SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

INDPRINCIPLES

INDUCTION PRINCIPLES

With the Curry-Howard correspondence and its realization in Coq in mind, we can now take a deeper look at induction principles.

```
Set Warnings "-notation-overridden,-parsing".
Require Export ProofObjects.
```

Basics

Every time we declare a new `Inductive` datatype, Coq automatically generates an *induction principle* for this type. This induction principle is a theorem like any other: If `t` is defined inductively, the corresponding induction principle is called `t_ind`. Here is the one for natural numbers:

```
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 `induction` tactic is a straightforward wrapper that, at its core, simply performs `apply t_ind`. To see this more clearly, let's experiment with directly using `apply nat_ind`, instead of the `induction` tactic, to carry out some proofs. Here, for example, is an alternate proof of a theorem that we saw in the Basics chapter.

```
Theorem mult_0_r' : ∀ n:nat,
  n * 0 = 0.
Proof.
  apply nat_ind.
  - (* n = O *) reflexivity.
  - (* n = S n' *) simpl. intros n' IHn'. rewrite → IHn'.
    reflexivity. Qed.
```

This proof is basically the same as the earlier one, but a few minor differences 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.

some explanation may help!!!

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. By contrast, the induction tactic works either with a variable in the context or a quantified variable in the goal.

These conveniences make induction nicer to use in practice than applying induction principles like nat_ind directly. But it is important to realize that, modulo these bits of bookkeeping, applying nat_ind is what we are really doing.

Exercise: 2 stars, optional (plus_one_r')

Complete this proof without using the induction tactic.

```
Theorem plus_one_r' : ∀ n:nat,
  n + 1 = S n.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Coq generates induction principles for every datatype defined with Inductive, including those that aren't recursive. Although of course we don't need induction to prove properties of non-recursive datatypes, the idea of an induction principle still makes sense for them: it gives a way to prove that a property holds for all values of the type.

These generated principles follow a similar pattern. If we define a type t with constructors c_1 ... cn, Coq generates a theorem with this shape:

```
t_ind : ∀ P : t → Prop,
          ... case for c1 ... →
          ... case for c2 ... → ...
          ... case for cn ... →
          ∀ n : t, P n
```

The specific shape of each case depends on the arguments to the corresponding constructor. Before trying to write down a general rule, let's look at some more examples. First, an example where the constructors take no arguments:

```
Inductive yesno : Type :=
  | yes : yesno
  | no : yesno.

Check yesno_ind.
(* ===> yesno_ind : forall P : yesno -> Prop,
                      P yes  ->
```

```
                   P no  ->
                   forall y : yesno, P y *)
```

Exercise: 1 star, optional (rgb)

Write out the induction principle that Coq will generate for the following datatype. Write down your answer on paper or type it into a comment, and then compare it with what Coq prints.

```
Inductive rgb : Type :=
  | red : rgb
  | green : rgb
  | blue : rgb.
Check rgb_ind.
```

□

Here's another example, this time with one of the constructors taking some arguments.

```
Inductive natlist : Type :=
  | nnil : natlist
  | ncons : nat → natlist → natlist.

Check natlist_ind.
(* ===> (modulo a little variable renaming)
   natlist_ind :
      forall P : natlist -> Prop,
         P nnil  ->
         (forall (n : nat) (l : natlist),
            P l -> P (ncons n l)) ->
         forall n : natlist, P n *)
```

Exercise: 1 star, optional (natlist1)

why???

Suppose we had written the above definition a little differently:

```
Inductive natlist1 : Type :=
  | nnil1 : natlist1
  | nsnoc1 : natlist1 → nat → natlist1.
```

Now what will the induction principle look like? □

From these examples, we can extract this general rule:

- The type declaration gives several constructors; each corresponds to one clause of the induction principle.
- Each constructor c takes argument types a_1 ... an.
- Each ai can be either t (the datatype we are defining) or some other type s.
- The corresponding case of the induction principle says:
  - "For all values x_1...xn of types a_1...an, if P holds for each of the inductive arguments (each xi of type t), then P holds for c x_1 ... xn".

Exercise: 1 star, optional (byntree_ind)

Write out the induction principle that Coq will generate for the following datatype. (Again, write down your answer on paper or type it into a comment, and then compare it with what Coq prints.)

```
Inductive byntree : Type :=
 | bempty : byntree
 | bleaf : yesno → byntree
 | nbranch : yesno → byntree → byntree → byntree.
```

□

Exercise: 1 star, optional (ex_set)

Here is an induction principle for an inductively defined set.

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

Give an `Inductive` definition of `ExSet`:

```
Inductive ExSet : Type :=
  (* FILL IN HERE *)
.
```

□

Polymorphism

Next, what about polymorphic datatypes?

The inductive definition of polymorphic lists

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

is very similar to that of `natlist`. The main difference is that, here, the whole definition is *parameterized* on a set `X`: that is, we are defining a *family* of inductive types `list X`, one for each `X`. (Note that, wherever `list` appears in the body of the declaration, it is always applied to the parameter `X`.) The induction principle is likewise parameterized on `X`:

```
list_ind :
  ∀ (X : Type) (P : list X → Prop),
     P [] →
     (∀ (x : X) (l : list X), P l → P (x :: l)) →
     ∀ l : list X, P l
```

Note that the *whole* induction principle is parameterized on `X`. That is, `list_ind` can be thought of as a polymorphic function that, when applied to a type `X`, gives us back an induction principle specialized to the type `list X`.

Exercise: 1 star, optional (tree)

Write out the induction principle that Coq will generate for the following datatype. Compare your answer with what Coq prints.

```
Inductive tree (X:Type) : Type :=
  | leaf : X → tree X
  | node : tree X → tree X → tree X.
Check tree_ind.
```

□

Exercise: 1 star, optional (mytype)

Find an inductive definition that gives rise to the following induction principle:

```
mytype_ind :
  ∀ (X : Type) (P : mytype X → Prop),
      (∀ x : X, P (constr1 X x)) →
      (∀ n : nat, P (constr2 X n)) →
      (∀ m : mytype X, P m →
         ∀ n : nat, P (constr3 X m n)) →
      ∀ m : mytype X, P m
```

□

Exercise: 1 star, optional (foo)

Find an inductive definition that gives rise to the following induction principle:

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

□

Exercise: 1 star, optional (foo')

Consider the following inductive definition:

```
Inductive foo' (X:Type) : Type :=
  | 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.)

```
foo'_ind :
   ∀ (X : Type) (P : foo' X → Prop),
         (∀ (l : list X) (f : foo' X),
               _______________________ →
               _______________________    ) →
         ___________________________________________ →
         ∀ f : foo' X, ________________________
```

□

Induction Hypotheses

Where does the phrase "induction hypothesis" fit into this story?

The induction principle for numbers

```
∀ P : nat → Prop,
     P 0   →
     (∀ n : nat, P n → P (S n))  →
     ∀ 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 proofs by induction more explicit by giving this expression a name. For example, instead of stating the theorem `mult_0_r` as "∀`n`, `n` * 0 = 0," we can write it as "∀`n`, `P_m0r n`", where `P_m0r` is defined as...

```
  Definition P_m0r (n:nat) : Prop :=
    n * 0 = 0.
```

... or equivalently:

```
  Definition P_m0r' : nat→Prop :=
    fun n ⇒ n * 0 = 0.
```

Now it is easier to see where `P_m0r` appears in the proof.

```
  Theorem mult_0_r'' : ∀ n:nat,
    P_m0r n.
  Proof.
    apply nat_ind.
    - (* n = O *) reflexivity.
    - (* n = S n' *)
      (* Note the proof state at this point! *)
      intros n IHn.
      unfold P_m0r in IHn. unfold P_m0r. simpl. apply IHn. Qed.
```

This extra naming step isn't something that we do in normal proofs, but it is useful to do it explicitly for an example or two, because it allows us to see exactly what the induction hypothesis is. If we prove ∀n, P_m0r n by induction on n (using either induction or apply nat_ind), we see that the first subgoal requires us to prove P_m0r 0 ("P holds for zero"), while the second subgoal requires us to prove ∀n', P_m0r n' → P_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'.

premise -> conclusion

More on the induction Tactic

The induction tactic actually does even more low-level bookkeeping for us than we discussed above.

Recall the informal statement of the induction principle for natural numbers:

- If P n is some proposition involving a natural number n, and we want to show that P holds for *all* numbers n, we can reason like this:
    - show that P O holds
    - show that, if P n' holds, then so does P (S n')
    - conclude that P n holds for all n.

So, when we begin a proof with intros n and then induction n, we are first telling Coq to consider a *particular* n (by introducing it into the context) and then telling it to prove something about *all* numbers (by using induction).

What Coq actually does in this situation, internally, is to "re-generalize" the variable we perform induction on. For example, in our original proof that plus is associative...

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  (* ...we first introduce all 3 variables into the context,
     which amounts to saying "Consider an arbitrary n, m, and
     p..." *)
  intros n m p.
  (* ...We now use the induction tactic to prove P n (that
     is, n + (m + p) = (n + m) + p) for _all_ n,
     and hence also for the particular n that is in the context
     at the moment. *)
  induction n as [| n'].
  - (* n = O *) reflexivity.
  - (* n = S n' *)
    (* In the second subgoal generated by induction -- the
       "inductive step" -- we must prove that P n' implies
       P (S n') for all n'.  The induction tactic
       automatically introduces n' and P n' into the context
       for us, leaving just P (S n') as the goal. *)
    simpl. rewrite → IHn'. reflexivity. Qed.
```

It also works to apply `induction` to a variable that is quantified in the goal.

```
Theorem plus_comm' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction n as [| n'].
  - (* n = O *) intros m. rewrite <- plus_n_O. reflexivity.
  - (* n = S n' *) intros m. simpl. rewrite → IHn'.
    rewrite <- plus_n_Sm. reflexivity. Qed.
```

Note that `induction` `n` leaves `m` still bound in the goal — i.e., what we are proving inductively is a statement beginning with ∀`m`.

wow!!!

If we do `induction` on a variable that is quantified in the goal *after* some other quantifiers, the `induction` tactic will automatically introduce the variables bound by these quantifiers into the context.

```
Theorem plus_comm'' : ∀ n m : nat,
  n + m = m + n.
Proof.
  (* Let's do induction on m this time, instead of n... *)
  induction m as [| m'].
  - (* m = O *) simpl. rewrite <- plus_n_O. reflexivity.
  - (* m = S m' *) simpl. rewrite <- IHm'.
    rewrite <- plus_n_Sm. reflexivity. Qed.
```

Exercise: 1 star, optional (plus_explicit_prop)

Rewrite both `plus_assoc'` and `plus_comm'` and their proofs in the same style as `mult_0_r''` above — that is, for each theorem, give an explicit `Definition` of the proposition being proved by induction, and state the theorem and proof in terms of this defined proposition.

```
(* FILL IN HERE *)
```

□

Induction Principles in `Prop`

Earlier, we looked in detail at the induction principles that Coq generates for inductively defined *sets*. The induction principles for inductively defined *propositions* like `ev` are a tiny bit more complicated. As with all induction principles, we want to use the induction principle on `ev` to prove things by inductively considering the possible shapes that something in `ev` can have. Intuitively speaking, however, what we want to prove are not statements about *evidence* but statements about *numbers*: accordingly, we want an induction principle that lets us prove properties of numbers by induction on evidence.

hmmm!!

For example, from what we've said so far, you might expect the inductive definition of ev...

```
      Inductive ev : nat → Prop :=
      | ev_0 : ev 0
      | ev_SS : ∀ n : nat, ev n → ev (S (S n)).
```

...to give rise to an induction principle that looks like this...

```
    ev_ind_max : ∀ P : (∀ n : nat, ev n → Prop),
         P O ev_0 →
         (∀ (m : nat) (E : ev m),
            P m E →
            P (S (S m)) (ev_SS m E)) →
         ∀ (n : nat) (E : ev n),
         P n E
```

... because:

- Since ev is indexed by a number n (every ev object E is a piece of evidence that some particular number n is even), the proposition P is parameterized by both n and E — that is, the induction principle can be used to prove assertions involving both an even number and the evidence that it is even.
- Since there are two ways of giving evidence of evenness (ev has two constructors), applying the induction principle generates two subgoals:
    - We must prove that P holds for O and ev_0.
    - We must prove that, whenever n is an even number and E is an evidence of its evenness, if P holds of n and E, then it also holds of S (S n) and ev_SS n E.
- If these subgoals can be proved, then the induction principle tells us that P is true for *all* even numbers n and evidence E of their evenness.

This is more flexibility than we normally need or want: it is giving us a way to prove logical assertions where the assertion involves properties of some piece of *evidence* of evenness, while all we really care about is proving properties of *numbers* that are even — we are interested in assertions about numbers, not about evidence. It would therefore be more convenient to have an induction principle for proving propositions P that are parameterized just by n and whose conclusion establishes P for all even numbers n:

```
       ∀ P : nat → Prop,
       ... →
       ∀ n : nat,
       even n → P n
```

For this reason, Coq actually generates the following simplified induction principle for ev:

```
Check ev_ind.
(* ===> ev_ind
        : forall P : nat -> Prop,
          P 0 ->
          (forall n : nat, ev n -> P n -> P (S (S n))) ->
          forall n : nat,
          ev n -> P n *)
```

In particular, Coq has dropped the evidence term `E` as a parameter of the the proposition `P`.

In English, `ev_ind` says:

- Suppose, `P` is a property of natural numbers (that is, `P n` is a `Prop` for every `n`). To show that `P n` holds whenever `n` is even, it suffices to show:
  - `P` holds for `0`,
  - for any `n`, if `n` is even and `P` holds for `n`, then `P` holds for `S (S n)`.

As expected, we can apply `ev_ind` directly instead of using `induction`. For example, we can use it to show that `ev'` (the slightly awkward alternate definition of evenness that we saw in an exercise in the \chap{IndProp} chapter) is equivalent to the cleaner inductive definition `ev`:

```
Theorem ev_ev' : ∀ n, ev n → ev' n.
Proof.
  apply ev_ind.
  - (* ev_0 *)
    apply ev'_0.
  - (* ev_SS *)
    intros m Hm IH.
    apply (ev'_sum 2 m).
    + apply ev'_2.
    + apply IH.
Qed.
```

The precise form of an `Inductive` definition can affect the induction principle Coq generates.

For example, in chapter IndProp, we defined ≤ as:

```
(* Inductive le : nat -> nat -> Prop :=
     | le_n : forall n, le n n
     | le_S : forall n m, (le n m) -> (le n (S m)). *)
```

This definition can be streamlined a little by observing that the left-hand argument `n` is the same everywhere in the definition, so we can actually make it a "general parameter" to the whole definition, rather than an argument to each constructor.

```
Inductive le (n:nat) : nat → Prop :=
  | le_n : le n n
  | le_S : ∀ m, (le n m) → (le n (S m)).

Notation "m ≤ n" := (le m n).
```

The second one is better, even though it looks less symmetric. Why? Because it gives us a simpler induction principle.

```
Check le_ind.
(* ===>  forall (n : nat) (P : nat -> Prop),
            P n ->
            (forall m : nat, n <= m -> P m -> P (S m)) ->
            forall n0 : nat, n <= n0 -> P n0 *)
```

Formal vs. Informal Proofs by Induction

Question: What is the relation between a formal proof of a proposition `P` and an informal proof of the same proposition `P`?

Answer: The latter should *teach* the reader how to produce the former.

Question: How much detail is needed??

Unfortunately, there is no single right answer; rather, there is a range of choices.

At one end of the spectrum, we can essentially give the reader the whole formal proof (i.e., the "informal" proof will amount to just transcribing the formal one into words). This may give the reader the ability to reproduce the formal one for themselves, but it probably doesn't *teach* them anything much.

At the other end of the spectrum, we can say "The theorem is true and you can figure out why for yourself if you think about it hard enough." This is also not a good teaching strategy, because often writing the proof requires one or more significant insights into the thing we're proving, and most readers will give up before they rediscover all the same insights as we did.

In the middle is the golden mean — a proof that includes all of the essential insights (saving the reader the hard work that we went through to find the proof in the first place) plus high-level suggestions for the more routine parts to save the reader from spending too much time reconstructing these (e.g., what the IH says and what must be shown in each case of an inductive proof), but not so much detail that the main ideas are obscured.

Since we've spent much of this chapter looking "under the hood" at formal proofs by induction, now is a good moment to talk a little about *informal* proofs by induction.

In the real world of mathematical communication, written proofs range from extremely longwinded and pedantic to extremely brief and telegraphic. Although the ideal is somewhere in between, while one is getting used to the style it is better to start out at the pedantic end. Also, during the learning phase, it is probably helpful to have a clear standard to compare against. With this in mind, we offer two templates — one for proofs by induction over *data* (i.e., where the thing we're doing induction on lives in

`Type`) and one for proofs by induction over *evidence* (i.e., where the inductively defined thing lives in `Prop`).

Induction Over an Inductively Defined Set

Template:

might be useful for future!!!

- *Theorem*: <Universally quantified proposition of the form "For all `n:S`, `P(n)`," where `S` is some inductively defined set.>

  Proof: By induction on `n`.

  <one case for each constructor `c` of `S`...>

  - Suppose `n` = `c` a_1 `... ak`, where <...and here we state the IH for each of the `a`'s that has type `S`, if any>. We must show <...and here we restate `P(c` a_1 `... ak)`>.

    <go on and prove `P(n)` to finish the case...>

  - <other cases similarly...> □

Example:

- *Theorem*: For all sets `X`, lists `l : list X`, and numbers `n`, if `length l` = `n` then `index (S n) l` = `None`.

  Proof: By induction on `l`.

  - Suppose `l` = `[]`. We must show, for all numbers `n`, that, if `length []` = `n`, then `index (S n) []` = `None`.

    This follows immediately from the definition of `index`.

  - Suppose `l` = `x :: l'` for some `x` and `l'`, where `length l'` = `n'` implies `index (S n') l'` = `None`, for any number `n'`. We must show, for all `n`, that, if `length (x::l')` = `n` then `index (S n) (x::l')` = `None`.

    Let `n` be a number with `length l` = `n`. Since

    ```
    length l = length (x::l') = S (length l'),
    ```

    it suffices to show that

    ```
    index (S (length l')) l' = None.
    ```

    But this follows directly from the induction hypothesis, picking `n'` to be `length l'`. □

Induction Over an Inductively Defined Proposition

Since inductively defined proof objects are often called "derivation trees," this form of proof is also known as *induction on derivations*.

Template:

- *Theorem*: <Proposition of the form "`Q → P`," where `Q` is some inductively defined proposition (more generally, "For all `x` `y` `z`, `Q x y z → P x y z`")>

  Proof: By induction on a derivation of `Q`. <Or, more generally, "Suppose we are given `x`, `y`, and `z`. We show that `Q x y z` implies `P x y z`, by induction on a derivation of `Q x y z`"...>

  <one case for each constructor `c` of `Q`...>

  - Suppose the final rule used to show `Q` is `c`. Then <...and here we state the types of all of the `a`'s together with any equalities that follow from the definition of the constructor and the IH for each of the `a`'s that has type `Q`, if there are any>. We must show <...and here we restate `P`>.

    <go on and prove `P` to finish the case...>
  - <other cases similarly...> □

Example

- *Theorem*: The `≤` relation is transitive — i.e., for all numbers `n`, `m`, and `o`, if `n ≤ m` and `m ≤ o`, then `n ≤ o`.

  Proof: By induction on a derivation of `m ≤ o`.

  - Suppose the final rule used to show `m ≤ o` is `le_n`. Then `m = o` and we must show that `n ≤ m`, which is immediate by hypothesis.
  - Suppose the final rule used to show `m ≤ o` is `le_S`. Then `o = S o'` for some `o'` with `m ≤ o'`. We must show that `n ≤ S o'`. By induction hypothesis, `n ≤ o'`.

    But then, by `le_S`, `n ≤ S o'`. □