SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

TACTICS

MORE BASIC TACTICS

This chapter introduces several additional proof strategies and tactics that allow us to begin proving more interesting properties of functional programs. We will see:

- how to use auxiliary lemmas in both "forward-style" and "backward-style" proofs;
- how to reason about data constructors (in particular, how to use the fact that they are injective and disjoint);
- how to strengthen an induction hypothesis (and when such strengthening is required); and
- more details on how to reason by case analysis.

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

The apply Tactic

We often encounter situations where the goal to be proved is *exactly* the same as some hypothesis in the context or some previously proved lemma.

```
Theorem silly1 : ∀ (n m o p : nat),
     n = m →
     [n;o] = [n;p] →
     [n;o] = [m;p].
Proof.
  intros n m o p eq₁ eq₂.
  rewrite <- eq₁.
```

you can think of the apply tactic as apply function with whole

Here, we could finish with "`rewrite → eq₂. reflexivity.`" as we have done several times before. We can achieve the same effect in a single step by using the `apply` tactic instead:

```
  apply eq₂. Qed.
```

The `apply` tactic also works with *conditional* hypotheses and lemmas: if the statement being applied is an implication, then the premises of this implication will be added to the list of subgoals needing to be proved.

```
Theorem silly2 : ∀ (n m o p : nat),
     n = m →
     (∀ (q r : nat), q = r → [q;o] = [r;p]) →
     [n;o] = [m;p].
Proof.
  intros n m o p eq1 eq2.
  apply eq2. apply eq1. Qed.
```

Typically, when we use `apply H`, the statement `H` will begin with a ∀that binds some *universal variables*. When Coq matches the current goal against the conclusion of `H`, it will try to find appropriate values for these variables. For example, when we do `apply eq2` in the following proof, the universal variable `q` in `eq2` gets instantiated with `n` and `r` gets instantiated with `m`.

```
Theorem silly2a : ∀ (n m : nat),
     (n,n) = (m,m) →
     (∀ (q r : nat), (q,q) = (r,r) → [q] = [r]) →
     [n] = [m].
Proof.
  intros n m eq1 eq2.
  apply eq2. apply eq1. Qed.
```

Exercise: 2 stars, optional (silly_ex)

Complete the following proof without using `simpl`.

```
Theorem silly_ex :
     (∀ n, evenb n = true → oddb (S n) = true) →
     evenb 3 = true →
     oddb 4 = true.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

To use the `apply` tactic, the (conclusion of the) fact being applied must match the goal exactly — for example, `apply` will not work if the left and right sides of the equality are swapped.

```
Theorem silly3_firsttry : ∀ (n : nat),
     true = beq_nat n 5 →
     beq_nat (S (S n)) 7 = true.
Proof.
  intros n H.
```

Here we cannot use `apply` directly, but we can use the `symmetry` tactic, which switches the left and right sides of an equality in the goal.

```
    symmetry.
    simpl. (* (This simpl is optional, since apply will perform
              simplification first, if needed.) *)
    apply H. Qed.
```

Exercise: 3 stars (apply_exercise1)

(*Hint*: You can use `apply` with previously defined lemmas, not just hypotheses in the context. Remember that `Search` is your friend.)

```
Theorem rev_exercise1 : ∀ (l l' : list nat),
     l = rev l' →
     l' = rev l.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 1 star, optional (apply_rewrite)

Briefly explain the difference between the tactics `apply` and `rewrite`. What are the situations where both can usefully be applied?

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

☐

The apply ... with ... Tactic

The following silly example uses two rewrites in a row to get from `[a,b]` to `[e,f]`.

```
Example trans_eq_example : ∀ (a b c d e f : nat),
     [a;b] = [c;d] →
     [c;d] = [e;f] →
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  rewrite → eq1. rewrite → eq2. reflexivity. Qed.
```

in apply you really need to have a implication
but for rewrite your just substituting an expression
with another one that you have the equality of them
as your hypothesis

Since this is a common pattern, we might like to pull it out as a lemma recording, once and for all, the fact that equality is transitive.

```
Theorem trans_eq : ∀ (X:Type) (n m o : X),
  n = m → m = o → n = o.
Proof.
  intros X n m o eq1 eq2. rewrite → eq1. rewrite → eq2.
  reflexivity. Qed.
```

Now, we should be able to use `trans_eq` to prove the above example. However, to do this we need a slight refinement of the `apply` tactic.

```
Example trans_eq_example' : ∀ (a b c d e f : nat),
     [a;b] = [c;d] →
     [c;d] = [e;f] →
     [a;b] = [e;f].
```

```
Proof.
  intros a b c d e f eq1 eq2.
```

If we simply tell Coq `apply trans_eq` at this point, it can tell (by matching the goal against the conclusion of the lemma) that it should instantiate `X` with `[nat]`, `n` with `[a,b]`, and `o` with `[e,f]`. However, the matching process doesn't determine an instantiation for `m`: we have to supply one explicitly by adding `with (m:=[c,d])` to the invocation of `apply`.

```
  apply trans_eq with (m:=[c;d]).
  apply eq1. apply eq2. Qed.
```

Actually, we usually don't have to include the name `m` in the `with` clause; Coq is often smart enough to figure out which instantiation we're giving. We could instead write: `apply trans_eq with [c;d]`.

Exercise: 3 stars, optional (apply_with_exercise)

```
Example trans_eq_exercise : ∀ (n m o p : nat),
     m = (minustwo o) →
     (n + p) = m →
     (n + p) = (minustwo o).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

The `inversion` Tactic

Recall the definition of natural numbers:

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

It is obvious from this definition that every number has one of two forms: either it is the constructor `O` or it is built by applying the constructor `S` to another number. But there is more here than meets the eye: implicit in the definition (and in our informal understanding of how datatype declarations work in other programming languages) are two more facts:

- The constructor `S` is *injective*. That is, if `S n = S m`, it must be the case that `n = m`.
- The constructors `O` and `S` are *disjoint*. That is, `O` is not equal to `S n` for any `n`.

Similar principles apply to all inductively defined types: all constructors are injective, and the values built from distinct constructors are never equal. For lists, the `cons` constructor is injective and `nil` is different from every non-empty list. For booleans, `true` and `false` are different. (Since neither `true` nor `false` take any arguments, their injectivity is not interesting.) And so on.

Coq provides a tactic called `inversion` that allows us to exploit these principles in proofs. To see how to use it, let's show explicitly that the `S` constructor is injective:

```
Theorem S_injective : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H.
```

By writing `inversion H` at this point, we are asking Coq to generate all equations that it can infer from `H` as additional hypotheses, replacing variables in the goal as it goes. In the present example, this amounts to adding a new hypothesis H_1 : n = m and replacing `n` by `m` in the goal.

```
  inversion H.
  reflexivity.
Qed.
```

Here's a more interesting example that shows how multiple equations can be derived at once.

```
Theorem inversion_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
Proof.
  intros n m o H. inversion H. reflexivity. Qed.
```

We can name the equations that `inversion` generates with an `as ...` clause:

what if the inversion generates multiple hypothesis like the case of ex1?

```
Theorem inversion_ex2 : ∀ (n m : nat),
  [n] = [m] →
  n = m.
Proof.
  intros n m H. inversion H as [Hnm]. reflexivity. Qed.
```

Exercise: 1 star (inversion_ex3)

```
Example inversion_ex3 : ∀ (X : Type) (x y z : X) (l j : list X),
  x :: y :: l = z :: j →
  y :: l = x :: j →
  x = y.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

When used on a hypothesis involving an equality between *different* constructors (e.g., `S n` = `O`), `inversion` solves the goal immediately. Consider the following proof:

```
Theorem beq_nat_0_l : ∀ n,
   beq_nat 0 n = true → n = 0.
Proof.
  intros n.
```

We can proceed by case analysis on `n`. The first case is trivial.

```
  destruct n as [| n'].
  - (* n = 0 *)
    intros H. reflexivity.
```

However, the second one doesn't look so simple: assuming `beq_nat 0 (S n') = true`, we must show `S n' = 0`, but the latter clearly contradictory! The way forward lies in the assumption. After simplifying the goal state, we see that `beq_nat 0 (S n') = true` has become `false = true`:

```
  - (* n = S n' *)
    simpl.
```

If we use `inversion` on this hypothesis, Coq notices that the subgoal we are working on is impossible, and therefore removes it from further consideration.

```
    intros H. inversion H. Qed.
```

This is an instance of a logical principle known as the *principle of explosion*, which asserts that a contradictory hypothesis entails anything, even false things!

```
Theorem inversion_ex4 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. inversion contra. Qed.

Theorem inversion_ex5 : ∀ (n m : nat),
  false = true →
  [n] = [m].
Proof.
  intros n m contra. inversion contra. Qed.
```

If you find the principle of explosion confusing, remember that these proofs are not actually showing that the conclusion of the statement holds. Rather, they are arguing that, if the nonsensical situation described by the premise did somehow arise, then the nonsensical conclusion would follow. We'll explore the principle of explosion of more detail in the next chapter.

Exercise: 1 star (inversion_ex6)

```
Example inversion_ex6 : ∀ (X : Type)
                          (x y z : X) (l j : list X),
  x :: y :: l = [] →
  y :: l = z :: j →
  x = z.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

To summarize this discussion, suppose `H` is a hypothesis in the context or a previously proven lemma of the form

```
      c a1 a2 ... an = d b1 b2 ... bm
```

for some constructors `c` and `d` and arguments `a1 ... an` and `b1 ... bm`. Then `inversion H` has the following effect:

- If `c` and `d` are the same constructor, then, by the injectivity of this constructor, we know that `a1 = b1`, `a2 = b2`, etc. The `inversion H` adds these facts to the context and tries to use them to rewrite the goal.
- If `c` and `d` are different constructors, then the hypothesis `H` is contradictory, and the current goal doesn't have to be considered at all. In this case, `inversion H` marks the current goal as completed and pops it off the goal stack.

The injectivity of constructors allows us to reason that `∀(n m : nat), S n = S m → n = m`. The converse of this implication is an instance of a more general fact about both constructors and functions, which we will find convenient in a few places below:

```
  Theorem f_equal : ∀ (A B : Type) (f: A → B) (x y: A),
    x = y → f x = f y.
  Proof. intros A B f x y eq. rewrite eq. reflexivity. Qed.
```

Using Tactics on Hypotheses

By default, most tactics work on the goal formula and leave the context unchanged. However, most tactics also have a variant that performs a similar operation on a statement in the context.

For example, the tactic `simpl in H` performs simplification in the hypothesis named `H` in the context.

```
  Theorem S_inj : ∀ (n m : nat) (b : bool),
       beq_nat (S n) (S m) = b →
       beq_nat n m = b.
  Proof.
    intros n m b H. simpl in H. apply H. Qed.
```

Similarly, `apply L in H` matches some conditional statement `L` (of the form `L1 → L2`, say) against a hypothesis `H` in the context. However, unlike ordinary `apply` (which rewrites a goal matching `L2` into a subgoal `L1`), `apply L in H` matches `H` against `L1` and, if successful, replaces it with `L2`.

In other words, `apply L in H` gives us a form of "forward reasoning": from `L1 → L2` and a hypothesis matching `L1`, it produces a hypothesis matching `L2`. By contrast, `apply L` is "backward reasoning": it says that if we know `L1→L2` and we are trying to prove `L2`, it suffices to prove `L1`.

Here is a variant of a proof from above, using forward reasoning throughout instead of backward reasoning.

```
Theorem silly3' : ∀ (n : nat),
  (beq_nat n 5 = true → beq_nat (S (S n)) 7 = true) →
  true = beq_nat n 5 →
  true = beq_nat (S (S n)) 7.
Proof.
  intros n eq H.
  symmetry in H. apply eq in H. symmetry in H.
  apply H. Qed.
```

Forward reasoning starts from what is *given* (premises, previously proven theorems) and iteratively draws conclusions from them until the goal is reached. Backward reasoning starts from the *goal*, and iteratively reasons about what would imply the goal, until premises or previously proven theorems are reached. If you've seen informal proofs before (for example, in a math or computer science class), they probably used forward reasoning. In general, idiomatic use of Coq tends to favor backward reasoning, but in some situations the forward style can be easier to think about.

ANY SHORTER PROOF?

Exercise: 3 stars, recommended (plus_n_n_injective)

Practice using "in" variants in this proof. (Hint: use `plus_n_Sm`.)

```
Theorem plus_n_n_injective : ∀ n m,
     n + n = m + m →
     n = m.
Proof.
  intros n. induction n as [| n'].
  (* FILL IN HERE *) Admitted.
```

□

Varying the Induction Hypothesis

Sometimes it is important to control the exact form of the induction hypothesis when carrying out inductive proofs in Coq. In particular, we need to be careful about which of the assumptions we move (using `intros`) from the goal to the context before invoking the `induction` tactic. For example, suppose we want to show that the `double` function is injective — i.e., that it maps different arguments to different results:

```
Theorem double_injective: ∀ n m,
  double n = double m → n = m.
```

The way we *start* this proof is a bit delicate: if we begin with

```
intros n. induction n.
```

all is well. But if we begin it with

```
intros n m. induction n.
```

we get stuck in the middle of the inductive case...

```
Theorem double_injective_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction n as [| n'].
  - (* n = O *) simpl. intros eq. destruct m as [| m'].
    + (* m = O *) reflexivity.
    + (* m = S m' *) inversion eq.
  - (* n = S n' *) intros eq. destruct m as [| m'].
    + (* m = O *) inversion eq.
    + (* m = S m' *) apply f_equal.
```

At this point, the induction hypothesis, `IHn'`, does *not* give us `n' = m'` — there is an extra `S` in the way — so the goal is not provable.

```
      Abort.
```

What went wrong?

The problem is that, at the point we invoke the induction hypothesis, we have already introduced `m` into the context — intuitively, we have told Coq, "Let's consider some particular `n` and `m`..." and we now have to prove that, if `double n = double m` for *these particular* `n` and `m`, then `n = m`.

The next tactic, `induction n` says to Coq: We are going to show the goal by induction on `n`. That is, we are going to prove, for *all* `n`, that the proposition

IMP

- `P n` = "if `double n = double m`, then `n = m`"

holds, by showing

- `P O`

  (i.e., "if `double O = double m` then `O = m`") and

- `P n → P (S n)`

  (i.e., "if `double n = double m` then `n = m`" implies "if `double (S n) = double m` then `S n = m`").

If we look closely at the second statement, it is saying something rather strange: it says that, for a *particular* `m`, if we know

- "if `double n = double m` then `n = m`"

then we can prove

- "if `double (S n) = double m` then `S n = m`".

To see why this is strange, let's think of a particular `m` — say, `5`. The statement is then saying that, if we know

- `Q` = "if `double n = 10` then `n = 5`"

then we can prove

- `R` = "if `double (S n) = 10` then `S n = 5`".

But knowing `Q` doesn't give us any help at all with proving `R`! (If we tried to prove `R` from `Q`, we would start with something like "Suppose `double (S n) = 10`..." but then we'd be stuck: knowing that `double (S n)` is `10` tells us nothing about whether `double n` is `10`, so `Q` is useless.)

Trying to carry out this proof by induction on `n` when `m` is already in the context doesn't work because we are then trying to prove a relation involving *every* `n` but just a *single* `m`.

The successful proof of `double_injective` leaves `m` in the goal statement at the point where the `induction` tactic is invoked on `n`:

```
Theorem double_injective : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n. induction n as [| n'].
  - (* n = O *) simpl. intros m eq. destruct m as [| m'].
    + (* m = O *) reflexivity.
    + (* m = S m' *) inversion eq.

  - (* n = S n' *) simpl.
```

Notice that both the goal and the induction hypothesis are different this time: the goal asks us to prove something more general (i.e., to prove the statement for *every* `m`), but the IH is correspondingly more flexible, allowing us to choose any `m` we like when we apply the IH.

```
    intros m eq.
```

Now we've chosen a particular `m` and introduced the assumption that `double n = double m`. Since we are doing a case analysis on `n`, we also need a case analysis on `m` to keep the two "in sync."

```
    destruct m as [| m'].
    + (* m = O *) simpl.
```

The 0 case is trivial:

```
      inversion eq.

    + (* m = S m' *)
      apply f_equal.
```

At this point, since we are in the second branch of the `destruct m`, the `m'` mentioned in the context is the predecessor of the `m` we started out talking about. Since we are also in the `S` branch of the induction, this is perfect: if we instantiate the generic `m` in the IH with the current `m'` (this instantiation is performed automatically by the `apply` in the next step), then `IHn'` gives us exactly what we need to finish the proof.

```
      apply IHn'. inversion eq. reflexivity. Qed.
```

What you should take away from all this is that we need to be careful about using induction to try to prove something too specific: To prove a property of n and m by induction on n, it is sometimes important to leave m generic.

The following exercise requires the same pattern.

Exercise: 2 stars (beq_nat_true)

```
Theorem beq_nat_true : ∀ n m,
    beq_nat n m = true → n = m.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

MAYBE LATER!!!!!!!!
REALLY DO THIS!!!!!!!

Exercise: 2 stars, advanced (beq_nat_true_informal)

Give a careful informal proof of beq_nat_true, being as explicit as possible about quantifiers.

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

☐

The strategy of doing fewer intros before an induction to obtain a more general IH doesn't always work by itself; sometimes some *rearrangement* of quantified variables is needed. Suppose, for example, that we wanted to prove double_injective by induction on m instead of n.

```
Theorem double_injective_take2_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction m as [| m'].
  - (* m = O *) simpl. intros eq. destruct n as [| n'].
    + (* n = O *) reflexivity.
    + (* n = S n' *) inversion eq.
  - (* m = S m' *) intros eq. destruct n as [| n'].
    + (* n = O *) inversion eq.
    + (* n = S n' *) apply f_equal.
        (* Stuck again here, just like before. *)
Abort.
```

The problem is that, to do induction on m, we must first introduce n. (If we simply say induction m without introducing anything first, Coq will automatically introduce n for us!)

What can we do about this? One possibility is to rewrite the statement of the lemma so that m is quantified before n. This works, but it's not nice: We don't want to have to twist the statements of lemmas to fit the needs of a particular strategy for proving them! Rather we want to state them in the clearest and most natural way.

What we can do instead is to first introduce all the quantified variables and then *re-generalize* one or more of them, selectively taking variables out of the context and putting them back at the beginning of the goal. The `generalize dependent` tactic does this.

```
Theorem double_injective_take2 : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m.
  (* n and m are both in the context *)
  generalize dependent n.
  (* Now n is back in the goal and we can do induction on
     m and get a sufficiently general IH. *)
  induction m as [| m'].
  - (* m = O *) simpl. intros n eq. destruct n as [| n'].
    + (* n = O *) reflexivity.
    + (* n = S n' *) inversion eq.
  - (* m = S m' *) intros n eq. destruct n as [| n'].
    + (* n = O *) inversion eq.
    + (* n = S n' *) apply f_equal.
      apply IHm'. inversion eq. reflexivity. Qed.
```

why would I need to start by induction on m?

why do I have intro n and then generalize it?
exercise gen_dep_practice illustrates this why

Let's look at an informal proof of this theorem. Note that the proposition we prove by induction leaves `n` quantified, corresponding to the use of generalize dependent in our formal proof.

Theorem: For any nats `n` and `m`, if `double n` = `double m`, then `n` = `m`.

Proof: Let `m` be a `nat`. We prove by induction on `m` that, for any `n`, if `double n` = `double m` then `n` = `m`.

- First, suppose `m` = `0`, and suppose `n` is a number such that `double n` = `double m`. We must show that `n` = `0`.

  Since `m` = `0`, by the definition of `double` we have `double n` = `0`. There are two cases to consider for `n`. If `n` = `0` we are done, since `m` = `0` = `n`, as required. Otherwise, if `n` = `S n'` for some `n'`, we derive a contradiction: by the definition of `double`, we can calculate `double n` = `S (S (double n'))`, but this contradicts the assumption that `double n` = `0`.

- Second, suppose `m` = `S m'` and that `n` is again a number such that `double n` = `double m`. We must show that `n` = `S m'`, with the induction hypothesis that for every number `s`, if `double s` = `double m'` then `s` = `m'`.

  By the fact that `m` = `S m'` and the definition of `double`, we have `double n` = `S (S (double m'))`. There are two cases to consider for `n`.

  If `n` = `0`, then by definition `double n` = `0`, a contradiction.

  Thus, we may assume that `n` = `S n'` for some `n'`, and again by the definition of `double` we have `S (S (double n'))` = `S (S (double m'))`, which implies by inversion that `double n'` = `double m'`. Instantiating the induction hypothesis with `n'` thus allows us to conclude that `n'` = `m'`, and it follows immediately that `S n'` = `S m'`. Since `S n'` = `n` and `S m'` = `m`, this is just what we wanted to show. ☐

Before we close this section and move on to some exercises, let's digress briefly and use `beq_nat_true` to prove a similar property of identifiers that we'll need in later chapters:

```
Theorem beq_id_true : ∀ x y,
  beq_id x y = true → x = y.
Proof.
  intros [m] [n]. simpl. intros H.
  assert (H' : m = n). { apply beq_nat_true. apply H. }
  rewrite H'. reflexivity.
Qed.
```

Exercise: 3 stars, recommended (gen_dep_practice)

Prove this by induction on `l`.

```
Theorem nth_error_after_last: ∀ (n : nat) (X : Type) (l : list
X),
     length l = n →
     nth_error l n = None.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Unfolding Definitions

It sometimes happens that we need to manually unfold a Definition so that we can manipulate its right-hand side. For example, if we define...

```
Definition square n := n * n.
```

... and try to prove a simple fact about `square`...

```
Lemma square_mult : ∀ n m, square (n * m) = square n * square m.
Proof.
  intros n m.
  simpl.
```

... we get stuck: `simpl` doesn't simplify anything at this point, and since we haven't proved any other facts about `square`, there is nothing we can `apply` or `rewrite` with.

To make progress, we can manually `unfold` the definition of `square`:

```
  unfold square.
```

Now we have plenty to work with: both sides of the equality are expressions involving multiplication, and we have lots of facts about multiplication at our disposal. In particular, we know that it is commutative and associative, and from these facts it is not hard to finish the proof.

```
  rewrite mult_assoc.
  assert (H : n * m * n = n * n * m).
```

where did the direction go?
when you don't have direction
it's ->

```
    { rewrite mult_comm. apply mult_assoc. }
    rewrite H. rewrite mult_assoc. reflexivity.
  Qed.
```

At this point, a deeper discussion of unfolding and simplification is in order.

You may already have observed that tactics like `simpl`, `reflexivity`, and `apply` will often unfold the definitions of functions automatically when this allows them to make progress. For example, if we define `foo m` to be the constant 5...

```
  Definition foo (x: nat) := 5.
```

then the `simpl` in the following proof (or the `reflexivity`, if we omit the `simpl`) will unfold `foo m` to `(fun x ⇒ 5) m` and then further simplify this expression to just 5.

```
  Fact silly_fact_1 : ∀ m, foo m + 1 = foo (m + 1) + 1.
  Proof.
    intros m.
    simpl.
    reflexivity.
  Qed.
```

However, this automatic unfolding is rather conservative. For example, if we define a slightly more complicated function involving a pattern match...

```
  Definition bar x :=
    match x with
    | O ⇒ 5
    | S _ ⇒ 5
    end.
```

...then the analogous proof will get stuck:

```
  Fact silly_fact_2_FAILED : ∀ m, bar m + 1 = bar (m + 1) + 1.
  Proof.
    intros m.
    simpl. (* Does nothing! *)
  Abort.
```

The reason that `simpl` doesn't make progress here is that it notices that, after tentatively unfolding `bar m`, it is left with a match whose scrutinee, `m`, is a variable, so the `match` cannot be simplified further. (It is not smart enough to notice that the two branches of the `match` are identical.) So it gives up on unfolding `bar m` and leaves it alone. Similarly, tentatively unfolding `bar (m+1)` leaves a `match` whose scrutinee is a function application (that, itself, cannot be simplified, even after unfolding the definition of +), so `simpl` leaves it alone.

At this point, there are two ways to make progress. One is to use `destruct m` to break the proof into two cases, each focusing on a more concrete choice of `m` (`O` vs `S _`). In each case, the `match` inside of `bar` can now make progress, and the proof is easy to complete.

```
  Fact silly_fact_2 : ∀ m, bar m + 1 = bar (m + 1) + 1.
  Proof.
```

```
    intros m.
    destruct m.
    - simpl. reflexivity.
    - simpl. reflexivity.
  Qed.
```

This approach works, but it depends on our recognizing that the `match` hidden inside `bar` is what was preventing us from making progress.

A more straightforward way to make progress is to explicitly tell Coq to unfold `bar`.

```
  Fact silly_fact_2' : ∀ m, bar m + 1 = bar (m + 1) + 1.
  Proof.
    intros m.
    unfold bar.
```

Now it is apparent that we are stuck on the `match` expressions on both sides of the =, and we can use `destruct` to finish the proof without thinking too hard.

```
    destruct m.
    - reflexivity.
    - reflexivity.
  Qed.
```

Using `destruct` on Compound Expressions

We have seen many examples where `destruct` is used to perform case analysis of the value of some variable. But sometimes we need to reason by cases on the result of some *expression*. We can also do this with `destruct`.

Here are some examples:

```
  Definition sillyfun (n : nat) : bool :=
    if beq_nat n 3 then false
    else if beq_nat n 5 then false
    else false.

  Theorem sillyfun_false : ∀ (n : nat),
    sillyfun n = false.
  Proof.
    intros n. unfold sillyfun.
    destruct (beq_nat n 3).
      - (* beq_nat n 3 = true *) reflexivity.
      - (* beq_nat n 3 = false *) destruct (beq_nat n 5).
        + (* beq_nat n 5 = true *) reflexivity.
        + (* beq_nat n 5 = false *) reflexivity. Qed.
```

After unfolding `sillyfun` in the above proof, we find that we are stuck on `if (beq_nat n 3) then ... else ...`. But either `n` is equal to `3` or it isn't, so we can use `destruct (beq_nat n 3)` to let us reason about the two cases.

In general, the `destruct` tactic can be used to perform case analysis of the results of arbitrary computations. If `e` is an expression whose type is some inductively defined type `T`, then, for each constructor `c` of `T`, `destruct e` generates a subgoal in which all occurrences of `e` (in the goal and in the context) are replaced by `c`.

Exercise: 3 stars, optional (combine_split)

Here is an implementation of the `split` function mentioned in chapter Poly:

```
Fixpoint split {X Y : Type} (l : list (X*Y))
               : (list X) * (list Y) :=
  match l with
  | [] ⇒ ([], [])
  | (x, y) :: t ⇒
      match split t with
      | (lx, ly) ⇒ (x :: lx, y :: ly)
      end
  end.
```

Prove that `split` and `combine` are inverses in the following sense:

```
Theorem combine_split : ∀ X Y (l : list (X * Y)) l₁ l₂,
  split l = (l₁, l₂) →
  combine l₁ l₂ = l.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

However, `destruct`ing compound expressions requires a bit of care, as such `destruct`s can sometimes erase information we need to complete a proof. For example, suppose we define a function `sillyfun1` like this:

```
Definition sillyfun1 (n : nat) : bool :=
  if beq_nat n 3 then true
  else if beq_nat n 5 then true
  else false.
```

Now suppose that we want to convince Coq of the (rather obvious) fact that `sillyfun1 n` yields `true` only when `n` is odd. By analogy with the proofs we did with `sillyfun` above, it is natural to start the proof like this:

```
Theorem sillyfun1_odd_FAILED : ∀ (n : nat),
     sillyfun1 n = true →
     oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (beq_nat n 3).
  (* stuck... *)
Abort.
```

We get stuck at this point because the context does not contain enough information to prove the goal! The problem is that the substitution performed by `destruct` is too brutal — it threw away every occurrence of `beq_nat n 3`, but we need to keep some memory of this expression and how it was destructed, because we need to be able to reason that, since `beq_nat n 3 = true` in this branch of the case analysis, it must be that `n = 3`, from which it follows that `n` is odd.

What we would really like is to substitute away all existing occurences of `beq_nat n 3`, but at the same time add an equation to the context that records which case we are in. The `eqn:` qualifier allows us to introduce such an equation, giving it a name that we choose.

```
Theorem sillyfun1_odd : ∀ (n : nat),
     sillyfun1 n = true →
     oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (beq_nat n 3) eqn:Heqe3.
  (* Now we have the same state as at the point where we got
     stuck above, except that the context contains an extra
     equality assumption, which is exactly what we need to
     make progress. *)
    - (* e3 = true *) apply beq_nat_true in Heqe3.
      rewrite → Heqe3. reflexivity.
    - (* e3 = false *)
     (* When we come to the second equality test in the body
        of the function we are reasoning about, we can use
        eqn: again in the same way, allow us to finish the
        proof. *)
      destruct (beq_nat n 5) eqn:Heqe5.
        + (* e5 = true *)
          apply beq_nat_true in Heqe5.
          rewrite → Heqe5. reflexivity.
        + (* e5 = false *) inversion eq. Qed.
```

Exercise: 2 stars (destruct_eqn_practice)

```
Theorem bool_fn_applied_thrice :
  ∀ (f : bool → bool) (b : bool),
  f (f (f b)) = f b.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Review

We've now seen many of Coq's most fundamental tactics. We'll introduce a few more in the coming chapters, and later on we'll see some more powerful *automation* tactics that make Coq help us with low-level details. But basically we've got what we need to get work done.

Here are the ones we've seen:

- `intros`: move hypotheses/variables from goal to context

- `reflexivity`: finish the proof (when the goal looks like `e = e`)
- `apply`: prove goal using a hypothesis, lemma, or constructor
- `apply... in H`: apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning)
- where did we see this? `apply... with...`: explicitly specify values for variables that cannot be determined by pattern matching
- `simpl`: simplify computations in the goal
- `simpl in H`: ... or a hypothesis
- `rewrite`: use an equality hypothesis (or lemma) to rewrite the goal
- `rewrite ... in H`: ... or a hypothesis
- `symmetry`: changes a goal of the form `t=u` into `u=t`
- `symmetry in H`: changes a hypothesis of the form `t=u` into `u=t`
- `unfold`: replace a defined constant by its right-hand side in the goal
- `unfold... in H`: ... or a hypothesis
- `destruct... as...`: case analysis on values of inductively defined types
- `destruct... eqn:...`: specify the name of an equation to be added to the context, recording the result of the case analysis
- `induction... as...`: induction on values of inductively defined types
- `inversion`: reason by injectivity and distinctness of constructors
- still have problem using assert :(`assert (H: e)` (or `assert (e) as H`): introduce a "local lemma" `e` and call it `H`
- `generalize dependent x`: move the variable `x` (and anything else that depends on it) from the context back to an explicit hypothesis in the goal formula

Additional Exercises

didn't even read them!!!

Exercise: 3 stars (beq_nat_sym)

```
Theorem beq_nat_sym : ∀ (n m : nat),
  beq_nat n m = beq_nat m n.
Proof.
  (* FILL IN HERE *) Admitted.
```
☐

Exercise: 3 stars, advanced, optional (beq_nat_sym_informal)

Give an informal proof of this lemma that corresponds to your formal proof above:

Theorem: For any nats n m, beq_nat n m = beq_nat m n.

Proof: (* FILL IN HERE *)

□

Exercise: 3 stars, optional (beq_nat_trans)

```
Theorem beq_nat_trans : ∀ n m p,
  beq_nat n m = true →
  beq_nat m p = true →
  beq_nat n p = true.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, advanced (split_combine)

We proved, in an exercise above, that for all lists of pairs, `combine` is the inverse of `split`. How would you formalize the statement that `split` is the inverse of `combine`? When is this property true?

Complete the definition of `split_combine_statement` below with a property that states that `split` is the inverse of `combine`. Then, prove that the property holds. (Be sure to leave your induction hypothesis general by not doing `intros` on more things than necessary. Hint: what property do you need of l_1 and l_2 for `split combine` l_1 l_2 = (l_1,l_2) to be true?)

```
Definition split_combine_statement : Prop
  (* (": Prop" means that we are giving a name to a
     logical proposition here.) *)
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Theorem split_combine : split_combine_statement.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, advanced (filter_exercise)

This one is a bit challenging. Pay attention to the form of your induction hypothesis.

```
Theorem filter_exercise : ∀ (X : Type) (test : X → bool)
                               (x : X) (l lf : list X),
     filter test l = x :: lf →
     test x = true.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars, advanced, recommended (forall_exists_challenge)

Define two recursive `Fixpoints`, `forallb` and `existsb`. The first checks whether every element in a list satisfies a given predicate:

```
        forallb oddb [1;3;5;7;9] = true

        forallb negb [false;false] = true

        forallb evenb [0;2;4;5] = false

        forallb (beq_nat 5) [] = true
```

The second checks whether there exists an element in the list that satisfies a given predicate:

```
        existsb (beq_nat 5) [0;2;3;6] = false

        existsb (andb true) [true;true;false] = true

        existsb oddb [1;0;0;0;0;3] = true

        existsb evenb [] = false
```

Next, define a *nonrecursive* version of `existsb` — call it `existsb'` — using `forallb` and `negb`.

Finally, prove a theorem `existsb_existsb'` stating that `existsb'` and `existsb` have the same behavior.

```
  (* FILL IN HERE *)
```
□