Contents

Chapter 1

Preface

1.1 Welcome

This is the entry point in a series of electronic textbooks on various aspects of *Software Foundations* – the mathematical underpinnings of reliable software. Topics in the series include basic concepts of logic, computer-assisted theorem proving, the Coq proof assistant, functional programming, operational semantics, logics for reasoning about programs, 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 series is that it is one hundred percent formalized and machine-checked: each text is literally a script for Coq. The books are 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 in each book 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.

This book, *Logical Foundations*, lays groundwork for the others, introducing the reader to the basic ideas of functional programming, constructive logic, and the Coq proof assistant.

1.2 Overview

Building reliable software is really hard. The scale and complexity of modern systems, the number of people involved, 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 recom-

mendations 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 *Software Foundations* series is focused on this last set of techniques.

The text is constructed around three 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.

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.

1.2.1 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 significantly 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 more deeply than you have probably done so far.

1.2.2 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.

1.2.3 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 lan-

guage 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*.

1.2.4 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.

1.3 Practicalities

1.3.1 Chapter Dependencies

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

1.3.2 System Requirements

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

- A current installation of Coq, available from the Coq home page. These files have been tested with Coq 8.8.1.
- 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.

    Users who like CoqIDE should consider running it with the "asynchronous" and "error resilience" modes disabled:

```
coqide -async-proofs off -async-proofs-command-error-resilience off Foo.v &
```

1.3.3 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 place. 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.

1.3.4 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 at http://softwarefoundations.cis.upenn.edu.

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; you should use this one instead of the public release version, so that you get any local updates during the semester.

1.4 Resources

1.4.1 Sample Exams

A large compendium of exams from many offerings of CIS500 ("Software Foundations") at the University of Pennsylvania can be found at https://www.seas.upenn.edu/~cis500/current/exams/index.htm There has been some drift of notations over the years, but most of the problems are still relevant to the current text.

1.4.2 Lecture Videos

Lectures for two intensive summer courses based on *Logical Foundations* (part of the DeepSpec summer school series) can be found at https://deepspec.org/event/dsss17 and https://deepspec.org/ev The video quality in the 2017 lectures is poor at the beginning but gets better in the later lectures.

1.5 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," and for Volume 4, "QuickChick: Property-Based Testing in Coq," the Author of Record is Benjamin C. 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) your github username.

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

1.6 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 also underway; you can preview it at https://coq-zh.github.io/SF-zh/.

1.7 Thanks

Development of the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep*

Specification.

Chapter 2

Basics: Functional Programming in Coq

2.1 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.

2.2 Data and Functions

2.2.1 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.

2.2.2 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
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

The type is called **day**, and its members are monday, tuesday, 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.

2.2.3 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`).

- It is fine to use additional definitions (of helper functions, useful lemmas, etc.) in your solutions. You can put these between the exercise header and the theorem you are asked to prove.

You will also notice that each chapter (like *Basics.v*) is accompanied by a *test script* (*BasicsTest.v*) that automatically calculates points for the finished homework problems in the chapter. These scripts are mostly for the auto-grading infrastructure that your instructor may use to help process assignments, but you may also like to use them to double-check that your file is well formatted before handing it in. In a terminal window either type *make BasicsTest.vo* or do the following:

coqc -Q . LF Basics.v coqc -Q . LF BasicsTest.v

There is no need to hand in *BasicsTest.v* itself (or *Preface.v*).

If your class is using the Canvas system to hand in assignments:

- If you submit multiple versions of the assignment, you may notice that they are given different names. This is fine: The most recent submission is the one that will be graded.
- To hand in multiple files at the same time (if more than one chapter is assigned in the same week), you need to make a single submission with all the files at once using the button "Add another file" just above the comment box.

2.2.4 Booleans

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

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

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 `Notation` command defines a new symbolic notation for an existing definition.

```
Notation "x && y" := (andb x y).
Notation "x || y" := (orb x y).

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, standard (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. (I.e., 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, standard (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.

□

2.2.5 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**."

2.2.6 New Types from Old

The types we have defined so far are examples of "enumerated types": their definitions explicitly enumerate a finite set of elements, each of which is just a bare constructor. Here is a more interesting type definition, where one of the constructors takes an argument:

```
Inductive rgb : Type :=
  | red
```

```
  | green
  | blue.

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

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

Every inductively defined type (**day**, **bool**, **rgb**, **color**, etc.) contains a set of *constructor expressions* built from *constructors* like `red`, primary, true, false, monday, etc.

The definitions of **rgb** and **color** say how expressions in the sets **rgb** and **color** can be built:

- `red`, green, and blue are the constructors of **rgb**;
- black, white, and primary are the constructors of **color**;
- the expression `red` belongs to the set **rgb**, as do the expressions green and blue;
- the expressions black and white belong to the set **color**;
- if p is an expression belonging to the set **rgb**, then primary p (pronounced "the constructor primary applied to the argument p") is an expression belonging to the set **color**; and
- expressions formed in these ways are the *only* ones belonging to the sets **rgb** and **color**.

We can define functions on colors using pattern matching just as we have done for **day** and **bool**.

```
Definition monochrome (c : color) : bool :=
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary q ⇒ false
  end.
```

Since the primary constructor takes an argument, a pattern matching primary should include either a variable (as above – note that we can choose its name freely) or a constant of appropriate type (as below).

```
Definition isred (c : color) : bool :=
  match c with
  | black ⇒ false
  | white ⇒ false
  | primary red ⇒ true
  | primary _ ⇒ false
```

```
  end.
```

The pattern primary _ here is shorthand for "primary applied to any **rgb** constructor except red." (The wildcard pattern _ has the same effect as the dummy pattern variable p in the definition of monochrome.)

2.2.7 Tuples

A single constructor with multiple parameters can be used to create a tuple type. As an example, consider representing the four bits in a nybble (half a byte). We first define a datatype **bit** that resembles **bool** (using the constructors B0 and B1 for the two possible bit values), and then define the datatype **nybble**, which is essentially a tuple of four bits.

```
Inductive bit : Type :=
  | B0
  | B1.

Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).

Check (bits B1 B0 B1 B0).
```

The bits constructor acts as a wrapper for its contents. Unwrapping can be done by pattern-matching, as in the all_zero function which tests a nybble to see if all its bits are O. Note that we are using underscore (_) as a *wildcard pattern* to avoid inventing variable names that will not be used.

```
Definition all_zero (nb : nybble) : bool :=
  match nb with
    | (bits B0 B0 B0 B0) ⇒ true
    | (bits _ _ _ _) ⇒ false
  end.

Compute (all_zero (bits B1 B0 B1 B0)).
Compute (all_zero (bits B0 B0 B0 B0)).
```

2.2.8 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.
```

2.2.9 Numbers

The types we have defined so far, "enumerated types" such as **day**, **bool**, and **bit**, and tuple types such as **nybble** built from them, share the property that each type has a finite set of values. The natural numbers are an infinite set, and we need to represent all of them in a datatype with a finite number of constructors. There are many representations of numbers to choose from. We are most familiar with decimal notation (base 10), using the digits 0 through 9, for example, to form the number 123. You may have encountered hexadecimal notation (base 16), in which the same number is represented as 7B, or octal (base 8), where it is 173, or binary (base 2), where it is 1111011. Using an enumerated type to represent digits, we could use any of these to represent natural numbers. There are circumstances where each of these choices can be useful.

Binary is valuable in computer hardware because it can in turn be represented with two voltage levels, resulting in simple circuitry. Analogously, we wish here to choose a representation that makes *proofs* simpler.

Indeed, there is a representation of numbers that is even simpler than binary, namely unary (base 1), in which only a single digit is used (as one might do while counting days in prison by scratching on the walls). To represent unary with a Coq datatype, we use two constructors. The capital-letter O constructor represents zero. When the S constructor is applied to the representation of the natural number n, the result is the representation of $n+1$. (S stands for "successor", or "scratch" if one is in prison.) Here is the complete datatype definition.

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

With this definition, 0 is represented by O, 1 by S O, 2 by S (S O), and so on.
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 can be put in front of a natural number to yield another one – if n is a natural number, then S n is too.

Again, let's look at this in a little more detail. 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**, **bool**, **color**, etc.

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
  | tick (foo : 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.
```

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 decimal numerals can be used as an alternative to the "unary" notation defined by the constructors S and O. Coq prints numbers in decimal form by default:

```
Check (S (S (S (S O)))).
```

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

```
Compute (minustwo 4).
```

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

```
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 decimal 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.

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, standard (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.
```

□

Again, we can make numerical expressions 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 "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 is a user-defined operation!

Here is a function eqb, which tests natural numbers for **eq**uality, yielding a boolean. Note the use of nested `match`es (we could also have used a simultaneous match, as we did in minus.)

```
Fixpoint eqb (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' ⇒ eqb n' m'
              end
  end.
```

Similarly, 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.
```

Since we'll be using these (especially eqb) a lot, let's give them infix notations.

```
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
Notation "x <=? y" := (leb x y) (at level 70) : nat_scope.

Example test_leb3': (4 <=? 2) = false.
Proof. simpl. reflexivity. Qed.
```

Exercise: 1 star, standard (ltb) The ltb function tests natural numbers for *l*ess-*t*han, yielding a boolean. Instead of making up a new `Fixpoint` for this one, define it in terms of a previously defined function. (It can be done with just one previously defined function, but you can use two if you need to.)

```
Definition ltb (n m : nat) : bool
  . Admitted.

Notation "x <? y" := (ltb x y) (at level 70) : nat_scope.

Example test_ltb1: (ltb 2 2) = false.
  Admitted.
Example test_ltb2: (ltb 2 4) = true.
  Admitted.
Example test_ltb3: (ltb 4 2) = false.
  Admitted.
```

□

2.3 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 mostly a matter of style; the keywords `Example` and `Theorem` (and a few others, including `Lemma`, `Fact`, and `Remark`) mean pretty much 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.

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

2.5 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. (We then use the `Abort` command to give up on it for the moment.)

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

The reason for this is that the definitions of both `eqb` 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 `eqb` 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 $(n + 1) =? 0$ 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, $(n + 1) =? 0$ 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,
  (n + 1) =? 0 = false.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  - 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.

In each subgoal, Coq remembers the assumption about n that is relevant for this subgoal – either $n = 0$ or n = S n' for some n'. The *eqn*:*E* annotation tells `destruct` to give the name E to this equation. (Leaving off the *eqn*:*E* annotation causes Coq to elide these assumptions in the subgoals. This slightly streamlines proofs where the assumptions are not explicitly used, but it is better practice to keep them for the sake of documentation, as they can help keep you oriented when working with the subgoals.)

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 second one simplifies (S n' + 1) =? 0 to false by first rewriting (S n' + 1) to S (n' + 1), then unfolding `eqb`, 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 eqn:E.
  - 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 eqn:Eb.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
  - destruct c eqn:Ec.
    + 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 eqn:Eb.
  { destruct c eqn:Ec.
    { reflexivity. }
    { reflexivity. } }
  { destruct c eqn:Ec.
    { 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 eqn:Eb.
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
    { destruct d eqn:Ed.
      - 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 eqn:E.

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. (You'll also note one downside of this shorthand: we lose the equation recording the assumption we are making in each subgoal, which we previously got from the *eqn*:*E* annotation.)

```
Theorem plus_1_neq_0' : ∀ n : 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, standard (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, standard (zero_nbeq_plus_1) `Theorem zero_nbeq_plus_1 : ∀ n : nat,`
`  0 =? (n + 1) = false.`
`Proof.`
Admitted.
□

2.5.1 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 $\times$ 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 $(x \times y)\%\textbf{nat}$, 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!

2.5.2 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, standard, 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. (If you choose to turn in this optional exercise as part of a homework assignment, make sure you comment out your solution so that it doesn't cause Coq to reject the whole file!)

2.6 More Exercises

Each SF chapter comes with a tester file (e.g. *BasicsTest.v*), containing scripts that check most of the exercises. You can run *make BasicsTest.vo* in a terminal and check its output to make sure you didn't miss anything.

Exercise: 1 star, standard (indentity_fn_applied_twice) 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.
```

□

Exercise: 1 star, standard (negation_fn_applied_twice) 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 $f\ x = \mathsf{negb}\ x$.

```
From Coq Require Export String.

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

□

Exercise: 3 stars, standard, optional (andb_eq_orb) Prove the following theorem. (Hint: This one can be a bit tricky, depending on how you approach it. You will probably need both `destruct` and `rewrite`, but destructing everything in sight is not the best way.)

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

□

Exercise: 3 stars, standard (binary) We can generalize our unary representation of natural numbers to the more efficient binary representation by treating a binary number as a sequence of constructors A and B (representing 0s and 1s), terminated by a Z. For comparison, in the unary representation, a number is a sequence of Ss terminated by an O.

For example:

decimal binary unary 0 Z O 1 B Z S O 2 A (B Z) S (S O) 3 B (B Z) S (S (S O)) 4 A (A (B Z)) S (S (S (S O))) 5 B (A (B Z)) S (S (S (S (S O)))) 6 A (B (B Z)) S (S (S (S (S (S O))))) 7 B (B (B Z)) S (S (S (S (S (S (S O)))))) 8 A (A (A (B Z))) S (S (S (S (S (S (S (S O))))))))

Note that the low-order bit is on the left and the high-order bit is on the right – the opposite of the way binary numbers are usually written. This choice makes them easier to manipulate.

```
Inductive bin : Type :=
  | Z
  | A (n : bin)
  | B (n : bin).
```

(a) Complete the definitions below of an increment function incr for binary numbers, and a function bin_to_nat to convert binary numbers to unary numbers.

```
Fixpoint incr (m:bin) : bin
  . Admitted.

Fixpoint bin_to_nat (m:bin) : nat
  . Admitted.
```

(b) 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.

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

□

Chapter 3

Induction: Proof by Induction

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

`From` *LF* `Require Export` Basics.

For the `Require Export` to work, Coq needs to be able to find a compiled version of *Basics.v*, called *Basics.vo*, in a directory associated with the prefix *LF*. This file is analogous to the *.class* files compiled from *.java* source files and the *.o* files compiled from .c files.

First create a file named *_CoqProject* containing the following line (if you obtained the whole volume "Logical Foundations" as a single archive, a *_CoqProject* should already exist and you can skip this step):

-Q . LF

This maps the current directory (".", which contains *Basics.v*, *Induction.v*, etc.) to the prefix (or "logical directory") "*LF*". PG and CoqIDE read *_CoqProject* automatically, so they know to where to look for the file *Basics.vo* corresponding to the library LF.Basics.

Once *_CoqProject* is thus created, there are various ways to build *Basics.vo*:

- In Proof General: The compilation can be made to happen automatically when you submit the `Require` line above to PG, by setting the emacs variable *coq-compile-*`before`*-require* to *t*.
- In CoqIDE: Open *Basics.v*; then, in the "Compile" menu, click on "Compile Buffer".
- From the command line: Generate a *Makefile* using the *coq_makefile* utility, that comes installed with Coq (if you obtained the whole volume as a single archive, a *Makefile* should already exist and you can skip this step):

  *coq_makefile -f _CoqProject *.v -o Makefile*

  Note: You should rerun that command whenever you add or remove Coq files to the directory.

  Then you can compile *Basics.v* by running *make* with the corresponding *.vo* file as a target:

  make Basics.vo

All files in the directory can be compiled by giving no arguments:

make

Under the hood, *make* uses the Coq compiler, *coqc*. You can also run *coqc* directly:

coqc -Q . LF Basics.v

But *make* also calculates dependencies between source files to compile them in the right order, so *make* should generally be prefered over explicit *coqc*.

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.

In particular, if you see a message like

Compiled library Foo makes inconsistent assumptions over library Bar

check whether you have multiple installations of Coq on your machine. It may be that commands (like *coqc*) that you execute in a terminal window are getting a different version of Coq than commands executed by Proof General or CoqIDE.

- Another common reason is that the library *Bar* was modified and recompiled without also recompiling *Foo* which depends on it. Recompile *Foo*, or everything if too many files are affected. (Using the third solution above: *make clean*; *make*.)

One more tip for CoqIDE users: If you see messages like *Error*: *Unable to locate library* Basics, a likely reason is inconsistencies between compiling things *within CoqIDE* vs *using coqc from the command line*. This typically happens when there are two incompatible versions of *coqc* installed on your system (one associated with CoqIDE, and one associated with *coqc* from the terminal). The workaround for this situation is compiling using CoqIDE only (i.e. choosing "make" from the menu), and avoiding using *coqc* directly at all.

3.1 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 $n = \mathsf{S}\ n'$ 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'] eqn:E.
  -
    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 $P(\mathsf{O})$ holds;
- show that, for any n', if $P(n')$ holds, then so does $P(\mathsf{S}\ n')$;
- 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 $P(\mathsf{O})$ and another where we must show $P(n') \to P(\mathsf{S}\ n')$. 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 $\mathsf{S}\ n'$, 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'* = (S *n'*) + 0, which simplifies to S *n'* = S (*n'* + 0), 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, standard, 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, standard (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, standard, optional (evenb_S) One inconvenient 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 star, standard (destruct_induction) Briefly explain the difference between the tactics `destruct` and `induction`.

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

□

3.2 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` $(0 + n = n)$ `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 $0 + n = n$. 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 $(n + m) + (p + q) = (m + n) + (p + q)$. 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` $\rightarrow$ 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 $n + m = m + n$ (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.
```

3.3 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, recommended (plus_comm_informal) Translate your solution for plus_comm into an informal proof:

Theorem: Addition is commutative.

Proof:

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

□

Exercise: 2 stars, standard, optional (eqb_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 = n =? n for any n.

Proof:

□

3.4 More Exercises

Exercise: 3 stars, standard, 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, standard, 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!)

```
Check leb.

Theorem leb_refl : ∀ n:nat,
  true = (n <=? n).
Proof.
  Admitted.

Theorem zero_nbeq_S : ∀ n: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,
  n <=? m = true → (p + n) <=? (p + m) = true.
Proof.
  Admitted.

Theorem S_nbeq_0 : ∀ n:nat,
```

```
  (S n) =? 0 = false.
Proof.
  Admitted.

Theorem mult_1_l : ∀ n:nat, 1 × n = n.
Proof.
  Admitted.

Theorem all3_spec : ∀ b c : bool,
    orb
      (andb b c)
      (orb (negb b)
           (negb c))
  = true.
Proof.
  Admitted.

Theorem mult_plus_distr_r : ∀ n m p : nat,
  (n + m) × p = (n × p) + (m × p).
Proof.
  Admitted.

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

Exercise: 2 stars, standard, optional (eqb_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 eqb_refl : ∀ n : nat,
  true = (n =? n).
Proof.
  Admitted.
  □
```

Exercise: 2 stars, standard, 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' : ∀ n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
```
Admitted.
□

Exercise: 3 stars, standard, recommended (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!

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

Exercise: 5 stars, advanced (binary_inverse) This is a further continuation of the previous exercises about binary numbers. You may find you need to go back and change your earlier definitions to get things to work here.

(a) First, write a function to convert natural numbers to binary numbers.

```
Fixpoint nat_to_bin (n:nat) : bin
```
. *Admitted.*

Prove that, if we start with any **nat**, convert it to binary, and convert it back, we get the same **nat** we started with. (Hint: If your definition of nat_to_bin involved any extra functions, you may need to prove a subsidiary lemma showing how such functions relate to nat_to_bin.)

```
Theorem nat_bin_nat : ∀ n, bin_to_nat (nat_to_bin n) = n.
Proof.
```
Admitted.

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

(b) One might naturally expect that we should also prove the opposite direction – that starting with a binary number, converting to a natural, and then back to binary should yield the same number we started with. However, this is not the case! Explain (in a comment) what the problem is.

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

(c) Define a normalization function – i.e., a function *normalize* going directly from **bin** to **bin** (i.e., *not* by converting to **nat** and back) such that, for any binary number b, converting b to a natural and then back to binary yields (*normalize* b). Prove it. (Warning: This part is a bit tricky – you may end up defining several auxiliary lemmas. One good way to find out what you need is to start by trying to prove the main statement, see where you get stuck, and see if you can find a lemma – perhaps requiring its own inductive proof – that will allow the main proof to make progress.) Don't define thi using nat_to_bin and bin_to_nat!

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

□

Chapter 4

Lists: Working with Structured Data

```
From LF Require Export Induction.
Module NatList.
```

4.1 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 with **nybble**, and here):

```
Inductive natprod : Type :=
| pair (n1 n2 : nat).
```

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 simple functions for extracting the first and second components of a pair.

```
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 will be used heavily, 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.

```
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.
```

Note that pattern-matching on a pair (with parentheses: (x, y)) is not to be confused with the "multiple pattern" syntax (with no parentheses: x, y) that we have seen previously.

The above examples illustrate pattern matching on a pair with elements x and y, whereas minus below (taken from Basics) performs pattern matching on the values n and m.

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

The distinction is minor, but it is worth knowing that they are not the same. For instance, the following definitions are ill-formed:

Definition bad_fst (p : natprod) : nat := match p with | x, y => x end.

Definition bad_minus (n m : nat) : nat := match n, m with | (O , _) => O | (S _, O) => n | (S n', S m') => bad_minus n' m' end.

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

If we state things in a 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 **nat**s, where it generates two subgoals, `destruct` generates just one subgoal here. That's because **natprod**s can only be constructed in one way.

Exercise: 1 star, standard (snd_fst_is_swap) `Theorem snd_fst_is_swap : ∀ (p : natprod),`

```
  (snd p, fst p) = swap_pair p.
Proof.
```

Admitted.

□

Exercise: 1 star, standard, optional (fst_swap_is_snd) `Theorem fst_swap_is_snd : ∀ (p : natprod),`

```
  fst (swap_pair p) = snd p.
Proof.
```

Admitted.

□

4.2 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
  | cons (n : nat) (l : 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 here is roughly what's going on in case you are interested. The `right associativity` annotation tells Coq

how to parenthesize expressions involving multiple 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.
```

Since app will be used extensively in what follows, it is again 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"). Since the empty list has no first element, 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, standard, 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 interleaves 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 involves 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 representation for a bag of numbers is as a list.

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

Exercise: 3 stars, standard, 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 – using max instead of sum – 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, standard, 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. (This exercise is optional, but students following the advanced track will need to fill in the definition of remove_one for a later exercise.)

```
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: 2 stars, standard, recommended (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!

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

4.3 Reasoning About Lists

As for 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 expression whose value is 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* proof scripts will not get you very far! It is important to step through the details of each one using Coq and think 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.

4.3.1 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 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

  (□ ++ l2) ++ l3 = □ ++ (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. □

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, for something a bit more challenging than the proofs we've seen so far, 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 state 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 we 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.

4.3.2 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` into your .v file and evaluating this line 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*-;.

4.3.3 List Exercises, Part 1

Exercise: 3 stars, standard (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, standard (eqblist) Fill in the definition of eqblist, which compares lists of numbers for equality. Prove that eqblist l l yields true for every list l.

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

Example test_eqblist1 :
  (eqblist nil nil = true).
   Admitted.

Example test_eqblist2 :
  eqblist [1;2;3] [1;2;3] = true.
   Admitted.

Example test_eqblist3 :
  eqblist [1;2;3] [1;2;4] = false.
   Admitted.

Theorem eqblist_refl : ∀ l:natlist,
  true = eqblist l l.
Proof.
   Admitted.
   □
```

4.3.4 List Exercises, Part 2

Here are a couple of little theorems to prove about your definitions about bags above.

Exercise: 1 star, standard (count_member_nonzero)

```
Theorem count_member_nonzero
: ∀ (s : bag),
  1 <=? (count 1 (1 :: s)) = true.
Proof.
   Admitted.
   □
```

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

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

Before doing the next exercise, make sure you've filled in the definition of remove_one above.

Exercise: 3 stars, advanced (remove_does_not_increase_count) `Theorem` remove_does_not_increa
∀ (s : bag),
  (*count* 0 (*remove_one* 0 s)) <=? (*count* 0 s) = true.
`Proof`.
  Admitted.
  □

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

Exercise: 4 stars, advanced (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.)

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

  □

4.4 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 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 (n : nat)
  | None.
```

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 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 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, standard (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, standard, 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.
```

4.5 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 (n : nat).
```

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 ids:

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

Exercise: 1 star, standard (eqb_id_refl) `Theorem eqb_id_refl : ∀ x, true = eqb_id x x.`
`Proof.`
Admitted.
□

Now we define the type of partial maps:

`Module PartialMap.`
`Export NatList.`

```
Inductive partial_map : Type :=
  | empty
  | record (i : id) (v : nat) (m : 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 by shadowing it with a new one (or simply 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 eqb_id x y
                    then Some v
                    else find x d'
  end.
```

Exercise: 1 star, standard (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, standard (update_neq) `Theorem update_neq :`
`∀ (d : partial_map) (x y : id) (o: nat),`

```
    eqb_id x y = false → find x (update d y o) = find x d.
Proof.
  Admitted.
  □ End PartialMap.
```

Exercise: 2 stars, standard (baz_num_elts) Consider the following inductive definition:

```
Inductive baz : Type :=
  | Baz1 (x : baz)
  | Baz2 (y : baz) (b : bool).
```

How *many* elements does the type **baz** have? (Explain in words, in a comment.)

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

□

Chapter 5

Poly: Polymorphism and Higher-Order Functions

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Lists.
```

5.1 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.

5.1.1 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
  | bool_cons (b : bool) (l : 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
| cons (x : X) (l : 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 `Type`s to `Inductive` definitions; or, to put it another way, **list** is a function from `Type`s to `Type`s. For any particular type X, the type **list** X is an `Inductive`ly defined set of lists whose elements are of type X.

```
Check list.
```

The parameter X in the definition of **list** automatically becomes a parameter to the constructors nil and cons – that is, nil and cons are now polymorphic constructors; when we use them, we must now provide a first argument that is the type of the list they are building. For example, nil **nat** constructs the empty list of type **nat**.

```
Check (nil nat).
```

Similarly, cons **nat** adds an element of type **nat** to a list of type **list nat**. Here is an example of forming a list containing just the natural number 3.

```
Check (cons nat 3 (nil nat)).
```

What might the type of nil be? We can read off the type **list** X from the definition, but this omits the binding for X which is the parameter to **list**. `Type` → **list** X does not explain the meaning of X. (X : `Type`) → **list** X comes closer. Coq's notation for this situation is ∀ X : `Type`, **list** X.

```
Check nil.
```

Similarly, the type of cons from the definition looks like X → **list** X → **list** X, but using this convention to explain the meaning of X results in the type ∀ X, X → **list** X → **list** X.

```
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.)

Having to supply a type argument for each use of a list constructor may seem an awkward burden, but we will soon see ways of reducing that burden.

```
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 an element of this type (and a number):

```
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.
```

Exercise: 2 stars, standard (mumble_grumble) Consider the following two inductively defined types.

```
Module MumbleGrumble.

Inductive mumble : Type :=
  | a
  | b (x : mumble) (y : nat)
  | c.

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

Which of the following are well-typed elements of **grumble** X for some type X? (Add YES or NO to each line.)

- 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.

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

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 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 a "hole" `_`, 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 holes, the `repeat` 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 holes 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 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'
  | cons' (x : X) (l : 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, standard, 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, standard, 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.
  □
```

5.1.2 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 : X) (y : 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, standard, optional (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, standard, 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.
□

5.1.3 Polymorphic Options

One last polymorphic type for now: *polymorphic options*, which generalize **natoption** from the previous chapter. (We put the definition inside a module because the standard library already defines **option** and it's this one that we want to use below.)

```
Module OptionPlayground.

Inductive option (X:Type) : Type :=
  | Some (x : X)
  | None.

Arguments Some {X} _.
Arguments None {X}.

End OptionPlayground.
```

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 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, standard, 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.
```
□

5.2 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.

5.2.1 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.
```

5.2.2 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 :=
  (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.
```

5.2.3 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 $n \Rightarrow n \times n$) can be read as "the function that, given a number n, yields $n \times n$."

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

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

```
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

Exercise: 2 stars, standard (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, standard (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 $X \to$ **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.
```

□

5.2.4 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, standard (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, standard, 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 for which map makes sense. Here is a 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, standard, 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.)

□

5.2.5 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, advanced (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?

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

5.2.6 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.
```

5.3 Additional Exercises

```
Module Exercises.
```

Exercise: 2 stars, standard (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. (Hint: It may help to know that `reflexivity` simplifies expressions a bit more aggressively than `simpl` does – i.e., you may find yourself in a situation where `simpl` does nothing but `reflexivity` solves the goal.)

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

Admitted.

□

Exercise: 3 stars, standard (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. (Hint: again, remember that `reflexivity` simplifies expressions a bit more aggressively than `simpl`.)

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

□

Exercise: 2 stars, advanced (currying) In Coq, a function $f : A \to B \to C$ really has the type $A \to (B \to C)$. That is, if you give f a value of type A, it will give you function f' : $B \to C$. 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_map1': map (plus 3) [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 : ∀ (X Y Z : Type)
                          (f : (X × Y) → Z) (p : X × Y),
  prod_uncurry (prod_curry f) p = f p.
Proof.
  Admitted.
```

□

Exercise: 2 stars, advanced (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 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

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

□

The following exercises explore 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 cnat := ∀ X : Type, (X → X) → X → 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 : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f x.
```

Similarly, two should apply f twice to its argument:

```
Definition two : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ 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 : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ 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 : cnat := @doit3times.
```

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

Exercise: 1 star, advanced (church_succ) Successor of a natural number: given a Church numeral n, the successor succ n is a function that iterates its argument once more than n. `Definition succ (n : cnat) : cnat`
. *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.
```

□

Exercise: 1 star, advanced (church_plus) Addition of two natural numbers: `Definition` plus (n m : cnat) : cnat
. *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*.

□

Exercise: 2 stars, advanced (church_mult) Multiplication: `Definition` mult (n m : cnat) : cnat
. *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*.

□

Exercise: 2 stars, advanced (church_exp) 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. Iterating over cnat itself is usually problematic.)

`Definition` exp (n m : cnat) : cnat
. *Admitted*.

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

`Example` exp_2 : *exp* three zero = one.
`Proof`. *Admitted*.

`Example` exp_3 : *exp* three two = *plus* (*mult* two (*mult* two two)) one.
`Proof`. *Admitted*.

□

`End` Church.

`End` Exercises.

Chapter 6

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.

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Poly.
```

6.1 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.
```

Typically, when we use `apply` H, the statement H will begin with a $\forall$ 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, standard, optional (silly_ex) Complete the following proof without using `simpl`.

```
Theorem silly_ex :
      (∀ n, evenb n = true → oddb (S n) = true) →
      oddb 3 = true →
      evenb 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 = (n =? 5) →
      (S (S n)) =? 7 = true.
Proof.
  intros n H.
```

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.  (This simpl is optional, since apply will perform simplification first, if needed.)
apply H. Qed.
```

Exercise: 3 stars, standard (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, standard, optional (apply_rewrite) Briefly explain the difference between the tactics `apply` and `rewrite`. What are the situations where both can usefully be applied?

6.2 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, standard, 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.
```

□

6.3 The `injection` and `discriminate` Tactics

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.

For example, we can prove the injectivity of S by using the pred function defined in *Basics.v*.

```
Theorem S_injective : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H1.
  assert (H2: n = pred (S n)). { reflexivity. }
  rewrite H2. rewrite H1. reflexivity.
Qed.
```

This technique can be generalized to any constructor by writing the equivalent of pred for that constructor – i.e., writing a function that "undoes" one application of the constructor. As a more convenient alternative, Coq provides a tactic called `injection` that allows us to exploit the injectivity of any constructor. Here is an alternate proof of the above theorem using `injection`:

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

By writing `injection` *H* at this point, we are asking Coq to generate all equations that it can infer from *H* using the injectivity of constructors. Each such equation is added as a premise to the goal. In the present example, adds the premise $n = m$.

```
  injection H. intros Hnm. apply Hnm.
Qed.
```

Here's a more interesting example that shows how `injection` can derive multiple equations at once.

```
Theorem injection_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
Proof.
  intros n m o H.
  injection H. intros H1 H2.
  rewrite H1. rewrite H2. reflexivity.
Qed.
```

The "as" variant of `injection` permits us to choose names for the introduced equations rather than letting Coq do it.

```
Theorem injection_ex2 : ∀ (n m : nat),
```

```
  [n] = [m] →
  n = m.
Proof.
  intros n m H.
  injection H as Hnm. rewrite Hnm.
  reflexivity. Qed.
```

Exercise: 1 star, standard (injection_ex3) `Example injection_ex3 : ∀ (X : Type) (x y z : X) (l j : list X),`

```
  x :: y :: l = z :: j →
  y :: l = x :: j →
  x = y.
Proof.
```

Admitted.

□

So much for injectivity of constructors. What about disjointness?

The principle of disjointness says that two terms beginning with different constructors (like O and S, or true and false) can never be equal. This means that, any time we find ourselves working in a context where we've *assumed* that two such terms are equal, we are justified in concluding anything we want to (because the assumption is nonsensical).

The `discriminate` tactic embodies this principle: It is used on a hypothesis involving an equality between different constructors (e.g., S n = O), and it solves the current goal immediately. For example:

```
Theorem eqb_0_l : ∀ n,
  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'] eqn:E.
  -
    intros H. reflexivity.
```

However, the second one doesn't look so simple: assuming 0 =? (S n') = true, we must show S n' = 0! The way forward is to observe that the assumption itself is nonsensical:

```
  -
    simpl.
```

If we use `discriminate` on this hypothesis, Coq confirms that the subgoal we are working on is impossible and removes it from further consideration.

```
    intros H. discriminate 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 discriminate_ex1 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. discriminate contra. Qed.

Theorem discriminate_ex2 : ∀ (n m : nat),
  false = true →
  [n] = [m].
Proof.
  intros n m contra. discriminate contra. Qed.
```

If you find the principle of explosion confusing, remember that these proofs are *not* showing that the conclusion of the statement holds. Rather, they are showing 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, standard (discriminate_ex3) `Example discriminate_ex3 :`

```
  ∀ (X : Type) (x y z : X) (l j : list X),
    x :: y :: l = [] →
    x = z.
Proof.
  Admitted.
```

□

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 convenient in a few places below:

```
Theorem f_equal : ∀ (A B : Type) (f: A → B) (x y: A),
  x = y → f x = f y.
Proof. intros A B f x y eq. rewrite eq. reflexivity. Qed.
```

6.4 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),
```

```
    (S n) =? (S m) = b →
    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 $X \to Y$, say) against a hypothesis H in the context. However, unlike ordinary `apply` (which rewrites a goal matching Y into a subgoal X), `apply` L `in` H matches H against X and, if successful, replaces it with Y.

In other words, `apply` L `in` H gives us a form of "forward reasoning": from $X \to Y$ and a hypothesis matching X, it produces a hypothesis matching X. By contrast, `apply` L is "backward reasoning": it says that if we know $X \to Y$ and we are trying to prove Y, it suffices to prove X.

Here is a variant of a proof from above, using forward reasoning throughout instead of backward reasoning.

```
Theorem silly3' : ∀ (n : nat),
  (n =? 5 = true → (S (S n)) =? 7 = true) →
  true = (n =? 5) →
  true = ((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, standard, recommended (plus_n_n_injective) Practice using "in" variants in this proof. (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.
```

□

6.5 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 double 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'] eqn:E.
    + reflexivity.
    + discriminate eq.
  - intros eq. destruct m as [| m'] eqn:E.
    + discriminate 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

- $P\ n \to P\ (\mathsf{S}\ n)$

  (i.e., "if $\mathsf{double}\ n = \mathsf{double}\ m$ then $n = m$" implies "if $\mathsf{double}\ (\mathsf{S}\ n) = \mathsf{double}\ m$ then $\mathsf{S}\ n = m$").

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 $\mathsf{double}\ n = \mathsf{double}\ m$ then $n = m$"

then we can prove

- "if $\mathsf{double}\ (\mathsf{S}\ n) = \mathsf{double}\ m$ then $\mathsf{S}\ n = m$".

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 $\mathsf{double}\ n = 10$ then $n = 5$"

then we can prove

- R = "if $\mathsf{double}\ (\mathsf{S}\ n) = 10$ then $\mathsf{S}\ n = 5$".

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 $\mathsf{double}\ (\mathsf{S}\ n) = 10$..." but then we'd be stuck: knowing that $\mathsf{double}\ (\mathsf{S}\ n)$ is 10 tells us nothing about whether $\mathsf{double}\ n$ 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 statement 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'] eqn:E.
    + reflexivity.
    + discriminate 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'] eqn:E.
+ simpl.
```

The 0 case is trivial:

```
  discriminate 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'. injection eq as goal. apply goal. Qed.
```

What you should take away from all this is that we need to be careful, when using induction, that we are not trying 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, standard (eqb_true) `Theorem eqb_true : ∀ n m,`
`n =? m = true → n = m.`

```
Proof.
```
Admitted.
□

Exercise: 2 stars, advanced (eqb_true_informal) Give a careful informal proof of eqb_true, being as explicit as possible about quantifiers.

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

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'] eqn:E.
    + reflexivity.
    + discriminate eq.
  - intros eq. destruct n as [| n'] eqn:E.
    + discriminate 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'] eqn:E.
    + reflexivity.
    + discriminate eq.
  - intros n eq. destruct n as [| n'] eqn:E.
    + discriminate eq.
    + apply f_equal.
      apply IHm'. injection eq as goal. apply goal. 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 injectivity 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 eqb_true to prove a similar property of identifiers that we'll need in later chapters:

```
Theorem eqb_id_true : ∀ x y,
  eqb_id x y = true → x = y.
Proof.
  intros [m] [n]. simpl. intros H.
  assert (H' : m = n). { apply eqb_true. apply H. }
  rewrite H'. reflexivity.
Qed.
```

Exercise: 3 stars, standard, 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.
```
□

6.6 Unfolding Definitions

It sometimes happens that we need to manually unfold a name that has been introduced by 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 appear to be 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 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, some 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` $x \Rightarrow 5$) 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 somewhat 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 cannot itself 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 eqn:E.
  - 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 eqn:E.
  - reflexivity.
  - reflexivity.
Qed.
```

6.7 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 n =? 3 then false
  else if n =? 5 then false
  else false.

Theorem sillyfun_false : ∀ (n : nat),
  sillyfun n = false.
Proof.
  intros n. unfold sillyfun.
  destruct (n =? 3) eqn:E1.
    - reflexivity.
    - destruct (n =? 5) eqn:E2.
      + reflexivity.
      + reflexivity. Qed.
```

After unfolding sillyfun in the above proof, we find that we are stuck on `if` $(n =? 3)$ `then` ... `else` .... But either n is equal to 3 or it isn't, so we can use `destruct` (`eqb` 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, standard, optional (combine_split) Here is an implementation of the `split` function mentioned in chapter Poly:

```
Fixpoint split {X Y : Type} (l : list (X×Y))
               : (list X) × (list Y) :=
  match l with
  | [] ⇒ ([], [])
  | (x, y) :: t ⇒
      match split t with
      | (lx, ly) ⇒ (x :: lx, y :: ly)
      end
  end.
```

Prove that `split` and combine are inverses in the following sense:

```
Theorem combine_split : ∀ X Y (l : list (X × Y)) l1 l2,
  split l = (l1, l2) →
  combine l1 l2 = l.
Proof.
  Admitted.
```

□

The *eqn*: part of the `destruct` tactic is optional: We've chosen to include it most of the time, just for the sake of documentation, but many Coq proofs omit it.

When `destructing` compound expressions, however, the information recorded by the *eqn*: can actually be critical: if we leave it out, then `destruct` 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 n =? 3 then true
  else if 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. If we start the proof like this (with no *eqn*: on the destruct)...

```
Theorem sillyfun1_odd_FAILED : ∀ (n : nat),
     sillyfun1 n = true →
     oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (n =? 3).
Abort.
```

... then we are 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 quite brutal – in this case, it thows away every occurrence of $n =? 3$, 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 $n =? 3 =$ true in this branch of the case analysis, it must be that $n = 3$, from which it follows that n is odd.

What we want here is to substitute away all existing occurences of $n =? 3$, but at the same time add an equation to the context that records which case we are in. This is precisely what the *eqn*: qualifier does.

```
Theorem sillyfun1_odd : ∀ (n : nat),
     sillyfun1 n = true →
     oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (n =? 3) eqn:Heqe3.
    - apply eqb_true in Heqe3.
      rewrite → Heqe3. reflexivity.
    -


      destruct (n =? 5) eqn:Heqe5.
        +
          apply eqb_true in Heqe5.
          rewrite → Heqe5. reflexivity.
        + discriminate eq. Qed.
```

Exercise: 2 stars, standard (destruct_eqn_practice) `Theorem bool_fn_applied_thrice :`

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

□

6.8 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
- `injection`: reason by injectivity on equalities between values of inductively defined types
- `discriminate`: reason by disjointness of constructors on equalities between values of inductively defined types
- `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

6.9 Additional Exercises

Exercise: 3 stars, standard (eqb_sym) `Theorem eqb_sym : ∀ (n m : nat),`
`(n =? m) = (m =? n).`
`Proof.`
Admitted.
□

Exercise: 3 stars, advanced, optional (eqb_sym_informal) Give an informal proof of this lemma that corresponds to your formal proof above:
Theorem: For any **nat**s n m, $(n =? m) = (m =? n)$.
Proof:

Exercise: 3 stars, standard, optional (eqb_trans) `Theorem eqb_trans : ∀ n m p,`
`n =? m = true →`
`m =? p = true →`
`n =? p = true.`
`Proof.`
Admitted.
□

Exercise: 3 stars, advanced (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.

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

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 (eqb 5) □ = true

The second checks whether there exists an element in the list that satisfies a given predicate:

existsb (eqb 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.

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool
  . Admitted.

Example test_forallb_1 : forallb oddb [1;3;5;7;9] = true.
Proof. Admitted.

Example test_forallb_2 : forallb negb [false;false] = true.
Proof. Admitted.
```

```
Example test_forallb_3 : forallb evenb [0;2;4;5] = false.
Proof. Admitted.

Example test_forallb_4 : forallb (eqb 5) [] = true.
Proof. Admitted.

Fixpoint existsb {X : Type} (test : X → bool) (l : list X) : bool
  . Admitted.

Example test_existsb_1 : existsb (eqb 5) [0;2;3;6] = false.
Proof. Admitted.

Example test_existsb_2 : existsb (andb true) [true;true;false] = true.
Proof. Admitted.

Example test_existsb_3 : existsb oddb [1;0;0;0;0;3] = true.
Proof. Admitted.

Example test_existsb_4 : existsb evenb [] = false.
Proof. Admitted.

Definition existsb' {X : Type} (test : X → bool) (l : list X) : bool
  . Admitted.

Theorem existsb_existsb' : ∀ (X : Type) (test : X → bool) (l : list X),
  existsb test l = existsb' test l.
Proof. Admitted.
```

□

Chapter 7

Logic: Logic in Coq

```
Set Warnings "-notation-overridden,-parsing".
From LF 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 ($P \to Q$), 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.

Simply *being* a proposition is one thing; being *provable* is something else!

```
Check 2 = 2.
```

```
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. injection H as H1. apply H1.
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**.)

7.1 Logical Connectives

7.1.1 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, standard (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 ∧ 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, standard, optional (proj2) `Lemma proj2 : ∀ P Q : Prop,`
$P \wedge Q \to Q$.
`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, standard (and_assoc) (In the following proof of associativity, notice how the *nested* `intros` pattern breaks the hypothesis $H : P \wedge (Q \wedge \mathsf{R})$ down into $HP : P$, $HQ : Q$, and $HR : \mathsf{R}$. 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 $\wedge$ 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.
```

7.1.2 Disjunction

Another important connective is the *disjunction*, or *logical or*, of two propositions: $\mathsf{A} \vee \mathsf{B}$ is true when either A or B is. (This infix notation stands for **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 explicitly with `destruct` or implicitly with an `intros` pattern:

```
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 here is 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, standard (mult_eq_0) `Lemma mult_eq_0 :`
`∀ n m, n × m = 0 → n = 0 ∨ m = 0.`

```
Proof.
  Admitted.
```
□

Exercise: 1 star, standard (or_commut) `Theorem or_commut : ∀ P Q : Prop,`
`P ∨ Q → Q ∨ P.`

```
Proof.
  Admitted.
```
□

7.1.3 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 some given proposition is *not* true. In Coq, such statements are expressed with the negation operator ¬.

To see how negation works, recall 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, $P \to Q$.

Coq actually makes a slightly different (but equivalent) choice, defining ¬ P as $P \to$ **False**, where **False** is a specific 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 use `destruct` on 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, standard, 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.
```

□

Inequality is a frequent enough example of negated statement that there is a special notation for it, $x \neq y$:

Notation "x <> y" := (~(x = y)).

We can use not to state that 0 and 1 are different elements of **nat**:

```
Theorem zero_not_one : 0 ≠ 1.
Proof.
```

The proposition $0 \neq 1$ is exactly the same as ~(0 = 1), that is not (0 = 1), which unfolds to $(0 = 1) \to$ **False**. (We use `unfold not` explicitly here to illustrate that point, but generally it can be omitted.) `unfold not.`

To prove an inequality, we may assume the opposite equality... `intros contra.`

... and deduce a contradiction from it. Here, the equality O = S O contradicts the disjointness of constructors O and S, so `discriminate` takes care of it. `discriminate` *contra*.
`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 (double_neg_inf) Write an informal proof of double_neg:
Theorem: P implies $\tilde{}\tilde{}P$, for any proposition P.

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

Exercise: 2 stars, standard, recommended (contrapositive) `Theorem` contrapositive
: ∀ (P Q : Prop),
  (P → Q) → (¬Q → ¬P).
Proof.
Admitted.
□

Exercise: 1 star, standard (not_both_true_and_false) `Theorem` not_both_true_and_false
: ∀ P : Prop,
  ¬ (P ∧ ¬P).
Proof.
Admitted.
□

Exercise: 1 star, advanced (informal_not_PNP) Write an informal proof (in English) of the proposition $\forall\ P$: Prop, ~$(P \wedge \neg P)$.

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

□

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 $\neg P$ that may be available in the context – in particular, assumptions of the form $x \neq y$.

```
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.
```

7.1.4 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 `Prop`s using conditionals or as a parameter to higher-order `Prop`s. We will see examples of such uses of **True** later on.

7.1.5 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'. discriminate H'.
Qed.
```

Exercise: 1 star, standard, 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, standard (or_distributes_over_and) `Theorem or_distributes_over_and : ∀ P Q R : Prop, P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ 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 Coq library that supports it:

```
From Coq Require Import 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 $\leftrightarrow$. 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.
```

7.1.6 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 $\exists\ x : T, P$. As with $\forall$, 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 $\exists\ x, P$, 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 $\exists\ t$. 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 $\exists\ x, P$ 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, standard, recommended (dist_not_exists) Prove that "P holds for all x" implies "there is no x for which P does not hold." (Hint: `destruct H as [x E]` works on existential assumptions!)

```
Theorem dist_not_exists : ∀ (X:Type) (P : X → Prop),
  (∀ x, P x) → ¬ (∃ x, ¬ P x).
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard (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.
  □
```

7.2 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 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, standard (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, standard (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, standard, recommended (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, standard (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.
□

7.3 Applying Theorems to Arguments

One feature of Coq that distinguishes it from some other popular proof assistants (e.g., ACL2 and Isabelle) 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 all 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 $\forall$ 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 **nat**s as arguments and get a **nat** back. Similarly, if we have an object of type $n = m \rightarrow n + n = m + m$ 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 :
  ∀ x y z, x + (y + z) = (z + y) + x.
```

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 x y z.
  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 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite plus_comm.
  assert (H : y + z = z + y).
  { 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 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite plus_comm.
  rewrite (plus_comm y z).
  reflexivity.
Qed.
```

Let us show another example of using a theorem or lemma like a function. The following theorem says: any list l containing some element must be nonempty.

```
Lemma in_not_nil :
  ∀ A (x : A) (l : list A), In x l → l ≠ [].
Proof.
  intros A x l H. unfold not. intro Hl. destruct l.
  - simpl in H. destruct H.
  - discriminate Hl.
Qed.
```

What makes this interesting is that one quantified variable (x) does not appear in the conclusion ($l \neq []$).

We can use this lemma to prove the special case where x is 42. Naively, the tactic `apply in_not_nil` will fail because it cannot infer the value of x. There are several ways to work around that...

```
Lemma in_not_nil_42 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  Fail apply in_not_nil.
```

```
Abort.

Lemma in_not_nil_42_take2 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply in_not_nil with (x := 42).
  apply H.
Qed.

Lemma in_not_nil_42_take3 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply in_not_nil in H.
  apply H.
Qed.

Lemma in_not_nil_42_take4 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply (in_not_nil nat 42).
  apply H.
Qed.

Lemma in_not_nil_42_take5 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply (in_not_nil _ _ _ H).
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. (The details of how this proof works are not critical – the goal here is just to illustrate what can be done.)

```
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 in later chapters.

7.4 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 to write down precise and rigorous proofs. For example, in the most popular foundation for 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, about equally 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 **even** n holds, where **even** : **nat** → `Prop` is a property describing even numbers.

However, there are some cases where translating standard mathematical reasoning into Coq can be 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.

7.4.1 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 :
  (fun x ⇒ 3 + x) = (fun x ⇒ (pred 4) + x).
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 built-in logic. 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 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**, 2+2=5, etc.!

Unfortunately, there is no simple way of telling whether an axiom is safe to add: hard work by highly-trained trained experts is generally required to establish the consistency of any particular combination of axioms.

Fortunately, 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, standard (tr_rev_correct) 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.

□

7.4.2 Propositions and Booleans

We've seen two different ways of expressing logical claims 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...

... that evenb n evaluates to true... `Example even_42_bool : evenb 42 = true.`

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

... or that there exists some k such that n = double k. `Example even_42_prop : ∃ k, 42 = double k.`

```
Proof. ∃ 21. reflexivity. Qed.
```

Of course, it would be pretty strange if these two characterizations of evenness did not describe the same set of natural numbers! Fortunately, we can prove that they do...

We first need two helper lemmas. `Theorem evenb_double : ∀ k, evenb (double k) = true.`

```
Proof.
  intros k. induction k as [|k' IHk'].
  - reflexivity.
  - simpl. apply IHk'.
Qed.
```

Exercise: 3 stars, standard (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.
```

In view of this theorem, we say that the boolean computation evenb n is reflected in the truth of the proposition $\exists\ k,\ n =$ double k.

Similarly, to state that two numbers n and m are equal, we can say either

- (1) that $n =?\ m$ returns true, or
- (2) that $n = m$.

Again, these two notions are equivalent.

```
Theorem eqb_eq : ∀ n1 n2 : nat,
  n1 =? n2 = true ↔ n1 = n2.
Proof.
  intros n1 n2. split.
  - apply eqb_true.
  - intros H. rewrite H. rewrite ← eqb_refl. reflexivity.
Qed.
```

However, even when the boolean and propositional formulations of a claim are equivalent from a purely logical perspective, they may not be equivalent *operationally*.

In the case of even numbers above, 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 (evenb n = true) $\rightarrow$ ($\exists\ k,\ n =$ double k).

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 element 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-script 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.

Another notable difference is that the negation of a "boolean fact" is straightforward to state and prove: simply flip the expected boolean result.

```
Example not_even_1001 : evenb 1001 = false.
Proof.
  reflexivity.
Qed.
```

In contrast, propositional negation may be more difficult to grasp.

```
Example not_even_1001' : ~(∃ k, 1001 = double k).
Proof.
  rewrite ← even_bool_prop.
  unfold not.
  simpl.
  intro H.
  discriminate H.
Qed.
```

Equality provides a complementary example: knowing that $n =? \ m =$ `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.

```
Lemma plus_eqb_example : ∀ n m p : nat,
```

```
    n =? m = true → n + p =? m + p = true.
Proof.
  intros n m p H.
    rewrite eqb_eq in H.
  rewrite H.
  rewrite eqb_eq.
  reflexivity.
Qed.
```

We won't cover reflection in much detail, but it serves as a good example showing the complementary strengths of booleans and general propositions.

Exercise: 2 stars, standard (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, standard (eqb_neq) The following theorem is an alternate "negative" formulation of eqb_eq that is more convenient in certain situations (we'll see examples in later chapters).

```
Theorem eqb_neq : ∀ x y : nat,
  x =? y = false ↔ x ≠ y.
Proof.
    Admitted.
    □
```

Exercise: 3 stars, standard (eqb_list) Given a boolean operator eqb for testing equality of elements of some type A, we can define a function eqb_list for testing equality of lists with elements in A. Complete the definition of the eqb_list function below. To make sure that your definition is correct, prove the lemma eqb_list_true_iff.

```
Fixpoint eqb_list {A : Type} (eqb : A → A → bool)
                  (l1 l2 : list A) : bool
  . Admitted.

Lemma eqb_list_true_iff :
```

```
  ∀ A (eqb : A → A → bool),
    (∀ a1 a2, eqb a1 a2 = true ↔ a1 = a2) →
    ∀ l1 l2, eqb_list eqb l1 l2 = true ↔ l1 = l2.
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard, 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?

7.4.3 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. discriminate 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) (n =? m)).
  symmetry.
  apply eqb_eq.
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, standard (excluded_middle_irrefutable) Proving 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 $\neg\ (P\ \vee\ \neg P)$ and $\neg\ \neg\ (P\ \vee\ \neg P)$ (since P implies $\neg\ \neg\ P$, by the exercise below), which would be a contradiction. But since we can't, it is safe to add $P\ \vee\ \neg P$ as an axiom.

```
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, standard, 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 := ∀ P Q: Prop,
  ((P→Q)->P)->P.

Definition double_negation_elimination := ∀ P:Prop,
  ~~P → P.

Definition de_morgan_not_and_not := ∀ P Q:Prop,
  ~(~P ∧ ¬Q) → P∨Q.

Definition implies_to_or := ∀ P Q:Prop,
  (P→Q) → (¬P∨Q).
```

Chapter 8

IndProp: Inductively Defined Propositions

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Logic.
Require Coq.omega.Omega.
```

8.1 Inductively Defined Propositions

In the Logic chapter, we looked at several ways of writing propositions, including conjunction, disjunction, and existential quantification. In this chapter, we bring yet another new tool into the mix: *inductive definitions*.

In past chapters, we have seen two ways of stating that a number n is even: We can say
(1) evenb n = true, or
(2) $\exists$ 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 new 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) even 0

even n

(ev_SS) even (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 **even**, 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) even 0

(ev_SS) even 2

(ev_SS) even 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 shortly.

8.1.1 Inductive Definition of Evenness

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 even : nat → Prop :=
| ev_0 : even 0
| ev_SS (n : nat) (H : even n) : even (S (S n)).
```

This definition is different in one crucial respect from previous uses of `Inductive`: the thing we are defining is not a `Type`, but rather a function from **nat** to `Prop` – that is, a property of numbers. We've already seen other inductive definitions that result in functions – for example, **list**, whose type is `Type → Type`. What is really new here is that, because the **nat** argument of **even** appears 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 **even**, we would have seen an error:

```
Fail Inductive wrong_ev (n : nat) : Prop :=
| wrong_ev_0 : wrong_ev 0
| wrong_ev_SS : wrong_ev n → wrong_ev (S (S n)).
```

In an `Inductive` definition, an argument to the type constructor on the left of the colon is called a "parameter", whereas an argument on the right is called an "index".

For example, in `Inductive` **list** (X : `Type`) := ..., X is a parameter; in `Inductive` **even** : **nat** → `Prop` := ..., the unnamed **nat** argument is an index.

We can think of the definition of **even** as defining a Coq property **even** : **nat** → Prop, together with primitive theorems ev_0 : **even** 0 and ev_SS : ∀ n, **even** n → **even** (S (S n)).

That definition can also be written as follows...

Inductive even : nat -> Prop := | ev_0 : even 0 | ev_SS : forall n, even n -> even (S (S n)).

... making explicit the type of the rule ev_SS.

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 **even** for particular numbers...

```
Theorem ev_4 : even 4.
Proof. apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

... or we can use function application syntax:

```
Theorem ev_4' : even 4.
Proof. apply (ev_SS 2 (ev_SS 0 ev_0)). Qed.
```

We can also prove theorems that have hypotheses involving **even**.

```
Theorem ev_plus4 : ∀ n, even n → even (4 + n).
Proof.
  intros n. simpl. intros Hn.
  apply ev_SS. apply ev_SS. apply Hn.
Qed.
```

Exercise: 1 star, standard (ev_double) `Theorem ev_double : ∀ n,`
  even (double n).

```
Proof.
```
  Admitted.
  □

8.2 Using Evidence in Proofs

Besides *constructing* evidence that numbers are even, we can also *reason about* such evidence.

Introducing **even** 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 **even**).

In other words, if someone gives us evidence E for the assertion **even** 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 **even** n').

This suggests that it should be possible to analyze a hypothesis of the form **even** 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.

8.2.1 Inversion on Evidence

Suppose we are proving some fact involving a number n, and we are given **even** n as a hypothesis. We already know how to perform case analysis on n using `destruct` or `induction`, generating separate subgoals for the case where $n = \mathsf{O}$ and the case where $n = \mathsf{S}\ n'$ for some n'. But for some proofs we may instead want to analyze the evidence that **even** n *directly*. As a tool, we can prove our characterization of evidence for **even** n, using `destruct`.

```
Theorem ev_inversion :
  ∀ (n : nat), even n →
    (n = 0) ∨ (∃ n', n = S (S n') ∧ even n').
Proof.
  intros n E.
  destruct E as [ | n' E'].
  -
    left. reflexivity.
  -
    right. ∃ n'. split. reflexivity. apply E'.
Qed.
```

The following theorem can easily be proved using `destruct` on evidence.

```
Theorem ev_minus2 : ∀ n,
  even n → even (pred (pred n)).
Proof.
  intros n E.
  destruct E as [| n' E'].
  - simpl. apply ev_0.
  - simpl. apply E'.
Qed.
```

However, this variation cannot easily be handled with `destruct`.

```
Theorem evSS_ev : ∀ n,
  even (S (S n)) → even 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.

We could patch this proof by replacing the goal **even** n, which does not mention the replaced term S (S n), by the equivalent goal **even** (pred (pred (S (S n)))), which does mention this term, after which `destruct` can make progress. But it is more straightforward to use our inversion lemma.

```
Theorem evSS_ev : ∀ n, even (S (S n)) → even n.
Proof. intros n H. apply ev_inversion in H. destruct H.
 - discriminate H.
 - destruct H as [n' [Hnm Hev]]. injection Hnm.
   intro Heq. rewrite Heq. apply Hev.
Qed.
```

Coq provides a tactic called `inversion`, which does the work of our inversion lemma and more besides.

The `inversion` tactic can detect (1) that the first case ($n = 0$) does not apply and (2) that the n' that appears in the ev_SS case must be the same as n. It has an "`as`" variant similar to `destruct`, allowing us to assign names rather than have Coq choose them.

```
Theorem evSS_ev' : ∀ n,
  even (S (S n)) → even n.
Proof.
  intros n E.
  inversion E as [| n' E'].
  apply E'.
Qed.
```

The `inversion` tactic can apply the principle of explosion to "obviously contradictory" hypotheses involving inductive properties, something that takes a bit more work using our inversion lemma. For example: `Theorem one_not_even : ¬ even 1.`

```
Proof.
  intros H. apply ev_inversion in H.
  destruct H as [ | [m [Hm _]]].
  - discriminate H.
  - discriminate Hm.
Qed.

Theorem one_not_even' : ¬ even 1.
```

```
  intros H. inversion H. Qed.
```

Exercise: 1 star, standard (inversion_practice) Prove the following result using `inversion`. For extra practice, prove it using the inversion lemma.

```
Theorem SSSSev__even : ∀ n,
  even (S (S (S (S n)))) → even n.
Proof.
  Admitted.
```
□

Exercise: 1 star, standard (even5_nonsense) Prove the following result using `inversion`.

```
Theorem even5_nonsense :
  even 5 → 2 + 2 = 9.
Proof.
  Admitted.
```
□

The `inversion` tactic does quite a bit of work. When applied to equalities, as a special case, it does the work of both `discriminate` and `injection`. In addition, it carries out the `intros` and `rewrite`s that are typically necessary in the case of `injection`. It can also be applied, more generally, to analyze evidence for inductively defined propositions. As examples, we'll use it to reprove some theorems from *Tactics.v*.

```
Theorem inversion_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
Proof.
  intros n m o H. inversion H. reflexivity. Qed.

Theorem inversion_ex2 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. inversion contra. Qed.
```

Here's how `inversion` works in general. Suppose the name H 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` H generates a subgoal in which H 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,
  even n → ∃ k, n = double k.
Proof.
```

We could try to proceed by case analysis or induction on n. But since **even** 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 **even**. 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 $\exists\ k$, S (S n') = double k, but the only available assumption is E', which states that **even** 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 a similar one that involves a *different* piece of evidence for **even**: namely 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.
Abort.
```

8.2.2 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.

To prove a property of n holds for all numbers for which **even** n holds, we can use induction on **even** n. This requires us to prove two things, corresponding to the two ways in which **even** n could have been constructed. If it was constructed by ev_0, then n=0, and the property must hold of 0. If it was constructed by ev_SS, then the evidence of **even** n is of the form ev_SS n' E', where n = S (S n') and E' is evidence for **even** n'. In this case, the inductive hypothesis says that the property we are trying to prove holds for n'.

Let's try our current lemma again:

```
Lemma ev_even : ∀ n,
  even 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 **even** 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,
  even 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, standard (ev_sum) `Theorem ev_sum : ∀ n m, even n → even m → even (n + m).`

```
Proof.
  Admitted.
```

□

Exercise: 4 stars, advanced, optional (even'_ev) In general, there may be multiple ways of defining a property inductively. For example, here's a (slightly contrived) alternative definition for **even**:

```
Inductive even' : nat → Prop :=
| even'_0 : even' 0
| even'_2 : even' 2
| even'_sum n m (Hn : even' n) (Hm : even' m) : even' (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 even'_ev : ∀ n, even' n ↔ even 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,
  even (n+m) → even n → even m.
Proof.
```
Admitted.
□

Exercise: 3 stars, standard, 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,
  even (n+m) → even (n+p) → even (m+p).
Proof.
```
Admitted.
□

8.3 Inductive Relations

A proposition parameterized by a number (such as **even**) 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 (H : le n m) : le n (S m).

Notation "m <= n" := (le m n).
```

Proofs of facts about $\leq$ using the constructors `le_n` and `le_S` follow the same patterns as proofs about properties, like **even** above. We can `apply` the constructors to prove $\leq$ goals (e.g., to show that 3<=3 or 3<=6), and we can use tactics like `inversion` to extract information from $\leq$ 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 : square_of n (n × n).

Inductive next_nat : nat → nat → Prop :=
  | nn n : next_nat n (S n).

Inductive next_even : nat → nat → Prop :=
```

```
| ne_1 n : even (S n) → next_even n (S n)
| ne_2 n (H : even (S (S n))) : next_even n (S (S n)).
```

Exercise: 2 stars, standard, optional (total_relation) Define an inductive binary relation *total_relation* that holds between every pair of natural numbers.

Exercise: 2 stars, standard, optional (empty_relation) Define an inductive binary relation *empty_relation* (on numbers) that never holds.

From the definition of **le**, we can sketch the behaviors of `destruct`, `inversion`, and `induction` on a hypothesis H providing evidence of the form **le** *e1 e2*. Doing `destruct` H will generate two cases. In the first case, *e1* = *e2*, and it will replace instances of *e2* with *e1* in the goal and context. In the second case, *e2* = S *n'* for some *n'* for which **le** *e1 n'* holds, and it will replace instances of *e2* with S *n'*. Doing `inversion` H will remove impossible cases and add generated equalities to the context for further use. Doing `induction` H will, in the second case, add the induction hypothesis that the goal holds when *e2* is replaced with *n'*.

Exercise: 3 stars, standard, 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 ∧ n2 < m.
Proof.
 unfold lt.
   Admitted.

Theorem lt_S : ∀ n m,
  n < m →
  n < S m.
Proof.
   Admitted.

Theorem leb_complete : ∀ n m,
  n <=? m = true → n ≤ m.
Proof.
   Admitted.
```

Hint: The next one may be easiest to prove by induction on m.

```
Theorem leb_correct : ∀ n m,
  n ≤ m →
  n <=? m = true.
Proof.
   Admitted.
```

Hint: This one can easily be proved without using `induction`.

```
Theorem leb_true_trans : ∀ n m o,
  n <=? m = true → m <=? o = true → n <=? o = true.
Proof.
   Admitted.
```

□

Exercise: 2 stars, standard, optional (leb_iff) `Theorem leb_iff : ∀ n m,`

```
  n <=? m = true ↔ n ≤ m.
Proof.
   Admitted.
```

□

```
Module R.
```

Exercise: 3 stars, standard, recommended (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 → nat → nat → Prop :=
   | c1 : R 0 0 0
```

```
| c2 m n o (H : R m n o) : R (S m) n (S o)
| c3 m n o (H : R m n o) : R m (S n) (S o)
| c4 m n o (H : R (S m) (S n) (S (S o))) : R m n o
| c5 m n o (H : R m n o) : 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.

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

Exercise: 3 stars, standard, 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: 2 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!

```
Inductive subseq : list nat → list nat → Prop :=

.

Theorem subseq_refl : ∀ (l : list nat), subseq l l.
Proof.
    Admitted.

Theorem subseq_app : ∀ (l1 l2 l3 : list nat),
  subseq l1 l2 →
  subseq l1 (l2 ++ l3).
Proof.
    Admitted.

Theorem subseq_trans : ∀ (l1 l2 l3 : list nat),
  subseq l1 l2 →
  subseq l2 l3 →
  subseq l1 l3.
Proof.
    Admitted.
□
```

Exercise: 2 stars, standard, optional (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]

8.4 Case Study: Regular Expressions

The **even** 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 definitions 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 sets of strings. Their syntax is defined as follows:

```
Inductive reg_exp {T : Type} : Type :=
  | EmptySet
  | EmptyStr
  | Char (t : T)
  | App (r1 r2 : reg_exp)
  | Union (r1 r2 : reg_exp)
  | Star (r : reg_exp).
```

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 $s = s\_1 ++ \ldots ++ s\_k$, 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 → Prop :=
```

```
  | MEmpty : exp_match [] EmptyStr
  | MChar x : exp_match [x] (Char x)
  | MApp s1 re1 s2 re2
               (H1 : exp_match s1 re1)
               (H2 : exp_match s2 re2) :
               exp_match (s1 ++ s2) (App re1 re2)
  | MUnionL s1 re1 re2
                 (H1 : exp_match s1 re1) :
                 exp_match s1 (Union re1 re2)
  | MUnionR re1 s2 re2
                 (H2 : exp_match s2 re2) :
                 exp_match s2 (Union re1 re2)
  | MStar0 re : exp_match [] (Star re)
  | MStarApp s1 s2 re
                  (H1 : exp_match s1 re)
                  (H2 : 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) $\square$ =~ 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) $\square$ =~ 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 for writing 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, standard (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),
  (∀ s, In s ss → s =~ re) →
  fold app ss [] =~ Star re.
```

```
Proof.
  Admitted.
  □
```

Exercise: 4 stars, standard, optional (reg_exp_of_list_spec) 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 as character literals 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) : 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) (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**, rather than induction on the regular expression *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, standard (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.
  □
```

8.4.1 The *remember* Tactic

One potentially confusing feature of the `induction` tactic is that it will let you try to perform an induction over a term that isn't sufficiently general. The effect of this is to lose information (much as `destruct` without an *eqn*: clause 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 (on evidence!). 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`-without-*eqn*: than like `inversion`.)

An awkward way to solve this problem is "manually generalizing" over the problematic expressions by adding explicit equality hypotheses to the lemma:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re re' : reg_exp),
  re' = Star re →
  s1 =~ 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.

```
Abort.
```

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:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp),
  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, allowing us to conclude immediately.

```
  - discriminate.
  - discriminate.
  - discriminate.
  - discriminate.
  - discriminate.
```

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*.

```
  -
    injection Heqre'. intros Heqre'' s H. apply H.
  -
    injection Heqre'. intros H0.
    intros s2 H1. rewrite ← app_assoc.
    apply MStarApp.
```

```
    + apply Hmatch1.
    + apply IH2.
      × rewrite H0. reflexivity.
      × apply H1.
Qed.
```

Exercise: 4 stars, standard, optional (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),
  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 $s =\tilde{} \; re$ 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.

```
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.
```

□

8.5 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 as 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 (fun x ⇒ n =? x) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (n =? m) eqn:H.
    +
      intros _. rewrite eqb_eq in H. rewrite H.
      left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
Qed.
```

In the first branch after `destruct`, we explicitly apply the `eqb_eq` lemma to the equation generated by destructing $n =? m$, to convert the assumption $n =? m = \mathsf{true}$ 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 $n =? m$. Instead of generating an equation such as $(n =? m) = \mathsf{true}$, which is generally not directly useful, this principle gives us right away the assumption we really need: $n = m$.

```
Inductive reflect (P : Prop) : bool → Prop :=
| ReflectT (H : P) : reflect P true
| ReflectF (H : ¬ 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: that is, P holds if and only if $\mathsf{b} = \mathsf{true}$. To see this, notice that, by definition, the only way we can produce evidence for **reflect** P true is by showing P and then 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. Similarly, the only way to show **reflect** P false is by combining evidence for $\neg P$ with the ReflectF constructor.

It is easy to formalize this intuition and show that the statements $P \leftrightarrow \mathsf{b} = \mathsf{true}$ and **reflect** P b are indeed equivalent. First, the left-to-right implication:

```
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'. discriminate.
Qed.
```

Now you prove the right-to-left implication:

Exercise: 2 stars, standard, 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 eqbP : ∀ n m, reflect (n = m) (n =? m).
Proof.
  intros n m. apply iff_reflect. rewrite eqb_eq. reflexivity.
Qed.
```

A smoother 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 (fun x ⇒ n =? x) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (eqbP n m) as [H | H].
    +
      intros _. rewrite H. left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
Qed.
```

Exercise: 3 stars, standard, recommended (eqbP_practice) Use eqbP as above to prove the following:

```
Fixpoint count n l :=
```

```
  match l with
  | [] ⇒ 0
  | m :: l' ⇒ (if n =? m then 1 else 0) + count n l'
  end.

Theorem eqbP_practice : ∀ n l,
  count n l = 0 → ~(In n l).
Proof.
```

Admitted.

□

This small example shows how reflection gives us a small gain in convenience; in larger developments, using **reflect** consistently can often lead to noticeably shorter and clearer proof scripts. We'll see many more examples in later chapters and in *Programming Language Foundations*.

The use of the **reflect** property has been 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.

8.6 Additional Exercises

Exercise: 3 stars, standard, recommended (nostutter_defn) 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. (This is different from not containing duplicates: the sequence [1;4;1] repeats the element 1 but does not stutter.) The property "**nostutter** mylist" means that mylist does not stutter. Formulate an inductive definition for **nostutter**.

```
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.

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

□

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`.)

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

□

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, standard, 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.

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

Exercise: 5 stars, standard, 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).

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

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., $\forall\ x\ l,\ (\mathsf{In}\ x\ l) \vee \neg\ (\mathsf{In}\ x\ l)$. 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.

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

8.6.1 Extended Exercise: A Verified Regular-Expression Matcher

We have now defined a match relation over regular expressions and polymorphic lists. We can use such a definition to manually prove that a given regex matches a given string, but it does not give us a program that we can run to determine a match autmatically.

It would be reasonable to hope that we can translate the definitions of the inductive rules for constructing evidence of the match relation into cases of a recursive function reflects the relation by recursing on a given regex. However, it does not seem straightforward to define such a function in which the given regex is a recursion variable recognized by Coq. As a result, Coq will not accept that the function always terminates.

Heavily-optimized regex matchers match a regex by translating a given regex into a state machine and determining if the state machine accepts a given string. However, regex matching can also be implemented using an algorithm that operates purely on strings and regexes without defining and maintaining additional datatypes, such as state machines. We'll implemement such an algorithm, and verify that its value reflects the match relation.

We will implement a regex matcher that matches strings represented as lists of ASCII characters: `Require Export Coq.Strings.Ascii.`

```
Definition string := list ascii.
```

The Coq standard library contains a distinct inductive definition of strings of ASCII

characters. However, we will use the above definition of strings as lists as ASCII characters in order to apply the existing definition of the match relation.

We could also define a regex matcher over polymorphic lists, not lists of ASCII characters specifically. The matching algorithm that we will implement needs to be able to test equality of elements in a given list, and thus needs to be given an equality-testing function. Generalizing the definitions, theorems, and proofs that we define for such a setting is a bit tedious, but workable.

The proof of correctness of the regex matcher will combine properties of the regex-matching function with properties of the `match` relation that do not depend on the matching function. We'll go ahead and prove the latter class of properties now. Most of them have straightforward proofs, which have been given to you, although there are a few key lemmas that are left for you to prove.

Each provable `Prop` is equivalent to **True**. `Lemma` provable_equiv_true : $\forall$ (P : `Prop`), P $\rightarrow$ (P $\leftrightarrow$ **True**).

```
Proof.
  intros.
  split.
  - intros. constructor.
  - intros _. apply H.
Qed.
```

Each `Prop` whose negation is provable is equivalent to **False**. `Lemma` not_equiv_false : $\forall$ (P : `Prop`), $\neg P \rightarrow$ (P $\leftrightarrow$ **False**).

```
Proof.
  intros.
  split.
  - apply H.
  - intros. destruct H0.
Qed.
```

EmptySet matches no string. `Lemma` null_matches_none : $\forall$ (s : string), (s =~ EmptySet) $\leftrightarrow$ **False**.

```
Proof.
  intros.
  apply not_equiv_false.
  unfold not. intros. inversion H.
Qed.
```

EmptyStr only matches the empty string. `Lemma` empty_matches_eps : $\forall$ (s : string), s =~ EmptyStr $\leftrightarrow$ s = [ ].

```
Proof.
  split.
  - intros. inversion H. reflexivity.
  - intros. rewrite H. apply MEmpty.
Qed.
```

EmptyStr matches no non-empty string. Lemma empty_nomatch_ne : ∀ (a : ascii) s, (a :: s =~ EmptyStr) ↔ False.

```
Proof.
  intros.
  apply not_equiv_false.
  unfold not. intros. inversion H.
Qed.
```

Char a matches no string that starts with a non-a character. Lemma char_nomatch_char : ∀ (a b : ascii) s, b ≠ a → (b :: s =~ Char a ↔ False).

```
Proof.
  intros.
  apply not_equiv_false.
  unfold not.
  intros.
  apply H.
  inversion H0.
  reflexivity.
Qed.
```

If Char a matches a non-empty string, then the string's tail is empty. Lemma char_eps_suffix : ∀ (a : ascii) s, a :: s =~ Char a ↔ s = [ ].

```
Proof.
  split.
  - intros. inversion H. reflexivity.
  - intros. rewrite H. apply MChar.
Qed.
```

App *re0 re1* matches string *s* iff *s* = *s0* ++ *s1*, where *s0* matches *re0* and *s1* matches *re1*. Lemma app_exists : ∀ (s : string) re0 re1,

```
    s =~ App re0 re1 ↔
    ∃ s0 s1 , s = s0 ++ s1 ∧ s0 =~ re0 ∧ s1 =~ re1.
Proof.
  intros.
  split.
  - intros. inversion H. ∃ s1, s2. split.
    × reflexivity.
    × split. apply H3. apply H4.
  - intros [ s0 [ s1 [ Happ [ Hmat0 Hmat1 ] ] ] ].
    rewrite Happ. apply (MApp s0 _ s1 _ Hmat0 Hmat1).
Qed.
```

Exercise: 3 stars, standard, optional (app_ne) App *re0 re1* matches a::*s* iff *re0* matches the empty string and a::*s* matches *re1* or *s*=*s0*++*s1*, where a::*s0* matches *re0* and *s1* matches *re1*.

Even though this is a property of purely the match relation, it is a critical observation behind the design of our regex matcher. So (1) take time to understand it, (2) prove it, and (3) look for how you'll use it later. `Lemma app_ne : ∀ (a : ascii) s re0 re1,`

```
  a :: s =~ (App re0 re1) ↔
  ([ ] =~ re0 ∧ a :: s =~ re1) ∨
  ∃ s0 s1, s = s0 ++ s1 ∧ a :: s0 =~ re0 ∧ s1 =~ re1.
Proof.
  Admitted.
```

□

s matches Union $re0$ $re1$ iff s matches $re0$ or s matches $re1$. `Lemma union_disj : ∀ (s : string) re0 re1,`

```
  s =~ Union re0 re1 ↔ s =~ re0 ∨ s =~ re1.
Proof.
  intros. split.
  - intros. inversion H.
    + left. apply H2.
    + right. apply H1.
  - intros [ H | H ].
    + apply MUnionL. apply H.
    + apply MUnionR. apply H.
Qed.
```

Exercise: 3 stars, standard, optional (star_ne) a::s matches Star re iff $s = s0 ++ s1$, where a::$s0$ matches re and $s1$ matches Star re. Like app_ne, this observation is critical, so understand it, prove it, and keep it in mind.

Hint: you'll need to perform induction. There are quite a few reasonable candidates for Prop's to prove by induction. The only one that will work is splitting the iff into two implications and proving one by induction on the evidence for a :: s =~ Star re. The other implication can be proved without induction.

In order to prove the right property by induction, you'll need to rephrase a :: s =~ Star re to be a Prop over general variables, using the *remember* tactic.

```
Lemma star_ne : ∀ (a : ascii) s re,
  a :: s =~ Star re ↔
  ∃ s0 s1, s = s0 ++ s1 ∧ a :: s0 =~ re ∧ s1 =~ Star re.
Proof.
  Admitted.
```

□

The definition of our regex matcher will include two fixpoint functions. The first function, given regex re, will evaluate to a value that reflects whether re matches the empty string. The function will satisfy the following property: `Definition refl_matches_eps m :=`
`∀ re : @reg_exp ascii, reflect ([ ] =~ re) (m re).`

Exercise: 2 stars, standard, optional (match_eps) Complete the definition of match_eps so that it tests if a given regex matches the empty string: `Fixpoint match_eps` (*re*: @**reg_exp** **ascii**) : **bool**

. *Admitted.*

□

Exercise: 3 stars, standard, optional (match_eps_refl) Now, prove that match_eps indeed tests if a given regex matches the empty string. (Hint: You'll want to use the reflection lemmas ReflectT and ReflectF.) `Lemma` match_eps_refl : refl_matches_eps *match_eps*.
`Proof`.

Admitted.

□

We'll define other functions that use match_eps. However, the only property of match_eps that you'll need to use in all proofs over these functions is match_eps_refl.

The key operation that will be performed by our regex matcher will be to iteratively construct a sequence of regex derivatives. For each character a and regex *re*, the derivative of *re* on a is a regex that matches all suffixes of strings matched by *re* that start with a. I.e., *re'* is a derivative of *re* on a if they satisfy the following relation:

`Definition` is_der *re* (*a* : **ascii**) *re'* :=
∀ *s*, *a* :: *s* =~ *re* ↔ *s* =~ *re'*.

A function d derives strings if, given character a and regex *re*, it evaluates to the derivative of *re* on a. I.e., d satisfies the following property: `Definition` derives *d* := ∀ *a re*, is_der *re* *a* (*d* *a* *re*).

Exercise: 3 stars, standard, optional (derive) Define derive so that it derives strings. One natural implementation uses match_eps in some cases to determine if key regex's match the empty string. `Fixpoint` derive (*a* : **ascii**) (*re* : @**reg_exp** **ascii**) : @**reg_exp** **ascii**

. *Admitted.*

□

The derive function should pass the following tests. Each test establishes an equality between an expression that will be evaluated by our regex matcher and the final value that must be returned by the regex matcher. Each test is annotated with the match fact that it reflects. `Example` c := ascii_of_nat 99.
`Example` d := ascii_of_nat 100.

"c" =~ EmptySet: `Example` test_der0 : *match_eps* (*derive* c (EmptySet)) = true.
`Proof`.

Admitted.

"c" =~ Char c: `Example` test_der1 : *match_eps* (*derive* c (Char c)) = true.
`Proof`.

Admitted.

"c" =~ Char d: `Example` test_der2 : *match_eps* (*derive* c (Char d)) = false.

```
Proof.
```
Admitted.

"c" =~ App (Char c) EmptyStr: `Example test_der3 : match_eps (derive c (App (Char c) EmptyStr)) = true.`
```
Proof.
```
Admitted.

"c" =~ App EmptyStr (Char c): `Example test_der4 : match_eps (derive c (App EmptyStr (Char c))) = true.`
```
Proof.
```
Admitted.

"c" =~ Star c: `Example test_der5 : match_eps (derive c (Star (Char c))) = true.`
```
Proof.
```
Admitted.

"cd" =~ App (Char c) (Char d): `Example test_der6 :`
`match_eps (derive d (derive c (App (Char c) (Char d)))) = true.`
```
Proof.
```
Admitted.

"cd" =~ App (Char d) (Char c): `Example test_der7 :`
`match_eps (derive d (derive c (App (Char d) (Char c)))) = false.`
```
Proof.
```
Admitted.

Exercise: 4 stars, standard, optional (derive_corr) Prove that derive in fact always derives strings.

Hint: one proof performs induction on *re*, although you'll need to carefully choose the property that you prove by induction by generalizing the appropriate terms.

Hint: if your definition of derive applies match_eps to a particular regex *re*, then a natural proof will apply match_eps_refl to *re* and destruct the result to generate cases with assumptions that the *re* does or does not match the empty string.

Hint: You can save quite a bit of work by using lemmas proved above. In particular, to prove many cases of the induction, you can rewrite a `Prop` over a complicated regex (e.g., *s* =~ Union *re0 re1*) to a Boolean combination of `Prop`'s over simple regex's (e.g., *s* =~ *re0* ∨ *s* =~ *re1*) using lemmas given above that are logical equivalences. You can then reason about these `Prop`'s naturally using `intro` and `destruct`. `Lemma derive_corr : derives derive.`
```
Proof.
```
Admitted.

□

We'll define the regex matcher using derive. However, the only property of derive that you'll need to use in all proofs of properties of the matcher is derive_corr.

A function *m* matches regexes if, given string *s* and regex *re*, it evaluates to a value that reflects whether *s* is matched by *re*. I.e., *m* holds the following property: `Definition`

matches_regex m : Prop :=
  $\forall$ (s : string) re, **reflect** (s =~ re) (m s re).

Exercise: 2 stars, standard, optional (regex_match) Complete the definition of regex_match so that it matches regexes. Fixpoint regex_match (s : string) (re : @**reg_exp** ascii) : **bool**
  . *Admitted.*
  □

Exercise: 3 stars, standard, optional (regex_refl) Finally, prove that regex_match in fact matches regexes.

Hint: if your definition of regex_match applies match_eps to regex re, then a natural proof applies match_eps_refl to re and destructs the result to generate cases in which you may assume that re does or does not match the empty string.

Hint: if your definition of regex_match applies derive to character x and regex re, then a natural proof applies derive_corr to x and re to prove that x :: s =~ re given s =~ derive x re, and vice versa. Theorem regex_refl : matches_regex *regex_match*.
Proof.
  Admitted.
  □

Chapter 9

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.

9.1 The Coq Standard Library

One small digression before we begin...

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (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.

```
From Coq Require Import Arith.Arith.
From Coq Require Import Bool.Bool.
Require Export Coq.Strings.String.
From Coq Require Import Logic.FunctionalExtensionality.
From Coq Require Import Lists.List.
Import ListNotations.
```

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.

9.2 Identifiers

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

To compare strings, we define the function eqb_string, which internally uses the function string_dec from Coq's string library.

```
Definition eqb_string (x y : string) : bool :=
  if string_dec x y then true else false.
```

(The function string_dec comes from Coq's string library. If you check the result type of string_dec, you'll see that it does not actually return a **bool**, but rather a type that looks like $\{x = y\} + \{x \neq y\}$, 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**.)

Now we need a few basic properties of string equality... `Theorem eqb_string_refl : ∀ s : string, true = eqb_string s s.`

```
Proof. intros s. unfold eqb_string. destruct (string_dec s s) as [|Hs].
  - reflexivity.
  - destruct Hs. reflexivity.
Qed.
```

The following useful property follows from an analogous lemma about strings:

```
Theorem eqb_string_true_iff : ∀ x y : string,
    eqb_string x y = true ↔ x = y.
Proof.
   intros x y.
   unfold eqb_string.
   destruct (string_dec x y) as [|Hs].
   - subst. split. reflexivity. reflexivity.
   - split.
     + intros contra. discriminate contra.
     + intros H. rewrite H in Hs. destruct Hs. reflexivity.
Qed.
```

Similarly:

```
Theorem eqb_string_false_iff : ∀ x y : string,
    eqb_string x y = false ↔ x ≠ y.
Proof.
  intros x y. rewrite ← eqb_string_true_iff.
  rewrite not_true_iff_false. reflexivity. Qed.
```

This handy variant follows just by rewriting:

```
Theorem false_eqb_string : ∀ x y : string,
    x ≠ y → eqb_string x y = false.
Proof.
  intros x y. rewrite eqb_string_false_iff.
  intros H. apply H. Qed.
```

9.3 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) := string → A.
```

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

```
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 : string) (v : A) :=
  fun x' ⇒ if eqb_string x x' then v else m x'.
```

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

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

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

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

First, we will use the following notation to create an empty total map with a default value. `Notation "'_' '!->' v" := (t_empty v)`
(`at level` 100, `right associativity`).

```
Example example_empty := (_ !-> false).
```

We then introduce a convenient notation for extending an existing map with some bindings. `Notation "x '!->' v ';' m" := (t_update m x v)`
(`at level` 100, *v* `at` *next* `level`, `right associativity`).

The examplemap above can now be defined as follows:

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

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

```
Example update_example1 : examplemap' "baz" = false.
Proof. reflexivity. Qed.

Example update_example2 : examplemap' "foo" = true.
Proof. reflexivity. Qed.

Example update_example3 : examplemap' "quux" = false.
Proof. reflexivity. Qed.

Example update_example4 : examplemap' "bar" = true.
Proof. reflexivity. Qed.
```

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

```
Lemma t_apply_empty : ∀ (A : Type) (x : string) (v : A),
    (_ !-> v) x = v.
Proof.
  Admitted.
```

□

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

```
Lemma t_update_eq : ∀ (A : Type) (m : total_map A) x v,
  (x !-> v ; m) x = v.
Proof.
  Admitted.
```
□

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

```
Theorem t_update_neq : ∀ (A : Type) (m : total_map A) x1 x2 v,
  x1 ≠ x2 →
  (x1 !-> v ; m) x2 = m x2.
Proof.
  Admitted.
```
□

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

```
Lemma t_update_shadow : ∀ (A : Type) (m : total_map A) x v1 v2,
  (x !-> v2 ; x !-> v1 ; m) = (x !-> v2 ; m).
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 eqb_id.

Exercise: 2 stars, standard, optional (eqb_stringP) Use the proof of eqbP in chapter IndProp as a template to prove the following:

```
Lemma eqb_stringP : ∀ x y : string,
  reflect (x = y) (eqb_string x y).
Proof.
  Admitted.
```
□

Now, given strings *x1* and *x2*, we can use the tactic `destruct` (eqb_stringP *x1 x2*) to simultaneously perform case analysis on the result of eqb_string *x1 x2* and generate hypotheses about the equality (in the sense of =) of *x1* and *x2*.

Exercise: 2 stars, standard (t_update_same) With the example in chapter IndProp as a template, use eqb_stringP 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 : ∀ (A : Type) (m : total_map A) x,
    (x !-> m x ; m) = m.
Proof.
    Admitted.
    □
```

Exercise: 3 stars, standard, recommended (t_update_permute) Use eqb_stringP 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 : ∀ (A : Type) (m : total_map A)
                                        v1 v2 x1 x2,
    x2 ≠ x1 →
    (x1 !-> v1 ; x2 !-> v2 ; m)
    =
    (x2 !-> v2 ; x1 !-> v1 ; m).
Proof.
    Admitted.
    □
```

9.4 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 : string) (v : A) :=
  (x !-> Some v ; m).
```

We introduce a similar notation for partial maps: `Notation "x '|->' v ';' m" := (update m x v)`
`  (at level 100, v at next level, right associativity).`

We can also hide the last case when it is empty. `Notation "x '|->' v" := (update empty x v) (at level 100).`

```
Example examplepmap :=
  ("Church" |-> true ; "Turing" |-> false).
```

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

```
Lemma apply_empty : ∀ (A : Type) (x : string),
    @empty A x = None.
Proof.
  intros. unfold empty. rewrite t_apply_empty.
  reflexivity.
Qed.

Lemma update_eq : ∀ (A : Type) (m : partial_map A) x v,
    (x |-> v ; m) x = Some v.
Proof.
  intros. unfold update. rewrite t_update_eq.
  reflexivity.
Qed.

Theorem update_neq : ∀ (A : Type) (m : partial_map A) x1 x2 v,
    x2 ≠ x1 →
    (x2 |-> v ; m) x1 = m x1.
Proof.
  intros A m x1 x2 v H.
  unfold update. rewrite t_update_neq. reflexivity.
  apply H. Qed.

Lemma update_shadow : ∀ (A : Type) (m : partial_map A) x v1 v2,
    (x |-> v2 ; x |-> v1 ; m) = (x |-> v2 ; m).
Proof.
  intros A m x v1 v2. unfold update. rewrite t_update_shadow.
  reflexivity.
Qed.

Theorem update_same : ∀ (A : Type) (m : partial_map A) x v,
    m x = Some v →
    (x |-> v ; m) = m.
Proof.
  intros A m x v H. unfold update. rewrite ← H.
  apply t_update_same.
Qed.

Theorem update_permute : ∀ (A : Type) (m : partial_map A)
                           x1 x2 v1 v2,
```

```
    x2 ≠ x1 →
    (x1 |-> v1 ; x2 |-> v2 ; m) = (x2 |-> v2 ; x1 |-> v1 ; m).
Proof.
  intros A m x1 x2 v1 v2. unfold update.
  apply t_update_permute.
Qed.
```

Chapter 10

ProofObjects: The Curry-Howard Correspondence

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export IndProp.
```

"*Algorithms are the computational content of proofs.*" –Robert Harper

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 **even**), 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 $\rightarrow$ 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 $\mathsf{A} \rightarrow \mathsf{B}$, 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 **even** property.

```
Print even.
```

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 **even** declares that `ev_0` : **even** 0. Instead of "`ev_0` has type **even** 0," we can say that "`ev_0` is a proof of **even** 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 (in Bib.v) for a brief history and 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 **even** n – and yields evidence for the proposition **even** (S (S n))."

Now let's look again at a previous proof involving **even**.

```
Theorem ev_4 : even 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.
```

Indeed, 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 **even** 2 and **even** 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, even n -> even (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': even 4.
Proof.
  apply (ev_SS 2 (ev_SS 0 ev_0)).
Qed.
```

10.1 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'' : even 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 this evidence.

```
Definition ev_4''' : even 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: 2 stars, standard (eight_is_even) Give a tactic proof and a proof object showing that **even** 8.

```
Theorem ev_8 : even 8.
Proof.
    Admitted.

Definition ev_8' : even 8
  . Admitted.
```

□

10.2 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 `Inductively` 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 `Inductively` defined propositions, and... functions!

For example, consider this statement:

```
Theorem ev_plus4 : ∀ n, even n → even (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, **even** n → **even** $(4 + n)$ – 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, even n → even (4 + n) :=
  fun (n : nat) ⇒ fun (H : even 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 : even n)
                      : even (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 interesting point becomes apparent: The second argument's type, **even** 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:

forall (x:nat), nat = forall (_:nat), nat = nat -> nat

For example, consider this proposition:

```
Definition ev_plus2 : Prop :=
  ∀ n, ∀ (E : even n), even (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, ∀ (_ : even n), even (n + 2).
```

Or, equivalently, we can write it in more familiar notation:

```
Definition ev_plus2'' : Prop :=
  ∀ n, even n → even (n + 2).
```

In general, "$P \to Q$" is just syntactic sugar for "$\forall\ (\_{:}P),\ Q$".

10.3 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.

10.4 Logical Connectives as Inductive Types

Inductive definitions are powerful enough to express most of the connectives we have seen so far. Indeed, only universal quantification (with implication as a special case) is built into Coq; all the others are defined inductively. We'll see these definitions in this section.

```
Module Props.
```

10.4.1 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 similarity 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 exactly 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) : Q ∧ P :=
  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, standard, optional (conj_fact) Construct a proof object demonstrating the following proposition.

```
Definition conj_fact : ∀ P Q R, P ∧ Q → Q ∧ R → P ∧ R
  . Admitted.
```
□

10.4.2 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, standard, 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 : ∀ P Q, P ∨ Q → Q ∨ P
  . Admitted.
```
□

10.4.3 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 $\exists\ x,\ P\ x$ desugars to an expression involving **ex**:

```
Check ex (fun n ⇒ even n).
```

Here's how to define an explicit proof object involving **ex**:

```
Definition some_nat_is_even : ∃ n, even n :=
  ex_intro even 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

Exercise: 2 stars, standard, optional (ex_ev_Sn) Complete the definition of the following proof object:

```
Definition ex_ev_Sn : ex (fun n ⇒ even (S n))
  . Admitted.
```

□

10.4.4 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.
```

10.5 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 slight 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 members of this family: applying the constructor eq_refl to a type X and a single value $x : X$, which yields evidence that x is equal to x.

Other types of the form **eq** x y where x and y are not the same are thus uninhabited.

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 shorthand for `apply eq_refl`.

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:Type) (x:X), []++[x] == x::[] :=
  fun (X:Type) (x:X) ⇒ eq_refl [x].
```

Exercise: 2 stars, standard (equality__leibniz_equality) The inductive definition of equality implies *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 equality__leibniz_equality : ∀ (X : Type) (x y: X),
  x == y → ∀ P:X→Prop, P x → P y.
Proof.
  Admitted.
```

□

Exercise: 5 stars, standard, optional (leibniz_equality__equality) Show that, in fact, the inductive definition of equality is *equivalent* to Leibniz equality:

```
Lemma leibniz_equality__equality : ∀ (X : Type) (x y: X),
  (∀ P:X→Prop, P x → P y) → x == y.
Proof.
```

Admitted.

□

`End` MyEquality.

10.5.1 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 $\mathsf{S}\ n = \mathsf{O}$), 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 $\mathsf{eq\_refl}$ 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.

Chapter 11

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.

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export ProofObjects.
```

11.1 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, standard, 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
  | no.

Check yesno_ind.
```

Exercise: 1 star, standard, 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
  | green
  | blue.
Check rgb_ind.
```

□

Here's another example, this time with one of the constructors taking some arguments.

```
Inductive natlist : Type :=
  | nnil
  | ncons (n : nat) (l : natlist).

Check natlist_ind.
```

Exercise: 1 star, standard, optional (natlist1) Suppose we had written the above definition a little differently:

```
Inductive natlist1 : Type :=
  | nnil1
  | nsnoc1 (l : natlist1) (n : nat).
```

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, standard, 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
 | bleaf (yn : yesno)
 | nbranch (yn : yesno) (t1 t2 : byntree).
```
□

Exercise: 1 star, standard, 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 :=
```

.

□

11.2 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, standard, 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 : X)
  | node (t1 t2 : tree X).
Check tree_ind.
```

□

Exercise: 1 star, standard, 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, standard, 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, standard, optional (foo') Consider the following inductive definition:

```
Inductive foo' (X:Type) : Type :=
```

```
| C1 (l : list X) (f : foo' X)
| C2.
```

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, ----------

□

11.3 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 "$\forall\ n,\ n \times 0 = 0$," we can write it as "$\forall\ n$, 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 $\forall\ n$, 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 $\forall\ n'$, P_m0r $n' \rightarrow$ 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'*.

11.4 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, standard, 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.

11.5 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 **even** are a tiny bit more complicated. As with all induction principles, we want to use the induction principle on **even** to prove things by inductively considering the possible shapes that something in **even** 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 **even**...

Inductive even : nat -> Prop := | ev_0 : even 0 | ev_SS : forall n : nat, even n -> even (S (S n)).

...to give rise to an induction principle that looks like this...

ev_ind_max : forall P : (forall n : nat, even n -> Prop), P O ev_0 -> (forall (m : nat) (E : even m), P m E -> P (S (S m)) (ev_SS m E)) -> forall (n : nat) (E : even n), P n E

... because:

- Since **even** is indexed by a number n (every **even** 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 (**even** 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 **even**:

```
Check even_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 **even'** (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 **even**: `Theorem ev_ev' : ∀ n, even n → even' n.`

```
Proof.
  apply even_ind.
  -
    apply even'_0.
  -
    intros m Hm IH.
    apply (even'_sum 2 m).
    + apply even'_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 (H : 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.
```

11.6 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`).

11.6.1 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'.

    □

11.6.2 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 $z \to P$ 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 $n \leq m$ and $m \leq o$, then $n \leq o$.

  Proof: By induction on a derivation of $m \leq o$.

  - Suppose the final rule used to show $m \leq o$ is le_n. Then $m = o$ and we must show that $n \leq m$, which is immediate by hypothesis.
  - Suppose the final rule used to show $m \leq o$ is le_S. Then $o =$ S o' for some o' with $m \leq o'$. We must show that $n \leq$ S o'. By induction hypothesis, $n \leq o'$. But then, by le_S, $n \leq$ S o'. □

Chapter 12

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 of *Programming Language Foundations*), 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.

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export IndProp.
```

12.1 Relations

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* ≤ *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 ≤.)

12.2 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 ≤ relation on numbers is not a partial function. (Assume, for a contradiction, that ≤ is a partial function. But then, since 0 ≤ 0 and 0 ≤ 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. }
  discriminate Nonsense. Qed.
```

Exercise: 2 stars, standard, optional (total_relation_not_partial) Show that the *total_relation* defined in (an exercise in) IndProp is not a partial function.

Exercise: 2 stars, standard, optional (empty_relation_partial) Show that the *empty_relation* defined in (an exercise in) IndProp 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, standard, optional (le_trans_hard_way) 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, standard, optional (lt_trans") 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, standard, optional (le_S_n)

```
Theorem le_S_n : ∀ n m,
  (S n ≤ S m) → (n ≤ m).
Proof.
```

Admitted.

□

Exercise: 2 stars, standard, optional (le_Sn_n_inf) Provide an informal proof of the following theorem:

Theorem: For every n, $\neg$ (S $n \leq n$)

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, standard, optional (le_Sn_n)

```
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) :=
  ∀ a b : X, (R a b) → (R b a).
```

Exercise: 2 stars, standard, optional (le_not_symmetric) `Theorem le_not_symmetric :`

```
  ¬ (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) :=
  ∀ a b : X, (R a b) → (R b a) → a = b.
```

Exercise: 2 stars, standard, optional (le_antisymmetric) `Theorem le_antisymmetric :`

```
  antisymmetric le.
Proof.
```

Admitted.

□

Exercise: 2 stars, standard, optional (le_step) `Theorem le_step : ∀ n m p,`

```
  n < m →
  m ≤ S p →
  n ≤ 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) ∧ (symmetric R) ∧ (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.
```

12.3 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 (H : R x y) : clos_refl_trans R x y
    | rt_refl x : clos_refl_trans R x x
    | rt_trans x y z
          (Hxy : clos_refl_trans R x y)
          (Hyz : 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)
      (Hxy : R x y) (Hrest : 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, standard, 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, standard, optional (rtc_rsc_coincide)

```
Theorem rtc_rsc_coincide :
        ∀ (X:Type) (R: relation X) (x y : X),
  clos_refl_trans R x y ↔ clos_refl_trans_1n R x y.
Proof.
  Admitted.
```

□

Chapter 13

Imp: Simple Imperative Programs

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

Z ::= X;; Y ::= 1;; WHILE ~(Z = 0) DO Y ::= Y * Z;; Z ::= Z - 1 END

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

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Bool.Bool.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.Arith.
From Coq Require Import Arith.EqNat.
From Coq Require Import omega.Omega.
From Coq Require Import Lists.List.
From Coq Require Import Strings.String.
Import ListNotations.
```

```
From LF Require Import Maps.
```

13.1 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.

13.1.1 Syntax

```
Module AExp.
```

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

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

Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).
```

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

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

The optional chapter ImpParser develops a simple lexical analyzer and parser that can perform this translation. You do *not* need to understand that chapter to understand this one, but if you haven't already taken a course where these techniques are covered (e.g., a 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 | ~ b | b && 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 with an infix +) while leaving other aspects of lexical analysis and parsing (like the relative precedence of +, -, and ×, the use of parens to group subexpressions, etc.) unspecified. Some additional information – and human intelligence – would be required to turn this description into a formal definition, e.g., for implementing a compiler.

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

- Conversely, the BNF version is lighter and easier to read. Its informality makes it flexible, a big advantage in situations like discussions at the blackboard, where conveying general ideas is more important than 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 kind of BNF they're using because there is no need to: a rough-and-ready informal understanding is all that's important.

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

13.1.2 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 ⇒ (aeval a1) =? (aeval a2)
  | BLe a1 a2 ⇒ (aeval a1) <=? (aeval a2)
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

13.1.3 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 eqn:Ea1.
    + destruct n eqn:En.
      × simpl. apply IHa2.
      × simpl. rewrite IHa2. reflexivity.
    +
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
    +
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
    +
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
  -
    simpl. rewrite IHa1. rewrite IHa2. reflexivity.
  -
    simpl. rewrite IHa1. rewrite IHa2. reflexivity. Qed.
```

13.2 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.

13.2.1 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 (rather than 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, 0 <=? n = true.
Proof.
  intros.
  destruct n.
    - simpl. reflexivity.
    - simpl. reflexivity.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma foo' : ∀ n, 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 eqn:Ea1;

      try (simpl; simpl in IHa1; rewrite IHa1;
           rewrite IHa2; reflexivity).
    + destruct n eqn:En;
      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 $n = 0$, 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, 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 (i.e., it does not terminate), this simply means that we have failed to construct a proof, not that we have constructed a wrong one.

Exercise: 3 stars, standard (optimize_0plus_b_sound) 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 that performs this 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, standard, optional (optimize) *Design exercise*: The optimization implemented by our optimize_0plus function is only one of many possible optimizations on arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct. (You will probably find it easiest to start small – add just a single, simple optimization and its correctness proof – and build up to something more interesting incrementally.)

13.2.2 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 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`."

13.2.3 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 by William Pugh *Pugh* 1991 (in Bib.v).

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 ≠) and ordering (≤), and

- the logical connectives ∧, ∨, ¬, and →,

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

```
Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. omega.
Qed.
```

(Note the `From` *Coq* `Require Import` omega.Omega. at the top of the file.)

13.2.4 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: For a variable x, find an assumption $x =$ `e` or `e` $= x$ in the context, replace x with `e` throughout the context and current goal, and clear the assumption.
- `subst`: Substitute away *all* assumptions of the form $x =$ `e` or `e` $= x$ (where x is a variable).
- `rename`... *into*...: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named x, then `rename` x *into* y will change all occurrences of x to y.
- `assumption`: Try to find a hypothesis H in the context that exactly matches the goal; if one is found, 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 of all of these as we go along.

13.3 Evaluation as a Relation

We have presented aeval and beval as functions defined by `Fixpoint`s. 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 :
      aevalR (ANum n) n
  | E_APlus (e1 e2: aexp) (n1 n2: nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2: aexp) (n1 n2: nat) :
      aevalR e1 n1 →
```

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

Module TooHardToRead.

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

Instead, we've chosen to leave the hypotheses anonymous, just giving their types. This style gives us less control over the names that Coq chooses during proofs involving **aevalR**, but it makes the definition itself quite a bit lighter.

```
End TooHardToRead.
```

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

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

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

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

13.3.1 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 IndProp 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 →. 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

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

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

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

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

□

13.3.2 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, standard (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.
```

13.3.3 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 with division:

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

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

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

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

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

```
End AEVALR_DIVISION.

Module AEVALR_EXTENDED.
```

Or suppose that we want to extend the arithmetic operations by a nondeterministic number generator *any* that, when evaluated, may yield any number. (Note that this is not the same as making a *probabilistic* choice among all possible numbers – we're not specifying any particular probability distribution for the results, just saying what results are *possible*.)

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

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

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

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

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

End AEVALR_EXTENDED.
```

At this point you maybe wondering: which style should I use by default? In the examples we've just seen, relational definitions turned out to be more useful than functional ones. For situations like these, where the thing being defined is not easy to express as a function, or indeed where it is *not* a function, there is no real choice. But what about when both styles are workable?

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

Another is that 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" from Gallina to executable code in OCaml or Haskell.

Ultimately, the choice often comes down to either the specifics of a particular situation or simply a question of taste. Indeed, in large 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.

13.4 Expressions With Variables

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.

13.4.1 States

Since we'll want to look variables up to find out their current values, we'll reuse maps from the Maps chapter, and strings will be used to represent 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 strings to **nat**, and will use 0 as default value in the store. For more complex programming languages, the state might have more structure.

```
Definition state := total_map nat.
```

13.4.2 Syntax

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

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

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

```
Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".
```

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

The definition of **bexp**s is unchanged (except that it now refers to the new **aexp**s):

```
Inductive bexp : Type :=
```

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

13.4.3 Notations

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

You do not need to understand exactly what these declarations do. Briefly, though, the `Coercion` declaration in Coq stipulates that a function (or constructor) can be implicitly used by the type system to coerce a value of the input type to a value of the output type. For instance, the coercion declaration for AId allows us to use plain strings when an **aexp** is expected; the string will implicitly be wrapped with AId.

The notations below are declared in specific *notation scopes*, in order to avoid conflicts with other interpretations of the same symbols. Again, it is not necessary to understand the details, but it is important to recognize that we are defining *new* intepretations for some familiar operators like +, -, ×, =., [<=], *etc.*

```
Coercion AId : string >-> aexp.
Coercion ANum : nat >-> aexp.

Definition bool_to_bexp (b : bool) : bexp :=
  if b then BTrue else BFalse.
Coercion bool_to_bexp : bool >-> bexp.

Bind Scope imp_scope with aexp.
Bind Scope imp_scope with bexp.
Delimit Scope imp_scope with imp.

Notation "x + y" := (APlus x y) (at level 50, left associativity) : imp_scope.
Notation "x - y" := (AMinus x y) (at level 50, left associativity) : imp_scope.
Notation "x * y" := (AMult x y) (at level 40, left associativity) : imp_scope.
Notation "x <= y" := (BLe x y) (at level 70, no associativity) : imp_scope.
Notation "x = y" := (BEq x y) (at level 70, no associativity) : imp_scope.
Notation "x && y" := (BAnd x y) (at level 40, left associativity) : imp_scope.
Notation "'~' b" := (BNot b) (at level 75, right associativity) : imp_scope.

Definition example_aexp := (3 + (X × 2))%imp : aexp.
Definition example_bexp := (true && ~(X ≤ 4))%imp : bexp.
```

One downside of these coercions is that they can make it a little harder for humans to calculate the types of expressions. If you get confused, try doing `Set Printing` *Coercions* to see exactly what is going on.

```
Set Printing Coercions.
```

```
Print example_bexp.

Unset Printing Coercions.
```

13.4.4 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 ⇒ (aeval st a1) =? (aeval st a2)
  | BLe a1 a2 ⇒ (aeval st a1) <=? (aeval st a2)
  | BNot b1 ⇒ negb (beval st b1)
  | BAnd b1 b2 ⇒ andb (beval st b1) (beval st b2)
  end.
```

We specialize our notation for total maps to the specific case of states, i.e. using (_ !-> 0) as empty state.

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

Now we can add a notation for a "singleton state" with just one variable bound to a value.

```
Notation "a '!->' x" := (t_update empty_st a x) (at level 100).

Example aexp1 :
    aeval (X !-> 5) (3 + (X × 2))%imp
  = 13.
Proof. reflexivity. Qed.

Example bexp1 :
    beval (X !-> 5) (true && ~(X ≤ 4))%imp
  = true.
Proof. reflexivity. Qed.
```

13.5 Commands

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

13.5.1 Syntax

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

c ::= SKIP | x ::= a | c ;; c | TEST b THEN c ELSE c FI | WHILE b DO c END

(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 *TEST* to avoid conflicting with the `if` and *IF* notations from the standard library.) For example, here's factorial in Imp:

Z ::= X;; Y ::= 1;; WHILE ~(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
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

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

```
Bind Scope imp_scope with com.
Notation "'SKIP'" :=
  CSkip : imp_scope.
Notation "x '::=' a" :=
  (CAss x a) (at level 60) : imp_scope.
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity) : imp_scope.
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity) : imp_scope.
Notation "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity) : imp_scope.
```

For example, here is the factorial function again, written as a formal definition to Coq:

```
Definition fact_in_coq : com :=
  (Z ::= X;;
  Y ::= 1;;
  WHILE ~(Z = 0) DO
```

```
  Y ::= Y × Z;;
  Z ::= Z - 1
END)%imp.
```

13.5.2 Desugaring notations

Coq offers a rich set of features to manage the increasing complexity of the objects we work with, such as coercions and notations. However, their heavy usage can make for quite overwhelming syntax. It is often instructive to "turn off" those features to get a more elementary picture of things, using the following commands:

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

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

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

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

13.5.3 The `Locate` command

Finding notations

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

```
Locate ";;".
```

```
Locate "WHILE".
```

Finding identifiers

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

13.5.4 More Examples

Assignment:

```
Definition plus2 : com :=
  X ::= X + 2.

Definition XtimesYinZ : com :=
  Z ::= X × Y.

Definition subtract_slowly_body : com :=
  Z ::= Z - 1 ;;
  X ::= X - 1.
```

Loops

```
Definition subtract_slowly : com :=
  (WHILE ~(X = 0) DO
    subtract_slowly_body
  END)%imp.

Definition subtract_3_from_5_slowly : com :=
  X ::= 3 ;;
  Z ::= 5 ;;
  subtract_slowly.
```

An infinite loop:

```
Definition loop : com :=
  WHILE true DO
    SKIP
  END.
```

13.6 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...

13.6.1 Evaluation as a Function (Failed Attempt)

Here's an attempt at defining an evaluation function for commands, omitting the *WHILE* case.

The following declaration is needed to be able to use the notations in match patterns.

```
Open Scope imp_scope.
Fixpoint ceval_fun_no_while (st : state) (c : com)
```

```
                              : state :=
  match c with
    | SKIP ⇒
        st
    | x ::= a1 ⇒
        (x !-> (aeval st a1) ; st)
    | c1 ;; c2 ⇒
        let st' := ceval_fun_no_while st c1 in
        ceval_fun_no_while st' c2
    | TEST 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.
Close Scope imp_scope.
```

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).

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

Operational Semantics

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

(E_Skip) st = *SKIP* => st
  aeval st a1 = n

(E_Ass) st = *x* := *a1* => (x !-> n ; st)
  st = c1 => st' st' = c2 => st"

(E_Seq) st = c1;;c2 => st"
  beval st b1 = true st = c1 => st'

(E_IfTrue) st = *TEST b1 THEN* c1 *ELSE* c2 *FI* => st'
  beval st b1 = false st = c2 => st'

(E_IfFalse) st = *TEST b1 THEN* c1 *ELSE* c2 *FI* => st'
  beval st b = false

(E_WhileFalse) st = *WHILE* b *DO* c *END* => st
  beval st b = true st = c => st' st' = *WHILE* b *DO* c *END* => st"

(E_WhileTrue) st = *WHILE* b *DO* c *END* => st"

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

```
Reserved Notation "st '=[' c ']=>' st'"
                  (at level 40).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ SKIP ]=> st
  | E_Ass : ∀ st a1 n x,
      aeval st a1 = n →
      st =[ x ::= a1 ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ;; c2 ]=> st''
```

```
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ TEST b THEN c1 ELSE c2 FI ]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]=> st' →
      st =[ TEST b THEN c1 ELSE c2 FI ]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ WHILE b DO c END ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ WHILE b DO c END ]=> st'' →
      st =[ WHILE b DO c END ]=> st''

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

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:
  empty_st =[
     X ::= 2;;
     TEST X ≤ 1
       THEN Y ::= 3
       ELSE Z ::= 4
     FI
  ]=> (Z !-> 4 ; X !-> 2).
Proof.
  apply E_Seq with (X !-> 2).
  -
    apply E_Ass. reflexivity.
  -
    apply E_IfFalse.
    reflexivity.
    apply E_Ass. reflexivity.
Qed.
```

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

```
  empty_st =[
    X ::= 0;; Y ::= 1;; Z ::= 2
```

```
  ]=> (Z !-> 2 ; Y !-> 1 ; X !-> 0).
Proof.
```
Admitted.
□

Exercise: 3 stars, standard, optional (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 :
  (X !-> 2) =[
```
pup_to_n
```
  ]=> (X !-> 0 ; Y !-> 3 ; X !-> 1 ; Y !-> 2 ; Y !-> 0 ; X !-> 2).
Proof.
```
Admitted.
□

13.6.3 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,
      st =[ c ]=> st1 →
      st =[ c ]=> 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. discriminate H5.
- 
    rewrite H in H5. discriminate H5.
- 
    apply IHE1. assumption.
- 
  reflexivity.
- 
  rewrite H in H2. discriminate H2.
- 
  rewrite H in H4. discriminate H4.
- 
    assert (st' = st'0) as EQ1.
    { apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption. Qed.
```

13.7 Reasoning About Imp Programs

We'll get deeper into more systematic and powerful techniques for reasoning about Imp programs in *Programming Language Foundations*, but we can get some distance just working with the bare definitions. This section explores some examples.

```
Theorem plus2_spec : ∀ st n st',
  st X = n →
  st =[ plus2 ]=> 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, standard, recommended (XtimesYinZ_spec) State and prove a specification of XtimesYinZ.

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

□

Exercise: 3 stars, standard, recommended (loop_never_stops) Theorem loop_never_stops
: ∀ *st st'*,
  ~(*st* =[loop]=> *st'*).
Proof.
  intros *st st' contra*. unfold loop in *contra*.
  remember (WHILE true DO SKIP END)%*imp* 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 `discriminate`).

  Admitted.
□

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

```
Open Scope imp_scope.
Fixpoint no_whiles (c : com) : bool :=
  match c with
  | SKIP ⇒
      true
  | _ ::= _ ⇒
      true
  | c1 ;; c2 ⇒
      andb (no_whiles c1) (no_whiles c2)
  | TEST _ THEN ct ELSE cf FI ⇒
      andb (no_whiles ct) (no_whiles cf)
  | WHILE _ DO _ END ⇒
      false
  end.
Close Scope imp_scope.
```

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

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

.

```
Theorem no_whiles_eqv:
  ∀ c, no_whiles c = true ↔ no_whilesR c.
Proof.
  Admitted.
```
□

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

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

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

□

13.8 Additional Exercises

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

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

would be written as

2 3 * 3 4 2 - * +

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

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

The goal of this exercise is to write a small compiler that translates **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 (n : nat)
| SLoad (x : string)
| SPlus
| SMinus
| SMult.
```

Write a function to evaluate programs in the stack language. It should take as input a state, a stack represented as a list of numbers (top stack item is the head of the list), and a

program represented as a list of instructions, and it should return the stack after executing the program. Test your function on the examples below.

Note that 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_st []
       [SPush 5; SPush 3; SPush 1; SMinus]
   = [2; 5].
   Admitted.

Example s_execute2 :
     s_execute (X !-> 3) [3;4]
       [SPush 4; SLoad X; SMult; SPlus]
   = [15; 4].
   Admitted.
```

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

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

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

```
Example s_compile1 :
  s_compile (X - (2 × Y))%imp
  = [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, standard, optional (short_circuit) Most modern programming languages use a "short-circuit" evaluation rule for boolean **and**: to evaluate BAnd *b1 b2*, first evaluate *b1*. If it evaluates to false, then the entire BAnd expression evaluates to false immediately, without evaluating *b2*. Otherwise, *b2* is evaluated to determine the result of the BAnd expression.

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

```
Module BreakImp.
```

Exercise: 4 stars, 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
  | CBreak
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : 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 "'TEST' 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
  | SBreak.

Reserved Notation "st '=[' c ']=>' st' '/' s"
         (at level 40, st' at next level).
```

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

The definition of the "*st* =[c]=> *st'* / *s*" relation is very similar to the one we gave above for the regular evaluation relation (*st* =[c]=> *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 *TEST* 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,
      st =[ CSkip ]=> st / SContinue


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

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

```
Theorem break_ignore : ∀ c st st' s,
     st =[ BREAK;; c ]=> st' / s →
     st = st'.
Proof.
  Admitted.

Theorem while_continue : ∀ b c st st' s,
  st =[ WHILE b DO c END ]=> st' / s →
  s = SContinue.
Proof.
  Admitted.

Theorem while_stops_on_break : ∀ b c st st',
  beval st b = true →
  st =[ c ]=> st' / SBreak →
  st =[ WHILE b DO c END ]=> st' / SContinue.
Proof.
  Admitted.
```

□

Exercise: 3 stars, advanced, optional (while_break_true)

```
Theorem while_break_true : ∀ b c st st',
  st =[ WHILE b DO c END ]=> st' / SContinue →
  beval st' b = true →
  ∃ st'', st'' =[ c ]=> st' / SBreak.
Proof.
  Admitted.
```

□

Exercise: 4 stars, advanced, optional (ceval_deterministic) Theorem ceval_deterministic:
∀ (*c*:**com**) *st st1 st2 s1 s2*,
st =[*c*]=> *st1* / *s1* →
st =[*c*]=> *st2* / *s2* →
st1 = *st2* ∧ *s1* = *s2*.
Proof.
Admitted.

□ End BreakImp.

Exercise: 4 stars, standard, optional (add_for_loop) Add C-style `for` loops to the language of commands, update the **ceval** definition to define the semantics of `for` loops, and add cases for `for` loops as needed so that all the proofs in this file are accepted by 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 14

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.

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Strings.String.
From Coq Require Import Strings.Ascii.
From Coq Require Import Arith.Arith.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.EqNat.
From Coq Require Import Lists.List.
Import ListNotations.
From LF Require Import Maps Imp.
```

14.1 Internals

14.1.1 Lexical Analysis

```
Definition isWhite (c : ascii) : bool :=
  let n := nat_of_ascii c in
  orb (orb (n =? 32)
           (n =? 9))
```

```
      (orb (n =? 10)
           (n =? 13)).
Notation "x '<=?' y" := (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.
```

14.1.2 Parsing

Options With Errors

An **option** type with error messages:

```
Inductive optionE (X:Type) : Type :=
  | SomeE (x : X)
  | NoneE (s : string).

Arguments SomeE {X}.
Arguments NoneE {X}.
```

Some syntactic sugar to make writing nested match-expressions on optionE more convenient.

```
Notation "' p <- e1 ;; e2"
   := (match e1 with
       | SomeE p ⇒ e2
       | NoneE err ⇒ NoneE err
       end)
```

```
  (right associativity, p pattern, at level 60, e1 at next level).
Notation "'TRY' ' p <- e1 ;; e2 'OR' e3"
  := (match e1 with
      | SomeE p ⇒ e2
      | NoneE _ ⇒ e3
      end)
  (right associativity, p pattern,
   at level 60, e1 at next level, 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 ps:

```
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 (string × list token) :=
match xs with
| [] ⇒ NoneE "Expected identifier"
| x::xs' ⇒
    if forallb isLowerAlpha (list_of_string x) then
      SomeE (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' ⇒
      TRY ' (i, rest) ← parseIdentifier xs ;;
          SomeE (AId i, rest)
      OR
      TRY ' (n, rest) ← parseNumber xs ;;
          SomeE (ANum n, rest)
```

```
      OR
      ' (e, rest) ← firstExpect "(" (parseSumExp steps') xs ;;
      ' (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' ⇒
    ' (e, rest) ← parsePrimaryExp steps' xs ;;
    ' (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' ⇒
    ' (e, rest) ← parseProductExp steps' xs ;;
    ' (es, rest') ←
      many (fun xs ⇒
              TRY ' (e,rest') ←
                  firstExpect "+"
                              (parseProductExp steps') xs ;;
                  SomeE ( (true, e), rest')
              OR
              ' (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' ⇒
      TRY ' (u,rest) ← expect "true" xs ;;
          SomeE (BTrue,rest)
      OR
      TRY ' (u,rest) ← expect "false" xs ;;
          SomeE (BFalse,rest)
      OR
      TRY ' (e,rest) ← firstExpect "~"
                                   (parseAtomicExp steps')
                                   xs ;;
          SomeE (BNot e, rest)
      OR
      TRY ' (e,rest) ← firstExpect "("
                                   (parseConjunctionExp steps')
                                   xs ;;
          ' (u,rest') ← expect ")" rest ;;
          SomeE (e, rest')
      OR
      ' (e, rest) ← parseProductExp steps' xs ;;
      TRY ' (e', rest') ← firstExpect "="
                                  (parseAExp steps') rest ;;
          SomeE (BEq e e', rest')
      OR
      TRY ' (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' ⇒
    ' (e, rest) ← parseAtomicExp steps' xs ;;
```

```
    ' (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' ⇒
    TRY ' (u, rest) ← expect "SKIP" xs ;;
        SomeE (SKIP%imp, rest)
    OR
    TRY ' (e,rest) ←
            firstExpect "TEST"
                        (parseBExp steps') xs ;;
        ' (c,rest') ←
            firstExpect "THEN"
                        (parseSequencedCommand steps') rest ;;
        ' (c',rest'') ←
            firstExpect "ELSE"
                        (parseSequencedCommand steps') rest' ;;
        ' (tt,rest''') ←
            expect "END" rest'' ;;
       SomeE(TEST e THEN c ELSE c' FI%imp, rest''')
    OR
    TRY ' (e,rest) ←
            firstExpect "WHILE"
                        (parseBExp steps') xs ;;
        ' (c,rest') ←
            firstExpect "DO"
```

```
                              (parseSequencedCommand steps') rest ;;
          ' (u,rest'') ←
              expect "END" rest' ;;
          SomeE(WHILE e DO c END%imp, rest'')
    OR
    TRY ' (i, rest) ← parseIdentifier xs ;;
          ' (e, rest') ← firstExpect ":=" (parseAExp steps') rest ;;
          SomeE ((i := e)%imp, rest')
    OR
          NoneE "Expecting a command"
end

with parseSequencedCommand (steps:nat)
                           (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    ' (c, rest) ← parseSimpleCommand steps' xs ;;
    TRY ' (c', rest') ←
              firstExpect ";;"
                              (parseSequencedCommand steps') rest ;;
          SomeE ((c ;; c')%imp, rest')
    OR
    SomeE (c, rest)
  end.

Definition bignumber := 1000.

Definition parse (str : string) : optionE com :=
  let tokens := tokenize str in
  match parseSequencedCommand bignumber tokens with
  | SomeE (c, []) ⇒ SomeE c
  | SomeE (_, t::_) ⇒ NoneE ("Trailing tokens remaining: " ++ t)
  | NoneE err ⇒ NoneE err
  end.
```

14.2 Examples

```
Example eg1 : parse " TEST x = y + 1 + 2 - y * 6 + 3 THEN x ::= x * 1;; y ::= 0 ELSE
SKIP END "
=
  SomeE (
      TEST "x" = "y" + 1 + 2 - "y" × 6 + 3 THEN
```

```
      "x" ::= "x" × 1;;
      "y" ::= 0
    ELSE
      SKIP
    FI)%imp.
Proof. cbv. reflexivity. Qed.

Example eg2 : parse " SKIP;; z::=x*y*(x*x);; WHILE x=x DO TEST (z <= z*z) && ~(x
= 2) THEN x ::= z;; y ::= z ELSE SKIP END;; SKIP END;; x::=z "
=
  SomeE (
    SKIP;;
    "z" ::= "x" × "y" × ("x" × "x");;
    WHILE "x" = "x" DO
      TEST ("z" ≤ "z" × "z") && ~("x" = 2) THEN
        "x" ::= "z";;
        "y" ::= "z"
      ELSE
        SKIP
      FI;;
      SKIP
    END;;
    "x" ::= "z")%imp.
Proof. cbv. reflexivity. Qed.
```

Chapter 15

ImpCEvalFun: An 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.

15.1 A Broken Evaluator

```
From Coq Require Import omega.Omega.
From Coq Require Import Arith.Arith.
From LF Require Import Imp Maps.
```

Here was our first try at an evaluation function for commands, omitting *WHILE*.

```
Open Scope imp_scope.
Fixpoint ceval_step1 (st : state) (c : com) : state :=
  match c with
    | SKIP ⇒
        st
    | l ::= a1 ⇒
        (l !-> aeval st a1 ; st)
    | c1 ;; c2 ⇒
        let st' := ceval_step1 st c1 in
        ceval_step1 st' c2
    | TEST 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.
Close Scope imp_scope.
```

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...

15.2 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 of gas – it doesn't really matter because the result is going to be wrong in either case!)

```
Open Scope imp_scope.
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) : state :=
  match i with
  | O ⇒ empty_st
  | S i' ⇒
    match c with
      | SKIP ⇒
          st
      | l ::= a1 ⇒
          (l !-> aeval st a1 ; st)
      | c1 ;; c2 ⇒
          let st' := ceval_step2 st c1 i' in
          ceval_step2 st' c2 i'
      | TEST 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.
Close Scope imp_scope.
```

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.

```
Open Scope imp_scope.
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 (l !-> aeval st a1 ; st)
      | c1 ;; c2 ⇒
          match (ceval_step3 st c1 i') with
          | Some st' ⇒ ceval_step3 st' c2 i'
          | None ⇒ None
          end
      | TEST 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.
Close Scope imp_scope.
```

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).

Open Scope imp_scope.
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 (l !-> aeval st a1 ; st)
      | c1 ;; c2 ⇒
          LETOPT st' <== ceval_step st c1 i' IN
          ceval_step st' c2 i'
      | TEST 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.
Close Scope imp_scope.
```

```
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, standard, 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, standard, optional (peven) Write an Imp program that sets Z to 0 if X is even and sets Z to 1 otherwise. Use test_ceval to test your program.

15.3 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') →
      st =[ c ]=> 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. discriminate 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.
    ×
      discriminate 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_WhileTrue with s. rewrite Heqr.
       reflexivity.
       apply IHi'. rewrite Heqr1. reflexivity.
       apply IHi'. simpl in H1. assumption. }
     { discriminate H1. }
    ×
      injection H1. intros H2. rewrite ← H2.
      apply E_WhileFalse. apply Heqr. Qed.
```

Exercise: 4 stars, standard (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.

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

```
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. discriminate 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.
      ×
        discriminate 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. discriminate Hceval. Qed.
```

Exercise: 3 stars, standard, 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 $\leq$ and plus.

```
Theorem ceval__ceval_step: ∀ c st st',
      st =[ c ]=> 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',
      st =[ c ]=> st'
  ↔ ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st'.
  split. apply ceval__ceval_step. apply ceval_step__ceval.
Qed.
```

15.4 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,
     st =[ c ]=> st1 →
     st =[ c ]=> 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.
```

Chapter 16

Extraction: Extracting ML from Coq

16.1 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 (mostly works), and Scheme (a bit out of date).

```
Require Coq.extraction.Extraction.
Extraction Language OCaml.
```

Now we load up the Coq environment with some definitions, either directly or by importing them from other modules.

```
From Coq Require Import Arith.Arith.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.EqNat.
From LF 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.

16.2 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 eqb ⇒ "( = )".
```

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*.

16.3 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 first need to set up the right correspondence between Coq strings and lists of OCaml characters.

```
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlString.
```

We also need one more variant of booleans.

```
Extract Inductive sumbool ⇒ "bool" ["true" "false"].
```

The extraction is the same as always.

```
From LF Require Import Imp.
From LF Require Import ImpParser.

From LF Require Import Maps.
```

`Extraction` "imp.ml" *empty_st 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.)

16.4 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!

16.5 Going Further

Further details about extraction can be found in the Extract chapter in *Verified Functional Algorithms* (*Software Foundations* volume 3).

Chapter 17

Auto: More Automation

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import omega.Omega.
From LF Require Import Maps.
From LF 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 in *Programming Language Foundations*.

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.

First, define a little Ltac macro to compress a common pattern into a single command.

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

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> 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; apply H1. }
    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.
```

17.1 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.
  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 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.

It is also possible to define specialized hint databases that can be activated only when needed. See the Coq reference manual for more.

```
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.
```

Let's take a first pass over ceval_deterministic to simplify the proof script.

```
Theorem ceval_deterministic': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> 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,
```

```
    st =[ c ]=> st1 →
    st =[ c ]=> 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.
```

17.2 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 Ltac.

```
Ltac rwinv H1 H2 := rewrite H1 in H2; inv H2.

Theorem ceval_deterministic'': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> 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 a bit better, but 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.
```

This `match goal` 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,
     st =[ c ]=> st1 →
     st =[ c ]=> 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,
     st =[ c ]=> st1 →
     st =[ c ]=> 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 $\forall\ x,\ ?P\ x \to ?L = ?\mathbf{R}$ 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 $\mathbf{R}$. 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,
     st =[ c ]=> st1 →
     st =[ c ]=> 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 more robust in the face of modest changes to our language. To test this, let's try adding a *REPEAT* command to the language.

```
Module Repeat.

Inductive com : Type :=
  | CSkip
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com)
```

```
  | CRepeat (c : com) (b : bexp).
```

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 "'TEST' 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 (TEST b1 THEN c1 ELSE c2 FI) st'
  | E_IfFalse : ∀ st st' b1 c1 c2,
      beval st b1 = false →
      ceval st c2 st' →
      ceval st (TEST b1 THEN c1 ELSE c2 FI) st'
  | E_WhileFalse : ∀ b1 st c1,
      beval st b1 = false →
      ceval st (WHILE b1 DO c1 END) st
  | E_WhileTrue : ∀ 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 "st '=[' c ']=>' st'" := (ceval st c st')
                                  (at level 40).
```

Our first attempt at the determinacy proof does not quite succeed: the E_RepeatEnd and E_RepeatLoop cases are not handled by our previous automation.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> 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.
```

Fortunately, to fix this, we just have to swap the invocations of *find_eqn* and *find_rwinv*.

```
Theorem ceval_deterministic': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> 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.

17.2.1 The `eapply` and `eauto` variants

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:
  empty_st =[
    X ::= 2;;
    TEST X ≤ 1
      THEN Y ::= 3
      ELSE Z ::= 4
    FI
  ]=> (Z !-> 4 ; X !-> 2).
Proof.
  apply E_Seq with (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", st = c1 => st' -> st' = c2 => st" -> st = c1 ;; c2 => 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:
  empty_st =[
    X ::= 2;;
    TEST X ≤ 1
      THEN Y ::= 3
      ELSE Z ::= 4
    FI
  ]=> (Z !-> 4 ; X !-> 2).
```

```
Proof.
  eapply E_Seq.    - apply E_Ass.      reflexivity.    - apply E_IfFalse. reflexivity.
apply E_Ass. reflexivity.
Qed.
```

The `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 similar variants. For example, here's a proof using `eauto`:

```
Hint Constructors ceval.
Hint Transparent state.
Hint Transparent total_map.

Definition st12 := (Y !-> 2 ; X !-> 1).
Definition st21 := (Y !-> 1 ; X !-> 2).

Example eauto_example : ∃ s',
  st21 =[
    TEST X ≤ Y
      THEN Z ::= Y - X
      ELSE Y ::= X + Z
    FI
  ]=> s'.
Proof. eauto. Qed.
```

The `eauto` tactic works just like `auto`, except that it uses `eapply` instead of `apply`.

Pro tip: One might think that, since `eapply` and `eauto` are more powerful than `apply` and `auto`, it would be a good idea to use them all the time. Unfortunately, they are also significantly slower – especially `eauto`. Coq experts tend to use `apply` and `auto` most of the time, only switching to the `e` variants when the ordinary variants don't do the job.

Chapter 18

Postscript

Congratulations: We've made it to the end!

18.1 Looking Back

We've covered quite a bit of ground so far. Here's a quick review...

- *Functional programming*:
  - "declarative" programming style (recursion over immutable data structures, rather than looping over mutable arrays or pointer structures)
  - higher-order functions
  - polymorphism
- *Logic*, the mathematical basis for software engineering:

  logic calculus
  - ——————- ˜ ————————-

  software engineering mechanical/civil engineering
  - inductively defined sets and relations
  - inductive proofs
  - proof objects
- *Coq*, an industrial-strength proof assistant
  - functional core language
  - core tactics
  - automation

18.2 Looking Forward

If what you've seen so far has whetted your interest, you have two choices for further reading in the *Software Foundations* series:

- *Programming Language Foundations* (volume 2, by a set of authors similar to this book's) covers material that might be found in a graduate course on the theory of programming languages, including Hoare logic, operational semantics, and type systems.
- *Verified Functional Algorithms* (volume 3, by Andrew Appel) builds on the themes of functional programming and program verification in Coq, addressing a range of topics that might be found in a standard data structures course, with an eye to formal verification.

18.3 Other sources

Here are some other good places to learn more...

- This book includes some 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.
- For questions about Coq, the #*coq* area of Stack Overflow (https://stackoverflow.com/questions/tagge is an excellent community resource.
- Here are some great books on functional programming
  - Learn You a Haskell for Great Good, by Miran Lipovaca *Lipovaca* 2011 (in Bib.v).
  - Real World Haskell, by Bryan O'Sullivan, John Goerzen, and Don Stewart *O'Sullivan* 2008 (in Bib.v)
  - ...and many other excellent books on Haskell, OCaml, Scheme, Racket, Scala, F sharp, etc., etc.
- And some further resources for Coq:
  - Certified Programming with Dependent Types, by Adam Chlipala *Chlipala* 2013 (in Bib.v).
  - Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran *Bertot* 2004 (in Bib.v).
- If you're interested in real-world applications of formal verification to critical software, see the Postscript chapter of *Programming Language Foundations*.
- For applications of Coq in building verified systems, the lectures and course materials for the 2017 DeepSpec Summer School are a great resource. https://deepspec.org/event/dsss17/index.

Chapter 19

Bib: Bibliography

19.1 Resources cited in this volume

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

Lipovaca 2011 Learn You a Haskell for Great Good! A Beginner's Guide, by Miran Lipovaca, No Starch Press, April 2011. http://learnyouahaskell.com

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