7 More On Induction

7.1 Quick Review

We’ve now seen a bunch of Coq’s fundamental tactics—enough, in fact, to do pretty much everything we’ll want for a while. We’ll introduce one or two more as we go along through the next few lectures, and later in the course we’ll introduce some more powerful automation tactics that make Coq do more of the low-level work in many cases, but basically this is the set we need. Figure 7-1 gives a summary.

7.2 Programming with Propositions

A proposition is a statement expressing a factual claim. In Coq, propositions are written as expressions of type Prop. Although we haven’t mentioned it explicitly, we have already seen numerous examples of such expressions.

Check (plus 2 2 = 4).

▶ plus 2 2 = 4
  : Prop

Check (ble_nat 3 2 = false).

▶ ble_nat 3 2 = false
  : Prop

Both provable and unprovable claims are perfectly good propositions. Simply being a proposition is one thing; being provable is something else! Both plus 2 2 = 4 and plus 2 2 = 5 are expressions of type Prop.

One important role for propositions in Coq is as the subjects of Theorems, Examples, etc. But they 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 (numbers, functions, types, type functions, ...).

Definition plus_fact : Prop := plus 2 2 = 4.

Now we can use this name in any situation where a proposition is expected—for example, as the subject of a theorem.

Theorem plus_fact_is_true :
  plus_fact.

(Because of the Definition, the proof of this theorem involves an unfold in addition to the usual reflexivity.)

So far, all the propositions we have seen are equality propositions. But we can build on equality propositions to make other sorts of claims. For example, what does it mean to claim that “a number \( n \) is even”? We have already defined a function that tests evenness, so one reasonable definition could be “\( n \) is even iff \( \text{evenb} \ n = \text{true} \)”
Definition even (n:nat) :=
  evenb n = true.

This defines even as a **parameterized proposition**. It can be thought of as a
**function** that, when applied to a number n, yields a proposition claiming that
n is even.

The **type** of even is nat→Prop. This type can be pronounced in two ways:
either simply “even is a function from numbers to propositions” or, perhaps
more helpfully, “even is a **family** of propositions, indexed by a number n.”

Functions returning propositions are completely first-class citizens in Coq;
we can do all the same sorts of things with them as with any other kinds of
functions. We can, for example, use them in other definitions.

Definition even_n__even_SSn (n:nat) :=
  (even n) → (even (S (S n))).

We can define them to take multiple arguments...

Definition between (n m o: nat) : Prop :=
  andb (ble_nat n o) (ble_nat o m) = true.

... and then partially apply them.

Definition teen : nat→Prop := between 13 19.

And we can pass propositions—even parameterized propositions—as argu-
ments to functions.

Definition true_for_zero (P:nat→Prop) : Prop :=
  P 0.

Definition preserved_by_S (P:nat→Prop) : Prop :=
  forall n′, P n′ → P (S n′).

Definition true_for_all_numbers (P:nat→Prop) : Prop :=
  forall n, P n.

Definition nat_induction (P:nat→Prop) : Prop :=
  (true_for_zero P)
   → (preserved_by_S P)
   → (true_for_all_numbers P).

The last of these is interesting. If we unfold all the definitions, here is what it
means in concrete terms.
Example nat_induction_example : forall (P:nat→Prop),
    nat_induction P
  = ( (P 0)
      → (forall n′, P n′ → P (S n′))
      → (forall n, P n)).

That is, nat_induction expresses exactly the principle of induction for natural numbers that we’ve been using for most of our proofs about numbers. Indeed, we can use the induction tactic to prove very straightforwardly that nat_induction P holds for all P.

Theorem our_nat_induction_works : forall (P:nat→Prop),
    nat_induction P.

7.3 Induction Axioms

In fact, the connection between nat_induction and Coq’s built-in principle of induction is even closer than this suggests: modulo bound variable names, they are precisely the same!

Check nat_ind.

▶ nat_ind : forall P : nat → Prop,
  P 0
  → (forall n : nat, P n → P (S n))
  → forall n : nat, P n

The first “:” here can be pronounced “...records the truth of the proposition...” In general, every time we declare a new datatype t with Inductive, Coq automatically generates an axiom t_ind (i.e., a theorem whose truth is assumed rather than being proved from other axioms). This axiom expresses the induction principle for t. The induction tactic is a straightforward wrapper that, at its core, simply performs apply t_ind.

To see this more clearly, let’s experiment a little with using apply nat_ind directly, instead of induction, to carry out some proofs. First, here is a direct proof of the validity of our formulation of the induction principle. The proof amounts to observing that, after unfolding the names we defined, our principle coincides with the built-in one.

Theorem our_nat_induction_works’ :
  forall P, nat_induction P.
Proof.
  intros P.
And here’s an alternate proof of a theorem that we saw in Chapter 2 (Exercise 2.9.1):

Theorem mult_0_r' : forall n:nat,  
    mult n 0 = 0.
Proof.
    apply nat_ind.
    Case "O". reflexivity.
    Case "S". simpl. intros n IHn. rewrite \rightarrow IHn.
        simpl. reflexivity. □

Several details in this proof are worth noting. First, in the induction step of the proof (the "S" case), we have to do a little bookkeeping manually (the intros) that induction does automatically. Second, we do not introduce n into the context before applying nat_ind—the conclusion of nat_ind is a quantified formula, and apply needs this conclusion to exactly match the shape of the goal state, including the quantifier. The induction tactic works either with a variable in the context or a quantified variable in the goal. Third, the apply tactic automatically chooses variable names for us (in the second subgoal, here), whereas induction lets us specify (with the as... clause) what names should be used. The automatic choice is actually a little unfortunate, since it re-uses the name n for a variable that is different from the n in the original theorem. This is why the Case annotation is just S—if we tried to write it out in the more explicit form that we’ve been using for most proofs, we’d have to write n = S n, which doesn’t make a lot of sense! All of these conveniences make inductive nicer to use in practice than applying induction principles like nat_ind directly. But it is important to realize that, modulo this little bit of bookkeeping, applying nat_ind is what we are really doing.

7.3.1 **EXERCISE [★★]:** Prove theorem plus_one_r' in Ind.v without using the induction tactic.

7.3.2 **EXERCISE [★★]:** Prove the same theorem again (plus_one_r'”) using our re-formulation of the induction principle, nat_induction (and without using induction or apply nat_ind).

7.4 **Induction Principles for Other Datatypes**

    From here on, most of the text is still in Ind.v...
7.4 Induction Principles for Other Datatypes

7.4.1 Exercise [★, Optional]: Write out the induction principle that Coq will generate for the following datatype:

```coq
Inductive rgb : Set :=
| red : rgb
| green : rgb
| blue : rgb.
```

Compare your answer with what Coq prints.

7.4.2 Exercise [★, Optional]: Suppose we had written `natlist` a little differently:

```coq
Inductive natlist1 : Set :=
| nnil1 : natlist1
| nsnoc1 : natlist1 → nat → natlist1.
```

What would the induction principle for `natlist1` look like?

7.4.3 Exercise [★, Optional]: Here is an induction principle for an inductively defined set:

```coq
ExSet_ind :
  forall P : ExSet → Prop,
  (forall b : bool, P (con1 b))
  → (forall (n : nat) (e : ExSet), P e → P (con2 n e))
  → forall e : ExSet, P e
```

Give an Inductive definition of `ExSet`.

7.4.4 Exercise [★, Optional]: Write out the induction principle that Coq will generate for the following datatype:

```coq
Inductive tree (X:Set) : Set :=
| leaf : X → tree X
| node : tree X → tree X → tree X.
```

Compare your answer with what Coq prints.

7.4.5 Exercise [★, Optional]: Find an inductive definition that gives rise to the following induction principle:

```coq
mytype_ind :
  forall (X : Set) (P : mytype X → Prop),
  (forall x : X, P (constr1 X x))
  → (forall n : nat, P (constr2 X n))
  → (forall m : mytype X, P m → forall n : nat,
      P (constr3 X m n))
  → forall m : mytype X, P m
7.4.6 Exercise [★, Optional]: Find an inductive definition that gives rise to the following induction principle:

```lean
foo_ind : 
 forall (X Y : Set) (P : foo X Y → Prop),
  (forall x : X, P (bar X Y x)) → 
  (forall y : Y, P (baz X Y y)) → 
  (forall f1 : nat → foo X Y,
   (forall n : nat, P (f1 n)) → P (quux X Y f1)) → 
  forall f2 : foo X Y, P f2
```

7.4.7 Exercise [★, Optional]: Consider the following inductive definition:

```lean
Inductive foo' (X:Set) : Set :=
  | C1 : list X → foo' X → foo' X
  | C2 : foo' X.
```

What induction principle will Coq generate for foo'? (Fill in the blanks, then check your answer with Coq.)

```lean
foo'_ind :
  forall (X : Set) (P : foo' X → Prop),
  ____________________________________________ →
  ____________________________________________ →
  ____________________________________________ →
  forall f : foo' X, __________________________
```

7.5 A Closer Look at Induction Hypotheses

The induction principle for numbers

```lean
forall P : nat → Prop,
  P 0 → 
  (forall n : nat, P n → P (S n)) → 
  forall n : nat, P n
```

is a generic statement that holds for all propositions P—or rather, strictly speaking, for all families of propositions P indexed by a number n. Each time we use this principle, we are choosing P to be a particular expression of type nat→Prop.

We can make this more explicit by giving this expression a name. For example, instead of stating the theorem `mult_0_r as “forall n, mult n 0 = 0,” we can write it as “forall n, P_m0r n”, where P_m0r is defined as

```lean
Definition P_m0r (n:nat) : Prop :=
  mult n 0 = 0.
```
or equivalently as:

\[
\text{Definition } P_{\text{m0r}}' : \text{nat} \rightarrow \text{Prop} := \n\text{fun n => mult n 0 = 0.}
\]

This extra naming step isn’t something that we’ll do in normal proofs, but it is something that we should be able to do, because it allows us to see exactly what is the induction hypothesis. If we prove \(\forall n, P_{\text{m0r}} n\) by induction on \(n\) (using either induction or apply \text{nat_ind}), we see that the first subgoal requires us to prove \(P_{\text{m0r}} 0\) (“\(P\) holds for zero”), while the second subgoal requires us to prove \(\forall n', P_{\text{m0r}} n' \rightarrow P_{\text{m0r}} (S n')\) (that is “\(P\) holds of \(S n'\) if it holds of \(n'\)” or, more elegantly, “\(P\) is preserved by \(S\)”). The induction hypothesis is the premise of this latter implication—the assumption that \(P\) holds of \(n'\), which we are allowed to use in proving that \(P\) holds for \(S n'\).

### 7.6 A Closer Look at the induction Tactic

#### 7.6.1 Exercise [★, Optional]: Reprove \(\text{plus\_comm}'\) and \(\text{plus\_comm}''\) in the style of \(\text{mult\_0\_r}''\): write out an explicit Definition of the proposition being proved by induction and state the theorem and proof in terms of this defined proposition.

#### 7.6.2 Exercise [★★★★, Optional]: Define (using \text{Fixpoint}) a recursive function \(\text{true\_upto\_n\_implies\_true\_everywhere}\) so that Coq accepts the example \(\text{true\_upto\_n\_example}\) in \text{Ind.v}.

### 7.7 Generalizing the Induction Hypothesis

#### 7.7.1 Exercise [★★]: Prove theorems \(\text{plus\_n\_n\_injective\_take2}\) and \(\text{index\_after\_last}\) in \text{Ind.v}.

#### 7.7.2 Exercise [★★★]: Provide an informal proof corresponding to your coq proof of \(\text{index\_after\_last}\) in \text{Ind.v}.

#### 7.7.3 Exercise [★★★, Optional]: Prove \(\text{length\_snoc}''\), \(\text{eqnat\_false\_S}\), and \(\text{length\_append\_cons}\) in \text{Ind.v}.

#### 7.7.4 Exercise [★★★★]: Prove theorem \(\text{length\_append\_twice}\) in \text{Ind.v}.
8 Evidence

8.1 Constructing Evidence

8.1.1 Exercise [★★]: Prove theorem double_even in Ind.v.

8.1.2 Exercise [★★★★, Optional]: Try to predict what proof object is constructed by your proof of double_even. Compare your answer with what Coq prints.

8.2 Manipulating Evidence

8.2.1 Exercise [★, Optional]: Consider the proof of ev_minus2 from Ind.v:

Theorem ev_minus2: forall n, ev n → ev (pred (pred n)).
Proof.
intros n E.
destruct E as [| n' E'].
Case "E = ev_0". simpl. apply ev_0.
Case "E = ev_SS n' E'". simpl. apply E'.
□

What would happen if we tried to destruct on n instead of E?

8.2.2 Exercise [★, Optional]: Consider the proof of ev_even from Ind.v:

Theorem ev_even : forall n, ev n → even n.
Proof.
intros n E. induction E as [| n' E'].
Case "E = ev_0". unfold even. reflexivity.
Case "E = ev_SS n' E'".
8.2.10 Exercise [★★★★]: Prove theorem ev_MyProp’ in Ind.v without using the induction tactic.

8.2.11 Exercise [★★★★]: A palindrome is a sequence that reads the same backwards as forwards.
1. Define an inductive proposition pal on list nat that captures what it means to be a palindrome. (Hint: You’ll need three cases.)
2. Prove that forall l, pal(l++(rev l)).
3. Prove that forall l, pal l → l = rev l. (The converse theorem, forall l, l = rev l → pal l, is much harder! We won’t have the tools to attack this for some time.)

Put your solution to this problem after the comment which describes it, near the bottom of Ind.v.