

PART I

Functional Programming

1 *Basics*

Coq can be seen as a combination of two things:

1. a simple but very expressive *programming language*, and
2. a set of tools for stating *logical assertions* (including assertions about the behavior of programs) and assembling evidence of their truth.

We will spend a few lectures working with the programming language and then begin looking seriously at the logical aspects of the system.

1.1 Days of the Week

In Coq’s programming language, almost nothing is built in—not even booleans or numbers! Instead, it provides powerful tools for defining new types of data and functions that process and transform them.

Let’s start with a very simple example. The following definition tells Coq that we are defining a new set of data values. The set is called `day` and its members are `monday`, `tuesday`, etc. The lines of the definition can be read “`monday is a day, tuesday is a day, etc.`”

func.v: 11

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

Having defined this set, we can write functions that operate on its members.

func.v: 20

```

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 work out these types even if they are not given explicitly, but we'll always include them to make reading easier.

Having defined a function, we might like to check how it works on some examples. There are actually three different ways to do this in Coq.

1. We can use the command `Eval` to evaluate a compound expression involving `next_weekday`. For example, if we give Coq the following input

```
Eval simpl in (next_weekday (next_weekday saturday)).
```

it will print the simplified result and its type:

```

▶ = tuesday
  : day

```

The keyword `simpl` (for “simplify”) tells Coq precisely how to evaluate the expression we give it. For the moment, `simpl` is the only one we'll need; later on we'll see some alternatives that are sometimes useful.

If you have a computer handy, now 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 the file `func.v` from the book's accompanying Coq sources, find the above example a little ways from the top, submit it to Coq, and observe the result.

`func.v: 34`

2. We can record what we *expect* the result to be in the form of a `Lemma`:

```

Lemma 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 it 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 few chapters), but essentially this can be read "The assertion we've just made can be proved by observing that both sides of the equality are the same after simplification."

3. Third, we can ask Coq to "extract," from a `Definition`, a program in some other, more mainstream, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler.

This facility is very interesting, since it gives us a way to construct *fully certified* programs in mainstream languages. This is actually one of the main uses for which Coq was developed. However, exploring it would take us too far from the main topics of this text, so we will say no more about it here.

1.2 Booleans

Similarly, we can define the type `bool` of booleans, with constants `true` and `false`.

func.v: 41

```
Inductive bool : Set :=
  | 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 in its standard library, together with a multitude of useful functions and lemmas. (Take a look at the file `Coq.Init.Datatypes` if you're interested.) Whenever possible, we'll name our own definitions and lemmas so that they exactly coincide with the ones in the standard library.

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

func.v: 45

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
```

```

      | false => true
    end.

func.v: 51 Definition andb (b1:bool) (b2:bool) : bool :=
      match b1 with
      | true => b2
      | false => false
    end.

func.v: 57 Definition orb (b1:bool) (b2:bool) : bool :=
      match b1 with
      | true => true
      | false => b2
    end.

```

The second and third illustrate the syntax for multi-argument function definitions.

The following four “unit tests” constitute a complete specification—a truth table—for the `orb` function:

```

func.v: 63 Lemma test_orb1: (orb true false) = true.
func.v: 65 Lemma test_orb2: (orb false false) = false.
func.v: 67 Lemma test_orb3: (orb false true) = true.
func.v: 69 Lemma test_orb4: (orb true true) = true.

```

The proofs of these lemmas are precisely the same as what we saw above. From now on, proofs will generally be omitted, unless they are particularly relevant to the discussion. They can always be found in full in the accompanying Coq sources.

- 1.2.1 EXERCISE [★]: In the Coq source file `func.v`, you will find a comment containing incomplete implementations of two more boolean functions, `norb` and `and3b`. Uncomment and finish them, making sure that Coq can verify the provided unit test lemmas. □

The `Check` command causes Coq to print the type of an expression. For example, when presented with

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

Coq prints

```
► negb true
   : bool
```

Functions like `negb` itself are also data values, just like `true` and `false`. Their types are called *function types*. The type of `negb`, written `bool→bool`, is pronounced “`bool` arrow `bool`” and 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 Numbers

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

func.v: 124

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

The clauses of this definition can be read:

1. `O` is a natural number (note that this is the letter “`O`,” not the numeral “`0`”).
2. `S` is a *constructor* that takes a natural number and yields another one—that is, if `n` is a natural number, then `S n` is too.

We can write simple functions that pattern match on natural numbers just as we did above:

func.v: 130

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

func.v: 136

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

Because natural numbers are such a pervasive form of data, Coq provides a tiny bit of special 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)))) .
```

```
► 4
   : nat
```

and expands arabic numerals in its input into appropriate sequences of applications of `S` to `O`:

```
Eval simpl in (minustwo 4) .
```

```
► = 2
   : nat
```

The constructor `S` has the same type as functions like `minustwo` and `pred`: both

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

yield

```
► ... : nat → nat
```

signifying that these are things that can be applied to a number to yield a number. However, there is a fundamental difference: functions like `pred` and `minustwo` come with *computation rules*—e.g., the definition of `pred` says that `pred n` can be simplified to match `n` with `| O => O | S m' => m'` end—while `S` has no such behavior attached. Although it is a function in the sense that it can be applied to an argument, it does not “do” anything at all!

What’s going on here is that every inductively defined set (`weekday`, `nat`, `bool`, and many others we’ll see below) is actually a set of *expressions*. The definition of `nat` says how expressions in the set `nat` can be constructed:

- 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`.

These three 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 like `true` and $S (S \text{false})$ do not.

For most function definitions over numbers, pure pattern matching is not enough: we 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`.

func.v: 150

```
Fixpoint evenb (n:nat) {struct n} : bool :=
  match n with
  | 0      => true
  | S 0    => false
  | S (S n') => evenb n'
  end.
```

The most important thing to note about this definition is the annotation `{struct n}` on the first line. This instructs Coq to check 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 `evenb` will eventually terminate.

We can define `oddb` by a similar `Fixpoint` declaration, but here is a simpler definition that will be easier to work with later:

func.v: 157

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

func.v: 159

```
Lemma test_oddb1: (oddb (S 0)) = true.
```

func.v: 161

```
Lemma test_oddb2: (oddb (S (S (S (S 0)))) = false.
```

func.v: 164

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

```
Fixpoint plus (m : nat) (n : nat) {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => S (plus m' n)
  end.
```

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

```
Eval simpl in (plus (S (S (S 0))) (S (S 0))).
```

```
► = 5
   : nat
```

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

```

      plus (S (S (S O))) (S (S O))
    = S (plus (S (S O)) (S (S O)))   by the second clause of the match
    = S (S (plus (S O) (S (S O))))   by the second clause of the match
    = S (S (S (plus O (S (S O))))))  by the second clause of the match
    = S (S (S (S (S O))))           by the first clause of the match.

```

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

func.v: 172

```

Fixpoint times (m n : nat) {struct m} : nat :=
  match m with
  | 0 => 0
  | S m' => plus (times m' n) n
  end.

```

Other arithmetic functions like `minus` and `exp` can be defined similarly (see `func.v`).

- 1.3.1 EXERCISE [★]: Recall that the factorial function is defined like this in conventional mathematical notation:

$$\begin{aligned} \mathit{factorial}(0) &= 1 \\ \mathit{factorial}(n) &= n * (\mathit{factorial}(n - 1)) \quad \text{if } n > 0 \end{aligned}$$

Translate this into Coq's notation. (An incomplete definition can be found in a comment in `func.v`) □

func.v: 204

When we say that Coq comes with nothing built-in, we really mean it: Even equality testing has to be defined!

```

Fixpoint beq_nat (m n : nat) {struct m} : bool :=
  match m with
  | 0 =>
    match n with
    | 0 => true
    | S n' => false
    end
  | S m' =>
    match n with
    | 0 => false
    | S n' => beq_nat m' n'
    end
  end.

```

- 1.3.2 EXERCISE [★]: Complete the definitions (in `func.v`) of the comparison functions `ble_nat` and `blt_nat`. □

1.4 Pairs of Numbers

func.v: 247

Each constructor of an inductive type can take any number of parameters—none (as with `true` and `0`), one (as with `S`), or more than one:

```
Inductive prodnat : Set :=
  pair : nat → nat → prodnat.
```

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`.”

Here are some simple function definitions illustrating pattern matching on two-argument constructors:

func.v: 250

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

func.v: 254

```
Definition snd (p : prodnat) : nat :=
  match p with
  | pair x y => y
  end.
```

It would be nice to be able to use the more standard mathematical notation (x, y) instead of `pair x y`. We can instruct Coq to allow this with a `Notation` declaration.

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

func.v: 272

The new notation is supported both in expressions like `fst (3, 4)` and in pattern matches:

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

1.5 Lists of Numbers

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

func.v: 280

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

func.v: 284

For example, here is a three-element list:

```
Definition l123 := cons 1 (cons 2 (cons 3 nil)).
```

As with pairs, it is more convenient to write lists in familiar mathematical notation. The following two 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 "[ x , .. , y ]" := (cons x .. (cons y nil) ..).
```

It is not necessary to fully understand the second line of the first declaration, 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:

func.v: 289

```
Definition l123' := 1 :: (2 :: (3 :: nil)).
```

func.v: 290

```
Definition l123'' := 1 :: 2 :: 3 :: nil.
```

func.v: 291

```
Definition l123''' := [1,2,3].
```

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

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

then `+` will bind tighter than `::`, and `1 + 2 :: [3]` will be parsed correctly as `(1 + 2) :: [3]` rather than `1 + (2 :: [3])`.

A number of functions are useful for manipulating lists. For example, the `head` function returns the first element (the “head”) of the list, while `tail` (“tail”) returns everything but the first element. (Note that this version of `head` does not behave quite like the one in the Coq standard library; we’ll return to this point in a later chapter.)

- 1.5.1 EXERCISE [★]: Complete the definitions of `nonzeros`, `oddmembers` and `countoddmembers` in `func.v`. □
- 1.5.2 EXERCISE [★★, RECOMMENDED]: Complete the definition of `alternate`. This exercise illustrates the fact that it sometimes requires a little extra thought to satisfy Coq’s requirement that all `Fixpoint` definitions be “obviously terminating.” There is an easy way to write the `alternate` function using just a single `match . . . end`, but Coq will not accept it as obviously terminating. Look for a slightly more verbose solution with two nested `match . . . end` constructs. Note that each `match` must be terminated by an `end`. □
- 1.5.3 EXERCISE [★★, RECOMMENDED]: Complete the definition of `reverse`. □
- 1.5.4 EXERCISE [★★]: A *bag* (or *multiset*) is a set where each element can appear any finite number of times. One reasonable implementation of bags is to represent a bag of numbers as a list.

func.v: 378

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

Stubs for several bag-manipulating functions (`count`, `union`, etc.) can be found in `func.v`. Complete them. □

1.6 Options

Here is another type definition that is quite useful in day-to-day programming:

func.v: 448

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

We can use `natoption` as a way of returning “error codes” from functions. For example, suppose we want to write a function that returns the `n`th element of some list. If we give it type `nat → natlist → nat`, then we’ll have to return some number when the list is too short!

func.v: 452

```
Fixpoint index_bad (n:nat) (l:natlist) {struct l} : nat :=
  match l with
  | nil => 42 (* arbitrary! *)
  | a :: l' => match beq_nat n 0 with true => a
              | false => index_bad (pred n) l' end
  end.
```

On the other hand, if we give it type $\text{nat} \rightarrow \text{natlist} \rightarrow \text{natoption}$, then we can return `None` when the list is too short and `Some a` when the list has enough members and `a` appears at position `n`.

func.v: 459

```
Fixpoint index (n:nat) (l:natlist) {struct l} : natoption :=
  match l with
  | nil => None
  | a :: l' => match beq_nat n 0 with true => Some a
              | false => index (pred n) l' end
  end.
```

func.v: 466

```
Lemma test_index1 : index 0 [4,5,6,7] = Some 4.
```

func.v: 468

```
Lemma test_index2 : index 3 [4,5,6,7] = Some 7.
```

func.v: 470

```
Lemma test_index3 : index 10 [4,5,6,7] = None.
```

By the way, the standard definition of `index` in the Coq library (where it is called `nth`) actually takes a different approach to this problem: instead of returning an `option`, it takes an additional *default* parameter to be returned if the list is too short. So its type is $\text{nat} \rightarrow \text{natlist} \rightarrow \text{nat} \rightarrow \text{nat}$. The `option`-returning technique we show here is cleaner and more flexible, although the default-passing approach is sometimes more compact.

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

func.v: 473

```
Fixpoint index' (n:nat) (l:natlist) {struct l} : natoption :=
  match l with
  | nil => None
  | a :: l' => if beq_nat n 0 then Some a else index (pred n) l'
  end.
```

Coq's conditionals are exactly like those in every other language, with one small generalization. Since the boolean type is not built in, Coq actually allows 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.

1.7 A Note on Equality

You may have noticed what appears to be an inconsistency in what we have said so far about equality. On the one hand, we noted that equality is not "built-in" in Coq. Thus, for example, in order to compare two `nats` for equality, we had to define an explicit function `beq_nat`. In fact, this particular

function is in the standard library, but if we want to compare values of other types for equality, we will need to write similar functions specific to those types.

- 1.7.1 EXERCISE [★★, RECOMMENDED]: Complete the definition of `beq_natlist` in `func.v`. □

On the other hand, from the very beginning of this chapter, we have been using the `=` operator within Lemmas to test the behavior of our functions, e.g.

```
Lemma test_reverse1: reverse [1,2,3] = [3,2,1].
```

This operator seems to check correctly for equality, and it works on all the types we've seen so far, including deep structures like lists. Thus some (quite general-purpose) form of equality *does* appear to be built-in to Coq after all! So why do we need to make definitions like `beq_nat`?

The answer, which will probably strike you as a bit vague at this point, is that `=` and functions like `beq_nat` are fundamentally different things. Applying the `=` operator to two arguments produces a *proposition*, whereas `beq_nat` and similar functions produce values in the inductively-defined type `bool`. Propositions appear in proofs (such as our test-case Lemmas) rather than in computations. In general, it doesn't make sense to use a proposition where a computable value is expected. For example, Coq won't let you test the result of an `=` operation using an `if` expression (try it!).

Coq keeps track of the difference by giving propositions a type `Prop` that is quite distinct from types like `bool`. This distinction should make more sense later on the book, where we will learn much more about propositions and proofs. Incidentally, we will eventually see that not even `=` is really built-in.

2 *Programming with Functions*

2.1 Higher-Order Functions

A *higher-order function* is one that takes a function as a parameter or returns a function as its result—i.e., it treats a function as data. Here is a very simple one.

func.v: 496

```
Definition doit3times (f:nat→nat) (n:nat) := f (f (f n)).
```

The function `doit3times` takes another function `f` as its first parameter and applies `f` to its second argument `n` three times in succession.

func.v: 498

```
Lemma test_doit3times1:   doit3times minustwo 9 = 3.
```

Next, here is a more useful higher-order function, which takes a list and a predicate (a function from `bool` to `bool`) and “filters” the list, returning just those elements for which the predicate returns `true`.

func.v: 501

```
Fixpoint filter (p: nat→bool) (l: natlist) {struct l}
  : natlist :=
  match l with
  | nil => nil
  | h::t => if p h then h :: (filter p t)
           else filter p t
  end.
```

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

func.v: 509

```
Lemma test_filter1 :   filter evenb [4,5,6,7] = [4,6].
```

We can use `filter` to give a pleasantly concise version of the `countoddmembers` function from Exercise 1.5.1.

func.v: 512

```

Definition countoddmembers' (l:natlist) : nat :=
  length (filter oddb l).

```

- 2.1.1 EXERCISE [★]: Complete the definition of `doublematches`, a variant of `filter` that takes a predicate `p` and a list `l` and returns a new list in which every element satisfying `p` is repeated twice. □

func.v: 532

Another extremely handy higher-order function is `map`.

```

Fixpoint map (f:nat→nat) (l:natlist) {struct l}
  : natlist :=
  match l with
  | nil    => nil
  | 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:

func.v: 539

```

Lemma test_map1:   map minustwo [1,2,3,4,5] = [0,0,1,2,3].

```

func.v: 541

```

Lemma test_map2:   map S [1,2,3,4] = [2,3,4,5].

```

- 2.1.2 EXERCISE [★]: Complete the definition of `filter_map`, a combination of `filter` and `map` that takes both a predicate `p` and a function `f` together with a list `l` and returns a new list containing the results of applying `f` just to those elements of `l` for which `p` returns `true`. □

In fact, the multiple-argument functions we have already seen are simple examples of higher-order functions. For instance, the type of `plus` is `nat→nat→nat`. Since `→` associates to the right, this type can equivalently be written `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.”

func.v: 565

```

Definition plus3 : nat→nat := plus 3.

```

func.v: 567

```

Lemma test_plus3 :   plus3 4 = 7.

```

func.v: 569

```

Lemma test_plus3' :  map (plus 3) [1,2,3,4] = [4,5,6,7].

```

- 2.1.3 EXERCISE [★]: Complete the definition of `countzeros` in `func.v`. □

2.2 Anonymous Functions

It is possible to construct a function “on the fly” without declaring it at the top level and giving it a name; this is analogous to the notation we’ve been using for writing down constant lists, etc.

```
Eval simpl in (map (fun n => times n n) [2,0,3,1]).
```

```
► = [4, 0, 9, 1]
   : natlist
```

The expression `fun n => times n n` here can be read “The function that, given a number n , returns `times n n`.”

[That’s probably not enough explanation, and it certainly needs an exercise or two.]

→

2.3 Example: Bags as Functions

Recall that a *bag* is a set where each element can appear any finite number of times. In Exercise 1.5.4, we saw how to implement bags of numbers by representing them as `natlists`. Higher-order functions can be used to give an alternate implementation of bags. In this version, a bag is a function from numbers to numbers:

func.v: 581

```
Definition bagf := nat → nat.
```

The idea is that a bag is represented as a function that, when applied to an argument n , this function tells you how many times n occurs in the bag.

This representation makes the `count` function trivial to write (we’ll call it `countf` to avoid a name clash with the one from the exercise above).

func.v: 583

```
Definition countf (v:nat) (s:bagf) : nat :=
  s v.
```

func.v: 586

The empty bag returns 0 when queried for the count of any element.

```
Definition emptybagf : bagf := (fun n => 0).
```

As an example of a non-empty bag, here is one way of writing a function representing a bag containing the elements 5 (once) and 6 (twice):

func.v: 592

```
Definition bagf566 : bagf :=
  fun n =>
    if beq_nat n 5 then 1
```

```
else if beq_nat n 6 then 2
else 0.
```

To add an element to a bag b , we build a function that, when queried for the count for some element n , first asks b for its count and then either adds 1 or not, as appropriate.

func.v: 605

```
Definition addf (m:nat) (b:bagf) :=
  fun (n:nat) =>
    match beq_nat m n with
    | false => b n
    | true  => S (b n)
    end.
```

- 2.3.1 EXERCISE [★★, RECOMMENDED]: Finish the definitions of the bag operations `unionf` and `remove_onef`. □
- 2.3.2 EXERCISE [★★]: Can you write a `subset` operation that works with this implementation of bags? □

3 *Programming with Types*

3.1 Polymorphism

It often happens that we need different variants of a given function with different type annotations. As a trivial example, we might want a `doit3times` function that works on booleans instead of numbers.

func.v: 669

```
Definition doit3times_bool (f:bool→bool) (n:bool) : bool :=  
  f (f (f n)).
```

Except for type annotations, this definition is precisely the same as the old `doit3times` on page 17. Defining multiple variants of functions in this way is annoying and error-prone. Accordingly, many programming languages, including Coq, allow us to give a single *polymorphic* (or *generic*) definition:

func.v: 672

```
Definition doit3times (X:Set) (f:X→X) (n:X) : X :=  
  f (f (f n)).
```

The polymorphic definition adds an extra parameter to the function, telling it what *set* to expect its third argument to come from (which is the same set that its second argument `f` accepts and returns). To use `doit3times`, we must now apply it an appropriate set in addition to its other arguments.

func.v: 675

```
Lemma test_doit3times1: doit3times nat minustwo 9 = 3.  
Lemma test_doit3times2: doit3times nat (plus 3) 2 = 11.  
Lemma test_doit3times3: doit3times bool negb true = false.
```

func.v: 677

func.v: 679

Let's have a look at the type Coq assigns to the polymorphic `doit3times`.

```
Check doit3times.
```

```
► doit3times  
  : forall X : Set, (X → X) → X → X
```

The prefix `forall X : Set` corresponds to the first parameter `X`. The whole type `forall X : Set, (X → X) → X → X` can be read as a refined version of the type `Set → (X → X) → X → X`. That is, it tells us that `doit3times` takes three arguments, the first of which is a set, the second a function, etc. Additionally, the `forall X` prefix *binds* the variable `X` in the rest of the type, telling us that the second parameter must be a function on the set given as the first parameter, etc. (From this intuition, it would arguably be better to write it with an arrow, something like `X : Set → (X → X) → X → X`, but you'll find that Coq's `forall` notation will feel natural with a little familiarity.)

3.2 Polymorphic Lists

Not only functions, but also inductive type definitions can be polymorphic. For example, here is a polymorphic list data type.

func.v: 687

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

This is exactly like the definition of `natlist` on page 12 except that the `nat` argument to the `cons` constructor has been replaced by an arbitrary set `X`, a binding for `X` has been added to the first line, and the occurrences of `natlist` in the types of the constructors have been replaced by `list X`.

When we use the constructors `nil` and `cons` to build lists, we need to specify what sort of lists we are building—that is, `nil` and `cons` are now *polymorphic constructors*.

```
Check nil.

► nil
  : forall X : Set, list X

Check cons.

► cons
  : forall X : Set, X → list X → list X
```

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

func.v: 694

```
Fixpoint length (X:Set) (l:list X) {struct l} : nat :=
  match l with
  | nil      => 0
  | cons h t => S (length X t)
```

```
end.
```

Note that the uses of `nil` and `cons` in `match` patterns do not require any type annotations: we already know that the list `l` contains elements of type `X`, so there's no reason to include `X` in the pattern. Similarly, here is a polymorphic `app`.

func.v: 700

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

- 3.2.1 EXERCISE [★★, RECOMMENDED]: Complete the polymorphic definitions of `repeat` and `reverse` in `func.v`. □

3.3 Implicit Type Arguments

Whenever we use a polymorphic function, we need to pass it one or more sets in addition to its other arguments. For example, the recursive call in the body of the `length` function above must pass along the set `X`. But this is a bit heavy and verbose: Since the second argument to `length` is a list of `X`s, 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,¹ we can write the *implicit argument* `_`, which can be read “Please figure out for yourself what set 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 set should replace the `_`.

func.v: 731

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

```
Fixpoint length' (X:Set) (l:list X) {struct l} : nat :=
  match l with
  | nil      => 0
  | cons h t => S (length' _ t)
```

1. Actually, `_` can be used in place of *any* argument, as long as there is enough local information that Coq can determine what value is intended; but this feature is mainly used for type arguments.

```
end.
```

In this instance, the savings of writing `_` instead of `X` is small. But in other cases the difference is significant. For example, suppose we want to write down a list containing the numbers 1, 2, and 3. Instead of writing

func.v: 737

```
Definition l123 :=
  cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).
```

func.v: 740

we can use implicit arguments to write:

```
Definition l123' := cons _ 1 (cons _ 2 (cons _ 3 (nil _))).
```

Better yet, we can tell Coq that certain arguments to constructors or functions should always be treated as implicit by using `Implicit Arguments` declarations.

```
Implicit Arguments nil [X].
Implicit Arguments cons [X].
Implicit Arguments app [X].
```

These declarations must follow the definition of the constructor or function in question. In the general form, several implicit parameters can be listed within the square brackets, separated by spaces. (In fact, if the list of parameters is omitted, Coq will use a heuristic to determine which arguments should be treated as implicit. This usually does the right thing, but for clarity and certainty, we'll specify the implicit arguments ourselves for now.)

Occasionally it is necessary to specify explicitly the value of a parameter that is normally implicit. A common case involves the `nil` constructor; since it carries no value, it is often impossible for Coq to determine its type parameter. For example, the attempted definition

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

will generate a Coq error message

```
Error: Cannot infer an instance for the implicit parameter X of nil
```

To get around this problem, you can always obtain a version of a constructor or function in which all parameters revert to being *explicit* by prepending an `@`-sign, and then specify the needed arguments.

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

Finally, we can use `Notation` declarations to obtain the same abbreviations for constructing polymorphic lists as we had before for lists of numbers.

```

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).

```

func.v: 755

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

```

Definition l123'' := [1,2,3].

```

3.4 Polymorphic Lists, Continued

Besides `length` and `repeat`, we can give polymorphic versions of all the other functions we defined for processing lists of numbers. Here is the polymorphic `filter` function:

func.v: 759

```

Fixpoint filter (X:Set) (test: X→bool) (l:list X)
  {struct l} : (list X) :=
  match l with
  | [] => []
  | h :: t => if test h then h :: (filter _ test t)
              else filter _ test t
  end.

```

func.v: 769

This version can be applied to numbers, same as before:

```

Lemma test_filter1:
  filter evenb [1,2,3,4] = [2,4].

```

But, being polymorphic, it can also be used with other types. For example, here we apply it to a list of lists and a predicate that filters out just those lists of length exactly one.

func.v: 772

```

Lemma test_filter2:
  filter (fun l => beq_nat (length l) 1)
    [ [1, 2], [3], [4], [5,6,7], [], [8] ]
  = [ [3], [4], [8] ].

```

func.v: 778

Here is the polymorphic `map` function:

```

Fixpoint map (X:Set) (Y:Set) (f:X→Y) (l:list X) {struct l}
  : (list Y) :=

```

```

match l with
| []      => []
| h :: t => (f h) :: (map _ _ f t)
end.

```

func.v: 795

Again, this version can be applied to numbers:

```

Lemma test_map1:
  map (plus 3) [2,0,2] = [5,3,5].

```

(Both implicit arguments here will be `nat`.) More interestingly, the element types of the input and output lists need not be the same (note that it takes *two* type arguments, `X` and `Y`). This `map` it can thus be applied to a list of numbers and a function from numbers to booleans to yield a list of booleans:

func.v: 798

```

Lemma test_map2:
  map oddb [2,1,2,5] = [false,true,false,true].

```

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:

func.v: 801

```

Lemma test_map3:
  map (fun n => repeat false n) [2,1,3]
= [ [false,false], [false], [false,false,false] ].

```

- 3.4.1 EXERCISE [★]: A useful variant of `map` is a function `flatmap` that (as in the last unit test for `map`) takes a function `f` from `X` to `list Y`, but instead of simply returning a list of lists of results it concatenates the lists resulting from all the applications of `f` and returns a single long list of results. Complete its definition in `func.v`. □

3.5 Polymorphic Pairs

func.v: 823

Similarly, the type definition we gave above for pairs of numbers can be generalized to *polymorphic pairs*:

```

Inductive prod (X Y : Set) : Set :=
  pair : X → Y → prod X Y.

```

We can use implicit arguments and `Notation` to help define the familiar concrete notation for pairs.

```

Implicit Arguments pair [X Y].
Notation "( x , y )" := (pair _ _ x y).

```

We can use the same `Notation` mechanism to define the standard notation for pair *types*:

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

(The annotation `: type_scope` tells Coq that this abbreviation should be used when parsing types.)

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

func.v: 832

```
Definition fst (X Y : Set) (p : X * Y) : X :=
  match p with (x,y) => x end.
```

func.v: 837

```
Definition snd (X Y : Set) (p : X * Y) : Y :=
  match p with (x,y) => y end.
```

3.5.1 EXERCISE [★★, RECOMMENDED]: Consider the following function definition:

func.v: 842

```
Fixpoint zip (X Y : Set) (lx : list X) (ly : list Y)
  {struct lx} : list (X*Y) :=
  match lx with
  | [] => []
  | x::tx => match ly with
    | [] => []
    | y::ty => (x,y) :: (zip _ _ tx ty)
    end
  end.
```

1. What is the type of `zip` (i.e., what does `Check zip` print?)
2. What does

```
Eval simpl in (zip [1,2] [false,false,true,true]).
```

print? (Use Coq to check your answers.)

3. The `zip` function transforms a pair of lists into a list of pairs. The inverse transformation, `unzip`, takes a list of pairs and returns a pair of lists. For example,

```
unzip [(1,false), (2,false)]
```

yields:

```
(([1,2], [true, false]))
```

Write out the definition of `unzip` and make sure that it passes the unit test `test_unzip` in `func.v`.

□

3.6 Polymorphic Options

func.v: 875

One last polymorphic type for now: *polymorphic options*. The type declaration generalizes the one for `natoption` on page 14.

```
Inductive option (X:Set) : Set :=
  | Some : X → option X
  | None : option X.
```

3.6.1 EXERCISE [★]: Complete the definition of the polymorphic `index` function in `func.v`. □

3.6.2 EXERCISE: Complete the definition of the polymorphic `head` function in `func.v`. Note that its type is not quite what you might expect from looking at the `head` function we wrote on `natlists`. This new type matches the one in the Coq standard library. Why couldn't we just generalize the original type? □

3.7 Example: Permutations of a List

I am still deciding what to do with this section. It should either get expanded with more text, examples, etc., or else it should get turned into an extended exercise.

Here is a more serious example of functional programming in Coq. The goal is to produce a function `[perm]` that, given a list, returns a list containing all of the permutations of the input list. See if you can understand how it works by starting with the `[perm]` function and then looking at the two auxiliary functions that implement its “inner loops.”

func.v: 917

```
Fixpoint inserteverywhere (X:Set) (v:X) (l:list X)
  {struct l} : (list (list X)) :=
  match l with
  | [] => [[v]]
```

```

| h :: t =>
  (v :: l) ::
  (map (cons h) (inserteverywhere _ v t))
end.

```

func.v: 934

```

Definition inserteverywhereall
  (X:Set) (v:X) (l:list (list X))
  : (list (list X)) :=
  flatmap (inserteverywhere v) l.

```

func.v: 949

```

Fixpoint perm (X: Set) (l:list X) {struct l}
  : (list (list X)) :=
  match l with
  | [] => [[]]
  | h :: t =>
    inserteverywhereall h (perm X t)
  end.

```

3.8 Currying and Uncurrying

Similarly, this material should either get some more words or (probably better) become an exercise.

func.v: 968

```

Definition curry (X Y Z : Set) (f : X * Y → Z)
  : X → Y → Z :=
  fun x => fun y => f (x,y).

```

func.v: 972

```

Definition uncurry (X Y Z : Set) (f : X → Y → Z)
  : (X * Y) → Z :=
  fun p =>
    match p with
    (x,y) => f x y
  end.

```

3.9 Non-Structural Recursion

This section needs to be either cut or filled in and expanded. Coq offers lots of ways of addressing the limitations of structural recursion; it doesn't make sense to cover all of them, and I'm not even certain that, for the purposes of this book, it is necessary to cover any of them. So it may be best to replace

this section by a short discussion of the issues and a pointer to where to read more. (If it gets kept as a full-blown section, it could also potentially move to the end of the first chapter, to avoid mixing polymorphism with termination measures. But I think it's better to wait till here, when students have just a bit more familiarity with the basic ideas of functional programming and a bit more facility with Coq.)

func.v: 986

```

Fixpoint all_interleavings_aux
  (c:nat) (X:Set) (l1 : list X) (l2 : list X)
  {struct c} : list (list X) :=
  match c with
  | 0 => [] (** Out of steam: return something silly *)
  | S c' =>
    match l1 with
    | nil => l2 :: []
    | h1 :: t1 =>
      match l2 with
      | nil => l1 :: []
      | h2 :: t2 =>
        (map (cons h1)
              (all_interleavings_aux c' _ t1 l2))
        ++ (map (cons h2)
                (all_interleavings_aux c' _ l1 t2))
      end end end.

```

func.v: 1006

```

Definition all_interleavings
  (X:Set) (l1 : list X) (l2 : list X)
  : (list (list X)) :=
  all_interleavings_aux (length (l1 ++ l2)) l1 l2.

```

Things to note:

- the termination parameter is the auxiliary `c` argument (`c` for counter)
- the annotation for the result type is needed because of the way that the function `all_interleavings_aux` is called in the body — it's a little too complicated for Coq to work out what the result type must be