits×. applys× substitution_preserves_typing.
  - forwards*: IHHt1.
  - forwards*: IHHt2.
  - forwards*: IHHt.
  - forwards*: IHHt.
  - forwards*: IHHt1.
  - forwards*: IHHt2.
  - forwards*: IHHt1.
  - ∃ (ST ++ T1::nil). inverts keep HST. splits.
    apply extends_app.
    applys_eq T_Loc 1.
      rewrite app_length. simpl. omega.
      unfold store_Tlookup. rewrite× nth_eq_last'.
    apply× store_well_typed_app.
  - forwards*: IHHt.
  - ∃ ST. splits×. lets [_ Hsty]: HST.
    applys_eq× Hsty 1. inverts× Ht.
  - forwards*: IHHt.
  - ∃ ST. splits×. applys× assign_pres_store_typing. inverts× Ht1.
  - forwards*: IHHt1.
  - forwards*: IHHt2.
Qed.
```

36.4.6 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 : ∀ ST t T st,
  has_type empty ST t T →
  store_well_typed ST st →
  (value t ∨ ∃ t', ∃ st', t / st ==> t' / st').
Proof.
  introv Ht HST. remember (@empty ty) as Gamma.
  induction Ht; subst Gamma; tryfalse; try solve [left*].
  - right. destruct× IHHt1 as [K||].
    inverts K; inverts Ht1.
```

```
      destruct× IHHt2.
  - right. destruct× 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.
```

36.4.7 Subtyping

```
Module SubtypingInversion.
  Require Import Sub.
```

Consider the inversion lemma for typing judgment of abstractions in a type system with subtyping.

```
Lemma abs_arrow : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
      subtype T1 S1
  ∧ has_type (update empty x S1) s2 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.
```

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` I with *lets* I: K H. The solution is 4 lines.

```
Lemma abs_arrow' : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
     subtype T1 S1
  ∧ has_type (update empty x S1) s2 T2.
Proof.
   admit.
Admitted.
```

The lemma substitution_preserves_typing has already been used to illustrate the working of *lets* and *applys* in chapter UseTactics. Optimize further this proof using automation (with the star symbol), and using the tactic *cases_if'*. The solution is 33 lines).

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
  has_type (update Gamma x U) t S →
  has_type empty v U →
  has_type Gamma ([x:=v]t) S.
Proof.
   admit.
Admitted.

End SubtypingInversion.
```

36.5 Advanced Topics in Proof Search

36.5.1 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 ∀ n m, P m → m ≠ 0 → P n, then `eauto` works. However, with ∀ n m, m ≠ 0 → P m → P n, the tactic `eauto` fails.

```
Lemma order_matters_1 : ∀ (P : nat→Prop),
  (∀ n m, P m → m ≠ 0 → P n) → P 2 → 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.

36.5.2 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 : nat → Prop.

Definition myFact := ∀ x, x ≤ 3 → 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 :
  (∀ x, P x) → 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 Coq that it should always try to unfold myFact when myFact appears in the goal.

```
Hint Unfold myFact.
```

This time, automation is able to see through the definition of myFact.

```
Lemma demo_hint_unfold_goal_2 :
  (∀ x, P x) → 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** → myFact.

```
Lemma demo_hint_unfold_context_1 :
  (True → myFact) → P 3.
Proof.
  intros.
  auto.   unfold myFact in *. auto. Qed.
```

There is actually one exception to the previous rule: a constant occuring 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.
```

36.5.3 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 ≤ 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*.

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*.

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 $H1 \rightarrow H2 \rightarrow$ **False** is a much more effective hint than $H1 \rightarrow \neg\ H2$, even though the two statments 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.

Qed.

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).

36.5.4 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*.

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*.

Now, consider the goal ∀ S T, *subtype* S T, which clearly has no hope of being solved. Let's call eauto on this goal.

Lemma *transitivity_bad_hint_1* : ∀ *S T*,
  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 *subtype* S T. Since reflexivity does not apply, `eauto` invokes transitivity, which produces two subgoals, *subtype* S ?X and *subtype* ?X T. Solving the first subgoal, *subtype* S ?X, is straightforward, it suffices to apply reflexivity. This unifies ?X with S. So, the second sugoal, *subtype* ?X T, becomes *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 Coq to try and apply the transitivity lemma on a goal of the form *subtype* S U only when the proof context already contains an assumption either of the form *subtype* S T or of the form *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 (subtype ?S ?U) =>
  match goal with
  | H: subtype S ?T |- _ => apply (@subtype_trans S T U)
  | H: subtype ?T U |- _ => apply (@subtype_trans S T U)
  end.
```

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 *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 indices 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.
```

36.6 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 `omega` 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.

36.6.1 Omega

The tactic `omega` supports natural numbers (type `nat`) as well as integers (type Z, available by including the module ZArith). It supports addition, substraction, equalities and inequalities.

Before using omega, one needs to import the module Omega, as follows.

```
Require Import Omega.
```

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 omega_demo_1 : ∀ (x y : nat),
  (y ≤ 4) → (x + x + 1 ≤ y) → (x ≠ 0) → (x = 1).
Proof. intros. omega. 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 omega_demo_2 : ∀ (x y z : nat),
  (x + y = z + z) → (x - y ≤ 4) → (x - z ≤ 2).
Proof. intros. omega. Qed.
```

One can proof **False** using omega 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 omega_demo_3 : ∀ (x y : nat),
  (x + 5 ≤ y) → (y - x < 3) → False.
Proof. intros. omega. Qed.
```

Note: omega can prove a goal by contradiction only if its conclusion reduces to **False**. The tactic omega always fails when the conclusion is an arbitrary proposition P, even though **False** implies any proposition P (by ex_falso_quodlibet).

```
Lemma omega_demo_4 : ∀ (x y : nat) (P : Prop),
  (x + 5 ≤ y) → (y - x < 3) → P.
Proof.
  intros.
  false. omega.
Qed.
```

36.6.2 Ring

Compared with omega, 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.

```
Module RingDemo.
  Require Import ZArith.
  Open Scope Z_scope.

Lemma ring_demo : ∀ (x y z : Z),
    x × (y + z) - z × 3 × x
```

$= x \times y - 2 \times x \times z.$
`Proof. intros. ring. Qed.`

`End` *RingDemo*.

36.6.3 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 :
   ∀ (f : nat→nat→nat) (g h : nat→nat) (x y z : nat),
   f (g x) (g y) = z →
   2 = g x →
   g y = h z →
   f 2 (h z) = z.
Proof. intros. congruence. Qed.
```

Moreover, `congruence` is able to exploit universally quantified equalities, for example ∀ `a`, `g a` = *h* `a`.

```
Lemma congruence_demo_2 :
   ∀ (f : nat→nat→nat) (g h : nat→nat) (x y z : nat),
   (∀ a, g a = h a) →
   f (g x) (g y) = z →
   g x = 2 →
   f 2 (h y) = z.
Proof. congruence. Qed.
```

Next is an example where `congruence` is very useful.

```
Lemma congruence_demo_4 : ∀ (f g : nat→nat),
  (∀ a, f a = g a) →
  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.

36.7 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.
- *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.
- `omega` 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.

$Date: 2016-07-1312:41:41-0400(Wed, 13Jul2016)$

Chapter 37

Library Top.PE

37.1 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 ::= APlus (ANum 3) (ANum 1) to the command Y ::= ANum 4. However, it does not propagate known constants along data flow. For example, it does not simplify the sequence

X ::= ANum 3;; Y ::= APlus (AId X) (ANum 1)

to

X ::= ANum 3;; Y ::= ANum 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 ::= ANum 3;; Y ::= AMinus (APlus (AId X) (ANum 1)) (AId Y)

to

X ::= ANum 3;; Y ::= AMinus (ANum 4) (AId Y)

without knowing the initial value of Y.

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Logic.FunctionalExtensionality.
Require Import Coq.Lists.List.
Import ListNotations.

Require Import Maps.
```

```
Require Import Imp.
Require Import Smallstep.
```

37.2 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.

37.2.1 Partial States

Conceptually speaking, we can think of such partial states as the type **id** → **option** nat (as opposed to the type **id** → 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 **id** × nat pairs.

```
Definition pe_state := list (id × nat).
```

The idea is that a variable **id** 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 **id** 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:id) : option nat :=
  match pe_st with
  | [] ⇒ None
  | (V',n')::pe_st ⇒ if beq_id 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 **id** to None.

```
Definition empty_pe_state : pe_state := [].
```

More generally, if the **list** representing a pe_state does not contain some **id**, then that pe_state must map that **id** to None. Before we prove this fact, we first define a useful tactic for reasoning with **id** equality. The tactic

compare V V'

means to reason by cases over beq_id 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 (beq_idP i j);
```

```
  [ subst j | ].
Theorem pe_domain: ∀ pe_st V n,
  pe_lookup pe_st V = Some n →
  In V (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 *beq* for testing equality of its elements, we can compute a boolean inb *beq* a l for testing whether In a l holds or not.

```
Fixpoint inb {A : Type} (beq : A → A → bool) (a : A) (l : list A) :=
  match l with
  | [] ⇒ false
  | a'::l' ⇒ beq a a' || inb beq a l'
  end.
```

It is easy to relate inb to In with the **reflect** property:

```
Lemma inbP : ∀ A : Type, ∀ beq : A→A→bool,
  (∀ a1 a2, reflect (a1 = a2) (beq a1 a2)) →
  ∀ a l, reflect (In a l) (inb beq a l).
Proof.
  intros A beq 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.
```

37.2.2 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
  | AId i ⇒ match pe_lookup pe_st i with
             | Some n ⇒ ANum n
             | None ⇒ AId i
             end
  | APlus a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
      | (a1', a2') ⇒ APlus a1' a2'
      end
  | AMinus a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
      | (a1', a2') ⇒ AMinus a1' a2'
      end
  | AMult a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 × n2)
      | (a1', a2') ⇒ AMult 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)] (APlus (APlus (AId X) (ANum 1)) (AId Y))
  = APlus (ANum 4) (AId Y).
Proof. reflexivity. Qed.

Example text_pe_aexp2:
  pe_aexp [(Y,3)] (APlus (APlus (AId X) (ANum 1)) (AId Y))
  = APlus (APlus (AId X) (ANum 1)) (ANum 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 i) 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 ::= ANum 3;; Y ::= AMinus (AId X) (AId Y);; X ::= ANum 4

to just

Y ::= AMinus (ANum 3) (AId Y);; X ::= ANum 4

by delaying the assignment to X until the end. To accomplish this simplification, we need the result of partial evaluating

pe_aexp (X,3) (AMinus (AId X) (AId Y))

to be equal to AMinus (ANum 3) (AId Y) and *not* the original expression AMinus (AId X) (AId Y). After all, it would be incorrect, not just inefficient, to transform

X ::= ANum 3;; Y ::= AMinus (AId X) (AId Y);; X ::= ANum 4

to

Y ::= AMinus (AId X) (AId Y);; X ::= ANum 4

even though the output expressions AMinus (ANum 3) (AId Y) and AMinus (AId X) (AId Y) both satisfy the correctness criterion that we just proved. Indeed, if we were to just define pe_aexp *pe_st* a = a then the theorem *pe_aexp_correct'* 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 (aeval *st* (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 (t_update empty_state Y 1) [(X,3);(Z,2)]
  = t_update (t_update (t_update empty_state Y 1) Z 2) X 3.
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 pe_update_correct: ∀ st pe_st V0,
  pe_update st pe_st V0 =
  match pe_lookup pe_st V0 with
  | Some n ⇒ n
  | None ⇒ st V0
  end.
Proof. intros. induction pe_st as [| [V n] pe_st]. reflexivity.
  simpl in *. unfold t_update.
  compare V0 V; auto. rewrite ← beq_id_refl; auto. rewrite false_beq_id; 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: ∀ 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: ∀ st pe_st,
  pe_consistent st pe_st → ∀ V, st V = 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: ∀ (pe_st:pe_state) (a:aexp) (st:state),
  aeval (pe_update st pe_st) a = aeval st (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 i); reflexivity.
```

Qed.

37.2.3 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
  | BTrue ⇒ BTrue
  | BFalse ⇒ BFalse
  | BEq a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ if beq_nat n1 n2 then BTrue else BFalse
      | (a1', a2') ⇒ BEq a1' a2'
      end
  | BLe a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ if leb n1 n2 then BTrue else BFalse
      | (a1', a2') ⇒ BLe a1' a2'
      end
  | BNot b1 ⇒
      match (pe_bexp pe_st b1) with
      | BTrue ⇒ BFalse
      | BFalse ⇒ BTrue
      | b1' ⇒ BNot b1'
      end
  | BAnd b1 b2 ⇒
      match (pe_bexp pe_st b1, pe_bexp pe_st b2) with
      | (BTrue, BTrue) ⇒ BTrue
      | (BTrue, BFalse) ⇒ BFalse
      | (BFalse, BTrue) ⇒ BFalse
      | (BFalse, BFalse) ⇒ BFalse
      | (b1', b2') ⇒ BAnd b1' b2'
      end
  end.

Example test_pe_bexp1:
  pe_bexp [(X,3)] (BNot (BLe (AId X) (ANum 3)))
  = BFalse.
Proof. reflexivity. Qed.

Example test_pe_bexp2: ∀ b,
  b = BNot (BLe (AId X) (APlus (AId X) (ANum 1))) →
  pe_bexp [] b = b.
```

```
Proof. intros b H. rewrite → H. reflexivity. Qed.
```

The correctness of pe_bexp is analogous to the correctness of pe_aexp above.

```
Theorem pe_bexp_correct: ∀ (pe_st:pe_state) (b:bexp) (st: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 a) as a';
         remember (pe_aexp pe_st a0) as a0';
         assert (Ha: aeval (pe_update st pe_st) a = aeval st a');
         assert (Ha0: aeval (pe_update st pe_st) a0 = aeval st a0');
           try (subst; apply pe_aexp_correct);
         destruct a'; destruct a0'; rewrite Ha; rewrite Ha0;
         simpl; try destruct (beq_nat n n0);
         try destruct (leb 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.
```

37.3 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

(X ::= ANum 3 ;; Y ::= AMult (AId Z) (APlus (AId X) (AId X))) / ☐ \\ (Y ::= AMult (AId Z) (ANum 6)) / (X,3)

to hold. The assignment to X appears in the final partial state, not the residual command.

37.3.1 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 ::= ANum 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 ::= AMult (AId Z) (APlus (AId X) (AId 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:id) : pe_state :=
  match pe_st with
  | [] ⇒ []
  | (V',n')::pe_st ⇒ if beq_id 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 beq_id 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 (beq_id V V0); reflexivity.
  - simpl. compare V V'.
    + rewrite IHpe_st.
      destruct (beq_idP V V0). reflexivity.
      rewrite false_beq_id; auto.
    + simpl. compare V0 V'.
      × rewrite false_beq_id; auto.
      × rewrite IHpe_st. reflexivity.
Qed.
```

```
Definition pe_add (pe_st:pe_state) (V:id) (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 beq_id 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 ← beq_id_refl; auto.
  - rewrite pe_remove_correct.
    repeat rewrite false_beq_id; 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 (beq_id 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 (beq_id V V0); reflexivity. Qed.
```

37.3.2 Conditional

Trickier than assignments to partially evaluate is the conditional, *IFB b1 THEN* c1 *ELSE* c2 *FI*. 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 ::= ANum 3;; IFB BLe (AId Y) (ANum 4) THEN Y ::= ANum 4;; IFB BEq (AId X) (AId Y) THEN Y ::= ANum 999 ELSE SKIP FI ELSE SKIP FI

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 ::= ANum 4 to the end of the *THEN* branch. So, the residual program will be something like

SKIP;; IFB BLe (AId Y) (ANum 4) THEN SKIP;; SKIP;; Y ::= ANum 4 ELSE SKIP FI

Programming this case in Coq 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 to 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:id) : bool :=
  match pe_lookup pe_st1 V, pe_lookup pe_st2 V with
  | Some x, Some y ⇒ negb (beq_nat x y)
  | None, None ⇒ false
  | _, _ ⇒ true
  end.

Theorem pe_disagree_domain: ∀ (pe_st1 pe_st2 : pe_state) (V:id),
  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 id) : list id :=
  match l with
  | [] ⇒ []
  | x::l ⇒
      x :: filter (fun y ⇒ if beq_id x y then false else true) (pe_unique l)
  end.

Theorem pe_unique_correct: ∀ l x,
  In x l ↔ In x (pe_unique l).
Proof. intros l x. induction l as [| h t]. reflexivity.
```

```
  simpl in *. split.
  -
    intros. inversion H; clear H.
      left. assumption.
      destruct (beq_idP h x).
         left. assumption.
         right. apply filter_In. split.
           apply IHt. assumption.
           rewrite false_beq_id; 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 id :=
  pe_unique (filter (pe_disagree_at pe_st1 pe_st2)
    (map (@fst _ _) pe_st1 ++ map (@fst _ _) pe_st2)).

Theorem pe_compare_correct: ∀ pe_st1 pe_st2 V,
  pe_lookup pe_st1 V = pe_lookup pe_st2 V ↔
  ¬ In V (pe_compare pe_st1 pe_st2).
Proof. intros pe_st1 pe_st2 V.
  unfold pe_compare. rewrite ← 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 ← beq_nat_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 beq_nat_true 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 id) : 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 beq_id 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 ← beq_id_refl. reflexivity.
  - rewrite false_beq_id; 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 _ _ beq_idP V (pe_compare pe_st1 pe_st2)).
  - reflexivity.
  - apply pe_compare_correct. auto. Qed.

Theorem pe_compare_update: ∀ pe_st1 pe_st2 st,
  pe_update st (pe_removes pe_st1 (pe_compare pe_st1 pe_st2)) =
  pe_update st (pe_removes pe_st2 (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 id) : com :=
  match ids with
  | [] ⇒ SKIP
```

```
  | V::ids ⇒ match pe_lookup pe_st V with
              | Some n ⇒ (assign pe_st ids;; V ::= ANum n)
              | None ⇒ 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 id) (st:state) : state :=
  fun V ⇒ if inb beq_id V ids then
              match pe_lookup pe_st V with
              | Some n ⇒ n
              | None ⇒ st V
              end
           else st V.

Theorem assign_removes: ∀ pe_st ids st,
  pe_update st pe_st =
  pe_update (assigned pe_st ids st) (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 _ _ beq_idP V ids); destruct (pe_lookup pe_st V); reflexivity.
Qed.

Lemma ceval_extensionality: ∀ c st st1 st2,
  c / st \\ st1 → (∀ V, st1 V = st2 V) → c / st \\ 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,
  assign pe_st ids / st \\ 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_Ass. simpl. reflexivity.
      intros V0. unfold t_update. compare V V0.
      × rewrite ← Heqlookup. rewrite ← beq_id_refl. reflexivity.
      × rewrite false_beq_id; simpl; congruence.
    + eapply ceval_extensionality. apply IHids.
      unfold assigned. intros V0. simpl. compare V V0.
      × rewrite ← Heqlookup.
```

```
      rewrite ← beq_id_refl.
      destruct (inbP _ _ beq_idP V ids); reflexivity.
    × rewrite false_beq_id; simpl; congruence.
Qed.
```

37.3.3 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_AssDynamic 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_AssStatic : ∀ pe_st a1 n1 l,
      pe_aexp pe_st a1 = ANum n1 →
      (l ::= a1) / pe_st \\ SKIP / pe_add pe_st l n1
  | PE_AssDynamic : ∀ pe_st a1 a1' l,
      pe_aexp pe_st a1 = a1' →
      (∀ n, a1' ≠ ANum 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 : ∀ pe_st pe_st' b1 c1 c2 c1',
      pe_bexp pe_st b1 = BTrue →
      c1 / pe_st \\ c1' / pe_st' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c1' / pe_st'
  | PE_IfFalse : ∀ pe_st pe_st' b1 c1 c2 c2',
      pe_bexp pe_st b1 = BFalse →
      c2 / pe_st \\ c2' / pe_st' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c2' / pe_st'
  | PE_If : ∀ pe_st pe_st1 pe_st2 b1 c1 c2 c1' c2',
      pe_bexp pe_st b1 ≠ BTrue →
      pe_bexp pe_st b1 ≠ BFalse →
      c1 / pe_st \\ c1' / pe_st1 →
      c2 / pe_st \\ c2' / pe_st2 →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st
        \\ (IFB 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) FI)
      / pe_removes pe_st1 (pe_compare pe_st1 pe_st2)

  where "c1 '/' st '\\' c1' '/' st'" := (pe_com c1 st c1' st').
Hint Constructors pe_com.
Hint Constructors ceval.
```

37.3.4 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 Coq. That is not hard to do, but it is not needed here.

```
Example pe_example1:
  (X ::= ANum 3 ;; Y ::= AMult (AId Z) (APlus (AId X) (AId X)))
  / [] \\ (SKIP;; Y ::= AMult (AId Z) (ANum 6)) / [(X,3)].
Proof. eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  eapply PE_AssDynamic. reflexivity. intros n H. inversion H. Qed.

Example pe_example2:
  (X ::= ANum 3 ;; IFB BLe (AId X) (ANum 4) THEN X ::= ANum 4 ELSE SKIP FI)
  / [] \\ (SKIP;; SKIP) / [(X,4)].
Proof. eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  eapply PE_IfTrue. reflexivity.
  eapply PE_AssStatic. reflexivity. Qed.

Example pe_example3:
  (X ::= ANum 3;;
   IFB BLe (AId Y) (ANum 4) THEN
      Y ::= ANum 4;;
      IFB BEq (AId X) (AId Y) THEN Y ::= ANum 999 ELSE SKIP FI
   ELSE SKIP FI) / []
  \\ (SKIP;;
       IFB BLe (AId Y) (ANum 4) THEN
          (SKIP;; SKIP);; (SKIP;; Y ::= ANum 4)
       ELSE SKIP;; SKIP FI)
      / [(X,3)].
Proof. erewrite f_equal2 with (f := fun c st => _ / _ \\ c / st).
  eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  eapply PE_If; intuition eauto; try solve_by_invert.
  econstructor. eapply PE_AssStatic. reflexivity.
  eapply PE_IfFalse. reflexivity. econstructor.
  reflexivity. reflexivity. Qed.
```

37.3.5 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',
    c' / st \\ 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'').

Hint Constructors pe_ceval.

Theorem pe_com_complete:
  ∀ c pe_st pe_st' c', c / pe_st \\ c' / pe_st' →
  ∀ st st'',
  (c / pe_update st pe_st \\ 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'') →
  (c / pe_update st pe_st \\ 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_Ass.
    rewrite → pe_aexp_correct. rewrite → H. reflexivity.
  - rewrite ← pe_update_update_remove. apply E_Ass.
    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_IfFalse. 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'',
  (c / pe_update st pe_st \\ 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.
```

37.4 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 BLe (ANum 1) (AId X) DO Y ::= AMult (AId Y) (AId Y);; X ::= AMinus (AId X) (ANum 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 ::= AMult (AId Y) (AId Y);; Y ::= AMult (AId Y) (AId Y);; Y ::= AMult (AId Y) (AId 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 ::= ANum 0;; WHILE BLe (ANum 1) (AId Y) DO Y ::= AMinus (AId Y) (ANum 1);; X ::= AMinus (ANum 1) (AId 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 BLe (ANum 1) (AId Y) DO Y ::= AMinus (AId Y) (ANum 1);; IF BLe (ANum 1) (AId Y) THEN Y ::= AMinus (AId Y) (ANum 1) ELSE X ::= ANum 1;; EXIT FI END;; X ::= ANum 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 '/' st '\\' c1' '/' st' '/' 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 : ∀ pe_st,
      SKIP / pe_st \\ SKIP / pe_st / SKIP
  | PE_AssStatic : ∀ pe_st a1 n1 l,
      pe_aexp pe_st a1 = ANum n1 →
      (l ::= a1) / pe_st \\ SKIP / pe_add pe_st l n1 / SKIP
  | PE_AssDynamic : ∀ pe_st a1 a1' l,
      pe_aexp pe_st a1 = a1' →
      (∀ n, a1' ≠ ANum n) →
      (l ::= a1) / pe_st \\ (l ::= a1') / pe_remove pe_st l / SKIP
```

```
| PE_Seq : ∀ pe_st pe_st' pe_st'' c1 c2 c1' c2' c'',
    c1 / pe_st \\ c1' / pe_st' / SKIP →
    c2 / pe_st' \\ c2' / pe_st'' / c'' →
    (c1 ;; c2) / pe_st \\ (c1' ;; c2') / pe_st'' / c''
| PE_IfTrue : ∀ pe_st pe_st' b1 c1 c2 c1' c'',
    pe_bexp pe_st b1 = BTrue →
    c1 / pe_st \\ c1' / pe_st' / c'' →
    (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c1' / pe_st' / c''
| PE_IfFalse : ∀ pe_st pe_st' b1 c1 c2 c2' c'',
    pe_bexp pe_st b1 = BFalse →
    c2 / pe_st \\ c2' / pe_st' / c'' →
    (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c2' / pe_st' / c''
| PE_If : ∀ pe_st pe_st1 pe_st2 b1 c1 c2 c1' c2' c'',
    pe_bexp pe_st b1 ≠ BTrue →
    pe_bexp pe_st b1 ≠ BFalse →
    c1 / pe_st \\ c1' / pe_st1 / c'' →
    c2 / pe_st \\ c2' / pe_st2 / c'' →
    (IFB b1 THEN c1 ELSE c2 FI) / pe_st
      \\ (IFB 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) FI)
           / pe_removes pe_st1 (pe_compare pe_st1 pe_st2)
           / c''
| PE_WhileEnd : ∀ pe_st b1 c1,
    pe_bexp pe_st b1 = BFalse →
    (WHILE b1 DO c1 END) / pe_st \\ SKIP / pe_st / SKIP
| PE_WhileLoop : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2' c2'',
    pe_bexp pe_st b1 = BTrue →
    c1 / pe_st \\ c1' / pe_st' / SKIP →
    (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 ≠ BFalse →
    pe_bexp pe_st b1 ≠ BTrue →
    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
      \\ (IFB 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'') FI)
              / pe_removes pe_st (pe_compare pe_st pe_st'')
              / c2''
  | PE_WhileFixedEnd : ∀ pe_st b1 c1,
      pe_bexp pe_st b1 ≠ BFalse →
      (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 = BTrue →
      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 BTrue DO SKIP END) / pe_st / SKIP

  | PE_WhileFixed : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2',
      pe_bexp pe_st b1 ≠ BFalse →
      pe_bexp pe_st b1 ≠ BTrue →
      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'').
Hint Constructors pe_com.
```

37.4.1 Examples

```
Ltac step i :=
  (eapply i; intuition eauto; try solve_by_invert);
  repeat (try eapply PE_Seq;
          try (eapply PE_AssStatic; simpl; reflexivity);
          try (eapply PE_AssDynamic;
               [ simpl; reflexivity
               | intuition eauto; solve_by_invert])).
Definition square_loop: com :=
  WHILE BLe (ANum 1) (AId X) DO
    Y ::= AMult (AId Y) (AId Y);;
    X ::= AMinus (AId X) (ANum 1)
  END.
```

```
Example pe_loop_example1:
  square_loop / []
  \\ (WHILE BLe (ANum 1) (AId X) DO
         (Y ::= AMult (AId Y) (AId Y);;
          X ::= AMinus (AId X) (ANum 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 ::= ANum 3;; square_loop) / []
  \\ (SKIP;;
       (Y ::= AMult (AId Y) (AId Y);; SKIP);;
       (Y ::= AMult (AId Y) (AId Y);; SKIP);;
       (Y ::= AMult (AId Y) (AId Y);; SKIP);;
       SKIP) / [(X,0)] / SKIP.
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st / SKIP).
  eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  step PE_WhileLoop.
  step PE_WhileLoop.
  step PE_WhileLoop.
  step PE_WhileEnd.
  inversion H. inversion H. inversion H.
  reflexivity. reflexivity. Qed.

Example pe_loop_example3:
  (Z ::= ANum 3;; subtract_slowly) / []
  \\ (SKIP;;
       IFB BNot (BEq (AId X) (ANum 0)) THEN
         (SKIP;; X ::= AMinus (AId X) (ANum 1));;
         IFB BNot (BEq (AId X) (ANum 0)) THEN
           (SKIP;; X ::= AMinus (AId X) (ANum 1));;
           IFB BNot (BEq (AId X) (ANum 0)) THEN
             (SKIP;; X ::= AMinus (AId X) (ANum 1));;
             WHILE BNot (BEq (AId X) (ANum 0)) DO
               (SKIP;; X ::= AMinus (AId X) (ANum 1));; SKIP
             END;;
             SKIP;; Z ::= ANum 0
           ELSE SKIP;; Z ::= ANum 1 FI;; SKIP
         ELSE SKIP;; Z ::= ANum 2 FI;; SKIP
       ELSE SKIP;; Z ::= ANum 3 FI) / [] / SKIP.
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st / SKIP).
  eapply PE_Seq. eapply PE_AssStatic. 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 ::= ANum 0;;
   WHILE BLe (AId X) (ANum 2) DO
     X ::= AMinus (ANum 1) (AId X)
   END) / [] \\ (SKIP;; WHILE BTrue DO SKIP END) / [(X,0)] / SKIP.
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st / SKIP).
  eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  step PE_WhileFixedLoop.
  step PE_WhileLoop.
  step PE_WhileFixedEnd.
  inversion H. reflexivity. reflexivity. reflexivity. Qed.
```

37.4.2 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'Ass : ∀ 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 →
      (IFB b1 THEN c1 ELSE c2 FI) / st \\ st' # n
  | E'IfFalse : ∀ st st' b1 c1 c2 n,
      beval st b1 = false →
```

```
      c2 / st \\ st' # n →
      (IFB b1 THEN c1 ELSE c2 FI) / st \\ st' # n
  | E'WhileEnd : ∀ b1 st c1,
      beval st b1 = false →
      (WHILE b1 DO c1 END) / st \\ st # 0
  | E'WhileLoop : ∀ 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).

Hint Constructors ceval_count.

Theorem ceval_count_complete: ∀ c st st',
  c / st \\ 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 → c / st \\ 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: ∀ pe_st1 pe_st2,
  pe_compare pe_st1 pe_st2 = [] →
  ∀ 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 : ∀ st' n',
    c' / st \\ st' →
    c'' / pe_update st' pe_st' \\ st'' # n' →
    n' ≤ n →
    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).

Hint Constructors pe_ceval_count.

Lemma pe_ceval_count_le: ∀ c' pe_st' c'' st st'' n n',
  n' ≤ n →
  c' / pe_st' / c'' / st \\ st'' # n' →
  c' / pe_st' / c'' / st \\ st'' # n.
Proof. intros c' pe_st' c'' st st'' n n' Hle H. inversion H.
  econstructor; try eassumption. omega. Qed.

Theorem pe_com_complete:
  ∀ c pe_st pe_st' c' c'', c / pe_st \\ c' / pe_st' / c'' →
  ∀ st st'' n,
  (c / pe_update st pe_st \\ st'' # n) →
  (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 → pe_bexp_correct, → 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. omega.
  - 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. 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. eassumption.
  -
    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    econstructor; eauto. omega.
  - inversion Heval; subst.
    + econstructor. apply E_IfFalse.
      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_IfTrue.
      rewrite <- pe_bexp_correct. assumption.
      repeat eapply E_Seq; eauto. apply eval_assign.
      rewrite -> pe_compare_update, <- assign_removes. eassumption.
      omega.
  - 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; [| omega]. 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; [| omega]. inversion H8.
      econstructor; [ eapply E_WhileLoop; eauto | eassumption | omega].
Qed.

Theorem pe_com_sound:
  ∀ c pe_st pe_st' c' c'', c / pe_st \\ c' / pe_st' / c'' →
  ∀ st st'' n,
  (c' / pe_st' / c'' / st \\ st'' # n) →
  (c / pe_update st pe_st \\ st'').
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_Ass.
    rewrite → pe_aexp_correct. rewrite → H. reflexivity.
  - rewrite ← pe_update_update_remove. apply E_Ass.
    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_WhileEnd.
    rewrite → pe_bexp_correct. rewrite → H. reflexivity.
  - eapply E_WhileLoop.
```

```
    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_WhileLoop. 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_WhileEnd.
        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_WhileEnd.
      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_WhileLoop. rewrite pe_bexp_correct. assumption. eauto.
      eapply IHHpe2. 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'',
  (c / pe_update st pe_st \\ st'') ↔
```

```
  (∃ st', c' / st \\ 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.
```

37.5 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.

37.5.1 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 : id → 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 (AMinus (AId Y) (ANum 1))
   (Assign X (AMinus (ANum 1) (AId 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_state parity_body
  = (t_update (t_update empty_state Y 0) X 1, loop).
Proof. reflexivity. Qed.
```

37.5.2 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 → option (block L).

Definition parity : program parity_label := fun l ⇒
  match l with
  | entry ⇒ Some (Assign X (ANum 0) (Goto loop))
  | loop ⇒ Some (If (BLe (ANum 1) (AId 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 → L → state → L → Prop :=
  | E_None: ∀ st l,
```

```
    p l = None →
    peval p st l st l
  | E_Some: ∀ st l k st' l' st'' l'',
    p l = Some k →
    keval st k = (st', l') →
    peval p st' l' st'' l'' →
    peval p st l st'' l''.

Example parity_eval: peval parity empty_state entry empty_state 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.
```

37.5.3 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 (AMinus (AId Y) (ANum 1)) (Goto ([(X,1)], loop)).
Proof. reflexivity. Qed.
```

```
Theorem pe_block_correct: ∀ (L:Type) st pe_st k st' pe_st' (l':L),
  keval st (pe_block pe_st k) = (st', (pe_st', l')) →
  keval (pe_update st pe_st) k = (pe_update st' 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.
```

$Date : 2017-01-3019 : 42 : 52-0500(Mon, 30Jan2017)$

Chapter 38

Library Top.Postscript

38.1 Postscript

Congratulations: We've made it to the end!

38.2 Looking Back

We've covered a lot of ground. Here's a quick review...

- *Functional programming*:

    - "declarative" programming style (recursion over persistent 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
- *Coq*, 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

38.3 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 is written in Gallina and extracted to an efficient OCaml program using Coq'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 Coq.

"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: (1) what OS kernel structure can offer the best support for extensibility, security, and resilience? (2) which semantic models and program logics can best capture these abstractions? (3) what are the right programming languages and environments for developing such certified kernels? and (4) 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 Coq 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: http://deepspec.org/
- Overview presentations:
    - http://deepspec.org/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."

http://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 Coq
- the Princeton Verified Software Toolchain

38.4 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.
- Cutting-edge conferences on programming languages and formal verification:
  - Principles of Programming Langauges (POPL)
  - Programming Language Design and Implementation (PLDI)
  - SPLASH/OOPSLA
  - International Conference on Functional Programming (ICFP)

  - Computer Aided Verification (CAV)
  - Interactive Theorem Proving (ITP)
  - Principles in Practice workshop (PiP)
  - CoqPL workshop

- More on functional programming

  - Learn You a Haskell for Great Good, by Miran Lipovaca *Lipovaca* 2011.
  - Real World Haskell, by Bryan O'Sullivan, John Goerzen, and Don Stewart *O'Sullivan* 2008
  - ...and many other excellent books on Haskell, OCaml, Scheme, Racket, Scala, F sharp, etc., etc.

- More on Hoare logic and program verification

  - The Formal Semantics of Programming Languages: An Introduction, by Glynn Winskel *Winskel* 1993.
  - 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
  - Types and Programming Languages, by Benjamin C. Pierce *Pierce* 2002.
  - Practical Foundations for Programming Languages, by Robert Harper *Harper* 2016.
  - Foundations for Programming Languages, by John C. Mitchell *Mitchell* 1996.

- More on Coq:

  - Verified Functional Algorithms, by Andrew Appel *Chlipala* 2013.
  - Certified Programming with Dependent Types, by Adam Chlipala *Chlipala* 2013.
  - Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran *Bertot* 2004.
  - Iron Lambda (http://iron.ouroborus.net/) is a collection of Coq 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).

Chapter 39

Library Top.Bib

39.1 Bib: Bibliography

Appel 2017 Andrew Appel. Verified Functional Algorithms (volume 3 in the Software Foundations series). https://www.cs.princeton.edu/~appel/vfa/

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. http://www.cis.upenn.edu/~bcpierce/papers/binders.pdf

Bertot 2004 Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran. Springer-Verlag, 2004. http://tinyurl.com/z3o7nqu

Chlipala 2013 Certified Programming with Dependent Types, by Adam Chlipala. MIT Press. 2013. http://tinyurl.com/zqdnyg2

Harper 2015 Practical Foundations for Programming Languages, by Robert Harper. Cambridge University Press. Second edition, 2016. http://tinyurl.com/z82xwta

Lipovaca 2011 Learn You a Haskell for Great Good! A Beginner's Guide, by Miran Lipovaca, No Starch Press, April 2011. http://learnyouahaskell.com

Mitchell 1996 Foundations for Programming Languages, by John C. Mitchell. MIT Press, 1996. http://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. http://tinyurl.com/gtnudmu

O'Sullivan 2008 Bryan O'Sullivan, John Goerzen, and Don Stewart: Real world Haskell - code you can believe in. O'Reilly 2008. http://book.realworldhaskell.org

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. http://dl.acm.org/citation.cfm?id=125848

Wadler 2015 Philip Wadler. "Propositions as types." Communications of the ACM 58, no. 12 (2015): 75-84. http://dl.acm.org/citation.cfm?id=2699407

Winskel 1993 The Formal Semantics of Programming Languages: An Introduction, by Glynn Winskel. MIT Press, 1993. http://tinyurl.com/j2k6ev7