TACTICS

MORE BASIC TACTICS

The `apply` Tactic

We can use `apply` when some hypothesis or an earlier lemma exactly matches the goal:

```
Theorem silly1 : ∀ (n m : nat),
  n = m →
  n = m.
Proof.
  intros n m eq.
```

Here, we could finish with "`rewrite → eq. reflexivity.`" as we have done several times before. Alternatively, we can finish in a single step by using the `apply` tactic:

```
  apply eq. Qed.
```

`apply` also works with *conditional* hypotheses:

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

Observe how Coq picks appropriate values for the ∀-quantified variables of the hypothesis:

```
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.
```

The goal must match the hypothesis *exactly* for `apply` to work:

```
Theorem silly3 : ∀ (n m : nat),
  n = m →
  m = n.
Proof.
  intros n m H.
```

Here we cannot use `apply` directly...

```
  Fail apply H.
```

but we can use the `symmetry` tactic, which switches the left and right sides of an equality in the goal.

```
  symmetry. apply H. Qed.
```

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.
```

Since this is a common pattern, we might like to pull it out as a lemma that records, 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.
```

Applying this lemma to the example above requires a slight refinement of `apply`:

```
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.
```

Doing `apply trans_eq` doesn't work! But...

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

does.

```
  apply eq1. apply eq2. Qed.
```

`transitivity` is also a 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.
  transitivity [c;d].
  apply eq1. apply eq2. Qed.
```

The `injection` and `discriminate` Tactics

The constructors of inductive types are *injective* (or *one-to-one*) and *disjoint*.

E.g., for `nat`...

- if `S n = S m` then it must be that `n = m`
- `O` is not equal to `S n` for any `n`

We can *prove* the injectivity of `S` by using the `pred` function defined in `Basics.v`.

```
Theorem S_injective : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H1.
  assert (H2: n = pred (S n)). { reflexivity. }
  rewrite H2. rewrite H1. simpl. reflexivity.
Qed.
```

As a convenience, the `injection` tactic allows us to exploit injectivity of any constructor (not just `S`).

```
Theorem S_injective' : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H.
  injection H as Hnm. apply Hnm.
Qed.
```

Here's a more interesting example that shows how `injection` can derive multiple equations at once.

```
Theorem injection_ex1 : ∀ (n m o : nat),
  [n;m] = [o;o] →
  n = m.
Proof.
  intros n m o H.
  (* WORK IN CLASS *) Admitted.
```

So much for injectivity of constructors. What about disjointness?

Two terms beginning with different constructors (like `O` and `S`, or `true` and `false`) can never be equal!

The `discriminate` tactic embodies this principle: It is used on a hypothesis involving an equality between different constructors (e.g., `false = true`), and it solves the current goal immediately. Some examples:

```
Theorem discriminate_ex1 : ∀ (n m : nat),
  false = true →
  n = m.
Proof.
  intros n m contra. discriminate contra. Qed.

Theorem discriminate_ex2 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. discriminate contra. Qed.
```

These examples are instances of a logical principle known as the *principle of explosion*, which asserts that a contradictory hypothesis entails anything (even manifestly false things!).

For a slightly more involved example, we can use `discriminate` to make a connection between the two different notions of equality (`=` and `=?`) on natural numbers.

```
Theorem eqb_0_l : ∀ n,
   0 =? n = true → n = 0.
Proof.
  intros n.

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

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

    intros H. discriminate H.
Qed.
```

Suppose Coq's proof state looks like

```
x : bool
y : bool
H : negb x = negb y
============================
 y = x
```

and we apply the tactic `injection H as Hxy`. What will happen?

(1) "No more subgoals."

(2) The tactic fails.

(3) Hypothesis `H` becomes `Hxy : x = y`.

(4) None of the above.

Now suppose Coq's proof state looks like

```
x : nat
y : nat
H : x + 1 = y + 1
============================
 y = x
```

and we apply the tactic `injection H as Hxy`. What will happen?

(1) "No more subgoals."

(2) The tactic fails.

(3) Hypothesis `H` becomes `Hxy : x = y`.

(4) None of the above.

Finally, suppose Coq's proof state looks like

```
x : nat
y : nat
H : 1 + x = 1 + y
============================
 y = x
```

and we apply the tactic `injection H as Hxy`. What will happen?

(1) "No more subgoals."

(2) The tactic fails.

(3) Hypothesis `H` becomes `Hxy : x = y`.

(4) None of the above.

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.

Theorem eq_implies_succ_equal : ∀ (n m : nat),
  n = m → S n = S m.
Proof. intros n m H. apply f_equal. apply H. Qed.
```

Or we can just use the `f_equal` tactic.

```
Theorem eq_implies_succ_equal' : ∀ (n m : nat),
  n = m → S n = S m.
Proof. intros n m H. f_equal. apply H. Qed.
```

Using Tactics on Hypotheses

Many tactics come with "`... in ...`" variants that work on hypotheses instead of goals.

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

The ordinary `apply` tactic is a form of "backward reasoning": it says "We're trying to prove `X` and we know `Y → X`, so if we can prove `Y` we'll be done."

By contrast, the variant `apply... in...` is "forward reasoning": it says "We know `Y` and we know `Y → X`, so we also know `X`."

```
Theorem silly4 : ∀ (n m p q : nat),
  (n = m → p = q) →
  m = n →
  q = p.
Proof.
  intros n m p q EQ H.
  symmetry in H. apply EQ in H. symmetry in H.
  apply H. Qed.
```

Varying the Induction Hypothesis

Recall this function for doubling a natural number (from the Induction chapter):

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

Suppose we want to show that `double` is injective -- i.e., that it maps different arguments to different results. The way we *start* this proof is a little bit delicate:

```
Theorem double_injective_FAILED : ∀ n m,
  double n = double m →
  n = m.
Proof.
  intros n m. induction n as [| n' IHn'].
  - (* n = O *) simpl. intros eq. destruct m as [| m'] eqn:E.
    + (* m = O *) reflexivity.
    + (* m = S m' *) discriminate eq.
  - (* n = S n' *) intros eq. destruct m as [| m'] eqn:E.
    + (* m = O *) discriminate 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?

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 statement involving *every* `n` but just a *single* `m`.

A successful proof of `double_injective` leaves `m` universally quantified 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' IHn'].
  - (* n = O *) simpl. intros m eq. destruct m as [| m'] eqn:E.
    + (* m = O *) reflexivity.
    + (* m = S m' *) discriminate eq.

  - (* n = S n' *)
    intros m eq.
    destruct m as [| m'] eqn:E.
    + (* m = O *)
    discriminate eq.
    + (* m = S m' *)
      apply f_equal.
      apply IHn'. simpl in eq. injection eq as goal. apply goal. Qed.
```

The thing to take away from all this is that you need to be careful, when using induction, that you are not trying to prove something too specific: When proving a property involving two variables `n` and `m` by induction on `n`, it is sometimes crucial to leave `m` generic.

The following theorem illustrates the same point:

Exercise: 2 stars, standard (eqb_true)

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

☐

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' IHm'].
  - (* m = O *) simpl. intros eq. destruct n as [| n'] eqn:E.
    + (* n = O *) reflexivity.
    + (* n = S n' *) discriminate eq.
  - (* m = S m' *) intros eq. destruct n as [| n'] eqn:E.
    + (* n = O *) discriminate eq.
    + (* n = S n' *) apply f_equal.
        (* We are stuck here, just like before. *)
Abort.
```

The problem is that, to do induction on `m`, we must first introduce `n`. (And 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' IHm'].
  - (* m = O *) simpl. intros n eq. destruct n as [| n'] eqn:E.
    + (* n = O *) reflexivity.
    + (* n = S n' *) discriminate eq.
  - (* m = S m' *) intros n eq. destruct n as [| n'] eqn:E.
    + (* n = O *) discriminate eq.
    + (* n = S n' *) apply f_equal.
      apply IHm'. injection eq as goal. apply goal. Qed.
```

Unfolding Definitions

It sometimes happens that we need to manually unfold a name that has been introduced by a `Definition` so that we can manipulate the expression it denotes. 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 appear to be stuck: `simpl` doesn't simplify anything, 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 it is not hard to finish the proof.

```
  rewrite mult_assoc.
  assert (H : n × m × n = n × n × m).
    { rewrite mul_comm. apply mult_assoc. }
  rewrite H. rewrite mult_assoc. reflexivity.
Qed.
```

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

We 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.
```

But this automatic unfolding is somewhat 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 cannot itself 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 eqn:E.
  - 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 forward 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 eqn:E.
  - reflexivity.
  - reflexivity.
Qed.
```

Using `destruct` on Compound Expressions

The `destruct` tactic can be used on expressions as well as variables:

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

Theorem sillyfun_false : ∀ (n : nat),
  sillyfun n = false.
Proof.
  intros n. unfold sillyfun.
  destruct (n =? 3) eqn:E1.
    - (* n =? 3 = true *) reflexivity.
    - (* n =? 3 = false *) destruct (n =? 5) eqn:E2.
      + (* n =? 5 = true *) reflexivity.
      + (* n =? 5 = false *) reflexivity. Qed.
```

The `eqn:` part of the `destruct` tactic is optional; although we've chosen to include it most of the time, for the sake of documentation, it can often be omitted without harm.

However, when `destruct`ing compound expressions, the information recorded by the `eqn:` can actually be critical: if we leave it out, then `destruct` can erase information we need to complete a proof.

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

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

Adding the `eqn:` qualifier saves this information so we can use it.

```
Theorem sillyfun1_odd : ∀ (n : nat),
  sillyfun1 n = true →
  odd n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (n =? 3) eqn:Heqe3.
    - (* e3 = true *) apply eqb_true in Heqe3.
      rewrite → Heqe3. reflexivity.
    - (* e3 = false *)
      destruct (n =? 5) eqn:Heqe5.
        + (* e5 = true *)
          apply eqb_true in Heqe5.
          rewrite → Heqe5. reflexivity.
        + (* e5 = false *) discriminate eq. Qed.
```

Micro Sermon

Mindless proof-hacking is a terrible temptation...

Try to resist!