4 *Programming with Types*

4.1 Polymorphism

We've been working a lot with lists of numbers, but clearly programs also need to be able to manipulate lists whose elements are drawn from other types—lists of strings, lists of booleans, lists of lists, etc. We could define a new inductive datatype for each of these...

```
Inductive boollist : Set :=
  | 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 data type.

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

This is exactly like the definition of `natlist` on page 28 except that the `nat` argument to the `cons` constructor has been replaced by an arbitrary set `X`, a binding for `X` has been added to the first line, and the occurrences of `natlist` in the types of the constructors have been replaced by `list X`. (We're able to 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.)

With this definition, 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
```

The "`forall X`" in these types should be read as an additional argument to the constructors that determines the expected types of the arguments that follow. When `nil` and `cons` are used, these arguments are supplied in the same way as the others. For example, the list containing `2` and `1` is written `(cons nat 2 (cons nat 1 (nil nat)))`.

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

```
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. (More formally, the set `X` is a parameter of the whole definition of `list`, not of the individual constructors. So when we pattern)

Just as we did with `nil` and `cons`, to use `length` we apply it first to a type and then to its list argument:

```
Example test_length1 :
    length nat (cons nat 1 (cons nat 2 (nil nat))) = 2.
```

To use our length with other kinds of lists, we simply instantiate it with an appropriate type parameter:

```
Example test_length2 :
    length bool (cons bool true (nil bool)) = 1.
```

Similarly, here is a polymorphic `app` function.

```
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.
```

`Poly.v` also gives polymorphic variants of `snoc` and `rev`.

4.2 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 _.

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)
  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

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

we can use implicit arguments to write:

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

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.

Better yet, we can use implicit arguments in the right-hand side of `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).
```

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

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

Indeed, we can go a step further and tell Coq that we *always* want it to infer some of the arguments of particular functions:

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

In what followed, we'll make a habit of adding an `Implicit Arguments` declaration after every polymorphic function. To avoid clutter, we won't show any of these in the typeset notes.

One small problem with declaring arguments `Implicit` is that, occasionally, there will not be enough local information to determine a type argument and we will need to tell Coq specially that we want to give it explicitly even though we've declared it to be `Implicit`. For example, if we write

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

Coq will give us an error, because it doesn't know what type argument to supply to `nil`. We can supply it explicitly by annotating the `nil` with an `@`, which tells Coq to ignore the `Implicit` declaration for this use of `nil`:

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

4.2.1 EXERCISE [★★]: Finish the missing parts of definitions and proofs in the section marked "`Polymorphism exercises`" in `Poly.v`. □

4.3 Polymorphic Pairs

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 to help define the familiar concrete notation for pairs.

```
  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.

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

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

4.3.1 EXERCISE [★★]: Consider the following function definition:

```
Fixpoint combine (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) :: (combine _ _ tx ty)
             end
  end.
```

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

2. What does

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

   print? (Use Coq to check your answers.)

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

```
split _ _ [(1,false),(2,false)]
```

yields:

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

Write out the definition of `split` and make sure that it passes the unit test `test_split` in `Poly.v`.

4. We can think of `combine` and `split` as inverses. We can prove part of this formally with the following theorem:

```
Theorem split_combine : forall X Y (l : list (X * Y)) l1 l2,
  split l = (l1, l2) →
  combine l1 l2 = l.
```

What other theorem must you prove to show that `combine` and `split` really are inverses? How would you state it? (Hint: for all lists of pairs, `combine` is `split`'s inverse. For what domain is `split` the inverse of `combine`?) □

4.4 Polymorphic Options

One last polymorphic type for now: *polymorphic options*. The type declaration generalizes the one for `natoption` on page 35.

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

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

4.4.2 EXERCISE [★]: Complete the definition of the polymorphic `hd_opt` function in `Poly.v`. □

5 *Programming With Functions*

5.1 Higher-Order Functions

Like many other modern programming languages—including, of course, all *functional languages*—Coq allows functions to be passed as arguments to other functions and returned as results from other functions. Functions that operate on other functions in this way are called *higher-order* functions. Here's a simple one:

```
Definition doit3times (X:Set) (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`.

```
Example test_doit3times1: doit3times nat minustwo 9 = 3.
Example test_doit3times2: doit3times bool negb true = false.
```

5.2 Partial application

In fact, the multiple-argument functions we have already seen are also 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."

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

```
Example test_plus3 :    plus3 4 = 7.
Example test_plus3' :   doit3times plus3 0 = 9.
```

5.2.1 EXERCISE [★★,OPTIONAL]: In Coq, a function `f : A → B → C` really has the type `A → (B → C)`. That is, if you give `f` a value of type `A`, it will give you function `f' : B → C`. If you then give `f'` a value of type `B`, it will return a value of type `C`. Processing a list of arguments with functions that return functions is called "currying", named in honor of the logician Haskell Curry.

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

1. We can define currying as follows:

   ```
   Definition prod_curry (X Y Z : Set)
     (f : X * Y → Z) (x : X) (y : Y) : Z := f (x, y).
   ```

   Define its inverse, `prod_uncurry`.

2. Prove that the two are inverses: the theorems `uncurry_curry` and `curry_uncurry`. □

5.3 Anonymous Functions

It is also 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.

```
Example test_anon_fun:
  doit3times (fun (n:nat) => mult n n) 2 = 256.
```

The expression `fun (n:nat) => mult n n` here can be read "The function that, given a number `n`, returns `times n n`."

One small that we don't actually need to bother declaring the type of the argument `n`; Coq can see that it must be `nat` by looking at the context.

```
Example test_anon_fun':
  doit3times (fun n => mult n n) 2 = 256.
```

5.4 Polymorphic Lists, Continued

We've seen some very simple higher-order functions. Here is a more useful one, 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`.

```
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.
```

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

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

We can use `filter` to give a pleasantly concise version of the `countoddmembers` function from Exercise 3.3.1.

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

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

```
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.
```

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 nat nat (plus 3) [2,0,2] = [5,3,5].
Example test_map2: map oddb [2,1,2,5] = [false,true,false,true].
```

Note that 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:

```
Example 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:

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

5.4.1 EXERCISE [★]: The definitions in `Poly.v` corresponding to this section (i.e., the polymorphic `filter` and `map` functions and their unit tests) use implicit arguments in many places. Replace every _ by an explicit set and use Coq to check that you've done so correctly. (You may also have to remove `Implicit Arguments` commands for Coq to accept explicit arguments.) This exercise is not to be turned in; it is probably easiest to do it on a *copy* of `Poly.v` that you can throw away afterwards. □

5.4.2 EXERCISE [★★,OPTIONAL]: Prove that the polymorphic definitions `map` and `rev` commute (`map_rev`). You will most likely need an auxiliary lemma. □

5.4.3 EXERCISE [★]: The function `map` maps a `list X` to a `list Y` using a function of type `X → Y`. We can define a similar function, `flat_map`, which maps a `list X` to a `list Y` using a function `f` of type `X → list Y`. Your definition should work by 'flattening' the results of `f`, like so:

```
    flat_map (fun n => [n,n,n]) [1,5,4]
      = [1, 1, 1, 5, 5, 5, 4, 4, 4].
```

5.5 The `unfold` Tactic

The precise behavior of the `simpl` tactic is somewhat subtle: even expert Coq users tend to just try it and see what it does in particular situations, rather than trying to predict in advance. However, one point is worth noting: `simpl` never expands names that have been declared as `Definition`s. For example, because `plus3` is a `Definition`, these two expressions do not simplify to the same result.

```
  Eval simpl in (plus 3 5).

▶     = 8
      : nat

  Eval simpl in (plus3 5).

▶     = plus3 5
      : nat
```

The `unfold` tactic can be used to explicitly replace a defined name by the right-hand side of its definition. For example, we need to use it to prove this fact about `plus3`:

```
Theorem unfold_example :
  plus3 5 = 8.
Proof.
  unfold plus3.
  reflexivity.
Qed.
```

5.6 Functions as Data

The higher-order functions we have seen so far all take functions as arguments. Now let's look at some examples involving returning functions as the results of other functions.

To begin, here is a function that takes a value `x` (drawn from some set `X`) and returns a function from `nat` to `X` that returns `x` whenever it is called.

```
Definition constfun (X : Set) (x : X) :=
  fun (k:nat) => x.
```

Similarly, but a bit more interestingly, here is a function that takes a function `f` from numbers to some set `X`, a number `k`, and a value `x`, and constructs a function that behaves exactly like `f` except that, when called with the argument `k`, it returns `x`.

```
Definition override (X : Set) (f : nat→X) (k:nat) (x:X) :=
  fun k′ => if beq_nat k k′ then x else f k′.
```

We'll use function overriding heavily in parts of the rest of the course, and we will end up using quite a few of its properties. For example, we'll need to know that, if we override a function at some argument `k` and then look up `k`, we get back the overriden value.

```
Theorem override_eq : forall (X:Set) x k (f : nat→X),
  (override f k x) k = x.
```

(The proof of this theorem is straightforward, but note that it requires `unfold` to unfold the definition of `override`.)

5.6.1 Exercise [★★]: Check that you understand each part of the following theorem and can paraphrase it in English. Then prove it in Coq.

```
Theorem override_example : forall (b:bool),
  (override (constfun b) 3 true) 2 = b.
```

□

5.6.2 EXERCISE [★★]: Prove the following theorem:

```
Theorem override_neq : forall (X:Set) x1 x2 k1 k2 (f : nat→X),
  f k1 = x1 →
  beq_nat k2 k1 = false →
  (override f k2 x2) k1 = x1.
```

□

In what follows, we will see several other, more interesting, properties of `override`. But to prove them, we'll need to know a few more of Coq's tactics; developing these is the main topic of the next chapter.

6 *More Coq Tactics*

6.1 Inversion

Recall the definition of natural numbers:

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

It is clear 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 other facts:

1. The constructor `S` is *injective*.[1] That is, the only way we can have `S n = S m` is if `n = m`.

2. 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 unequal. (Since neither `true` nor `false` take any arguments, their injectivity is not an issue.) Coq provides a tactic, called `inversion`, that allows us to exploit these principles in making proofs.

The `inversion` tactic is used like this. Suppose `H` is a hypothesis in the context (or a previously proven lemma) of the form `e = f`, where `e` and `f` are

1. Recall that, in mathematics, saying that a function f is injective means that its results are equal only when its arguments are—that is, $f\ x = f\ y$ implies $x = y$.

expressions of some inductively defined type. Moreover, suppose `e` and `f` are “concrete,” in the sense that `e = c a1 a2 ... an` for some constructor `c` and arguments `a1` through `an` and similarly `f = d b1 b2 ... bm` for some constructor `d` and arguments `b1` through `bm`. Then `inversion H` instructs Coq to “invert” this equality to extract the information it gives us about the subcomponents of `e` and `f`:

1. If `c` and `d` are the same constructor, then we know, by the injectivity of this constructor, that `a1 = b1`, `a2 = b2`, etc., and `inversion H` adds these facts to the context.
2. If `c` and `d` are different constructors, then the hypothesis `e=f` is contradictory. That is, a false assumption has crept into the context, and this means that any goal whatsoever is provable! In this case, `inversion H` marks the current goal and completed and pops it off the goal stack.

The `inversion` tactic is probably easier to understand from at examples than from general descriptions like the above. `Poly.v` contains several small examples illustrating its behavior and a larger example, `beq_nat_eq`, showing how it is used in a more interesting proof.

6.1.1 EXERCISE [★]: Prove `sillyex1` using the `inversion` tactic to destruct the equalities. □

6.1.2 EXERCISE [★]: Prove `sillyex2` using the `inversion` tactic to solve a goal with contradictory hypotheses. □

6.1.3 EXERCISE [★★]: Write an informal proof of `beq_nat_eq`. □

6.1.4 EXERCISE [★]: Prove the following theorem:

```
Theorem override_same : forall (X:Set) x1 k1 k2 (f : nat→X),
  f k1 = x1 →
  (override f k1 x1) k2 = f k2.
```

6.1.5 EXERCISE [★★]: Prove `beq_nat_eq'`; note the care used when stating the theorem and introducing variables. □

6.1.6 EXERCISE [★★]: Do the exercises in the `Practice session` in `Poly.v`. □

6.2 Applying Tactics in Hypotheses

By default, most tactics work on the goal formula and leave the context unchanged. But tactics often come with 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.

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

In other words, `apply L in H` gives us a form of *forward reasoning*: from `L1→L2` and a hypothesis matching `L1`, it gives us a hypothesis matching `L2`. By contrast, `apply L` is *backward reasoning*: it says that, if we know `L1→L2` and we are trying to prove `L2`, it suffices to prove `L1`. In general, Coq tends to favor backward reasoning, but in some situations the forward style can be easier to think about.

`Poly.v` contains a number of additional examples.

6.2.1 EXERCISE [★★]: Prove the following theorem using the `unfold ... in ...` tactic:

```
Theorem plus_n_n_injective : forall n m,
     plus n n = plus m m
  → n = m.
```

6.3 Using `destruct` on Compound Expressions

We have seen many examples where the [destruct] tactic 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`.

`Poly.v` contains several examples.

6.3.1 EXERCISE [★]: Prove `sillyfun_odd` using `destruct` on compound expressions. (Don't cheat and use `sillyfun_false`!) □

6.3.2 EXERCISE [★]: Prove the `override_shadow` theorem. □

6.3.3 EXERCISE [★★]: Prove `split_combine` (you may want to look back at exercise 4.3.1). □

6.4 The `remember` Tactic

We have seen how 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 set `T`, then, for each constructor `c` of `T`, the tactic destruct e generates a subgoal in which all occurrences of e (in the goal and in the context) are replaced by c.

Sometimes, however, this substitution process loses information that we need in order to complete the proof. For example, suppose we define a function sillyfun1 like this...

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

... and suppose that we want to convince Coq of the rather obvious observation 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 : forall (n : nat),
     sillyfun1 n = true
  → oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (beq_nat n 3).
Admitted.
```

At this point, we are stuck: the context does not contain enough information to prove the goal! The problem is that the substitution peformed by destruct is too brutal – it threw away every occurrence of beq_nat n 3, but we need to keep at least one of these because we need to be able to reason that since, in this branch of the case analysis, beq_nat n 3 = true, it must be that n = 3, from which it follows that n is odd.

What we would really like is not to use destruct directly on beq_nat n 3 and substitute away all occurrences of this expression, but rather to use destruct on something else that is *equal* to beq_nat n 3—e.g., if we had a variable that we knew was equal to beq_nat n 3, we could destruct this variable instead.

The remember tactic allows us to introduce such a variable.

→ [*Stopped here. See* Poly.v *for the rest...*]

6.4.1 EXERCISE [★★,OPTIONAL]: Prove filter_exercise. □

6.5 The apply ... with ... Tactic

6.5.1 EXERCISE [★★]: Solve the exercises at the end of the apply ... with ... section in Poly.v. □

6.6 Challenge problem

6.6.1 EXERCISE [★★★, CHALLENGE PROBLEM]: Define two recursive `Fixpoint`s, `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 at least one element in the list that satisfies the 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, write a *nonrecursive* `Definition`, `existsb'`, using `forallb` and `negb`. Prove that `existsb'` and `existsb` have the same behavior. □