7

More On Induction

NOTE: This is a preliminary version of this chapter. A complete, revised version will be released on Wednesday.

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.
7.2 Programming with Propositions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intros</td>
<td>move hypotheses/variables from goal to context</td>
</tr>
<tr>
<td>reflexivity</td>
<td>finish the proof (when the goal looks like $e = e$)</td>
</tr>
<tr>
<td>apply</td>
<td>prove goal using a hypothesis, lemma, or constructor</td>
</tr>
<tr>
<td>apply... in H</td>
<td>apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning)</td>
</tr>
<tr>
<td>apply... with...</td>
<td>explicitly specify values for variables that cannot be determined by pattern matching</td>
</tr>
<tr>
<td>simpl</td>
<td>simplify computations in the goal</td>
</tr>
<tr>
<td>simpl in H</td>
<td>... or a hypothesis</td>
</tr>
<tr>
<td>rewrite</td>
<td>use an equality to rewrite the goal</td>
</tr>
<tr>
<td>rewrite ... in H</td>
<td>... or a hypothesis</td>
</tr>
<tr>
<td>unfold</td>
<td>replace a defined constant by its RHS in the goal</td>
</tr>
<tr>
<td>unfold... in H</td>
<td>... or a hypothesis</td>
</tr>
<tr>
<td>destruct... as...</td>
<td>case analysis on values of inductively defined types</td>
</tr>
<tr>
<td>induction... with...</td>
<td>induction on values of inductively defined types</td>
</tr>
<tr>
<td>inversion</td>
<td>reason by injectivity and distinctness of constructors</td>
</tr>
<tr>
<td>remember (e) as x</td>
<td>give a name (x) to an expression (e) so that we can destruct x without “losing” e</td>
</tr>
<tr>
<td>assert (e) as H</td>
<td>introduce a “local lemma” e and call it H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intros</td>
<td>move hypotheses/variables from goal to context</td>
</tr>
<tr>
<td>reflexivity</td>
<td>finish the proof (when the goal looks like $e = e$)</td>
</tr>
<tr>
<td>apply</td>
<td>prove goal using a hypothesis, lemma, or constructor</td>
</tr>
<tr>
<td>apply... in H</td>
<td>apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning)</td>
</tr>
<tr>
<td>apply... with...</td>
<td>explicitly specify values for variables that cannot be determined by pattern matching</td>
</tr>
<tr>
<td>simpl</td>
<td>simplify computations in the goal</td>
</tr>
<tr>
<td>simpl in H</td>
<td>... or a hypothesis</td>
</tr>
<tr>
<td>rewrite</td>
<td>use an equality to rewrite the goal</td>
</tr>
<tr>
<td>rewrite ... in H</td>
<td>... or a hypothesis</td>
</tr>
<tr>
<td>unfold</td>
<td>replace a defined constant by its RHS in the goal</td>
</tr>
<tr>
<td>unfold... in H</td>
<td>... or a hypothesis</td>
</tr>
<tr>
<td>destruct... as...</td>
<td>case analysis on values of inductively defined types</td>
</tr>
<tr>
<td>induction... with...</td>
<td>induction on values of inductively defined types</td>
</tr>
<tr>
<td>inversion</td>
<td>reason by injectivity and distinctness of constructors</td>
</tr>
<tr>
<td>remember (e) as x</td>
<td>give a name (x) to an expression (e) so that we can destruct x without “losing” e</td>
</tr>
<tr>
<td>assert (e) as H</td>
<td>introduce a “local lemma” e and call it H</td>
</tr>
</tbody>
</table>

**Figure 7-1** Tactics we’ve seen so far

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 exam-
ple, 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 $\text{nat} \rightarrow \text{Prop}$. 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) \rightarrow (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\rightarrow\text{Prop} := between 13 19.

And we can pass propositions—even parameterized propositions—as arguments to functions.

Definition true_for_zero (P:nat\rightarrow\text{Prop}) : \text{Prop} :=
  P 0.

Definition preserved_by_S (P:nat\rightarrow\text{Prop}) : \text{Prop} :=
  forall n′, P n′ \rightarrow P (S n′).

Definition true_for_all_numbers (P:nat\rightarrow\text{Prop}) : \text{Prop} :=
  forall n, P n.

Definition nat_induction (P:nat\rightarrow\text{Prop}) : \text{Prop} :=
  (true_for_zero P)
  \rightarrow (preserved_by_S P)
  \rightarrow (true_for_all_numbers P).
7.3 Induction Axioms

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.
unfold nat_induction, true_for_zero,
preserved_by_S, true_for_all_numbers.
apply nat_ind. □

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 Logic.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

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

\[
\text{Inductive tree } (X: \text{Set}) : \text{Set} := \\
| \text{leaf} : X \to \text{tree } X \\
| \text{node} : \text{tree } X \to \text{tree } X \to \text{tree } X.
\]

Compare your answer with what Coq prints.

7.4.2 Exercise [★, Optional]: Suppose we had written \text{natlist} a little differently:

\[
\text{Inductive natlist1} : \text{Set} := \\
| \text{nnil1} : \text{natlist1} \\
| \text{nsnoc1} : \text{natlist1} \to \text{nat} \to \text{natlist1}.
\]

What would the induction principle for \text{natlist1} look like?

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

\[
\text{ExSet\_ind} : \\
\forall P : \text{ExSet} \to \text{Prop} , \\
(\forall b : \text{bool}, P (\text{con1 } b)) \\
\to (\forall (n : \text{nat}) (e : \text{ExSet}), P e \to P (\text{con2 } n e)) \\
\to \forall e : \text{ExSet}, P e
\]

Give an Inductive definition of \text{ExSet}.

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

\[
\text{Inductive tree } (X: \text{Set}) : \text{Set} := \\
| \text{leaf} : X \to \text{tree } X \\
| \text{node} : \text{tree } X \to \text{tree } X \to \text{tree } X.
\]

Compare your answer with what Coq prints.

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

\[
\text{mytype\_ind} : \\
\forall (X : \text{Set}) (P : \text{mytype } X \to \text{Prop}),
\]
7.4.6 Exercise [★, Optional]: Find an inductive definition that gives rise to the following induction principle:

\[\text{foo\_ind} :\]
\[
\forall (X : \text{Set}) (Y : \text{Set}) (P : \text{foo } X Y \rightarrow \text{Prop}),
\]
\[
\left(\forall x : X, P (\text{bar } X Y x)\right) \\
\left(\forall y : Y, P (\text{baz } X Y y)\right) \\
\left(\forall f1 : \text{nat} \rightarrow \text{foo } X Y, \right.
\]
\[
\left(\forall n : \text{nat}, P (f1 n) \rightarrow P (\text{quux } X Y f1)\right) \\
\left(\forall f2 : \text{foo } X Y, P f2\right)
\]

7.5 A Closer Look at Induction Hypotheses

The induction principle for numbers

\[
\forall P : \text{nat} \rightarrow \text{Prop}, \\
P 0 \\
\left(\forall n : \text{nat}, P n \rightarrow P (S n)\right) \\
\forall n : \text{nat}, P n
\]

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 \(\text{nat} \rightarrow \text{Prop}\).

We can make this more explicit by giving this expression a name. For example, instead of stating the theorem \text{mult\_0\_r} as “\(\forall n, \text{mult } n 0 = 0\),” we can write it as “\(\forall n, \text{P\_m0r } n\), where \text{P\_m0r} is defined as

\[
\text{Definition P\_m0r (n:nat) : Prop :=} \\
\text{mult n 0 = 0.}
\]

or equivalently as:

\[
\text{Definition P\_m0r′ : nat\rightarrowProp :=} \\
\text{fun n => mult n 0 = 0.}
\]

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 \(\forall n, \text{P\_m0r } n\) by induction on \(n\) (using either induction or apply nat\_ind), we see that the first
subgoal requires us to prove \( \text{P}_\text{m0r} 0 \) ("\( P \) holds for zero"), while the second subgoal requires us to prove \( \forall n', \text{P}_\text{m0r} n' \rightarrow \text{P}_\text{m0r} n' (S n') \) (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

7.7 Generalizing the Induction Hypothesis

7.7.1 Exercise [★★★]: Prove theorems plus_n_n_injective_take2 and index_after_last in Logic.v.

7.7.2 Exercise [★★★]: Provide an informal proof corresponding to your coq proof of index_after_last in Logic.v.

7.7.3 Exercise [★★★, Optional]: Prove \text{length}_\text{snoc}''', eqnat_false_S, and \text{length}_\text{append}_\text{cons} in Logic.v.

7.7.4 Exercise [★★★★]: Prove theorem \text{length}_\text{append}twice in Logic.v.