Contents

Chapter 1

Library SoftwareFoundationsExercises.Basics

1.1 Basics: Functional Programming in Coq

1.2 Introduction

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

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

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

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

1.3 Data and Functions

1.3.1 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.

1.3.2 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 : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3.3 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.

1.3.4 Booleans

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

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

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

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

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

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

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

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

```
Example test_orb1: (orb true false) = true.
```

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

We can also introduce some familiar syntax for the boolean operations we have just defined. The `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 (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 :=
  match b1 with
  | true ⇒ (negb b2)
  | false ⇒ true
  end.
```

```
Example test_nandb1: (nandb true false) = true.
Proof. simpl. reflexivity. Qed.
Example test_nandb2: (nandb false false) = true.
Proof. simpl. reflexivity. Qed.
Example test_nandb3: (nandb false true) = true.
Proof. simpl. reflexivity. Qed.
Example test_nandb4: (nandb true true) = false.
Proof. simpl. reflexivity. Qed.
```

☐

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

```
Definition andb3 (b1:bool) (b2:bool) (b3:bool) : bool :=
  match b1 with
  | true ⇒ (andb b2 b3)
  | false ⇒ false
  end.
```

```
Example test_andb31: (andb3 true true true) = true.
Proof. simpl. reflexivity. Qed.
Example test_andb32: (andb3 false true true) = false.
Proof. simpl. reflexivity. Qed.
Example test_andb33: (andb3 true false true) = false.
Proof. simpl. reflexivity. Qed.
Example test_andb34: (andb3 true true false) = false.
Proof. simpl. reflexivity. Qed.
```

☐

1.3.5 Function Types

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

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

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

```
Check negb.
```

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

1.3.6 Compound Types

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 : rgb
  | green : rgb
  | blue : rgb.
```

```
Inductive color : Type :=
  | black : color
  | white : color
  | primary : rgb → color.
```

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 p ⇒ false
  end.
```

Since the *primary* constructor takes an argument, a pattern matching *primary* should include either a variable (as above) 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*.)

1.3.7 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.
```

1.3.8 Numbers

An even more interesting way of defining a type is to allow its constructors to take arguments from the very same type – that is, to allow the rules describing its elements to be *inductive*.

For example, we can define (a unary representation of) natural numbers as follows:

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

The clauses of this definition can be read:

- O is a natural number (note that this is the letter "O," not the numeral "0").
- S 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 : nat'
  | tick : nat' → nat'.
```

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

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

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

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

```
End NatPlayground.
```

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

```
Check (S (S (S (S O)))).
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 arabic numerals: the numeral 1 is not a computation; it's a piece of data. When we write 111 to mean the number one hundred and eleven, we are using 1, three times, to write down a concrete representation of a number.)

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

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

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

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

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

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

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

```
Module NatPlayground2.

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

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

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

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

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

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

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

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

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

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

```
End NatPlayground2.

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

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

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

Translate this into Coq.

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

Example test_factorial1: (factorial 3) = 6.
Proof. simpl. reflexivity. Qed.
Example test_factorial2: (factorial 5) = (mult 10 12).
Proof. simpl. reflexivity. Qed.
```

□

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

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

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

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

(The level, associativity, and *nat_scope* annotations control how these notations are treated by Coq's parser. The details are not important for our purposes, but interested readers can refer to the "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 *beq_nat*, which tests *nat*ural numbers for *eq*uality, yielding a *b*oolean. Note the use of nested matches (we could also have used a simultaneous match, as we did in *minus*.)

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

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

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

Example test_leb1: (leb 2 2) = true.
Proof. simpl. reflexivity. Qed.
Example test_leb2: (leb 2 4) = true.
Proof. simpl. reflexivity. Qed.
```

```
Example test_leb3: (leb 4 2) = false.
Proof. simpl. reflexivity. Qed.
```

Exercise: 1 star (blt_nat) The *blt_nat* function tests *nat*ural numbers for *l*ess-*t*han, yielding a *b*oolean. Instead of making up a new `Fixpoint` for this one, define it in terms of a previously defined function.

```
Definition blt_nat (n m : nat) : bool :=
  match n, m with
  | n', m' ⇒ (andb (negb (beq_nat n' m')) (leb n' m'))
  end.
Example test_blt_nat1: (blt_nat 2 2) = false.
Proof. simpl. reflexivity. Qed.
Example test_blt_nat2: (blt_nat 2 4) = true.
Proof. simpl. reflexivity. Qed.
Example test_blt_nat3: (blt_nat 4 2) = false.
Proof. simpl. reflexivity. Qed.
```

☐

1.4 Proof by Simplification

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

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

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

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

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

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

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

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

First, we've used the keyword `Theorem` instead of `Example`. This difference is 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. simpl. 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.

1.5 Proof by Rewriting

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

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

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

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

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

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

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

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

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

```
Theorem plus_id_exercise : ∀ n m o : nat,
  n = m → m = o → n + m = m + o.
Proof.
  intros m n o.
  intros H.
  rewrite → H.
  intros H0.
  rewrite → H0.
  reflexivity.
  Qed.
```

☐

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

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

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

Exercise: 2 stars (mult_S_1)

```
Theorem mult_S_1 : ∀ n m : nat,
  m = S n →
  m × (1 + n) = m × m.
Proof.
  intros n m.
  rewrite ← plus_1_l.
  intros H.
  rewrite → H.
  reflexivity.
  Qed.
```

☐

1.6 Proof by Case Analysis

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

intros x y. destruct y as |*y*.

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

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

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

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

```
  - reflexivity.
  - reflexivity.
Qed.
```

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

```
Theorem andb_true_elim2 : ∀ b c : bool,
  andb b c = true → c = true.
Proof.
  intros [] [].
  - intros H. reflexivity.
  - simpl. intros H. rewrite ← H. reflexivity.
  - intros H. reflexivity.
  - simpl. intros H. rewrite ← H. reflexivity.
Qed.
```

□

Exercise: 1 star (zero_nbeq_plus_1) `Theorem zero_nbeq_plus_1 : ∀ n : nat,`

```
  beq_nat 0 (n + 1) = false.
Proof.
  intros [|n].
  - reflexivity.
  - reflexivity.
Qed.
```

□

1.6.1 More on Notation (Optional)

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

Recall the notation definitions for infix plus and times:

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

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

Each notation symbol is also associated with a *notation scope*. Coq tries to guess what scope is meant from context, so when it sees $S(O \times O)$ 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 $(x \times y)\%nat$, 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!

1.6.2 Fixpoints and Structural Recursion (Optional)

Here is a copy of the definition of addition:

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

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

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

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

□

1.7 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: 2 stars (boolean_functions) Use the tactics you have learned so far to prove the following theorem about boolean functions.

```
Theorem identity_fn_applied_twice :
  ∀ (f : bool → bool),
  (∀ (x : bool), f x = x) →
  ∀ (b : bool), f (f b) = b.
Proof.
  intros f x b. rewrite → x. destruct b.
  - rewrite x. reflexivity.
  - rewrite x. reflexivity.
Qed.
```

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

```
Theorem negation_fn_applied_twice :
  ∀ (f : bool → bool),
  (∀ (x : bool), f x = negb x) →
  ∀ (b : bool), f (f b) = b.
Proof.
  intros f x b. destruct b.
  - rewrite x. rewrite x. reflexivity.
  - rewrite x. rewrite x. reflexivity.
Qed.
From Coq Require Export String.

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

□

Exercise: 3 stars, 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.
```

```
  intros b c. destruct b.
  - simpl. intros H. rewrite H. reflexivity.
  - simpl. intros H. rewrite H. reflexivity.
Qed.
```

☐

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

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

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

(Hint: Recall that the definition of *nat* above,

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

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

One caveat: If you use O or S as constructor names in your definition, it will confuse the auto-grader script. Please choose different names.

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

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

```
Inductive bin : Type :=
  | E : bin
  | R : bin
  | RR : bin → bin.

Fixpoint incr (bit_field:bin) : bin :=
  match bit_field with
  | E ⇒ R
```

```
  | R => RR E
  | RR n' => RR (incr n')
  end.

Fixpoint bin_to_nat (bit_field:bin) : nat :=
  match bit_field with
  | E => O
  | R => S O
  | RR n' => (mult 2 (bin_to_nat n'))
  end.

Example test_bin_incr1: (incr (RR E)) = (RR R).
Proof. simpl. reflexivity. Qed.
Example test_bin_incr2: (incr (RR R)) = (RR (RR E)).
Proof. simpl. reflexivity. Qed.
Example test_bin_incr3: (incr E) = R.
Proof. simpl. reflexivity. Qed.
Example test_bin_incr4: (incr R) = (RR E).
Proof. simpl. reflexivity. Qed.
Example test_bin_incr5: (incr (RR (RR E))) = (RR (RR R)).
Proof. simpl. reflexivity. Qed.

Example test_bin_to_nat1: (bin_to_nat E) = 0.
Proof. simpl. reflexivity. Qed.
Example test_bin_to_nat2: (bin_to_nat R) = 1.
Proof. simpl. reflexivity. Qed.

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

Chapter 2

Library SoftwareFoundationsExercises.Induction

2.1 Induction: Proof by Induction

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

`Require Export` *Basics.*

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

- In CoqIDE:

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

- From the command line: Either

  make Basics.vo

  (assuming you've downloaded the whole LF directory and have a working *make* command) or

  coqc Basics.v

  (which should work regardless).

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 Coq.Init.Bar

you should check whether you have multiple installations of Coq on your machine. If so, 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.

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.

2.2 Proof by Induction

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

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

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

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

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

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

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

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

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

- show that $P(O)$ holds;
- show that, for any n', if $P(n')$ holds, then so does $P(S\ n')$;

- conclude that $P(n)$ holds for all n.

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

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

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

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

In the second subgoal, n is replaced by $S\ n'$, and the assumption $n' + 0 = n'$ is added to the context with the name *IHn'* (i.e., the Induction Hypothesis for n'). These two names are specified in the second part of the `as...` clause. The goal in this case becomes $S\ n' = (S\ n') + 0$, 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, recommended (basic_induction) Prove the following using induction. You might need previously proven results.

```
Theorem mult_0_r : ∀ n:nat,
  n × 0 = 0.
Proof.
  intros n. induction n as [| n' IHn'].
  - simpl. reflexivity.
```

```
  - simpl. rewrite -> IHn'. reflexivity.
Qed.

Theorem plus_n_Sm : ∀ n m : nat,
  S (n + m) = n + (S m).
Proof.
  intros n m. induction n as [| n' IHn'].
  - simpl. reflexivity.
  - simpl. rewrite -> IHn'. reflexivity.
Qed.

Theorem plus_comm : ∀ n m : nat,
  n + m = m + n.
Proof.
  intros n m. induction n as [| n' IHn'].
  - induction m as [| m' IHm'].
    + reflexivity.
    + simpl. rewrite <- IHm'. simpl. reflexivity.
  - simpl. rewrite -> IHn'. rewrite <- plus_n_Sm. reflexivity.
Qed.

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

□

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

```
Fixpoint double (n:nat) :=
  match n with
  | O => O
  | S n' => S (S (double n'))
  end.
```

Use induction to prove this simple fact about *double*:

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

```
Proof.
  intros n. induction n as [| n' IHn'].
  {
    simpl. reflexivity.
  }
  {
    simpl. rewrite → IHn'. rewrite → plus_n_Sm. reflexivity.
  }
Qed.
```

☐

Exercise: 2 stars, 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.
  intros n. induction n as [| n' IHn'].
  {
    simpl. reflexivity.
  }
  {
    rewrite → IHn'. simpl. rewrite → negb_involutive. reflexivity.
  }
Qed.
```

☐

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

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

☐

2.3 Proofs Within Proofs

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

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

The `assert` tactic introduces two sub-goals. The first is the assertion itself; by prefixing it with H: we name the assertion H. (We can also name the assertion with `as` just as we did above with `destruct` and `induction`, i.e., `assert` $(0 + n = n)$ `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` → *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.
```

2.4 Formal vs. Informal Proof

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

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

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

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

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

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

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

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

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

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

```
intros n m p. induction n as [| n' IHn'].
-
  reflexivity.
-
  simpl. rewrite → IHn'. reflexivity. Qed.
```

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

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

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

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

  Proof: By induction on n.

  - First, suppose $n = 0$. We must show

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

    This follows directly from the definition of +.

  - Next, suppose $n = S\ n'$, where

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

    We must show

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

    By the definition of +, this follows from

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

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

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

Exercise: 2 stars, advanced, 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 (prod nat string) := None.
```

□

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

Theorem: *true* = *beq_nat* *n* *n* for any *n*.

Proof: ☐

2.5 More Exercises

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

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

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

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

☐

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

```
Check leb.

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

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

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

Admitted.

`Theorem` *plus_ble_compat_l* : $\forall$ *n m p* : *nat*,
  leb n m = *true* $\rightarrow$ *leb* (*p* + *n*) (*p* + *m*) = *true*.
`Proof.`
  Admitted.

`Theorem` *S_nbeq_0* : $\forall$ *n*:*nat*,
  beq_nat (*S n*) 0 = *false*.
`Proof.`
  Admitted.

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

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

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

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

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

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

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

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

`Theorem` *plus_swap'* : $\forall$ *n m p* : *nat*,
$n + (m + p) = m + (n + p)$.
`Proof.`
Admitted.
□

Exercise: 3 stars, 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* (*prod nat string*) := *None*.
□

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

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

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

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

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

`Definition` *manual_grade_for_binary_inverse* : *option* (*prod nat string*) := *None*.

□

Chapter 3

Library SoftwareFoundationsExercises.Lists

3.1 Lists: Working with Structured Data

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

3.2 Pairs of Numbers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Proof.
  intros p. destruct p as [n m]. simpl. reflexivity.
Qed.
```

☐

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

```
Proof.
  intros p. destruct p as [n m]. simpl. reflexivity.
Qed.
```

☐

3.3 Lists of Numbers

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

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

For example, here is a three-element list:

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

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

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

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

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
```

```
Definition mylist3 := [1;2;3].
```

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

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

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

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

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

Repeat

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

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

Length

The *length* function calculates the length of a list.

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

Append

The *app* function concatenates (appends) two lists.

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

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

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

Head (with default) and Tail

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

```
Definition hd (default:nat) (l:natlist) : nat :=
  match l with
  | nil ⇒ default
  | h :: t ⇒ h
  end.

Definition tl (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ t
  end.

Example test_hd1: hd 0 [1;2;3] = 1.
Proof. reflexivity. Qed.
Example test_hd2: hd 0 [] = 0.
Proof. reflexivity. Qed.
Example test_tl: tl [1;2;3] = [2;3].
Proof. reflexivity. Qed.
```

Exercises

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

```
Fixpoint nonzeros (l:natlist) : natlist :=
  match l with
```

```
  | nil ⇒ nil
  | O :: t ⇒ (nonzeros t)
  | h :: t ⇒ h :: (nonzeros t)
  end.
Example test_nonzeros: nonzeros [0;1;0;2;3;0;0] = [1;2;3].
Proof. simpl. reflexivity. Qed.
Fixpoint oddmembers (l:natlist) : natlist :=
  match l with
    | nil ⇒ nil
    | h :: t ⇒
      match (oddb h) with
        | true ⇒ h :: (oddmembers t)
        | false ⇒ (oddmembers t)
      end
  end.
Example test_oddmembers: oddmembers [0;1;0;2;3;0;0] = [1;3].
Proof. simpl. reflexivity. Qed.
Definition countoddmembers (l:natlist) : nat := (length (oddmembers l)).
Example test_countoddmembers1:
  countoddmembers [1;0;3;1;4;5] = 4.
Proof. simpl. reflexivity. Qed.
Example test_countoddmembers2:
  countoddmembers [0;2;4] = 0.
Proof. simpl. reflexivity. Qed.
Example test_countoddmembers3:
  countoddmembers nil = 0.
Proof. simpl. reflexivity. Qed.
```

☐

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

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

```
Fixpoint alternate (l1 l2 : natlist) : natlist :=
  match l1, l2 with
    | nil, nil ⇒ nil
```

```
    | x :: x', nil => x :: x'
    | nil, y :: y' => y :: y'
    | x :: x', y :: y' => x :: y :: (alternate x' y')
  end.

Example test_alternate1:
  alternate [1;2;3] [4;5;6] = [1;4;2;5;3;6].
Proof. simpl. reflexivity. Qed.

Example test_alternate2:
  alternate [1] [4;5;6] = [1;4;5;6].
Proof. simpl. reflexivity. Qed.

Example test_alternate3:
  alternate [1;2;3] [4] = [1;4;2;3].
Proof. simpl. reflexivity. Qed.

Example test_alternate4:
  alternate [] [20;30] = [20;30].
Proof. simpl. reflexivity. Qed.
```

☐

Bags via Lists

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

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

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

```
Fixpoint count (v:nat) (s:bag) : nat :=
  match v, s with
    | v, nil => O
    | v, s :: s' =>
      match (beq_nat v s) with
        | true => S (count v s')
        | false => (count v s')
      end
  end.
```

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

```
Example test_count1: count 1 [1;2;3;1;4;1] = 3.
Proof. simpl. reflexivity. Qed.
Example test_count2: count 6 [1;2;3;1;4;1] = 0.
Proof. simpl. reflexivity. Qed.
```

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 := (alternate).

Example test_sum1: count 1 (sum [1;2;3] [1;4;1]) = 3.
Proof. simpl. reflexivity. Qed.

Definition add (v:nat) (s:bag) : bag := v :: s.

Example test_add1: count 1 (add 1 [1;4;1]) = 3.
Proof. simpl. reflexivity. Qed.
Example test_add2: count 5 (add 1 [1;4;1]) = 0.
Proof. simpl. reflexivity. Qed.

Definition member (v:nat) (s:bag) : bool :=
  match (count v s) with
    | O ⇒ false
    | S n' ⇒ true
  end.

Example test_member1: member 1 [1;4;1] = true.
Proof. simpl. reflexivity. Qed.

Example test_member2: member 2 [1;4;1] = false.
Proof. simpl. reflexivity. Qed.
```

☐

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

When *remove_one* is applied to a bag without the number to remove, it should return the same bag unchanged. `Definition` *manual_grade_for_bag_theorem* : *option* (*prod nat string*) := *None*.

☐

Exercise: 3 stars, 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!

☐

3.4 Reasoning About Lists

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

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

...because the [] is substituted into the "scrutinee" (the 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 example proof scripts will not get you very far! It is important to work through the details of each one, using Coq and thinking about what each step achieves. Otherwise it is more or less guaranteed that the exercises will make no sense when you get to them. 'Nuff said.

3.4.1 Induction on Lists

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

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

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

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

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

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

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

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

Proof: By induction on *l1*.

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

  $(\square ++ \text{l2}) ++ \text{l3} = \square ++ (\text{l2} ++ \text{l3})$,

  which follows directly from the definition of $++$.

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

  $(\text{l1'} ++ \text{l2}) ++ \text{l3} = \text{l1'} ++ (\text{l2} ++ \text{l3})$

  (the induction hypothesis). We must show

  $((\text{n} :: \text{l1'}) ++ \text{l2}) ++ \text{l3} = (\text{n} :: \text{l1'}) ++ (\text{l2} ++ \text{l3})$.

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

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


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

So let's take the equation relating ++ and *length* that would have enabled us to make progress and 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 natlists*, not just those that result from an application of *rev*. This should seem natural, because the truth of the goal clearly doesn't depend on the list having been reversed. Moreover, it is easier to prove the more general property.

Now we can complete the original proof.

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

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

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

Proof: By induction on *l1*.

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

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

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

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

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

  We must show

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

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

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

Proof: By induction on *l*.

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

  length (rev □) = length □,

  which follows directly from the definitions of *length* and *rev*.

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

  length (rev l') = length l'.

  We must show

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

  By the definition of *rev*, this follows from

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

  which, by the previous lemma, is the same as

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

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

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

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

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

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

3.4.2 `Search`

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

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

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

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

3.4.3 List Exercises, Part 1

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

`Theorem` *app_nil_r* : ∀ l : *natlist*,

```
  l ++ [] = l.
Proof.
  intros l. induction l as [| n l' IHl'].
    - simpl. reflexivity.
    - simpl. rewrite → IHl'. reflexivity.
Qed.

Theorem rev_app_distr: ∀ l1 l2 : natlist,
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
  intros l1 l2. induction l1 as [| n l' IHl'].
  - simpl. rewrite app_nil_r. reflexivity.
  - simpl. rewrite → IHl'. rewrite app_assoc. reflexivity.
Qed.

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

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.
  intros l1 l2 l3 l4. rewrite app_assoc. rewrite app_assoc. reflexivity.
Qed.
```

An exercise about your implementation of *nonzeros*:

```
Lemma nonzeros_app : ∀ l1 l2 : natlist,
  nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2).
Proof.
  intros l1 l2. induction l1 as [| n l' IHl'].
  - simpl. reflexivity.
  - simpl. rewrite IHl'. destruct n.
    + reflexivity.
    + simpl. reflexivity.
Qed.
```

☐

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

```
Fixpoint beq_natlist (l1 l2 : natlist) : bool :=
  match l1, l2 with
  | nil, nil ⇒ true
  | x :: x', nil ⇒ false
  | nil, y :: y' ⇒ false
  | x :: x', y :: y' ⇒
    match (beq_nat x y) with
      | true ⇒ (beq_natlist x' y')
      | false ⇒ false
    end
  end.

Example test_beq_natlist1 :
  (beq_natlist nil nil = true).
Proof. simpl. reflexivity. Qed.

Example test_beq_natlist2 :
  beq_natlist [1;2;3] [1;2;3] = true.
Proof. simpl. reflexivity. Qed.

Example test_beq_natlist3 :
  beq_natlist [1;2;3] [1;2;4] = false.
Proof. simpl. reflexivity. Qed.

Theorem beq_natlist_refl : ∀ l:natlist,
  true = beq_natlist l l.
Proof.
  intros l. induction l as [| n l' IHl'].
  - simpl. reflexivity.
  - simpl. rewrite IHl'. rewrite ← beq_nat_refl. reflexivity.
Qed.
```

☐

3.4.4 List Exercises, Part 2

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

Exercise: 1 star (count_member_nonzero)

```
Theorem count_member_nonzero : ∀ (s : bag),
  leb 1 (count 1 (1 :: s)) = true.
Proof.
  intros s. simpl. reflexivity.
Qed.
```

☐

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

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

Exercise: 3 stars, advanced (remove_does_not_increase_count)

Exercise: 3 stars, 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 (prod nat string) := None.
```

□

3.5 Options

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

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

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

```
Inductive natoption : Type :=
```

```
  | Some : nat → natoption
  | None : natoption.
```

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

```
Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match beq_nat n O with
               | true ⇒ Some a
               | false ⇒ nth_error l' (pred n)
               end
  end.
Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
Proof. reflexivity. Qed.
Example test_nth_error2 : nth_error [4;5;6;7] 3 = Some 7.
Proof. reflexivity. Qed.
Example test_nth_error3 : nth_error [4;5;6;7] 9 = None.
Proof. reflexivity. Qed.
```

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

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

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

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

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

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

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

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

Example test_hd_error1 : hd_error [] = None.
   Admitted.

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

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

☐

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

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

☐

```
End NatList.
```

3.6 Partial Maps

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

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

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

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

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

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

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

```
Proof.
  Admitted.
```

□

Now we define the type of partial maps:

```
Module PartialMap.
Export NatList.
```

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

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

The *update* function overrides the entry for a given key in a partial map 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 beq_id x y
                     then Some v
                     else find x d'
  end.
```

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

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

□

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

```
  ∀ (d : partial_map) (x y : id) (o: nat),
```

$beq\_id\ x\ y = false \rightarrow find\ x\ (update\ d\ y\ o) = find\ x\ d.$

`Proof.`

Admitted.

□ `End` *PartialMap.*

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

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

How *many* elements does the type *baz* have? (Explain your answer in words, preferrably English.)

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

□

Chapter 4

Library SoftwareFoundationsExercises.Poly

4.1 Poly: Polymorphism and Higher-Order Functions

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

4.2 Polymorphism

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

4.2.1 Polymorphic Lists

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

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

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

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

```
Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X → list X → list X.
```

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

What sort of thing is *list* itself? One good way to think about it is that *list* is a *function* from `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* becomes a parameter to the constructors *nil* and *cons* – that is, *nil* and *cons* are now polymorphic constructors, that need to be supplied with the type of the list they are building. As an 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 (mumble_grumble) Consider the following two inductively defined types.

```
Module MumbleGrumble.

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

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

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

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

- *e bool* (*b c* 0)
- *c*

End *MumbleGrumble*.

Definition *manual_grade_for_mumble_grumble* : *option* (*prod 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 the "implicit argument" _, which can be read as "Please try to figure out for yourself what belongs here." More precisely, when Coq encounters a _, it will attempt to *unify* all locally available information – the type of the function being applied, the types of the other arguments, and the type expected by the context in which the application appears – to determine what concrete type should replace the _.

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

repeat' X x count : list X :=

we can also replace the types with _

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

to tell Coq to attempt to infer the missing information.

Using implicit arguments, the `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 argument synthesis to write this:

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

Implicit Arguments

We can go further and even avoid writing _'s in most cases by telling Coq *always* to infer the type argument(s) of a given function.

The *Arguments* directive specifies the name of the function (or constructor) and then lists its argument names, with curly braces around any arguments to be treated as implicit. (If some arguments of a definition don't have a name, as is often the case for constructors, they can be marked with a wildcard pattern _.)

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

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

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

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

```
Fixpoint repeat''' {X : Type} (x : X) (count : nat) : list X :=
```

```
  match count with
  | 0 ⇒ nil
  | S count' ⇒ cons x (repeat''' x count')
  end.
```

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

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

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

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

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

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

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

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

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

Example test_rev2:
  rev (cons true nil) = cons true nil.
```

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

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

Supplying Type Arguments Explicitly

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

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

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

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

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

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

```
Check @nil.
```

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

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

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

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

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

Exercises

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

```
Theorem app_nil_r : ∀ (X:Type), ∀ l:list X,
```

```
  l ++ [] = l.
Proof.
  intros X. induction l as [| n l' IHl'].
  - simpl. reflexivity.
  - simpl. rewrite → IHl'. reflexivity.
Qed.

Theorem app_assoc : ∀ A (l m n:list A),
  l ++ m ++ n = (l ++ m) ++ n.
Proof.
  intros A l m n. induction l as [| q l' IHl'].
  - simpl. reflexivity.
  - simpl. rewrite → IHl'. reflexivity.
Qed.

Lemma app_length : ∀ (X:Type) (l1 l2 : list X),
  length (l1 ++ l2) = length l1 + length l2.
Proof.
  intros X l1 l2. induction l1 as [| n l' IHl'].
  - simpl. reflexivity.
  - simpl. rewrite → IHl'. reflexivity.
Qed.
```

□

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

```
Theorem rev_app_distr: ∀ X (l1 l2 : list X),
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
  intros. induction l1.
  - simpl. rewrite → app_nil_r. reflexivity.
  - simpl. rewrite → IHl1. rewrite → app_assoc. reflexivity.
Qed.

Theorem rev_involutive : ∀ X : Type, ∀ l : list X,
  rev (rev l) = l.
Proof.
  intros. induction l.
  - simpl. reflexivity.
  - simpl. rewrite → rev_app_distr. rewrite → IHl. reflexivity.
Qed.
```

□

4.2.2 Polymorphic Pairs

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

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

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

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

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

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

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

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

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

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, 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, 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) :=
  match l with
  | nil ⇒ ([],[])
  | (x, y) :: l' ⇒ match (split l') with
    | (lx, ly) ⇒ (x :: lx, y :: ly)
    end
  end.

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

□

4.2.3 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 → option X
  | None : option X.

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 beq_nat n O then Some a else nth_error l' (pred n)
  end.
```

```
Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
Proof. reflexivity. Qed.
Example test_nth_error2 : nth_error [[1];[2]] 1 = Some [2].
Proof. reflexivity. Qed.
Example test_nth_error3 : nth_error [true] 2 = None.
Proof. reflexivity. Qed.
```

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

```
Definition hd_error {X : Type} (l : list X) : option X :=
  match l with
  | nil ⇒ None
  | h :: l ⇒ Some h
  end.
```

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.
Proof. reflexivity. Qed.
Example test_hd_error2 : hd_error [[1];[2]] = Some [1].
Proof. reflexivity. Qed.
```

□

4.3 Functions as Data

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

4.3.1 Higher-Order Functions

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

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

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

```
Check @doit3times.
```

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

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

4.3.2 Filter

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

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

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

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

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

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

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

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

length (*filter oddb l*).

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

4.3.3 Anonymous Functions

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

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

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

The expression (fun $n \Rightarrow n \times n$) 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 ⇒ beq_nat (length l) 1)
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

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

```
Definition filter_even_gt7 (l : list nat) : list nat :=
  (filter
    (fun x ⇒ (andb (evenb x) (negb (leb x 7)))) l).

Example test_filter_even_gt7_1 :
  filter_even_gt7 [1;2;6;9;10;3;12;8] = [10;12;8].
Proof. simpl. reflexivity. Qed.

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

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

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

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

Given a set X, a test function of type $X \to bool$ 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 :=
  ((filter test l), (filter (fun y ⇒ (negb (test y))) l)).
Example test_partition1: partition oddb [1;2;3;4;5] = ([1;3;5], [2;4]).
Proof. simpl. reflexivity. Qed.
Example test_partition2: partition (fun x ⇒ false) [5;9;0] = ([], [5;9;0]).
Proof. simpl. reflexivity. Qed.
```
□

4.3.4 Map

Another handy higher-order function is called *map*.

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

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

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

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

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

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

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

Exercises

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

```
Theorem map_rev : ∀ (X Y : Type) (f : X → Y) (l : list X),
  map f (rev l) = rev (map f l).
Proof.
  intros X Y f l. induction l as [| n l' IHl'].
  - simpl. reflexivity.
  - simpl. rewrite ← IHl'.
Abort.
```

☐

Exercise: 2 stars, recommended (flat_map) The function *map* maps a *list X* to a *list Y* using a function of type $X \rightarrow Y$. We can define a similar function, *flat_map*, which maps a *list X* to a *list Y* using a function *f* of type $X \rightarrow$ *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) :=
  match l with
    | nil ⇒ []
    | x :: l' ⇒ (app (f x) (flat_map f l'))
  end.
```

```
Example test_flat_map1:
  flat_map (fun n ⇒ [n;n;n]) [1;5;4]
  = [1; 1; 1; 5; 5; 5; 4; 4; 4].
Proof. simpl. reflexivity. Qed.
```

☐

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

```
Definition option_map {X Y : Type} (f : X → Y) (xo : option X)
                         : option Y :=
  match xo with
```

```
    | None ⇒ None
    | Some x ⇒ Some (f x)
  end.
```

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

4.3.5 Fold

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

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

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

fold plus 1;2;3;4 0

yields

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

Some more examples:

```
Check (fold andb).

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

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

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

Exercise: 1 star, 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 (prod nat string) := None.
```
☐

4.3.6 Functions That Construct Functions

Most of the higher-order functions we have talked about so far take functions as arguments. Let's look at some examples that involve *returning* functions as the results of other functions. To begin, here is a function that takes a value x (drawn from some type X) and returns a function from *nat* to X that yields x whenever it is called, ignoring its *nat* argument.

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

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

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

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

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

```
Check plus.
```

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

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

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

4.4 Additional Exercises

```
Module Exercises.
```

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

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

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

Prove the correctness of *fold_length*.

```
Theorem fold_length_correct : ∀ X (l : list X),
  fold_length l = length l.
Proof.
  intros. induction l.
  - simpl. reflexivity.
  - simpl. rewrite ← IHl. reflexivity.
Qed.
```

☐

Exercise: 3 stars (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 :=
  fold (fun x y ⇒ (f x) :: y) l [].
```

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

```
Definition manual_grade_for_fold_map : option (prod 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 $A \to B \to C$ as $(A \times B) \to C$. 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 :=
  (f (fst p) (snd p)).
```

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.
  intros. simpl. reflexivity.
Qed.

Theorem curry_uncurry : ∀ (X Y Z : Type)
                        (f : (X × Y) → Z) (p : X × Y),
  prod_uncurry (prod_curry f) p = f p.
Proof.
  intros. destruct p as [x y]. reflexivity.
Qed.
```

☐

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 beq_nat n O then Some a else nth_error l' (pred n) end.

Write an informal proof of the following theorem:

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

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

☐

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

```
Module Church.
Definition nat := ∀ 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 : nat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f x.
```

Similarly, *two* should apply f twice to its argument:

```
Definition two : nat :=
  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 : nat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ x.
```

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

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

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

Successor of a natural number:

```
Definition succ (n : nat) : nat :=
  (fun X f x ⇒ n X f (f x)).

Example succ_1 : succ zero = one.
Proof. reflexivity. Qed.

Example succ_2 : succ one = two.
Proof. reflexivity. Qed.

Example succ_3 : succ two = three.
Proof. reflexivity. Qed.
```

Addition of two natural numbers:

```
Definition plus (n m : nat) : nat :=
  (fun X f x ⇒ m X f (n X f x)).

Example plus_1 : plus zero one = one.
Proof. reflexivity. Qed.
```

```
Example plus_2 : plus two three = plus three two.
Proof. reflexivity. Qed.

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

Multiplication:

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

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

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

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

Exponentiation:

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

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

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

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

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

End Church.

Definition manual_grade_for_succ_plus_mult_exp : option (prod nat string) := None.
```

□

```
End Exercises.
```

Chapter 5

Library SoftwareFoundationsExercises.Tactics

5.1 Tactics: More Basic Tactics

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

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

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

5.2 The apply Tactic

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

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

rewrite ← *eq1*.

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

```
apply eq2. Qed.
```

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

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

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

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

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

```
Theorem silly_ex :
     (∀ n, evenb n = true → oddb (S n) = true) →
     oddb 3 = true →
     evenb 4 = true.
Proof.
  intros n eq1. apply eq1.
Qed.
```

□

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

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

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 (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.
  intros l l' eq1. symmetry. rewrite eq1. apply rev_involutive.
Qed.
```

☐

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

☐

5.3 The `apply with` Tactic

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

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

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

```
Theorem trans_eq : ∀ (X:Type) (n m o : X),
  n = m → m = o → n = o.
Proof.
```

```
  intros X n m o eq1 eq2. rewrite → eq1. rewrite → eq2.
  reflexivity. Qed.
```

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

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

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

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

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

Exercise: 3 stars, optional (apply_with_exercise) Example *trans_eq_exercise* : ∀ (n m o p : *nat*),

```
     m = (minustwo o) →
     (n + p) = m →
     (n + p) = (minustwo o).
Proof.
  intros n m o p eq1 eq2.
  rewrite ← eq1. apply eq2.
Qed.
```

☐

5.4 The inversion Tactic

Recall the definition of natural numbers:

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

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

- The constructor S is *injective*. That is, if $S\ n = S\ m$, it must be the case that $n = m$.

- The constructors O and S are *disjoint*. That is, O is not equal to $S\ n$ for any n.

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

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

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

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

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

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

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

We can name the equations that `inversion` generates with an `as` ... clause:

```
Theorem inversion_ex2 : ∀ (n m : nat),
  [n] = [m] →
  n = m.
Proof.
  intros n m H. inversion H as [Hnm]. reflexivity. Qed.
```

Exercise: 1 star (inversion_ex3) `Example inversion_ex3 : ∀ (X : Type) (x y z w : X) (l j : list X),`

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

```
  intros X x y z w l j eq1 eq2.
  inversion eq2. inversion eq1. reflexivity.
Qed.
```

☐

When used on a hypothesis involving an equality between *different* constructors (e.g., S $n = O$), `inversion` solves the goal immediately. Consider the following proof:

```
Theorem beq_nat_0_l : ∀ n,
   beq_nat 0 n = true → n = 0.
Proof.
  intros n.
```

We can proceed by case analysis on n. The first case is trivial.

```
  destruct n as [| n'].
  -
    intros H. reflexivity.
```

However, the second one doesn't look so simple: assuming *beq_nat* 0 (S n') = *true*, we must show S n' = 0, but the latter clearly contradictory! The way forward lies in the assumption. After simplifying the goal state, we see that *beq_nat* 0 (S n') = *true* has become *false* = *true*:

```
  -
    simpl.
```

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

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

This is an instance of a logical principle known as the *principle of explosion*, which asserts that a contradictory hypothesis entails anything, even false things!

```
Theorem inversion_ex4 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. inversion contra. Qed.

Theorem inversion_ex5 : ∀ (n m : nat),
  false = true →
  [n] = [m].
Proof.
  intros n m contra. inversion contra. Qed.
```

If you find the principle of explosion confusing, remember that these proofs are not actually showing that the conclusion of the statement holds. Rather, they are arguing that, if the nonsensical situation described by the premise did somehow arise, then the nonsensical conclusion would follow. We'll explore the principle of explosion of more detail in the next chapter.

Exercise: 1 star (inversion_ex6) Example *inversion_ex6* : ∀ (*X* : Type) (*x y z* : *X*) (*l j* : *list X*),

```
  x :: y :: l = [] →
  y :: l = z :: j →
  x = z.
Proof.
  intros X x y z l j eq1 eq2.
  inversion eq2.
  inversion eq1.
Qed.
```

☐

To summarize this discussion, suppose *H* is a hypothesis in the context or a previously proven lemma of the form

c a1 a2 ... an = d b1 b2 ... bm

for some constructors *c* and *d* and arguments *a1* ... *an* and *b1* ... *bm*. Then `inversion` *H* has the following effect:

- If *c* and *d* are the same constructor, then, by the injectivity of this constructor, we know that *a1* = *b1*, *a2* = *b2*, etc. The `inversion` *H* adds these facts to the context and tries to use them to rewrite the goal.
- If *c* and *d* are different constructors, then the hypothesis *H* is contradictory, and the current goal doesn't have to be considered at all. In this case, `inversion` *H* marks the current goal as completed and pops it off the goal stack.

The injectivity of constructors allows us to reason that ∀ (*n m* : *nat*), *S n* = *S m* → *n* = *m*. The converse of this implication is an instance of a more general fact about both constructors and functions, which we will find 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.
```

5.5 Using Tactics on Hypotheses

By default, most tactics work on the goal formula and leave the context unchanged. However, most tactics also have a variant that performs a similar operation on a statement in the context.

For example, the tactic `simpl in` *H* performs simplification in the hypothesis named *H* in the context.

```
Theorem S_inj : ∀ (n m : nat) (b : bool),
      beq_nat (S n) (S m) = b →
      beq_nat n m = b.
```

```
Proof.
  intros n m b H. simpl in H. apply H. Qed.
```

Similarly, `apply` L `in` H matches some conditional statement L (of the form $L1 \rightarrow L2$, say) against a hypothesis H in the context. However, unlike ordinary `apply` (which rewrites a goal matching $L2$ into a subgoal $L1$), `apply` L `in` H matches H against $L1$ and, if successful, replaces it with $L2$.

In other words, `apply` L `in` H gives us a form of "forward reasoning": from $L1 \rightarrow L2$ and a hypothesis matching $L1$, it produces a hypothesis matching $L2$. By contrast, `apply` L is "backward reasoning": it says that if we know $L1 \rightarrow L2$ and we are trying to prove $L2$, it suffices to prove $L1$.

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

```
Theorem silly3' : ∀ (n : nat),
  (beq_nat n 5 = true → beq_nat (S (S n)) 7 = true) →
  true = beq_nat n 5 →
  true = beq_nat (S (S n)) 7.
Proof.
  intros n eq H.
  symmetry in H. apply eq in H. symmetry in H.
  apply H. Qed.
```

Forward reasoning starts from what is *given* (premises, previously proven theorems) and iteratively draws conclusions from them until the goal is reached. Backward reasoning starts from the *goal*, and iteratively reasons about what would imply the goal, until premises or previously proven theorems are reached. If you've seen informal proofs before (for example, in a math or computer science class), they probably used forward reasoning. In general, idiomatic use of Coq tends to favor backward reasoning, but in some situations the forward style can be easier to think about.

Exercise: 3 stars, recommended (plus_n_n_injective) Practice using "in" variants in this proof. (Hint: use *plus_n_Sm*.)

```
Theorem plus_n_n_injective : ∀ n m,
      n + n = m + m →
      n = m.
Proof.
  intros n m H. induction n as [|n'].
  - destruct m.
    + reflexivity.
    + inversion H.
  - destruct m.
    + inversion H.
    + simpl in H. rewrite ← plus_n_Sm in H. rewrite ← plus_n_Sm in H.
```

```
    inversion H.
Admitted.
```
□

5.6 Varying the Induction Hypothesis

Sometimes it is important to control the exact form of the induction hypothesis when carrying out inductive proofs in Coq. In particular, we need to be careful about which of the assumptions we move (using `intros`) from the goal to the context before invoking the `induction` tactic. For example, suppose we want to show that the *double* function is injective – i.e., that it maps different arguments to different results:

Theorem double_injective: forall n m, double n = double m -> n = m.

The way we *start* this proof is a bit delicate: if we begin with

intros n. induction n.

all is well. But if we begin it with

intros n m. induction n.

we get stuck in the middle of the inductive case...

```
Theorem double_injective_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction n as [| n'].
  - simpl. intros eq. destruct m as [| m'].
    + reflexivity.
    + inversion eq.
  - intros eq. destruct m as [| m'].
    + inversion eq.
    + apply f_equal.
```

At this point, the induction hypothesis, *IHn'*, does *not* give us *n' = m'* – there is an extra *S* in the way – so the goal is not provable.

```
    Abort.
```

What went wrong?

The problem is that, at the point we invoke the induction hypothesis, we have already introduced *m* into the context – intuitively, we have told Coq, "Let's consider some particular *n* and *m*..." and we now have to prove that, if *double n = double m* for *these particular n* and *m*, then *n = m*.

The next tactic, `induction` *n* says to Coq: We are going to show the goal by induction on *n*. That is, we are going to prove, for *all n*, that the proposition

- *P n* = "if *double n = double m*, then *n = m*"

holds, by showing

- $P\ O$

  (i.e., "if *double* O = *double* m then $O = m$") and

- $P\ n \to P\ (S\ n)$

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

If we look closely at the second statement, it is saying something rather strange: it says that, for a *particular* m, if we know

- "if *double* n = *double* m then $n = m$"

then we can prove

- "if *double* $(S\ n)$ = *double* m then $S\ n = m$".

To see why this is strange, let's think of a particular m – say, 5. The statement is then saying that, if we know

- Q = "if *double* n = 10 then n = 5"

then we can prove

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

But knowing Q doesn't give us any help at all with proving R! (If we tried to prove R from Q, we would start with something like "Suppose *double* $(S\ n)$ = 10..." but then we'd be stuck: knowing that *double* $(S\ n)$ is 10 tells us nothing about whether *double* n is 10, so Q is useless.)

Trying to carry out this proof by induction on n when m is already in the context doesn't work because we are then trying to prove a relation involving *every* n but just a *single* m.

The successful proof of *double_injective* leaves m in the goal statement at the point where the `induction` tactic is invoked on n:

```
Theorem double_injective : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n. induction n as [| n'].
  - simpl. intros m eq. destruct m as [| m'].
    + reflexivity.
    + inversion eq.
  - simpl.
```

Notice that both the goal and the induction hypothesis are different this time: the goal asks us to prove something more general (i.e., to prove the statement for *every* m), but the IH is correspondingly more flexible, allowing us to choose any m we like when we apply the IH.

```
intros m eq.
```

Now we've chosen a particular m and introduced the assumption that *double* n = *double* m. Since we are doing a case analysis on n, we also need a case analysis on m to keep the two "in sync."

```
destruct m as [| m'].
+ simpl.
```

The 0 case is trivial:

```
  inversion eq.
+
  apply f_equal.
```

At this point, since we are in the second branch of the `destruct` m, the m' mentioned in the context is the predecessor of the m we started out talking about. Since we are also in the S branch of the induction, this is perfect: if we instantiate the generic m in the IH with the current m' (this instantiation is performed automatically by the `apply` in the next step), then *IHn'* gives us exactly what we need to finish the proof.

```
  apply IHn'. inversion eq. reflexivity. Qed.
```

What you should take away from all this is that we need to be careful about using induction to try to prove something too specific: To prove a property of n and m by induction on n, it is sometimes important to leave m generic.

The following exercise requires the same pattern.

Exercise: 2 stars (beq_nat_true)

```
Theorem beq_nat_true : ∀ n m,
    beq_nat n m = true → n = m.
Proof.
  intros. induction n.
  - destruct m.
    + reflexivity.
    + inversion H.
  - destruct m.
    + inversion H.
    + apply f_equal. simpl in H.
Admitted.
```

☐

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

```
Definition manual_grade_for_informal_proof : option (prod 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'].
    + reflexivity.
    + inversion eq.
  - intros eq. destruct n as [| n'].
    + inversion eq.
    + apply f_equal.
Abort.
```

The problem is that, to do induction on m, we must first introduce n. (If we simply say `induction` m without introducing anything first, Coq will automatically introduce n for us!)

What can we do about this? One possibility is to rewrite the statement of the lemma so that m is quantified before n. This works, but it's not nice: We don't want to have to twist the statements of lemmas to fit the needs of a particular strategy for proving them! Rather we want to state them in the clearest and most natural way.

What we can do instead is to first introduce all the quantified variables and then *re-generalize* one or more of them, selectively taking variables out of the context and putting them back at the beginning of the goal. The `generalize dependent` tactic does this.

```
Theorem double_injective_take2 : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m.
  generalize dependent n.
  induction m as [| m'].
  - simpl. intros n eq. destruct n as [| n'].
    + reflexivity.
    + inversion eq.
  - intros n eq. destruct n as [| n'].
    + inversion eq.
    + apply f_equal.
      apply IHm'. inversion eq. reflexivity. Qed.
```

Let's look at an informal proof of this theorem. Note that the proposition we prove by induction leaves n quantified, corresponding to the use of generalize dependent in our formal proof.

Theorem: For any nats n and m, if $double\ n = double\ m$, then $n = m$.

Proof: Let m be a *nat*. We prove by induction on m that, for any n, if $double\ n = double\ m$ then $n = m$.

- First, suppose $m = 0$, and suppose n is a number such that $double\ n = double\ m$. We must show that $n = 0$.

  Since $m = 0$, by the definition of *double* we have $double\ n = 0$. There are two cases to consider for n. If $n = 0$ we are done, since $m = 0 = n$, as required. Otherwise, if $n = S\ n'$ for some n', we derive a contradiction: by the definition of *double*, we can calculate $double\ n = S\ (S\ (double\ n'))$, but this contradicts the assumption that $double\ n = 0$.

- Second, suppose $m = S\ m'$ and that n is again a number such that $double\ n = double\ m$. We must show that $n = S\ m'$, with the induction hypothesis that for every number s, if $double\ s = double\ m'$ then $s = m'$.

  By the fact that $m = S\ m'$ and the definition of *double*, we have $double\ n = S\ (S\ (double\ m'))$. There are two cases to consider for n.

  If $n = 0$, then by definition $double\ n = 0$, a contradiction.

  Thus, we may assume that $n = S\ n'$ for some n', and again by the definition of *double* we have $S\ (S\ (double\ n')) = S\ (S\ (double\ m'))$, which implies by inversion that $double\ n' = double\ m'$. Instantiating the induction hypothesis with n' thus allows us to conclude that $n' = m'$, and it follows immediately that $S\ n' = S\ m'$. Since $S\ n' = n$ and $S\ m' = m$, this is just what we wanted to show. ☐

Before we close this section and move on to some exercises, let's digress briefly and use *beq_nat_true* to prove a similar property of identifiers that we'll need in later chapters:

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

Exercise: 3 stars, recommended (gen_dep_practice) Prove this by induction on l.

```
Theorem nth_error_after_last: ∀ (n : nat) (X : Type) (l : list X),
     length l = n →
     nth_error l n = None.
Proof.
```

```
  intros. generalize dependent n. induction l.
  - simpl. reflexivity.
  - intros. induction n.
    + inversion H.
    + apply S_injective in H. apply IHl. apply H.
Qed.
```

☐

5.7 Unfolding Definitions

It sometimes happens that we need to manually unfold a Definition so that we can manipulate its right-hand side. For example, if we define...

```
Definition square n := n × n.
```

... and try to prove a simple fact about *square*...

```
Lemma square_mult : ∀ n m, square (n × m) = square n × square m.
Proof.
  intros n m.
  simpl.
```

... we get stuck: `simpl` doesn't simplify anything at this point, and since we haven't proved any other facts about *square*, there is nothing we can `apply` or `rewrite` with.

To make progress, we can manually `unfold` the definition of *square*:

```
  unfold square.
```

Now we have plenty to work with: both sides of the equality are expressions involving multiplication, and we have lots of facts about multiplication at our disposal. In particular, we know that it is commutative and associative, and from these facts it is not hard to finish the proof.

```
  rewrite mult_assoc.
  assert (H : n × m × n = n × n × m).
  { rewrite mult_comm. apply mult_assoc. }
  rewrite H. rewrite mult_assoc. reflexivity.
Qed.
```

At this point, a deeper discussion of unfolding and simplification is in order.

You may already have observed that tactics like `simpl`, `reflexivity`, and `apply` will often unfold the definitions of functions automatically when this allows them to make progress. For example, if we define *foo m* to be the constant 5...

```
Definition foo (x: nat) := 5.
```

then the `simpl` in the following proof (or the `reflexivity`, if we omit the `simpl`) will unfold *foo m* to (`fun` $x \Rightarrow 5$) 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 rather conservative. For example, if we define a slightly more complicated function involving a pattern match...

```
Definition bar x :=
  match x with
  | O ⇒ 5
  | S _ ⇒ 5
  end.
```

...then the analogous proof will get stuck:

```
Fact silly_fact_2_FAILED : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  simpl. Abort.
```

The reason that `simpl` doesn't make progress here is that it notices that, after tentatively unfolding *bar m*, it is left with a match whose scrutinee, *m*, is a variable, so the `match` cannot be simplified further. It is not smart enough to notice that the two branches of the `match` are identical, so it gives up on unfolding *bar m* and leaves it alone. Similarly, tentatively unfolding *bar* $(m+1)$ leaves a `match` whose scrutinee is a function application (that, itself, cannot be simplified, even after unfolding the definition of $+$), so `simpl` leaves it alone.

At this point, there are two ways to make progress. One is to use `destruct` *m* to break the proof into two cases, each focusing on a more concrete choice of *m* (*O* vs *S* _). In each case, the `match` inside of *bar* can now make progress, and the proof is easy to complete.

```
Fact silly_fact_2 : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  destruct m.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

This approach works, but it depends on our recognizing that the `match` hidden inside *bar* is what was preventing us from making progress.

A more straightforward way to make progress is to explicitly tell Coq to unfold *bar*.

```
Fact silly_fact_2' : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  unfold bar.
```

Now it is apparent that we are stuck on the `match` expressions on both sides of the =, and we can use `destruct` to finish the proof without thinking too hard.

```
  destruct m.
  - reflexivity.
  - reflexivity.
Qed.
```

5.8 Using `destruct` on Compound Expressions

We have seen many examples where `destruct` is used to perform case analysis of the value of some variable. But sometimes we need to reason by cases on the result of some *expression*. We can also do this with `destruct`.

Here are some examples:

```
Definition sillyfun (n : nat) : bool :=
  if beq_nat n 3 then false
  else if beq_nat n 5 then false
  else false.

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

After unfolding *sillyfun* in the above proof, we find that we are stuck on `if` (*beq_nat* n 3) `then` ... `else` .... But either n is equal to 3 or it isn't, so we can use `destruct` (*beq_nat* n 3) to let us reason about the two cases.

In general, the `destruct` tactic can be used to perform case analysis of the results of arbitrary computations. If e is an expression whose type is some inductively defined type T, then, for each constructor c of T, `destruct` e generates a subgoal in which all occurrences of e (in the goal and in the context) are replaced by c.

Exercise: 3 stars, optional (combine_split) 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.
  intros. unfold split. destruct l.
  - inversion H. reflexivity.
  -
Abort.
```

☐

However, destructing compound expressions requires a bit of care, as such destructs can sometimes erase information we need to complete a proof. For example, suppose we define a function *sillyfun1* like this:

```
Definition sillyfun1 (n : nat) : bool :=
  if beq_nat n 3 then true
  else if beq_nat n 5 then true
  else false.
```

Now suppose that we want to convince Coq of the (rather obvious) fact that *sillyfun1 n* yields *true* only when n is odd. By analogy with the proofs we did with *sillyfun* above, it is natural to start the proof like this:

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

We get stuck at this point because the context does not contain enough information to prove the goal! The problem is that the substitution performed by destruct is too brutal – it threw away every occurrence of *beq_nat n* 3, but we need to keep some memory of this expression and how it was destructed, because we need to be able to reason that, since *beq_nat n* 3 = *true* in this branch of the case analysis, it must be that $n = 3$, from which it follows that n is odd.

What we would really like is to substitute away all existing occurences of *beq_nat n* 3, but at the same time add an equation to the context that records which case we are in. The *eqn*: qualifier allows us to introduce such an equation, giving it a name that we choose.

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

      destruct (beq_nat n 5) eqn:Heqe5.
        +
          apply beq_nat_true in Heqe5.
          rewrite → Heqe5. reflexivity.
        + inversion eq. Qed.
```

Exercise: 2 stars (destruct_eqn_practice)

```
Theorem bool_fn_applied_thrice :
  ∀ (f : bool → bool) (b : bool),
  f (f (f b)) = f b.
Proof.
  intros. destruct (f b) eqn:H.
  - destruct b.
    + rewrite H. apply H.
    + destruct (f true) eqn:H2.
      × apply H2.
      × apply H.
  - destruct b.
    + destruct (f false) eqn:H2.
      × apply H.
      × apply H2.
    + rewrite H. apply H.
Qed.
```

□

5.9 Review

We've now seen many of Coq's most fundamental tactics. We'll introduce a few more in the coming chapters, and later on we'll see some more powerful *automation* tactics that make Coq help us with low-level details. But basically we've got what we need to get work done.

Here are the ones we've seen:

- `intros`: move hypotheses/variables from goal to context

- `reflexivity`: finish the proof (when the goal looks like $e = e$)
- `apply`: prove goal using a hypothesis, lemma, or constructor
- `apply`... `in` H: apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning)
- `apply`... `with`...: explicitly specify values for variables that cannot be determined by pattern matching
- `simpl`: simplify computations in the goal
- `simpl in` H: ... or a hypothesis
- `rewrite`: use an equality hypothesis (or lemma) to rewrite the goal
- `rewrite` ... `in` H: ... or a hypothesis
- `symmetry`: changes a goal of the form t=u into u=t
- `symmetry in` H: changes a hypothesis of the form t=u into u=t
- `unfold`: replace a defined constant by its right-hand side in the goal
- `unfold`... `in` H: ... or a hypothesis
- `destruct`... `as`...: case analysis on values of inductively defined types
- `destruct`... *eqn*:...: specify the name of an equation to be added to the context, recording the result of the case analysis
- `induction`... `as`...: induction on values of inductively defined types
- `inversion`: reason by injectivity and distinctness of constructors
- `assert` (H: e) (or `assert` (e) `as` H): introduce a "local lemma" e and call it H
- `generalize dependent` x: move the variable x (and anything else that depends on it) from the context back to an explicit hypothesis in the goal formula

5.10 Additional Exercises

Exercise: 3 stars (beq_nat_sym) `Theorem` *beq_nat_sym* : ∀ (*n m* : *nat*),
  beq_nat n m = *beq_nat m n*.

```
Proof.
  intros. induction n.
  - induction m.
    + reflexivity.
    + reflexivity.
  - induction m.
    + reflexivity.
    + simpl.
Admitted.
```

☐

Exercise: 3 stars, advanced, optional (beq_nat_sym_informal) Give an informal proof of this lemma that corresponds to your formal proof above:

Theorem: For any *nats* *n m*, *beq_nat n m* = *beq_nat m n*.

Proof: ☐

Exercise: 3 stars, optional (beq_nat_trans) `Theorem` *beq_nat_trans* : ∀ *n m p*,
  beq_nat n m = *true* →
  beq_nat m p = *true* →
  beq_nat n p = *true*.

```
Proof.
  intros. apply beq_nat_true in H. apply beq_nat_true in H0. rewrite H. rewrite H0.
rewrite ← beq_nat_refl. reflexivity.
Qed.
```

☐

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 :=
  ∀ (X Y: Type) (l1: list X) (l2: list Y), (length l1) = (length l2) →
    (split (combine l1 l2)) = (l1, l2).

Theorem split_combine : split_combine_statement.
Proof.
```

```
  unfold split_combine_statement. intros a b. induction l1.
  - destruct l2.
    + reflexivity.
    + intros. inversion H.
  - destruct l2.
    + intros. inversion H.
    + simpl. rewrite IHl1.
      × reflexivity.
      ×
Admitted.
```

```
Definition manual_grade_for_split_combine : option (prod 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 (beq_nat 5) □ = true

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

existsb (beq_nat 5) 0;2;3;6 = false

existsb (andb true) *true*;*true*;*false* = true

existsb oddb 1;0;0;0;0;3 = true

existsb evenb □ = false

Next, define a *nonrecursive* version of *existsb* – call it *existsb'* – using *forallb* and *negb*.

Finally, prove a theorem *existsb_existsb'* stating that *existsb'* and *existsb* have the same behavior.

```
Definition manual_grade_for_forall_exists_challenge : option (prod nat string) := None.
```

□

Chapter 6

Library SoftwareFoundationsExercises.Logic

6.1 Logic: Logic in Coq

Set *Warnings* "-notation-overridden,-parsing".
Require Export *Tactics*.

In previous chapters, we have seen many examples of factual claims (*propositions*) and ways of presenting evidence of their truth (*proofs*). In particular, we have worked extensively with *equality propositions* of the form $e1 = e2$, with implications ($P \rightarrow Q$), and with quantified propositions ($\forall\ x,\ P$). In this chapter, we will see how Coq can be used to carry out other familiar forms of logical reasoning.

Before diving into details, let's talk a bit about the status of mathematical statements in Coq. Recall that Coq is a *typed* language, which means that every sensible expression in its world has an associated type. Logical claims are no exception: any statement we might try to prove in Coq has a type, namely Prop, the type of *propositions*. We can see this with the Check command:

Check $3 = 3$.

Check $\forall\ n\ m : nat,\ n + m = m + n$.

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 $\forall\ n : nat,\ n = 2$.

Check $3 = 4$.

Indeed, propositions don't just have types: they are *first-class objects* that can be manipulated in the same ways as the other entities in Coq's world.

So far, we've seen one primary place that propositions can appear: in Theorem (and Lemma and Example) declarations.

```
Theorem plus_2_2_is_4 :
  2 + 2 = 4.
Proof. reflexivity. Qed.
```

But propositions can be used in many other ways. For example, we can give a name to a proposition using a Definition, just as we have given names to expressions of other sorts.

```
Definition plus_fact : Prop := 2 + 2 = 4.
Check plus_fact.
```

We can later use this name in any situation where a proposition is expected – for example, as the claim in a Theorem declaration.

```
Theorem plus_fact_is_true :
  plus_fact.
Proof. reflexivity. Qed.
```

We can also write *parameterized* propositions – that is, functions that take arguments of some type and return a proposition.

For instance, the following function takes a number and returns a proposition asserting that this number is equal to three:

```
Definition is_three (n : nat) : Prop :=
  n = 3.
Check is_three.
```

In Coq, functions that return propositions are said to define *properties* of their arguments.

For instance, here's a (polymorphic) property defining the familiar notion of an *injective function*.

```
Definition injective {A B} (f : A → B) :=
  ∀ x y : A, f x = f y → x = y.

Lemma succ_inj : injective S.
Proof.
  intros n m H. inversion H. reflexivity.
Qed.
```

The equality operator = is also a function that returns a Prop.

The expression $n = m$ is syntactic sugar for *eq* n m (defined using Coq's Notation mechanism). Because *eq* can be used with elements of any type, it is also polymorphic:

```
Check @eq.
```

(Notice that we wrote @*eq* instead of *eq*: The type argument A to *eq* is declared as implicit, so we need to turn off implicit arguments to see the full type of *eq*.)

6.2 Logical Connectives

6.2.1 Conjunction

The *conjunction*, or *logical and*, of propositions A and B is written $A \wedge B$, 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 $A \wedge B$ is also true.

```
Lemma and_intro : ∀ A B : Prop, A → B → A ∧ B.
Proof.
  intros A B HA HB. split.
  - apply HA.
  - apply HB.
Qed.
```

Since applying a theorem with hypotheses to some goal has the effect of generating as many subgoals as there are hypotheses for that theorem, we can apply *and_intro* to achieve the same effect as `split`.

```
Example and_example' : 3 + 4 = 7 ∧ 2 × 2 = 4.
Proof.
  apply and_intro.
  - reflexivity.
  - reflexivity.
Qed.
```

Exercise: 2 stars (and_exercise)
```
Example and_exercise :
  ∀ n m : nat, n + m = 0 → n = 0 ∧ m = 0.
Proof.
  intros. split.
  - destruct n.
    + reflexivity.
    + inversion H.
  - destruct m.
```

```
    + reflexivity.
    + rewrite plus_comm in H. inversion H.
Qed.
```

□

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 $A \wedge B$, writing `destruct` H `as` $[HA\ HB]$ will remove H from the context and add two new hypotheses: HA, stating that A is true, and HB, stating that B is true.

```
Lemma and_example2 :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  intros n m H.
  destruct H as [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

As usual, we can also destruct H right when we introduce it, instead of introducing and then destructing it:

```
Lemma and_example2' :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  intros n m [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

You may wonder why we bothered packing the two hypotheses $n = 0$ and $m = 0$ into a single conjunction, since we could have also stated the theorem with two separate premises:

```
Lemma and_example2'' :
  ∀ n m : nat, n = 0 → m = 0 → n + m = 0.
Proof.
  intros n m Hn Hm.
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

For this theorem, both formulations are fine. But it's important to understand how to work with conjunctive hypotheses because conjunctions often arise from intermediate steps in proofs, especially in bigger developments. Here's a simple example:

```
Lemma and_example3 :
  ∀ n m : nat, n + m = 0 → n × m = 0.
Proof.
```

```
  intros n m H.
  assert (H' : n = 0 ∧ m = 0).
  { apply and_exercise. apply H. }
  destruct H' as [Hn Hm].
  rewrite Hn. reflexivity.
Qed.
```

Another common situation with conjunctions is that we know $A \land B$ but in some context we need just A (or just B). The following lemmas are useful in such cases:

```
Lemma proj1 : ∀ P Q : Prop,
  P ∧ Q → P.
Proof.
  intros P Q [HP HQ].
  apply HP. Qed.
```

Exercise: 1 star, optional (proj2)

```
Lemma proj2 : ∀ P Q : Prop,
  P ∧ Q → Q.
Proof.
  intros. destruct H. apply H0.
Qed.
```

□

Finally, we sometimes need to rearrange the order of conjunctions and/or the grouping of multi-way conjunctions. The following commutativity and associativity theorems are handy in such cases.

```
Theorem and_commut : ∀ P Q : Prop,
  P ∧ Q → Q ∧ P.
Proof.
  intros P Q [HP HQ].
  split.
    - apply HQ.
    - apply HP. Qed.
```

Exercise: 2 stars (and_assoc) (In the following proof of associativity, notice how the *nested* `intros` pattern breaks the hypothesis $H : P \land (Q \land R)$ down into $HP : P$, $HQ : Q$, and $HR : R$. Finish the proof from there.)

```
Theorem and_assoc : ∀ P Q R : Prop,
  P ∧ (Q ∧ R) → (P ∧ Q) ∧ R.
Proof.
  intros P Q R [HP [HQ HR]]. split.
  - split.
    + apply HP.
    + apply HQ.
```

```
  - apply HR.
Qed.
```

☐

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.
```

6.2.2 Disjunction

Another important connective is the *disjunction*, or *logical or*, of two propositions: $A \vee B$ is true when either A or B is. (Alternatively, we can write *or* A B, where *or* : `Prop` $\rightarrow$ `Prop` $\rightarrow$ `Prop`.)

To use a disjunctive hypothesis in a proof, we proceed by case analysis, which, as for *nat* or other data types, can be done with `destruct` or `intros`. Here is an example:

```
Lemma or_example :
  ∀ n m : nat, n = 0 ∨ m = 0 → n × m = 0.
Proof.
  intros n m [Hn | Hm].
  -
    rewrite Hn. reflexivity.
  -
    rewrite Hm. rewrite ← mult_n_O.
    reflexivity.
Qed.
```

Conversely, to show that a disjunction holds, we need to show that one of its sides does. This is done via two tactics, `left` and `right`. As their names imply, the first one requires proving the left side of the disjunction, while the second requires proving its right side. Here is a trivial use...

```
Lemma or_intro : ∀ A B : Prop, A → A ∨ B.
Proof.
  intros A B HA.
  left.
  apply HA.
Qed.
```

... and a slightly more interesting example requiring both `left` and `right`:

```
Lemma zero_or_succ :
  ∀ n : nat, n = 0 ∨ n = S (pred n).
Proof.
  intros [|n].
  - left. reflexivity.
  - right. reflexivity.
```

```
Qed.
```

Exercise: 1 star (mult_eq_0) Lemma *mult_eq_0* :
  $\forall$ *n m*, $n \times m = 0 \rightarrow n = 0 \vee m = 0$.

```
Proof.
  intros. induction n.
  - left. reflexivity.
  - destruct m.
    + right. reflexivity.
    + inversion H.
Qed.
```

☐

Exercise: 1 star (or_commut) Theorem *or_commut* : $\forall$ *P Q* : Prop,
  $P \vee Q \rightarrow Q \vee P$.

```
Proof.
  intros P Q [HP | HQ].
  - right. apply HP.
  - left. apply HQ.
Qed.
```

☐

6.2.3 Falsehood and Negation

So far, we have mostly been concerned with proving that certain things are *true* – addition is commutative, appending lists is associative, etc. Of course, we may also be interested in *negative* results, showing that certain propositions are *not* true. In Coq, such negative statements are expressed with the negation operator $\neg$.

To see how negation works, recall the discussion of the *principle of explosion* from the *Tactics* chapter; it asserts that, if we assume a contradiction, then any other proposition can be derived. Following this intuition, we could define $\neg$ *P* ("not *P*") as $\forall$ *Q*, $P \rightarrow Q$. Coq actually makes a slightly different choice, defining $\neg$ *P* as $P \rightarrow$ *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` (or `inversion`) 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, 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.
  intros. apply H in H0. destruct H0.
Qed.
```

☐

This is how we use *not* to state that 0 and 1 are different elements of *nat*:

```
Theorem zero_not_one : ~(0 = 1).
Proof.
  intros contra. inversion contra.
Qed.
```

Such inequality statements are frequent enough to warrant a special notation, $x \neq y$:

```
Check (0 ≠ 1).
```

```
Theorem zero_not_one' : 0 ≠ 1.
Proof.
  intros H. inversion H.
Qed.
```

It takes a little practice to get used to working with negation in Coq. Even though you can see perfectly well why a statement involving negation is true, it can be a little tricky at first to get things into the right configuration so that Coq can understand it! Here are proofs of a few familiar facts to get you warmed up.

```
Theorem not_False :
  ¬ False.
Proof.
  unfold not. intros H. destruct H. Qed.
```

```
Theorem contradiction_implies_anything : ∀ P Q : Prop,
  (P ∧ ¬P) → Q.
Proof.
  intros P Q [HP HNA]. unfold not in HNA.
  apply HNA in HP. destruct HP. Qed.
```

```
Theorem double_neg : ∀ P : Prop,
  P → ~~P.
Proof.
  intros P H. unfold not. intros G. apply G. apply H. Qed.
```

Exercise: 2 stars, advanced, recommended (double_neg_inf) Write an informal proof of *double_neg*:

Theorem: P implies ~~P, for any proposition P.

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

☐

Exercise: 2 stars, recommended (contrapositive)

```
Theorem contrapositive : ∀ (P Q : Prop),
  (P → Q) → (~Q → ¬P).
Proof.
  intros P Q H1 H2 H3. apply H2 in H1.
  - apply H1.
  - apply H3.
Qed.
```

☐

Exercise: 1 star (not_both_true_and_false)

```
Theorem not_both_true_and_false : ∀ P : Prop,
  ¬ (P ∧ ¬P).
Proof.
  intros. intros [H1 H2].
  unfold not in H2. apply H2 in H1. apply H1.
Qed.
```

☐

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

```
Definition manual_grade_for_informal_not_PNP : option (prod 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.
```

6.2.4 Truth

Besides *False*, Coq's standard library also defines *True*, a proposition that is trivially true. To prove it, we use the predefined constant I : *True*:

```
Lemma True_is_true : True.
Proof. apply I. Qed.
```

Unlike *False*, which is used extensively, *True* is used quite rarely, since it is trivial (and therefore uninteresting) to prove as a goal, and it carries no useful information as a hypothesis. But it can be quite useful when defining complex `Prop`s using conditionals or as a parameter to higher-order `Prop`s. We will see examples of such uses of *True* later on.

6.2.5 Logical Equivalence

The handy "if and only if" connective, which asserts that two propositions have the same truth value, is just the conjunction of two implications.

```
Module MyIff.

Definition iff (P Q : Prop) := (P → Q) ∧ (Q → P).

Notation "P <-> Q" := (iff P Q)
                      (at level 95, no associativity)
```

: *type_scope*.

```
End MyIff.

Theorem iff_sym : ∀ P Q : Prop,
  (P ↔ Q) → (Q ↔ P).
Proof.
  intros P Q [HAB HBA].
  split.
  - apply HBA.
  - apply HAB. Qed.

Lemma not_true_iff_false : ∀ b,
  b ≠ true ↔ b = false.
Proof.
  intros b. split.
  - apply not_true_is_false.
  -
    intros H. rewrite H. intros H'. inversion H'.
Qed.
```

Exercise: 1 star, optional (iff_properties) Using the above proof that ↔ is symmetric (*iff_sym*) as a guide, prove that it is also reflexive and transitive.

```
Theorem iff_refl : ∀ P : Prop,
  P ↔ P.
Proof.
  intros. split.
  - intros. apply H.
  - intros. apply H.
Qed.

Theorem iff_trans : ∀ P Q R : Prop,
  (P ↔ Q) → (Q ↔ R) → (P ↔ R).
Proof.
  intros. split.
  - intros. apply H in H1. apply H0 in H1. apply H1.
  - intros. apply H0 in H1. apply H in H1. apply H1.
Qed.
```

☐

Exercise: 3 stars (or_distributes_over_and) Theorem *or_distributes_over_and* : ∀ P Q R : Prop,

```
  P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ R).
Proof.
  intros. split.
```

```
  - intros. destruct H.
    + split.
      × left. apply H.
      × left. apply H.
    + split. destruct H.
      × right. apply H.
      × right. inversion H. apply H1.
  - intros. destruct H as [[H1 | H2] [H3 | H4]].
    + left. apply H1.
    + left. apply H1.
    + left. apply H3.
    + right. split.
      × apply H2.
      × apply H4.
Qed.
```

□

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:

```
Require Import Coq.Setoids.Setoid.
```

Here is a simple example demonstrating how these tactics work with *iff*. First, let's prove a couple of basic iff equivalences...

```
Lemma mult_0 : ∀ n m, n × m = 0 ↔ n = 0 ∨ m = 0.
Proof.
  split.
  - apply mult_eq_0.
  - apply or_example.
Qed.

Lemma or_assoc :
  ∀ P Q R : Prop, P ∨ (Q ∨ R) ↔ (P ∨ Q) ∨ R.
Proof.
  intros P Q R. split.
  - intros [H | [H | H]].
    + left. left. apply H.
    + left. right. apply H.
    + right. apply H.
  - intros [[H | H] | H].
    + left. apply H.
    + right. left. apply H.
    + right. right. apply H.
```

Qed.

We can now use these facts with `rewrite` and `reflexivity` to give smooth proofs of statements involving equivalences. Here is a ternary version of the previous *mult_0* result:

Lemma *mult_0_3* :
  $\forall$ n m p, $n \times m \times p = 0 \leftrightarrow n = 0 \lor m = 0 \lor p = 0$.
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* :
  $\forall$ n m : *nat*, $n \times m = 0 \rightarrow n = 0 \lor m = 0$.
Proof.
  intros n m H. apply *mult_0*. apply H.
Qed.

6.2.6 Existential Quantification

Another important logical connective is *existential quantification*. To say that there is some x of type T such that some property P holds of x, we write $\exists$ 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* : $\exists$ n : *nat*, $4 = n + n$.
Proof.
  $\exists$ 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* : $\forall$ n,
  ($\exists$ m, $n = 4 + m$) $\rightarrow$
  ($\exists$ o, $n = 2 + o$).
Proof.
  intros n [m Hm]. $\exists$ $(2 + m)$.
  apply Hm. Qed.

Exercise: 1 star, 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.
  intros. unfold not in *. intros. inversion H0. apply H1 in H. apply H.
Qed.
```

☐

Exercise: 2 stars (dist_exists_or) Prove that existential quantification distributes over disjunction.

```
Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
  (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
Proof.
  intros. split.
  - intros [x [H1 | H2]]. left. ∃ x. apply H1. right. ∃ x. apply H2.
  - intros. destruct H.
    + destruct H. ∃ x. left. apply H.
    + destruct H. ∃ x. right. apply H.
Qed.
```

☐

6.3 Programming with Propositions

The logical connectives that we have seen provide a rich vocabulary for defining complex propositions from simpler ones. To illustrate, let's look at how to express the claim that an element x occurs in a list l. Notice that this property has a simple recursive structure:

- If l is the empty list, then x cannot occur on it, so the property "x appears in l" is simply false.
- Otherwise, l has the form $x' :: l'$. In this case, x occurs in l if either it is equal to x' or it occurs in l'.

We can translate this directly into a straightforward recursive function taking an element and a list and returning a proposition:

```
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] ⇒ False
  | x' :: l' ⇒ x' = x ∨ In x l'
  end.
```

When *In* is applied to a concrete list, it expands into a concrete sequence of nested disjunctions.

```
Example In_example_1 : In 4 [1; 2; 3; 4; 5].
Proof.
  simpl. right. right. right. left. reflexivity.
Qed.
```

```
Example In_example_2 :
  ∀ n, In n [2; 4] →
  ∃ n', n = 2 × n'.
Proof.
  simpl.
  intros n [H | [H | []]].
  - ∃ 1. rewrite ← H. reflexivity.
  - ∃ 2. rewrite ← H. reflexivity.
Qed.
```

(Notice the use of the empty pattern to discharge the last case *en passant.*)

We can also prove more generic, higher-level lemmas about *In*.

Note, in the next, how *In* starts out applied to a variable and only gets expanded when we do case analysis on this variable:

```
Lemma In_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
    In x l →
    In (f x) (map f l).
Proof.
  intros A B f l x.
  induction l as [|x' l' IHl'].
  -
    simpl. intros [].
  -
    simpl. intros [H | H].
    + rewrite H. left. reflexivity.
    + right. apply IHl'. apply H.
Qed.
```

This way of defining propositions recursively, though convenient in some cases, also has some drawbacks. In particular, it is subject to Coq's usual restrictions regarding the definition of recursive functions, e.g., the requirement that they be "obviously terminating." In the next chapter, we will see how to define propositions *inductively*, a different technique with its own set of strengths and limitations.

Exercise: 2 stars (In_map_iff)

```
Lemma In_map_iff :
  ∀ (A B : Type) (f : A → B) (l : list A) (y : B),
```

$In\ y\ (map\ f\ l) \leftrightarrow$
$\exists\ x, f\ x = y \land In\ x\ l.$

Proof.

Admitted.

□

Exercise: 2 stars (In_app_iff) Lemma *In_app_iff* : $\forall\ A\ l\ l'\ (a{:}A),$
$In\ a\ (l{+}{+}l') \leftrightarrow In\ a\ l \lor In\ a\ l'.$

Proof.

Admitted.

□

Exercise: 3 stars, recommended (All) Recall that functions returning propositions can be seen as *properties* of their arguments. For instance, if P has type $nat \to$ 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 $\{T : \texttt{Type}\}$ $(P : T \to \texttt{Prop})$ $(l : list\ T)$: Prop
. *Admitted.*

Lemma *All_In* :
$\forall\ T\ (P : T \to \texttt{Prop})\ (l : list\ T),$
$(\forall\ x, In\ x\ l \to P\ x) \leftrightarrow$
All $P\ l.$

Proof.

Admitted.

□

Exercise: 3 stars (combine_odd_even) Complete the definition of the *combine_odd_even* function below. It takes as arguments two properties of numbers, *Podd* and *Peven*, and it should return a property P such that $P\ n$ is equivalent to $Podd\ n$ when n is odd and equivalent to $Peven\ n$ otherwise.

Definition *combine_odd_even* $(Podd\ Peven : nat \to \texttt{Prop}) : nat \to$ Prop
. *Admitted.*

To test your definition, prove the following facts:

Theorem *combine_odd_even_intro* :
$\forall\ (Podd\ Peven : nat \to \texttt{Prop})\ (n : nat),$
$(oddb\ n = true \to Podd\ n) \to$
$(oddb\ n = false \to Peven\ n) \to$
$combine\_odd\_even\ Podd\ Peven\ n.$

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.
```

☐

6.4 Applying Theorems to Arguments

One feature of Coq that distinguishes it from many other proof assistants is that it treats *proofs* as first-class objects.

There is a great deal to be said about this, but it is not necessary to understand it in detail in order to use Coq. This section gives just a taste, while a deeper exploration can be found in the optional chapters *ProofObjects* and *IndPrinciples*.

We have seen that we can use the `Check` command to ask Coq to print the type of an expression. We can also use `Check` to ask what theorem a particular identifier refers to.

```
Check plus_comm.
```

Coq prints the *statement* of the *plus_comm* theorem in the same way that it prints the *type* of any term that we ask it to `Check`. Why?

The reason is that the identifier *plus_comm* actually refers to a *proof object* – a data structure that represents a logical derivation establishing of the truth of the statement $\forall\ n\ m : nat,\ n + m = m + n$. 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 $nat \rightarrow nat \rightarrow nat$, 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.
```

You can "use theorems as functions" in this way with almost all tactics that take a theorem name as an argument. Note also that theorem application uses the same inference mechanisms as function application; thus, it is possible, for example, to supply wildcards as arguments to be inferred, or to declare some hypotheses to a theorem as implicit by default. These features are illustrated in the proof below.

```
Example lemma_application_ex :
  ∀ {n : nat} {ns : list nat},
    In n (map (fun m ⇒ m × 0) ns) →
    n = 0.
Proof.
  intros n ns H.
  destruct (proj1 _ _ (In_map_iff _ _ _ _ _) H)
            as [m [Hm _]].
  rewrite mult_0_r in Hm. rewrite ← Hm. reflexivity.
Qed.
```

We will see many more examples of the idioms from this section in later chapters.

6.5 Coq vs. Set Theory

Coq's logical core, the *Calculus of Inductive Constructions*, differs in some important ways from other formal systems that are used by mathematicians for writing down precise and rigorous proofs. For example, in the most popular foundation for mainstream paper-and-pencil mathematics, Zermelo-Fraenkel Set Theory (ZFC), a mathematical object can potentially be a member of many different sets; a term in Coq's logic, on the other hand, is a member of at most one type. This difference often leads to slightly different ways of capturing informal mathematical concepts, but these are, by and large, quite natural and easy to work with. For example, instead of saying that a natural number n belongs to the set of even numbers, we would say in Coq that *ev n* holds, where *ev* : *nat* → Prop is a property describing even numbers.

However, there are some cases where translating standard mathematical reasoning into Coq can be either cumbersome or sometimes even impossible, unless we enrich the core logic with additional axioms. We conclude this chapter with a brief discussion of some of the most significant differences between the two worlds.

6.5.1 Functional Extensionality

The equality assertions that we have seen so far mostly have concerned elements of inductive types (*nat*, *bool*, etc.). But since Coq's equality operator is polymorphic, these are not the only possibilities – in particular, we can write propositions claiming that two *functions* are equal to each other:

```
Example function_equality_ex1 : plus 3 = plus (pred 4).
Proof. reflexivity. Qed.
```

In common mathematical practice, two functions f and g are considered equal if they produce the same outputs:

(forall x, f x = g x) -> f = g

This is known as the principle of *functional extensionality*.

Informally speaking, an "extensional property" is one that pertains to an object's observable behavior. Thus, functional extensionality simply means that a function's identity is completely determined by what we can observe from it – i.e., in Coq terms, the results we obtain after applying it.

Functional extensionality is not part of Coq's basic axioms. This means that some "reasonable" propositions are not provable.

```
Example function_equality_ex2 :
  (fun x ⇒ plus x 1) = (fun x ⇒ plus 1 x).
Proof.
Abort.
```

However, we can add functional extensionality to Coq's core logic using the `Axiom` command.

```
Axiom functional_extensionality : ∀ {X Y: Type}
                                    {f g : X → Y},
  (∀ (x:X), f x = g x) → f = g.
```

Using `Axiom` has the same effect as stating a theorem and skipping its proof using *Admitted*, but it alerts the reader that this isn't just something we're going to come back and fill in later!

We can now invoke functional extensionality in proofs:

```
Example function_equality_ex2 :
  (fun x ⇒ plus x 1) = (fun x ⇒ plus 1 x).
Proof.
  apply functional_extensionality. intros x.
  apply plus_comm.
Qed.
```

Naturally, we must be careful when adding new axioms into Coq's logic, as they may render it *inconsistent* – that is, they may make it possible to prove every proposition, including *False*!

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

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 (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.
Proof.
  intros. apply functional_extensionality. intros. unfold tr_rev.
Admitted.
```

□

6.5.2 Propositions and Booleans

We've seen two different ways of encoding logical facts in Coq: with *booleans* (of type *bool*), and with *propositions* (of type Prop).

For instance, to claim that a number n is even, we can say either

- (1) that *evenb n* returns *true*, or
- (2) that there exists some k such that n = *double k*. Indeed, these two notions of evenness are equivalent, as can easily be shown with a couple of auxiliary lemmas.

Of course, it would be very 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 (evenb_double_conv) Theorem *evenb_double_conv* : ∀ n,

```
  ∃ k, n = if evenb n then double k
                else S (double k).
Proof.
  intros. induction n.
```

```
  - ∃ 0. reflexivity.
  - rewrite evenb_S. destruct evenb.
    + simpl. ∃ n.
   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 ∃ *k*, *n* = *double k*.

Similarly, to state that two numbers *n* and *m* are equal, we can say either (1) that *beq_nat n m* returns *true* or (2) that *n* = *m*. Again, these two notions are equivalent.

```
Theorem beq_nat_true_iff : ∀ n1 n2 : nat,
  beq_nat n1 n2 = true ↔ n1 = n2.
Proof.
  intros n1 n2. split.
  - apply beq_nat_true.
  - intros H. rewrite H. rewrite ← beq_nat_refl. reflexivity.
Qed.
```

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

Equality provides an extreme example: knowing that *beq_nat n m* = *true* is generally of little direct help in the middle of a proof involving *n* and *m*; however, if we convert the statement to the equivalent form *n* = *m*, we can rewrite with it.

The case of even numbers is also interesting. Recall that, when proving the backwards direction of *even_bool_prop* (i.e., *evenb_double*, going from the propositional to the boolean claim), we used a simple induction on *k*. On the other hand, the converse (the *evenb_double_conv* exercise) required a clever generalization, since we can't directly prove (∃ *k*, *n* = *double k*) → *evenb n* = *true*.

For these examples, the propositional claims are more useful than their boolean counterparts, but this is not always the case. For instance, we cannot test whether a general proposition is true or not in a function definition; as a consequence, the following code fragment is rejected:

```
Fail Definition is_even_prime n :=
  if n = 2 then true
  else false.
```

Coq complains that $n = 2$ has type `Prop`, while it expects an elements of *bool* (or some other inductive type with two elements). The reason for this error message has to do with the *computational* nature of Coq's core language, which is designed so that every function that it can express is computable and total. One reason for this is to allow the extraction of executable programs from Coq developments. As a consequence, `Prop` in Coq does *not* have a universal case analysis operation telling whether any given proposition is true or false, since such an operation would allow us to write non-computable functions.

Although general non-computable properties cannot be phrased as boolean computations, it is worth noting that even many *computable* properties are easier to express using `Prop` than *bool*, since recursive function definitions are subject to significant restrictions in Coq. For instance, the next chapter shows how to define the property that a regular expression matches a given string using `Prop`. Doing the same with *bool* would amount to writing a regular expression matcher, which would be more complicated, harder to understand, and harder to reason about.

Conversely, an important side benefit of stating facts using booleans is enabling some proof automation through computation with Coq terms, a technique known as *proof by reflection*. Consider the following statement:

```
Example even_1000 : ∃ k, 1000 = double k.
```

The most direct proof of this fact is to give the value of k explicitly.

```
Proof. ∃ 500. reflexivity. Qed.
```

On the other hand, the proof of the corresponding boolean statement is even simpler:

```
Example even_1000' : evenb 1000 = true.
Proof. reflexivity. Qed.
```

What is interesting is that, since the two notions are equivalent, we can use the boolean formulation to prove the other one without mentioning the value 500 explicitly:

```
Example even_1000'' : ∃ k, 1000 = double k.
Proof. apply even_bool_prop. reflexivity. Qed.
```

Although we haven't gained much in terms of proof size in this case, larger proofs can often be made considerably simpler by the use of reflection. As an extreme example, the Coq proof of the famous *4-color theorem* uses reflection to reduce the analysis of hundreds of different cases to a boolean computation. We won't cover reflection in great detail, but it serves as a good example showing the complementary strengths of booleans and general propositions.

Exercise: 2 stars (logical_connectives) The following lemmas relate the propositional connectives studied in this chapter to the corresponding boolean operations.

```
Lemma andb_true_iff : ∀ b1 b2:bool,
  b1 && b2 = true ↔ b1 = true ∧ b2 = true.
Proof.
  intros. split.
```

```
  - intros.
Admitted.

Lemma orb_true_iff : ∀ b1 b2,
  b1 || b2 = true ↔ b1 = true ∨ b2 = true.
Proof.
   Admitted.
```
□

Exercise: 1 star (beq_nat_false_iff) The following theorem is an alternate "negative" formulation of *beq_nat_true_iff* that is more convenient in certain situations (we'll see examples in later chapters).

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

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

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

Lemma beq_list_true_iff :
  ∀ A (beq : A → A → bool),
    (∀ a1 a2, beq a1 a2 = true ↔ a1 = a2) →
    ∀ l1 l2, beq_list beq l1 l2 = true ↔ l1 = l2.
Proof.
   Admitted.
```
□

Exercise: 2 stars, recommended (All_forallb) Recall the function *forallb*, from the exercise *forall_exists_challenge* in chapter *Tactics*:

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool :=
  match l with
  | [] ⇒ true
  | x :: l' ⇒ andb (test x) (forallb test l')
  end.
```

Prove the theorem below, which relates *forallb* to the All property of the above exercise.

```
Theorem forallb_true_iff : ∀ X test (l : list X),
    forallb test l = true ↔ All (fun x ⇒ test x = true) l.
Proof.
    Admitted.
```

Are there any important properties of the function *forallb* which are not captured by this specification?

□

6.5.3 Classical vs. Constructive Logic

We have seen that it is not possible to test whether or not a proposition P holds while defining a Coq function. You may be surprised to learn that a similar restriction applies to *proofs*! In other words, the following intuitive reasoning principle is not derivable in Coq:

```
Definition excluded_middle := ∀ P : Prop,
  P ∨ ¬ P.
```

To understand operationally why this is the case, recall that, to prove a statement of the form $P \vee Q$, we use the `left` and `right` tactics, which effectively require knowing which side of the disjunction holds. But the universally quantified P in *excluded_middle* is an *arbitrary* proposition, which we know nothing about. We don't have enough information to choose which of `left` or `right` to apply, just as Coq doesn't have enough information to mechanically decide whether P holds or not inside a function.

However, if we happen to know that P is reflected in some boolean term b, then knowing whether it holds or not is trivial: we just have to check the value of b.

```
Theorem restricted_excluded_middle : ∀ P b,
  (P ↔ b = true) → P ∨ ¬ P.
Proof.
  intros P [] H.
  - left. rewrite H. reflexivity.
  - right. rewrite H. intros contra. inversion contra.
Qed.
```

In particular, the excluded middle is valid for equations $n = m$, between natural numbers n and m.

```
Theorem restricted_excluded_middle_eq : ∀ (n m : nat),
  n = m ∨ n ≠ m.
Proof.
  intros n m.
  apply (restricted_excluded_middle (n = m) (beq_nat n m)).
  symmetry.
  apply beq_nat_true_iff.
Qed.
```

It may seem strange that the general excluded middle is not available by default in Coq; after all, any given claim must be either true or false. Nonetheless, there is an advantage in not assuming the excluded middle: statements in Coq can make stronger claims than the analogous statements in standard mathematics. Notably, if there is a Coq proof of $\exists\ x,\ P\ x$, it is possible to explicitly exhibit a value of x for which we can prove $P\ x$ – in other words, every proof of existence is necessarily *constructive*.

Logics like Coq's, which do not assume the excluded middle, are referred to as *constructive logics*.

More conventional logical systems such as ZFC, in which the excluded middle does hold for arbitrary propositions, are referred to as *classical*.

The following example illustrates why assuming the excluded middle may lead to non-constructive proofs:

Claim: There exist irrational numbers a and b such that a ^ b is rational.

Proof: It is not difficult to show that $sqrt\ 2$ is irrational. If $sqrt\ 2$ ^ $sqrt\ 2$ is rational, it suffices to take $a = b = sqrt\ 2$ and we are done. Otherwise, $sqrt\ 2$ ^ $sqrt\ 2$ is irrational. In this case, we can take $a = sqrt\ 2$ ^ $sqrt\ 2$ and $b = sqrt\ 2$, since a ^ $b = sqrt\ 2$ ^ $(sqrt\ 2 \times sqrt\ 2) = sqrt\ 2$ ^ $2 = 2$. □

Do you see what happened here? We used the excluded middle to consider separately the cases where $sqrt\ 2$ ^ $sqrt\ 2$ is rational and where it is not, without knowing which one actually holds! Because of that, we wind up knowing that such a and b exist but we cannot determine what their actual values are (at least, using this line of argument).

As useful as constructive logic is, it does have its limitations: There are many statements that can easily be proven in classical logic but that have much more complicated constructive proofs, and there are some that are known to have no constructive proof at all! Fortunately, like functional extensionality, the excluded middle is known to be compatible with Coq's logic, allowing us to add it safely as an axiom. However, we will not need to do so in this book: the results that we cover can be developed entirely within constructive logic at negligible extra cost.

It takes some practice to understand which proof techniques must be avoided in constructive reasoning, but arguments by contradiction, in particular, are infamous for leading to non-constructive proofs. Here's a typical example: suppose that we want to show that there exists x with some property P, i.e., such that $P\ x$. We start by assuming that our conclusion is false; that is, $\neg\ \exists\ x,\ P\ x$. From this premise, it is not hard to derive $\forall\ x,\ \neg\ P\ x$. If we manage to show that this intermediate fact results in a contradiction, we arrive at an existence proof without ever exhibiting a value of x for which $P\ x$ holds!

The technical flaw here, from a constructive standpoint, is that we claimed to prove $\exists\ x,\ P\ x$ using a proof of $\neg\ \neg\ (\exists\ x,\ P\ x)$. Allowing ourselves to remove double negations from arbitrary statements is equivalent to assuming the excluded middle, as shown in one of the exercises below. Thus, this line of reasoning cannot be encoded in Coq without assuming additional axioms.

Exercise: 3 stars (excluded_middle_irrefutable) 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 $\neg (P \lor \neg P)$ and $\neg \neg (P \lor \neg P)$ (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 $P \lor \neg P$ as an axiom.

`Theorem` *excluded_middle_irrefutable*: $\forall$ (P:`Prop`),
$\neg \neg (P \lor \neg P)$.
`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 $\rightarrow$
$\forall$ (X:`Type`) (P : $X \rightarrow$ `Prop`),
$\neg (\exists\ x, \neg\ P\ x) \rightarrow (\forall\ x,\ P\ x)$.
`Proof.`
Admitted.
□

Exercise: 5 stars, optional (classical_axioms) For those who like a challenge, here is an exercise taken from the Coq'Art book by Bertot and Casteran (p. 123). Each of the following four statements, together with *excluded_middle*, can be considered as characterizing classical logic. We can't prove any of them in Coq, but we can consistently add any one of them as an axiom if we wish to work in classical logic.

Prove that all five propositions (these four plus *excluded_middle*) are equivalent.

`Definition` *peirce* := $\forall\ P\ Q$: `Prop`,
$((P \rightarrow Q)\text{-}{>}P)\text{-}{>}P$.

`Definition` *double_negation_elimination* := $\forall\ P$:`Prop`,
$\tilde{}\tilde{}P \rightarrow P$.

`Definition` *de_morgan_not_and_not* := $\forall\ P\ Q$:`Prop`,
$\tilde{}(\tilde{}P \land \neg Q) \rightarrow P \lor Q$.

`Definition` *implies_to_or* := $\forall\ P\ Q$:`Prop`,

$(P \rightarrow Q) \rightarrow (\tilde{}P \vee Q)$.

$\square$

Chapter 7

Library SoftwareFoundationsExercises.IndProp

7.1 IndProp: Inductively Defined Propositions

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

7.2 Inductively Defined Propositions

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

Recall that we have seen two ways of stating that a number n is even: We can say (1) $evenb\ n = true$, or (2) $\exists\ k,\ n = double\ k$. 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 definition of evenness works, let's imagine using it to show that 4 is even. By rule *ev_SS*, it suffices to show that 2 is even. This, in turn, is again guaranteed by rule *ev_SS*, as long as we can show that 0 is even. But this last fact follows directly from the *ev_0* rule.

We will see many definitions like this one during the rest of the course. For purposes of informal discussions, it is helpful to have a lightweight notation that makes them easy to read and write. *Inference rules* are one such notation:

(ev_0) ev 0

ev n

(ev_SS) ev (S (S n))

Each of the textual rules above is reformatted here as an inference rule; the intended reading is that, if the *premises* above the line all hold, then the *conclusion* below the line follows. For example, the rule *ev_SS* says that, if n satisfies *ev*, then $S\ (S\ n)$ also does. If a rule has no premises above the line, then its conclusion holds unconditionally.

We can represent a proof using these rules by combining rule applications into a *proof tree.* Here's how we might transcribe the above proof that 4 is even:

(ev_0) ev 0

(ev_SS) ev 2

(ev_SS) ev 4

Why call this a "tree" (rather than a "stack", for example)? Because, in general, inference rules can have multiple premises. We will see examples of this below.

Putting all of this together, we can translate the definition of evenness into a formal Coq definition using an `Inductive` declaration, where each constructor corresponds to an inference rule:

```
Inductive ev : nat → Prop :=
| ev_0 : ev 0
| ev_SS : ∀ n : nat, ev n → ev (S (S n)).
```

This definition is different in one crucial respect from previous uses of `Inductive`: its result is not a `Type`, but rather a function from *nat* to `Prop` – that is, a property of numbers. Note that we've already seen other inductive definitions that result in functions, such as *list*, whose type is `Type` → `Type`. What is new here is that, because the *nat* argument of *ev* appears *unnamed*, to the *right* of the colon, it is allowed to take different values in the types of different constructors: 0 in the type of *ev_0* and $S\ (S\ n)$ in the type of *ev_SS*.

In contrast, the definition of *list* names the X parameter *globally*, to the *left* of the colon, forcing the result of *nil* and *cons* to be the same (*list* X). Had we tried to bring *nat* to the left in defining *ev*, we would have seen an error:

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

("Parameter" here is Coq jargon for an argument on the left of the colon in an `Inductive` definition; "index" is used to refer to arguments on the right of the colon.)

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

Such "constructor theorems" have the same status as proven theorems. In particular, we can use Coq's `apply` tactic with the rule names to prove *ev* for particular numbers...

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

... or we can use function application syntax:

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

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

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

More generally, we can show that any number multiplied by 2 is even:

Exercise: 1 star (ev_double)

```
Theorem ev_double : ∀ n,
  ev (double n).
Proof.
  intros. rewrite double_plus. induction n.
  - apply ev_0.
  - rewrite ← plus_n_Sm. apply ev_SS. apply IHn.
Qed.
```

□

7.3 Using Evidence in Proofs

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

Introducing *ev* with an `Inductive` declaration tells Coq not only that the constructors *ev_0* and *ev_SS* are valid ways to build evidence that some number is even, but also that these two constructors are the *only* ways to build evidence that numbers are even (in the sense of *ev*).

In other words, if someone gives us evidence E for the assertion *ev n*, then we know that E must have one of two shapes:

- E is *ev_0* (and n is O), or
- E is *ev_SS n' E'* (and n is S (S n'), where E' is evidence for *ev n'*).

This suggests that it should be possible to analyze a hypothesis of the form *ev n* much as we do inductively defined data structures; in particular, it should be possible to argue by *induction* and *case analysis* on such evidence. Let's look at a few examples to see what this means in practice.

7.3.1 Inversion on Evidence

Suppose we are proving some fact involving a number n, and we are given *ev n* as a hypothesis. We already know how to perform case analysis on n using the `inversion` tactic, generating separate subgoals for the case where $n = O$ and the case where $n = S\ n'$ for some n'. But for some proofs we may instead want to analyze the evidence that *ev n directly*.

By the definition of *ev*, there are two cases to consider:

- If the evidence is of the form *ev_0*, we know that $n = 0$.
- Otherwise, the evidence must have the form *ev_SS n' E'*, where $n = S\ (S\ n')$ and *E'* is evidence for *ev n'*.

We can perform this kind of reasoning in Coq, again using the `inversion` tactic. Besides allowing us to reason about equalities involving constructors, `inversion` provides a case-analysis principle for inductively defined propositions. When used in this way, its syntax is similar to `destruct`: We pass it a list of identifiers separated by | characters to name the arguments to each of the possible constructors.

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

In words, here is how the inversion reasoning works in this proof:

- If the evidence is of the form *ev_0*, we know that $n = 0$. Therefore, it suffices to show that *ev* (*pred* (*pred* 0)) holds. By the definition of *pred*, this is equivalent to showing that *ev* 0 holds, which directly follows from *ev_0*.
- Otherwise, the evidence must have the form *ev_SS n' E'*, where $n = S\ (S\ n')$ and *E'* is evidence for *ev n'*. We must then show that *ev* (*pred* (*pred* (*S* (*S n'*)))) holds, which, after simplification, follows directly from *E'*.

This particular proof also works if we replace `inversion` by `destruct`:

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

The difference between the two forms is that `inversion` is more convenient when used on a hypothesis that consists of an inductive property applied to a complex expression (as opposed to a single variable). Here's is a concrete example. Suppose that we wanted to prove the following variation of *ev_minus2*:

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

Intuitively, we know that evidence for the hypothesis cannot consist just of the *ev_0* constructor, since *O* and *S* are different constructors of the type *nat*; hence, *ev_SS* is the only case that applies. Unfortunately, `destruct` is not smart enough to realize this, and it still generates two subgoals. Even worse, in doing so, it keeps the final goal unchanged, failing to provide any useful information for completing the proof.

```
Proof.
  intros n E.
  destruct E as [| n' E'].
  -

Abort.
```

What happened, exactly? Calling `destruct` has the effect of replacing all occurrences of the property argument by the values that correspond to each constructor. This is enough in the case of *ev_minus2'* because that argument, *n*, is mentioned directly in the final goal. However, it doesn't help in the case of *evSS_ev* since the term that gets replaced ($S\ (S\ n)$) is not mentioned anywhere.

The `inversion` tactic, on the other hand, can detect (1) that the first case does not apply, and (2) that the *n'* that appears on the *ev_SS* case must be the same as *n*. This allows us to complete the proof:

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

By using `inversion`, we can also apply the principle of explosion to "obviously contradictory" hypotheses involving inductive properties. For example:

```
Theorem one_not_even : ¬ ev 1.
Proof.
  intros H. inversion H. Qed.
```

Exercise: 1 star (SSSSev__even) Prove the following result using `inversion`.

```
Theorem SSSSev__even : ∀ n,
```

```
  ev (S (S (S (S n)))) → ev n.
Proof.
  intros. inversion H. inversion H1. apply H3.
Qed.
```
☐

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

```
Theorem even5_nonsense :
  ev 5 → 2 + 2 = 9.
Proof.
  intros. inversion H. inversion H1. inversion H3.
Qed.
```
☐

The way we've used `inversion` here may seem a bit mysterious at first. Until now, we've only used `inversion` on equality propositions, to utilize injectivity of constructors or to discriminate between different constructors. But we see here that `inversion` can also be applied to analyzing evidence for inductively defined propositions.

Here's how `inversion` works in general. Suppose the name *I* refers to an assumption *P* in the current context, where *P* has been defined by an `Inductive` declaration. Then, for each of the constructors of *P*, `inversion` *I* generates a subgoal in which *I* has been replaced by the exact, specific conditions under which this constructor could have been used to prove *P*. Some of these subgoals will be self-contradictory; `inversion` throws these away. The ones that are left represent the cases that must be proved to establish the original goal. For those, `inversion` adds all equations into the proof context that must hold of the arguments given to *P* (e.g., *S* (*S n'*) = *n* in the proof of *evSS_ev*).

The *ev_double* exercise above shows that our new notion of evenness is implied by the two earlier ones (since, by *even_bool_prop* in chapter *Logic*, we already know that those are equivalent to each other). To show that all three coincide, we just need the following lemma:

```
Lemma ev_even_firsttry : ∀ n,
  ev n → ∃ k, n = double k.
Proof.
```

We could try to proceed by case analysis or induction on *n*. But since *ev* is mentioned in a premise, this strategy would probably lead to a dead end, as in the previous section. Thus, it seems better to first try inversion on the evidence for *ev*. Indeed, the first case can be solved trivially.

```
  intros n E. inversion E as [| n' E'].
  -
    ∃ 0. reflexivity.
  - simpl.
```

Unfortunately, the second case is harder. We need to show ∃ *k*, *S* (*S n'*) = *double k*, but the only available assumption is *E'*, which states that *ev n'* holds. Since this isn't directly useful, it seems that we are stuck and that performing case analysis on E was a waste of time.

If we look more closely at our second goal, however, we can see that something interesting happened: By performing case analysis on E, we were able to reduce the original result to an similar one that involves a *different* piece of evidence for *ev*: *E'*. More formally, we can finish our proof by showing that

exists k', n' = double k',

which is the same as the original statement, but with *n'* instead of *n*. Indeed, it is not difficult to convince Coq that this intermediate result suffices.

```
    assert (I : (∃ k', n' = double k') →
                (∃ k, S (S n') = double k)).
    { intros [k' Hk']. rewrite Hk'. ∃ (S k'). reflexivity. }
    apply I.
Abort.
```

7.3.2 Induction on Evidence

If this looks familiar, it is no coincidence: We've encountered similar problems in the `Induction` chapter, when trying to use case analysis to prove results that required induction. And once again the solution is... induction!

The behavior of `induction` on evidence is the same as its behavior on data: It causes Coq to generate one subgoal for each constructor that could have used to build that evidence, while providing an induction hypotheses for each recursive occurrence of the property in question.

Let's try our current lemma again:

```
Lemma ev_even : ∀ n,
  ev n → ∃ k, n = double k.
Proof.
  intros n E.
  induction E as [|n' E' IH].
  -
    ∃ 0. reflexivity.
  -
    destruct IH as [k' Hk'].
    rewrite Hk'. ∃ (S k'). reflexivity.
Qed.
```

Here, we can see that Coq produced an *IH* that corresponds to *E'*, the single recursive occurrence of *ev* in its own definition. Since *E'* mentions *n'*, the induction hypothesis talks about *n'*, as opposed to *n* or some other number.

The equivalence between the second and third definitions of evenness now follows.

```
Theorem ev_even_iff : ∀ n,
```

$ev\ n \leftrightarrow \exists\ k,\ n = double\ k.$

```
Proof.
  intros n. split.
  - apply ev_even.
  - intros [k Hk]. rewrite Hk. apply ev_double.
Qed.
```

As we will see in later chapters, induction on evidence is a recurring technique across many areas, and in particular when formalizing the semantics of programming languages, where many properties of interest are defined inductively.

The following exercises provide simple examples of this technique, to help you familiarize yourself with it.

Exercise: 2 stars (ev_sum) `Theorem` *ev_sum* : $\forall\ n\ m,\ ev\ n \rightarrow ev\ m \rightarrow ev\ (n + m).$

```
Proof.
  intros. induction H.
  - apply H0.
  - apply ev_SS. apply IHev.
Qed.
```

☐

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

```
Inductive ev' : nat → Prop :=
| ev'_0 : ev' 0
| ev'_2 : ev' 2
| ev'_sum : ∀ n m, ev' n → ev' m → ev' (n + m).
```

Prove that this definition is logically equivalent to the old one. (You may want to look at the previous theorem when you get to the induction step.)

```
Theorem ev'_ev : ∀ n, ev' n ↔ ev n.
Proof.
  intros. split.
    - intros. induction H.
      + apply ev_0.
      + apply ev_SS. apply ev_0.
      + apply ev_sum. apply IHev'1. apply IHev'2.
    - intros. induction H.
      + apply ev'_0.
      + assert (H1 : (∀ n, S (S n) = 2 + n)).
        × intros. reflexivity.
        × rewrite H1. apply ev'_sum. apply ev'_2. apply IHev.
```

```
Qed.
```
☐

Exercise: 3 stars, advanced, recommended (ev_ev__ev) Finding the appropriate thing to do induction on is a bit tricky here:

```
Theorem ev_ev__ev : ∀ n m,
  ev (n+m) → ev n → ev m.
Proof.
  intros. induction H0.
  - simpl in H. apply H.
  - apply IHev. inversion H. apply H2.
Qed.
```
☐

Exercise: 3 stars, optional (ev_plus_plus) This exercise just requires applying existing lemmas. No induction or even case analysis is needed, though some of the rewriting may be tedious.

```
Theorem ev_plus_plus : ∀ n m p,
  ev (n+m) → ev (n+p) → ev (m+p).
Proof.
  intros. apply ev_sum.
Admitted.
```
☐

7.4 Inductive Relations

A proposition parameterized by a number (such as *ev*) can be thought of as a *property* – i.e., it defines a subset of *nat*, namely those numbers for which the proposition is provable. In the same way, a two-argument proposition can be thought of as a *relation* – i.e., it defines a set of pairs for which the proposition is provable.

```
Module Playground.
```

One useful example is the "less than or equal to" relation on numbers.

The following definition should be fairly intuitive. It says that there are two ways to give evidence that one number is less than or equal to another: either observe that they are the same number, or give evidence that the first is less than or equal to the predecessor of the second.

```
Inductive le : nat → nat → Prop :=
  | le_n : ∀ n, le n n
  | le_S : ∀ n m, (le n m) → (le n (S m)).

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

Proofs of facts about $\leq$ using the constructors *le_n* and *le_S* follow the same patterns as proofs about properties, like *ev* above. We can `apply` the constructors to prove $\leq$ 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 $(2 \leq 1) \rightarrow 2+2=5$.)

Here are some sanity checks on the definition. (Notice that, although these are the same kind of simple "unit tests" as we gave for the testing functions we wrote in the first few lectures, we must construct their proofs explicitly – `simpl` and `reflexivity` don't do the job, because the proofs aren't just a matter of simplifying computations.)

```
Theorem test_le1 :
  3 ≤ 3.
Proof.
  apply le_n. Qed.

Theorem test_le2 :
  3 ≤ 6.
Proof.
  apply le_S. apply le_S. apply le_S. apply le_n. Qed.

Theorem test_le3 :
  (2 ≤ 1) → 2 + 2 = 5.
Proof.
  intros H. inversion H. inversion H2. Qed.
```

The "strictly less than" relation $n < m$ can now be defined in terms of *le*.

```
End Playground.

Definition lt (n m:nat) := le (S n) m.

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

Here are a few more simple relations on numbers:

```
Inductive square_of : nat → nat → Prop :=
  | sq : ∀ n:nat, square_of n (n × n).

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

Inductive next_even : nat → nat → Prop :=
  | ne_1 : ∀ n, ev (S n) → next_even n (S n)
  | ne_2 : ∀ n, ev (S (S n)) → next_even n (S (S n)).
```

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

☐

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

□

Exercise: 3 stars, optional (le_exercises) Here are a number of facts about the $\leq$ 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.
  intros. rewrite ← H0. apply H.
Qed.

Theorem O_le_n : ∀ n,
  0 ≤ n.
Proof.
  intros. induction n.
  - reflexivity.
  - apply le_S. apply IHn.
Qed.

Theorem n_le_m__Sn_le_Sm : ∀ n m,
  n ≤ m → S n ≤ S m.
Proof.
  intros. induction H.
  - reflexivity.
  - apply le_S. apply IHle.
Qed.

Theorem Sn_le_Sm__n_le_m : ∀ n m,
  S n ≤ S m → n ≤ m.
Proof.
  intros. inversion H.
  - reflexivity.
  - apply le_trans with (n := (S n)).
    + apply le_S. reflexivity.
    + apply H1.
Qed.

Theorem le_plus_l : ∀ a b,
  a ≤ a + b.
Proof.
  intros. induction b.
  - rewrite ← plus_n_O. reflexivity.
  - rewrite ← plus_n_Sm. apply le_S. apply IHb.
Qed.

Theorem plus_lt : ∀ n1 n2 m,
  n1 + n2 < m →
```

```
  n1 < m ∧ n2 < m.
Proof.
unfold lt. intros. split.
- induction n2.
  + rewrite ← plus_n_O in H. apply H.
  + apply IHn2.
Admitted.
```

```
Theorem lt_S : ∀ n m,
  n < m →
  n < S m.
Proof.
  unfold lt. intros. apply le_S. apply H.
Qed.
```

```
Theorem leb_complete : ∀ n m,
  leb n m = true → n ≤ m.
Proof.
  intros. induction n.
  - apply O_le_n.
  - induction m.
    + inversion H.
    + inversion H.
Admitted.
```

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

```
Theorem leb_correct : ∀ n m,
  n ≤ m →
  leb n m = true.
Proof.
  intros.
Admitted.
```

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

```
Theorem leb_true_trans : ∀ n m o,
  leb n m = true → leb m o = true → leb n o = true.
Proof.
  intros. apply leb_complete in H. apply leb_complete in H0. apply leb_correct. apply
le_trans with (n := m). apply H. apply H0.
Qed.
```

☐

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

```
  leb n m = true ↔ n ≤ m.
Proof.
```

```
  intros. split.
  - apply leb_complete.
  - apply leb_correct.
Qed.
```

☐

```
Module R.
```

Exercise: 3 stars, 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, R m n o → R (S m) n (S o)
   | c3 : ∀ m n o, R m n o → R m (S n) (S o)
   | c4 : ∀ m n o, R (S m) (S n) (S (S o)) → R m n o
   | c5 : ∀ m n o, 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 (prod nat string) := None.
```

☐

Exercise: 3 stars, optional (R_fact) The relation R above actually encodes a familiar function. Figure out which function; then state and prove this equivalence in Coq?

```
Definition fR : nat → nat → nat
  . Admitted.

Theorem R_equiv_fR : ∀ m n o, R m n o ↔ fR m n = o.
Proof.
   Admitted.
```

☐

```
End R.
```

Exercise: 4 stars, advanced (subsequence) A list is a *subsequence* of another list if all of the elements in the first list occur in the same order in the second list, possibly with some extra elements in between. For example,

1;2;3

is a subsequence of each of the lists

1;2;3 1;1;1;2;2;3 1;2;7;3 5;6;1;9;9;2;7;3;8

but it is *not* a subsequence of any of the lists

1;2 1;3 5;6;2;1;7;3;8.

- Define an inductive proposition *subseq* on *list nat* that captures what it means to be a subsequence. (Hint: You'll need three cases.)
- Prove *subseq_refl* that subsequence is reflexive, that is, any list is a subsequence of itself.
- Prove *subseq_app* that for any lists *l1*, *l2*, and *l3*, if *l1* is a subsequence of *l2*, then *l1* is also a subsequence of *l2* ++ *l3*.
- (Optional, harder) Prove *subseq_trans* that subsequence is transitive – that is, if *l1* is a subsequence of *l2* and *l2* is a subsequence of *l3*, then *l1* is a subsequence of *l3*. Hint: choose your induction carefully!

`Definition` *manual_grade_for_subsequence* : *option* (*prod nat string*) := *None*.
☐

Exercise: 2 stars, 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]

☐

7.5 Case Study: Regular Expressions

The *ev* property provides a simple example for illustrating inductive definitions and the basic techniques for reasoning about them, but it is not terribly exciting – after all, it is equivalent to the two non-inductive 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 strings, defined as follows:

```
Inductive reg_exp {T : Type} : Type :=
| EmptySet : reg_exp
| EmptyStr : reg_exp
| Char : T → reg_exp
| App : reg_exp → reg_exp → reg_exp
| Union : reg_exp → reg_exp → reg_exp
| Star : reg_exp → 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* ++ ... ++ *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*,
exp_match s1 re1 →
exp_match s2 re2 →
exp_match (*s1* ++ *s2*) (*App re1 re2*)
| *MUnionL* : ∀ *s1 re1 re2*,
exp_match s1 re1 →
exp_match s1 (*Union re1 re2*)
| *MUnionR* : ∀ *re1 s2 re2*,
exp_match s2 re2 →
exp_match s2 (*Union re1 re2*)
| *MStar0* : ∀ *re*, *exp_match* [] (*Star re*)
| *MStarApp* : ∀ *s1 s2 re*,
exp_match s1 re →
exp_match s2 (*Star re*) →
exp_match (*s1* ++ *s2*) (*Star re*).

Again, for readability, we can also display this definition using inference-rule notation. At the same time, let's introduce a more readable infix notation.

`Notation` "s =~ re" := (*exp_match s re*) (`at level` 80).

(MEmpty) □ =~ EmptyStr

(MChar) x =~ Char x
s1 =~ re1 s2 =~ re2

(MApp) s1 ++ s2 =~ App re1 re2
s1 =~ re1

(MUnionL) s1 =~ Union re1 re2
s2 =~ re2

(MUnionR) s2 =~ Union re1 re2

(MStar0) □ =~ Star re
s1 =~ re s2 =~ Star re

(MStarApp) s1 ++ s2 =~ Star re

Notice that these rules are not *quite* the same as the informal ones that we gave at the beginning of the section. First, we don't need to include a rule explicitly stating that no string matches *EmptySet*; we just don't happen to include any rule that would have the effect of some string matching *EmptySet*. (Indeed, the syntax of inductive definitions doesn't even *allow* us to give such a "negative rule.")

Second, the informal rules for *Union* and *Star* correspond to two constructors each: *MUnionL* / *MUnionR*, and *MStar0* / *MStarApp*. The result is logically equivalent to the original rules but more convenient to use in Coq, since the recursive occurrences of *exp_match* are given as direct arguments to the constructors, making it easier to perform induction on evidence. (The *exp_match_ex1* and *exp_match_ex2* exercises below ask you to prove that the constructors given in the inductive declaration and the ones that would arise from a more literal transcription of the informal rules are indeed equivalent.)

Let's illustrate these rules with a few examples.

```
Example reg_exp_ex1 : [1] =~ Char 1.
Proof.
  apply MChar.
Qed.
```

```
Example reg_exp_ex2 : [1; 2] =~ App (Char 1) (Char 2).
Proof.
  apply (MApp [1] _ [2]).
  - apply MChar.
  - apply MChar.
Qed.
```

(Notice how the last example applies *MApp* to the strings [1] and [2] directly. Since the goal mentions [1; 2] instead of [1] ++ [2], Coq wouldn't be able to figure out how to split the string on its own.)

Using `inversion`, we can also show that certain strings do *not* match a regular expression:

```
Example reg_exp_ex3 : ¬ ([1; 2] =~ Char 1).
Proof.
  intros H. inversion H.
Qed.
```

We can define helper functions 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 (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.
  intros. intros H. inversion H.
Qed.
```

```
Lemma MUnion' : ∀ T (s : list T) (re1 re2 : @reg_exp T),
  s =~ re1 ∨ s =~ re2 →
  s =~ Union re1 re2.
Proof.
  intros. destruct H.
  - apply MUnionL. apply H.
  - apply MUnionR. apply H.
Qed.
```

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.
  intros. induction ss.
  - apply MStar0.
  - apply MStarApp.
    + apply H. left. reflexivity.
    + apply IHss. intros. apply H. right. apply H0.
Qed.
```

☐

Exercise: 4 stars, 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*, as opposed to *re*: The latter would only provide an induction hypothesis for strings that match *re*, which would not allow us to reason about the case *In x s2*.

```
  -
    simpl. rewrite In_app_iff in Hin.
    destruct Hin as [Hin | Hin].
    +
      apply (IH1 Hin).
    +
      apply (IH2 Hin).
Qed.
```

Exercise: 4 stars (re_not_empty) Write a recursive function *re_not_empty* that tests whether a regular expression matches some string. Prove that your function is correct.

```
Fixpoint re_not_empty {T : Type} (re : @reg_exp T) : bool :=
  match re with
```

```
  | EmptySet ⇒ false
  | EmptyStr ⇒ true
  | Char x ⇒ true
  | App r1 r2 ⇒ (re_not_empty r1) && (re_not_empty r2)
  | Union r1 r2 ⇒ (re_not_empty r1) || (re_not_empty r2)
  | Star x ⇒ true
end.
```

```
Lemma re_not_empty_correct : ∀ T (re : @reg_exp T),
  (∃ s, s =~ re) ↔ re_not_empty re = true.
Proof.
  intros. split.
  - intros. inversion H. induction H0.
    + reflexivity.
    + reflexivity.
    + simpl. rewrite IHexp_match1. rewrite IHexp_match2. reflexivity.
      ×
Admitted.
```

□

7.5.1 The *remember* Tactic

One potentially confusing feature of the `induction` tactic is that it happily lets you try to set up an induction over a term that isn't sufficiently general. The effect of this is to lose information (much as `destruct` can do), and leave you unable to complete the proof. Here's an example:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : @reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
```

Just doing an `inversion` on *H1* won't get us very far in the recursive cases. (Try it!). So we need induction (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` than like `inversion`.)

We can solve this problem by generalizing over the problematic expressions with an explicit equality:

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

Invoking the tactic *remember e* `as` *x* causes Coq to (1) replace all occurrences of the expression *e* by the variable *x*, and (2) add an equation *x = e* to the context. Here's how we can use it to show the above result:

```
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, which allows us to conclude immediately.

```
  - inversion Heqre'.
  - inversion Heqre'.
  - inversion Heqre'.
  - inversion Heqre'.
  - inversion Heqre'.
```

The interesting cases are those that correspond to *Star*. Note that the induction hypothesis *IH2* on the *MStarApp* case mentions an additional premise *Star re'' = Star re'*, which results from the equality generated by *remember*.

```
  -
    inversion Heqre'. intros s H. apply H.
  -
    inversion Heqre'. rewrite H0 in IH2, Hmatch1.
    intros s2 H1. rewrite <- app_assoc.
    apply MStarApp.
    + apply Hmatch1.
    + apply IH2.
      * reflexivity.
      * apply H1.
Qed.
```

Exercise: 4 stars, 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'' : forall T (s : list T) (re : reg_exp),
  s =~ Star re ->
  exists ss : list (list T),
    s = fold app ss []
    /\ forall s', In s' ss -> s' =~ re.
Proof.
  intros. remember (Star re) as re'. induction H.
  - inversion Heqre'.
Admitted.
```

□

Exercise: 5 stars, advanced (pumping) One of the first really interesting theorems in the theory of regular expressions is the so-called *pumping lemma*, which states, informally, that any sufficiently long string s matching a regular expression re can be "pumped" by repeating some middle section of s an arbitrary number of times to produce a new string also matching re.

To begin, we need to define "sufficiently long." Since we are working in a constructive logic, we actually need to be able to calculate, for each regular expression re, the minimum length for strings s to guarantee "pumpability."

```
Module Pumping.

Fixpoint pumping_constant {T} (re : @reg_exp T) : nat :=
  match re with
  | EmptySet => 0
  | EmptyStr => 1
  | Char _ => 2
  | App re1 re2 =>
      pumping_constant re1 + pumping_constant re2
  | Union re1 re2 =>
      pumping_constant re1 + pumping_constant re2
  | Star _ => 1
  end.
```

Next, it is useful to define an auxiliary function that repeats a string (appends it to itself) some number of times.

```
Fixpoint napp {T} (n : nat) (l : list T) : list T :=
  match n with
  | 0 => []
  | S n' => l ++ napp n' l
  end.

Lemma napp_plus: ∀ T (n m : nat) (l : list T),
  napp (n + m) l = napp n l ++ napp m l.
Proof.
  intros T n m l.
  induction n as [|n IHn].
  - reflexivity.
  - simpl. rewrite IHn, app_assoc. reflexivity.
Qed.
```

Now, the pumping lemma itself says that, if s =~ re and if the length of s is at least the pumping constant of re, then s can be split into three substrings $s1$ ++ $s2$ ++ $s3$ in such a way that $s2$ can be repeated any number of times and the result, when combined with $s1$ and $s3$ will still match re. Since $s2$ is also guaranteed not to be the empty string, this gives us a (constructive!) way to generate strings matching re that are as long as we like.

```
Lemma pumping : ∀ T (re : @reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
```

```
∃ s1 s2 s3,
  s = s1 ++ s2 ++ s3 ∧
  s2 ≠ [] ∧
  ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
```

To streamline the proof (which you are to fill in), the `omega` tactic, which is enabled by the following `Require`, is helpful in several places for automatically completing tedious low-level arguments involving equalities or inequalities over natural numbers. We'll return to `omega` in a later chapter, but feel free to experiment with it now if you like. The first case of the induction gives an example of how it is used.

```
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.
```

□

7.6 Case Study: Improving Reflection

We've seen in the *Logic* chapter that we often need to relate boolean computations to statements in `Prop`. But performing this conversion 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 (beq_nat n) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (beq_nat n m) eqn:H.
    +
      intros _. rewrite beq_nat_true_iff in H. rewrite H.
      left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
```

```
Qed.
```

In the first branch after `destruct`, we explicitly apply the *beq_nat_true_iff* lemma to the equation generated by destructing *beq_nat n m*, to convert the assumption *beq_nat n m* = *true* into the assumption $n = m$; then we had to `rewrite` using this assumption to complete the case.

We can streamline this by defining an inductive proposition that yields a better case-analysis principle for *beq_nat n m*. Instead of generating an equation such as *beq_nat n m* = *true*, which is generally not directly useful, this principle gives us right away the assumption we really need: $n = m$.

```
Inductive reflect (P : Prop) : bool → Prop :=
| ReflectT : P → reflect P true
| ReflectF : ¬ P → reflect P false.
```

The *reflect* property takes two arguments: a proposition P and a boolean b. Intuitively, it states that the property P is *reflected* in (i.e., equivalent to) the boolean b: that is, P holds if and only if b = *true*. To see this, notice that, by definition, the only way we can produce evidence that *reflect P true* holds is by showing that P is true and using the *ReflectT* constructor. If we invert this statement, this means that it should be possible to extract evidence for P from a proof of *reflect P true*. Conversely, the only way to show *reflect P false* is by combining evidence for ¬ P with the *ReflectF* constructor.

It is easy to formalize this intuition and show that the two statements are indeed equivalent:

```
Theorem iff_reflect : ∀ P b, (P ↔ b = true) → reflect P b.
Proof.
  intros P b H. destruct b.
  - apply ReflectT. rewrite H. reflexivity.
  - apply ReflectF. rewrite H. intros H'. inversion H'.
Qed.
```

Exercise: 2 stars, recommended (reflect_iff) `Theorem` *reflect_iff* : ∀ P b, *reflect P b* → (P ↔ b = *true*).

```
Proof.
  intros. split. destruct H.
  - intros. reflexivity.
  - intros. apply H in H0. inversion H0.
  - intros. destruct H. apply H. inversion H0.
Qed.
```

☐

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 ¬ P in the second).

```
Lemma beq_natP : ∀ n m, reflect (n = m) (beq_nat n m).
Proof.
  intros n m. apply iff_reflect. rewrite beq_nat_true_iff. reflexivity.
Qed.
```

The new proof of *filter_not_empty_In* now goes as follows. Notice how the calls to `destruct` and `apply` are combined into a single call to `destruct`.

(To see this clearly, look at the two proofs of *filter_not_empty_In* with Coq and observe the differences in proof state at the beginning of the first case of the `destruct`.)

```
Theorem filter_not_empty_In' : ∀ n l,
  filter (beq_nat n) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (beq_natP n m) as [H | H].
    +
      intros _. rewrite H. left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
Qed.
```

Exercise: 3 stars, recommended (beq_natP_practice) Use *beq_natP* as above to prove the following:

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

```
Theorem beq_natP_practice : ∀ n l,
  count n l = 0 → ~(In n l).
Proof.
  intros. induction l.
  - unfold not. intros. inversion H0.
  - simpl in H. destruct (beq_natP n x).
    + inversion H.
    + intros H2. apply IHl in H. inversion H2.
      × apply H0. symmetry in H1. apply H1.
      × apply H in H1. apply H1.
Qed.
```

□

In this small example, this technique gives us only a rather small gain in convenience for the proofs we've seen; however, using *reflect* consistently often leads to noticeably shorter and clearer scripts as proofs get larger. We'll see many more examples in later chapters and in *Programming Language Foundations*.

The use of the *reflect* property was popularized by *SSReflect*, a Coq library that has been used to formalize important results in mathematics, including as the 4-color theorem and the Feit-Thompson theorem. The name SSReflect stands for *small-scale reflection*, i.e., the pervasive use of reflection to simplify small proof steps with boolean computations.

7.7 Additional Exercises

Exercise: 3 stars, 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 :=
  | nostutter_null : nostutter nil
  | nostutter_1 : ∀ x, nostutter [x]
  | nostutter_2 : ∀ x y z, x ≠ y → nostutter (y :: z) → nostutter (x :: y :: z).
```

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].
Proof. repeat constructor; apply beq_nat_false_iff; auto. Qed.

Example test_nostutter_2: nostutter (@nil nat).
Proof. repeat constructor; apply beq_nat_false_iff; auto. Qed.

Example test_nostutter_3: nostutter [5].
Proof. repeat constructor; apply beq_nat_false_iff; auto. Qed.

Example test_nostutter_4: not (nostutter [3;1;1;4]).
Proof. intro.
  repeat match goal with
    h: nostutter _ ⊢ _ ⇒ inversion h; clear h; subst
  end.
```

contradiction H1 ; `auto`.
`Qed`.

`Definition` *manual_grade_for_nostutter* : *option* (*prod 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*: $X \rightarrow bool$, 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* (*prod 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, 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* (*prod nat string*) := *None*.
□

Exercise: 5 stars, optional (palindrome_converse) Again, the converse direction is significantly more difficult, due to the lack of evidence. Using your definition of *pal* from the previous exercise, prove that

forall l, l = rev l -> pal l.

□

Exercise: 4 stars, advanced, optional (NoDup) Recall the definition of the *In* property from the *Logic* chapter, which asserts that a value *x* appears at least once in a list *l*:

Your first task is to use *In* to define a proposition *disjoint X l1 l2*, which should be provable exactly when *l1* and *l2* are lists (with elements of type X) that have no elements in common.

Next, use *In* to define an inductive proposition *NoDup X l*, which should be provable exactly when *l* is a list (with elements of type *X*) where every member is different from every other. For example, *NoDup nat* [1;2;3;4] and *NoDup bool* [] should be provable, while *NoDup nat* [1;2;1] and *NoDup bool* [*true*;*true*] should not be.

Finally, state and prove one or more interesting theorems relating *disjoint*, *NoDup* and ++ (list append).

Definition *manual_grade_for_NoDup_disjoint_etc* : *option* (*prod 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.
  intros. induction l.
  - inversion H.
```

```
  - destruct H.
Admitted.
```

Now define a property *repeats* such that *repeats X l* asserts that *l* contains at least one repeated element (of type *X*).

```
Inductive repeats {X:Type} : list X → Prop :=

.
```

Now, here's a way to formalize the pigeonhole principle. Suppose list *l2* represents a list of pigeonhole labels, and list *l1* represents the labels assigned to a list of items. If there are more items than labels, at least two items must have the same label – i.e., list *l1* must contain repeats.

This proof is much easier if you use the *excluded_middle* hypothesis to show that *In* is decidable, i.e., ∀ *x l*, (*In x l*) ∨ ¬ (*In x l*). However, it is also possible to make the proof go through *without* assuming that *In* is decidable; if you manage to do this, you will not need the *excluded_middle* hypothesis.

```
Theorem pigeonhole_principle: ∀ (X:Type) (l1 l2:list X),
    excluded_middle →
    (∀ x, In x l1 → In x l2) →
    length l2 < length l1 →
    repeats l1.
Proof.
    intros X l1. induction l1 as [|x l1' IHl1'].
    Admitted.

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

□

7.7.1 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 represeneted 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* : ∀ (*P* : `Prop`), *P* → (*P* ↔ *True*).

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

Each `Prop` whose negation is provable is equivalent to *False*. `Lemma` *not_equiv_false* : ∀ (*P* : `Prop`), ¬*P* → (*P* ↔ *False*).

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

EmptySet matches no string. `Lemma` *null_matches_none* : ∀ (*s* : *string*), (*s* =~ *EmptySet*) ↔ *False*.

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

EmptyStr only matches the empty string. `Lemma` *empty_matches_eps* : ∀ (*s* : *string*), *s* =~ *EmptyStr* ↔ *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, 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* : $\forall$ (*a* : *ascii*) *s re0 re1*,

$a :: s =\tilde{}\ (App\ re0\ re1) \leftrightarrow$
$([\,] =\tilde{}\ re0 \land a :: s =\tilde{}\ re1) \lor$
$\exists\ s0\ s1,\ s = s0 ++ s1 \land a :: s0 =\tilde{}\ re0 \land s1 =\tilde{}\ re1.$

Proof.

Admitted.

□

s matches *Union re0 re1* iff *s* matches *re0* or *s* matches *re1*. Lemma *union_disj* : $\forall$ (*s* : *string*) *re0 re1*,

$s =\tilde{}\ Union\ re0\ re1 \leftrightarrow s =\tilde{}\ re0 \lor s =\tilde{}\ re1.$

Proof.

intros. split.
- intros. inversion *H*.
  + left. apply *H2*.
  + right. apply *H2*.
- intros [*H* | *H*].
  + apply *MUnionL*. apply *H*.
  + apply *MUnionR*. apply *H*.

Qed.

Exercise: 3 stars, 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 $a :: s =\tilde{}\ Star\ re$. The other implication can be proved without induction.

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

Lemma *star_ne* : $\forall$ (*a* : *ascii*) *s re*,

$a :: s =\tilde{}\ Star\ re \leftrightarrow$
$\exists\ s0\ s1,\ s = s0 ++ s1 \land a :: s0 =\tilde{}\ re \land s1 =\tilde{}\ Star\ re.$

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, 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, 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, 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*)) = *false*.
`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, 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, 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, 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.

□