SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

can we do strong induction in coq?

TABLE OF CONTENTS INDEX ROADMAP

INDUCTION

PROOF BY INDUCTION

Before getting started, we need to import all of our definitions from the previous chapter:

```
Require Export Basics.
```

For the `Require Export` to work, you first need to use `coqc` to compile `Basics.v` into `Basics.vo`. This is like making a `.class` file from a `.java` file, or a `.o` file from a `.c` file. There are two ways to do it:

- In CoqIDE:

  Open `Basics.v`. In the "Compile" menu, click on "Compile Buffer".

- From the command line: Either

  ```
  make Basics.vo
  ```

  (assuming you've downloaded the whole LF directory and have a working `make` command) or

  ```
  coqc Basics.v
  ```

  (which should work from any terminal window).

If you have trouble (e.g., if you get complaints about missing identifiers later in the file), it may be because the "load path" for Coq is not set up correctly. The `Print LoadPath.` command may be helpful in sorting out such issues.

In particular, if you see a message like

```
Compiled library Foo makes inconsistent assumptions over library Coq.Init.Bar
```

you should check whether you have multiple installations of Coq on your machine. If so, it may be that commands (like `coqc`) that you execute in a terminal window are getting a different version of Coq than commands executed by Proof General or CoqIDE.

One more tip for CoqIDE users: If you see messages like `Error: Unable to locate library Basics`, a likely reason is inconsistencies between compiling things *within CoqIDE* vs *using coqc* from the command line. This typically happens when there are two incompatible versions of `coqc` installed on your system (one associated with CoqIDE, and one associated with `coqc` from the terminal). The workaround for this situation is compiling using CoqIDE only (i.e. choosing "make" from the menu), and avoiding using `coqc` directly at all.

Proof by Induction

We proved in the last chapter that `0` is a neutral element for `+` on the left, using an easy argument based on simplification. We also observed that proving the fact that it is also a neutral element on the *right*...

```
Theorem plus_n_O_firsttry : ∀ n:nat,
  n = n + 0.
```

... can't be done in the same simple way. Just applying `reflexivity` doesn't work, since the `n` in `n + 0` is an arbitrary unknown number, so the `match` in the definition of `+` can't be simplified.

```
Proof.
  intros n.
  simpl. (* Does nothing! *)
Abort.
```

And reasoning by cases using `destruct n` doesn't get us much further: the branch of the case analysis where we assume `n = 0` goes through fine, but in the branch where `n = S n'` for some `n'` we get stuck in exactly the same way.

```
Theorem plus_n_O_secondtry : ∀ n:nat,
  n = n + 0.
Proof.
  intros n. destruct n as [| n'].
  - (* n = 0 *)
    reflexivity. (* so far so good... *)
  - (* n = S n' *)
    simpl. (* ...but here we are stuck again *)
Abort.
```

We could use `destruct n'` to get one step further, but, since `n` can be arbitrarily large, if we just go on like this we'll never finish.

To prove interesting facts about numbers, lists, and other inductively defined sets, we usually need a more powerful reasoning principle: *induction*.

Recall (from high school, a discrete math course, etc.) the *principle of induction over 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, for any `n'`, if `P(n')` holds, then so does `P(S n')`;
- conclude that `P(n)` holds for all `n`.

In Coq, the steps are the same: we begin with the goal of proving `P(n)` for all `n` and break it down (by applying the `induction` tactic) into two separate subgoals: one where we must show `P(O)` and another where we must show `P(n') → P(S n')`. Here's how this works for the theorem at hand:

```
Theorem plus_n_O : ∀ n:nat, n = n + 0.
Proof.
  intros n. induction n as [| n' IHn'].
  - (* n = 0 *) reflexivity.
  - (* n = S n' *) simpl. rewrite <- IHn'. reflexivity. Qed.
```

Like `destruct`, the `induction` tactic takes an `as...` clause that specifies the names of the variables to be introduced in the subgoals. Since there are two subgoals, the `as...` clause has two parts, separated by `|`. (Strictly speaking, we can omit the `as...` clause and Coq will choose names for us. In practice, this is a bad idea, as Coq's automatic choices tend to be confusing.)

In the first subgoal, `n` is replaced by `0`. No new variables are introduced (so the first part of the `as...` is empty), and the goal becomes `0 = 0 + 0`, which follows by simplification.

In the second subgoal, `n` is replaced by `S n'`, and the assumption `n' + 0 = n'` is added to the context with the name `IHn'` (i.e., the Induction Hypothesis for `n'`). These two names are specified in the second part of the `as...` clause. The goal in this case becomes `S n' = (S n') + 0`, which simplifies to `S n' = S (n' + 0)`, which in turn follows from `IHn'`.

```
Theorem minus_diag : ∀ n,
  minus n n = 0.
Proof.
  (* WORKED IN CLASS *)
  intros n. induction n as [| n' IHn'].
  - (* n = 0 *)
    simpl. reflexivity.
  - (* n = S n' *)
    simpl. rewrite → IHn'. reflexivity. Qed.
```

(The use of the `intros` tactic in these proofs is actually redundant. When applied to a goal that contains quantified variables, the `induction` tactic will automatically move them into the context as needed.)

Exercise: 2 stars, recommended (basic_induction)

Prove the following using induction. You might need previously proven results.

```
Theorem mult_0_r : ∀ n:nat,
  n * 0 = 0.
Proof.
```

```
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: mult_0_r *)

Theorem plus_n_Sm : ∀ n m : nat,
  S (n + m) = n + (S m).
Proof.
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: plus_n_Sm *)

Theorem plus_comm : ∀ n m : nat,
  n + m = m + n.
Proof.
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: plus_comm *)

Theorem plus_assoc : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: plus_assoc *)
```

□

Exercise: 2 stars (double_plus)

Consider the following function, which doubles its argument:

```
Fixpoint double (n:nat) :=
  match n with
  | O ⇒ O
  | S n' ⇒ S (S (double n'))
  end.
```

Use induction to prove this simple fact about `double`:

```
Lemma double_plus : ∀ n, double n = n + n .
Proof.
  (* FILL IN HERE *) Admitted.
```

□

h: x = y
some expression
rewrite -> : looks for x in some expression
and if it finds it will replace with y
rewrite <- : will look for y in some expression
and if found replace with x!!!

Exercise: 2 stars, optional (evenb_S)

One inconvenient aspect of our definition of `evenb n` is the recursive call on `n - 2`. This makes proofs about `evenb n` harder when done by induction on `n`, since we may need an induction hypothesis about `n - 2`. The following lemma gives an alternative characterization of `evenb (S n)` that works better with induction:

```
Theorem evenb_S : ∀ n : nat,
  evenb (S n) = negb (evenb n).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 1 star (destruct_induction)

Briefly explain the difference between the tactics `destruct` and `induction`.

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

□

Proofs Within Proofs

In Coq, as in informal mathematics, large proofs are often broken into a sequence of theorems, with later proofs referring to earlier theorems. But sometimes a proof will require some miscellaneous fact that is too trivial and of too little general interest to bother giving it its own top-level name. In such cases, it is convenient to be able to simply state and prove the needed "sub-theorem" right at the point where it is used. The `assert` tactic allows us to do this. For example, our earlier proof of the `mult_0_plus` theorem referred to a previous theorem named `plus_O_n`. We could instead use `assert` to state and prove `plus_O_n` in-line:

```
Theorem mult_0_plus' : ∀ n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  assert (H: 0 + n = n). { reflexivity. }
  rewrite → H.
  reflexivity. Qed.
```

The `assert` tactic introduces two sub-goals. The first is the assertion itself; by prefixing it with `H:` we name the assertion `H`. (We can also name the assertion with `as` just as we did above with `destruct` and `induction`, i.e., `assert (0 + n = n) as H`.) Note that we surround the proof of this assertion with curly braces `{ ... }`, both for readability and so that, when using Coq interactively, we can see more easily when we have finished this sub-proof. The second goal is the same as the one at the point where we invoke `assert` except that, in the context, we now have the assumption `H` that `0 + n = n`. That is, `assert` generates one subgoal where we must prove the asserted fact and a second subgoal where we can use the asserted fact to make progress on whatever we were trying to prove in the first place.

Another example of `assert`...

For example, suppose we want to prove that `(n + m) + (p + q) = (m + n) + (p + q)`. The only difference between the two sides of the `=` is that the arguments `m` and `n` to the first inner `+` are swapped, so it seems we should be able to use the commutativity of addition (`plus_comm`) to rewrite one into the other. However, the `rewrite` tactic is not very smart about *where* it applies the rewrite. There are three uses of `+` here, and it turns out that doing `rewrite → plus_comm` will affect only the *outer* one...

```
Theorem plus_rearrange_firsttry : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  (* We just need to swap (n + m) for (m + n)... seems
     like plus_comm should do the trick! *)
```

```
  rewrite → plus_comm.
  (* Doesn't work...Coq rewrote the wrong plus! *)
Abort.
```

To use `plus_comm` at the point where we need it, we can introduce a local lemma stating that `n + m = m + n` (for the particular `m` and `n` that we are talking about here), prove this lemma using `plus_comm`, and then use it to do the desired rewrite.

```
Theorem plus_rearrange : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  assert (H: n + m = m + n).
  { rewrite → plus_comm. reflexivity. }
  rewrite → H. reflexivity. Qed.
```

Formal vs. Informal Proof

"*Informal proofs are algorithms; formal proofs are code.*"

What constitutes a successful proof of a mathematical claim? The question has challenged philosophers for millennia, but a rough and ready definition could be this: A proof of a mathematical proposition `P` is a written (or spoken) text that instills in the reader or hearer the certainty that `P` is true — an unassailable argument for the truth of `P`. That is, a proof is an act of communication.

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 that `P` can be mechanically derived from a certain set of formal logical rules, and the proof is a recipe that guides the program in checking this fact. Such recipes are *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, and will thus necessarily be *informal*. Here, the criteria for success are less clearly specified. A "valid" 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. Some readers 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 other readers, 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, since it is easier for them to fill in the details for themselves than to wade through a written presentation of them. Ultimately, there is no universal standard, because there 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 — at least 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 are using Coq in this course, we will be working heavily with formal proofs. But this doesn't mean we can completely forget about informal ones! Formal proofs are useful in many ways, but they are *not* very efficient ways of communicating ideas between human beings.

For example, here is a proof that addition is associative:

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof. intros n m p. induction n as [| n' IHn']. reflexivity.
  simpl. rewrite → IHn'. reflexivity. Qed.
```

Coq is perfectly happy with this. For a human, however, it is difficult to make much sense of it. We can use comments and bullets to show the structure a little more clearly...

```
Theorem plus_assoc'' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - (* n = 0 *)
    reflexivity.
  - (* n = S n' *)
    simpl. rewrite → IHn'. reflexivity. Qed.
```

... and if you're used to Coq you may be able to 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.

A (pedantic) mathematician might write the proof something like this:

- *Theorem*: For any `n`, `m` and `p`,

  ```
  n + (m + p) = (n + m) + p.
  ```

  Proof: By induction on `n`.

  - First, suppose `n = 0`. We must show

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

    This follows directly from the definition of `+`.

  - Next, suppose `n = S n'`, where

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

    We must show

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

    By the definition of `+`, this follows from

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

which is immediate from the induction hypothesis. *Qed*.

The overall form of the proof is basically similar, and of course this is no accident: 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).

Exercise: 2 stars, advanced, recommended (plus_comm_informal)

Translate your solution for `plus_comm` into an informal proof:

Theorem: Addition is commutative.

Proof: `(* FILL IN HERE *)`

□

Exercise: 2 stars, optional (beq_nat_refl_informal)

Write an informal proof of the following theorem, using the informal proof of `plus_assoc` as a model. Don't just paraphrase the Coq tactics into English!

Theorem: `true` = `beq_nat n n` for any `n`.

Proof: `(* FILL IN HERE *)`

□ double check to see if it's true

mult_plus_distr_r ?? easier approach?

cannot do it with assert??? and also replace??

```
Theorem plus_swap' : forall n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
  intros n m p. induction n as [| n' IH].
  - reflexivity.
  - simpl. assert (H: S (n + m) = n + (S m)).
  {intros n m. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite -> IHn'. reflexivity.}
 rewrite -> IH. reflexivity.
Qed.
```

—> even doesn't work after renaming n in assert to n' —> btw, really annoying!!