3 *Pairs and Lists*

3.1 Formal vs. Informal Proofs

The question of what, exactly, constitutes a *proof* of a mathematical claim has challenged philosophers throughout the ages. A rough and ready definition, though, could be this: a proof of a mathematical proposition `P` is a written (or, sometimes, spoken) text that instills in the reader or hearer the certainty that `P` is true. That is, a proof is an act of communication.

Now, 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 a simple mechanical check that `P` can be derived from a certain set of formal logical rules, and the proof is a recipe that guides the program in performing this check. Such recipies are called *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, thus necessarily *informal*. Here, the criterial for success are less clearly specified. A "good" 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. One reader 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 another reader, 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, because it is easier to fill in the details for themselves. Ultimately, there is no universal standard, because 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, 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 will be using Coq in this course, we will be working heavily with formal proofs. But this doesn't mean we can ignore the informal ones! Formal proofs are useful in many ways, but they are *not* very efficient ways of communicating ideas between human beings.

For example, consider this statement:

```
Theorem plus_assoc' : forall n m p : nat,
  plus n (plus m p) = plus (plus n m) p.
```

Coq is perfectly happy with this as a proof:

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

For a human, however, it is difficult to make much sense of this. If you're used to Coq you can probably 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. Instead, a mathematician would write it as in Figure 3-1: The overall form of the proof is basically similar. (This is no accident, of course: 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.

3.1.1 EXERCISE [★★]: In `Lists.v`, you will find a Coq proof of a theorem called `plus_comm'`, which was an exercise last week. In a comment following the formal proof, write the corresponding informal proof, using the informal proof of `plus_assoc'` as a model. □

3.1.2 EXERCISE [★★]: In `Lists.v`, you will find a careful informal proof of a theorem called `beq_nat_refl`. Write a corresponding Coq proof. □

3.2 Pairs of Numbers

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

Proof: By induction on `n`.

- First, suppose $\mathtt{n} = \mathtt{0}$. We must show

  `plus 0 (plus m p) = plus (plus 0 m) p.`

  This follows directly from the definition of `plus`.

- Next, suppose $\mathtt{n} = \mathtt{S}\ \mathtt{n}'$, with

  `plus n' (plus m p) = plus (plus n' m) p.`

  We must show

  `plus (S n') (plus m p) = plus (plus (S n') m) p.`

  By the definition of `plus`, this follows from

  `S (plus n' (plus m p)) = S (plus (plus n' m) p),`

  which is immediate from the induction hypothesis.

Figure 3-1 A mathematician's inductive proof

```
Inductive natprod : Set :=
  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`."

Here are some simple function definitions illustrating 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.
```

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 instruct Coq to allow this with a `Notation` declaration.

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

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

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

3.2.1 EXERCISE [★★]: Prove `snd_fst_is_swap` in `Lists.v`. □

3.2.2 EXERCISE [★★, OPTIONAL]: Proof `fst_swap_is_snd` in `Lists.v`. □

3.3 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."

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

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:

```
Definition l123'   := 1 :: (2 :: (3 :: nil)).
Definition l123''  := 1 :: 2 :: 3 :: nil.
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 `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) {struct count} : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

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

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

The `app` function concatenates two lists.

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

In fact, `app` will be used so pervasively in some parts of what follows that it is convenient to have an infix operator for it.

```
  Notation "x ++ y" := (app x y)
                      (right associativity, at level 60).
```

Two more small examples. The hd function returns the first element (the "head") of the list, while tl ("tail") returns everything but the first element.

```
Definition hd (l:natlist) : nat :=
  match l with
  | nil => 0  (* arbitrarily *)
  | h :: t => h
  end.

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

3.3.1 EXERCISE [★]: Complete the definitions of nonzeros, oddmembers and countoddmembers in Lists.v. □

3.3.2 EXERCISE [★★]: 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. □

3.3.3 EXERCISE [★★, OPTIONAL]: 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.

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

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

3.4 Reasoning About Lists

Just as with numbers, simple facts about list-processing functions can sometimes be proved entirely by simplification. For example, simplification is enough for this theorem...

```
Theorem nil_app : forall l:natlist,
  [] ++ l = l.
```

... because the `[]` is substituted into the match position in the definition of `app`, allowing the match itself to be simplified. Also like 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 : forall l:natlist,
  pred (length l) = length (tl l).
Proof.
  intros l. destruct l as [| n l′].
  Case "l = nil".
    reflexivity.
  Case "l = cons n l′".
    reflexivity.
Qed.
```

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.

Proofs by induction over non-numeric data types are perhaps a little less familiar than natural number induction, but the basic idea is equally simple. Each `Inductive` declaration defines a set of data values that can be built up from the declared constructors: a number can be either `O` or `S` applied to a number; a boolean can be either `true` or `false`; 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′`, asssuming that `P` is true for `l′`. Since larger lists can only be built up from smaller ones, stopping eventually with `nil`, these two things together establish the truth of `P` for all lists `l`.

For example, the associativity of the ++ operation...

```
Theorem ass_app : forall l1 l2 l3 : natlist,
  l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.
```

... can be shown by induction on `l1`:

```
Proof.
  intros l1 l2 l3. induction l1 as [| n l1'].
  Case "l1 = nil".
    reflexivity.
  Case "l1 = cons n l1'".
    simpl. rewrite → IHl1'. reflexivity.
Qed.
```

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. A human-readable (informal) proof needs to include more explicit—in particular, it helps the reader a lot to be reminded exactly what the induction hypothesis is in the second case.

3.4.1 THEOREM: For all `l1`, `l2`, and `l3`,

`l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.`

Proof: By induction on `l1`.

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

  `[] ++ (l2 ++ l3) = ([] ++ l2) ++ l3,`

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

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

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

  We must show

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

  By the definition of `++`, this follows from

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

  which is immediate from the induction hypothesis. □

For a slightly more involved example of an inductive proof over lists, suppose we define a "cons on the right" function `snoc` like this...

```
Fixpoint snoc (l:natlist) (v:nat) {struct l} : natlist :=
  match l with
  | nil    => [v]
  | h :: t => h :: (snoc t v)
  end.
```

... and use it to define a list-reversing function `rev` line this:

```
Fixpoint rev (l:natlist) {struct l} : natlist :=
  match l with
  | nil    => nil
  | h :: t => snoc (rev t) h
  end.
```

We can now prove that reversing a list doesn't change it's length as follows. First we prove a lemma relating `length` and `snoc`.

3.4.2 LEMMA: For all numbers `n` and lists `l`,

`length (snoc l n) = S (length l).`

Proof: By induction on `l`.

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

  `length (snoc [] n) = S (length []),`

  which follows directly from the definitions of `length` and `snoc`.

- Next, suppose `l` = `n′::l′`, with

  `length (snoc l′ n) = S (length l′).`

  We must show

  `length (snoc (n′ :: l′) n) = S (length (n′ :: l′)).`

  By the definitions of `length` and `snoc`, this follows from

  `S (length (snoc l′ n)) = S (S (length l′)),`

  which is immediate from the induction hypothesis. □

Now we can use this lemma to prove the fact we wanted about `length` and `rev`.

3.4.3 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 (snoc (rev l') n) = S (length l'),`

  which, by the previous lemma, is the same as

  `S (length (rev l')) = S (length l').`

  This is immediate from the induction hypothesis. □

Obviously, the style of these proofs is rather longwinded and pedantic. After we've seen a few of them, we might begin to find it easier to follow proofs that give a little less detail overall (since we can easily work them 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 more like this:

3.4.4 THEOREM: For all lists `l`,

`length (rev l) = length l.`

Proof: First, observe that

`length (snoc l n) = S (length l)`

for any `l`. This follows by a straightforward induction on `l`.

The main property now follows by another straightforward 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 on how similar the proof at hand is to ones that the audience will already be familiar with. The more pedantic style is usually a safe fallback.

3.4.5 EXERCISE [★★]: Find the section marked "`A bunch of exercises`" in `Lists.v` and complete all the proofs you find there. □

3.4.6 EXERCISE [★★]: (1) Find a non-trivial equational property involving `::`, `snoc`, and `++`. (2) Prove it. Fill in your theorem and proof in the file `Lists.v` just after the string "`Design exercise`." □

3.4.7 EXERCISE [★★, OPTIONAL]: If you did Exercise 3.3.3, then prove the theorems `count_member_nonzero` and `remove_decreases_count` in `Lists.v`. □

3.5 Options

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

```
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!

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

On the other hand, if we give it type `nat→natlist→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`.

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

```
Example test_index1 :    index 0 [4,5,6,7]  = Some 4.
Example test_index2 :    index 3 [4,5,6,7]  = Some 7.
Example test_index3 :    index 10 [4,5,6,7] = None.
```

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

```
Fixpoint index′ (n:nat) (l:natlist) {struct l} : natoption :=
  match l with
  | nil => None
  | a :: l′ => if beq_nat n O 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.

3.5.1 EXERCISE [★★]: Complete the definition of `hd_opt` in `Lists.v`. □

3.5.2 EXERCISE [★★]: Prove `option_elim_hd` in `Lists.v`. □

3.5.3 EXERCISE [★★]: Define a function `beq_natlist` for comparing lists of numbers for equality. Prove that `beq_natlist l l` yields `true` for every list `l`. (Stubs are provided in `Lists.v`.) □

3.6 The `apply` Tactic

This section still needs to be written. Please have a look at the corresponding material in `Lists.v`*, where you'll find plenty of descriptive text...*

3.6.1 EXERCISE [★★]: Prove `silly_ex` in `Lists.v` without using the `simpl` tactic. □

3.6.2 EXERCISE [★★★]: Prove `rev_exercise1` and `beq_nat_sym` in `Lists.v`. □

3.6.3 EXERCISE [★★★]: Provide an informal proof of `beq_nat_sym` in `Lists.v`. □

3.7 Varying the Induction Hypothesis

One subtlety in these inductive proofs is worth noticing here. For example, look back at the proof of the `app_ass` theorem. The induction hypothesis (in the second subgoal generated by the `induction` tactic) is

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

That is it makes a statement about `l1'` together with the *particular* lists `l2` and `l3`. The lists `l2` and `l3`, which were introduced into the context by the `intros` at the top of the proof, are "held constant" in the induction hypothesis. If we set up the proof slightly differently by introducing just `n` into the context at the top, then we get an induction hypothesis that makes a stronger claim:

```
forall l2 l3, l1' ++ l2 ++ l3 = (l1' ++ l2) ++ l3.
```

(Use Coq to see the difference for yourself.) In the present case, the difference between the two proofs is minor, since the definition of the `++` function just examines its first argument and doesn't do anything interesting with its second argument. But we'll soon come to situations where setting up the induction hypothesis one way or the other can make the difference between a proof working and failing.

3.7.1 EXERCISE [★★]: Finish the proof of `ass_app'` at the end of `Lists.v`. □