Coq Tactics Index

Stage 1: Proving Easy Goals

reflexivity
assumption
discriminate
constructor

Stage 2: Transforming Your Goal

apply
subst
rewrite
simpl

cut

Stage 3: Breaking Apart Your Goal

destruct
inversion
induction

Stage 4: Powerful Automatic Tactics

auto
intuition
omega

---

reflexivity

Use `reflexivity` when your goal is to prove that something equals itself.

In this example we will prove that any term `x` of type `Set` is equal to itself. After we intro the variable we can prove the goal using `reflexivity`.

```
Lemma everything_is_itself:
  forall x: Set, x = x.
Proof.
  intro.
  reflexivity.
Qed.
```

```
1 subgoal
x : Set
-----------(1/1)
x = x
```

Use it when: your goal is something like `a = a`.

Advanced usage: `reflexivity` will work even if your goal is not syntactically identical on the left and right side of the equality. Both sides just have to *evaluate* to the same term.

In this example we will apply `reflexivity` to a more complicated math equation: (3 + (0 + 2)) = (1 + 4).

```
Inductive nat : Set :=
  | O
  | S : nat -> nat.

Fixpoint add (a: nat) (b: nat) : nat :=
  match a with
    | O => b
    | S x => S (add x b)
  end.

Lemma complex_math:
    (add
        (S (S (S O)))
        (add O (S (S O)))) =
    add (S O) (S (S (S (S O)))).
Proof.
    reflexivity.
```

```
Qed.

No more subgoals.
```

assumption

If the thing you are trying to prove is already in your context, use `assumption` to finish the proof.

In this example we show that if we assume `p` we can prove `p`. We use `assumption` to tell Coq that our goal is already true in our context because we assumed it!

```
Lemma everything_implies_itself:
  forall p: Prop, p -> p.
Proof.
  intros.
  assumption.
Qed.
```

```
1 subgoal
p : Prop
H : p
-----------(1/1)
p
```

Use it when: your goal is already in your "context" of terms you already know.

discriminate

If you have an equality in your context that isn't true, you can prove anything using `discriminate`.

For `discriminate` to work, the terms must be "structurally" different. This means that both terms are elements of an inductive set but they are built differently, using different constructors (e.g. `true` and `false`, or `(S O)` and `(S (S O))`).

In this example we show that if we assume `true = false` then we can prove anything. Note that we don't specify what `a` is, it really can be anything!

```
Inductive bool: Set :=
  | true
  | false.

Lemma incorrect_equality_implies_anything:
  forall a, false = true -> a.
Proof.
  intros.
  discriminate.
Qed.
```

```
1 subgoal
a : Type
H : false = true
-----------(1/1)
a
```

constructor

When your goal is to show that you can build up a term that has some type and you have a constructor to do just that, use `constructor`!

In this example we will prove that two is even. First we say what it means for a number to be even. We define zero to be even, and the proof of that is the term `even_O`. The next line says that if we can prove that `n` is even than we can also prove that `(S (S n))` (or `n + 2`) is even.

To prove our lemma, we first call `constructor`. Coq sees that our goal matches the rightmost side of a constructor (namely `even_S`). Thus it transforms our goal into the left side of that constructor, so instead of proving that `(S (S O))` is even now we only need to prove that `O` is even. We use `constructor` again and this time Coq sees that our goal matches the right side of a different constructor, `even_O`. This constructor has no preconditions (since zero is defined to be even, gotta start somewhere) so we are done!

```
Inductive even : nat -> Prop:=
 | even_O: even O
 | even_S: forall n, even n -> even (S(S n)).

Lemma two_is_even:
  even (S (S O)).
Proof.
  constructor.
  constructor.
Qed.
```

```
1 subgoal
-----------(1/1)
even O
```

Use it when: your goal matches the right side of a constructor for some type.

apply

If we have a hypothesis that says that `x` implies `y`, we know that to prove `y` all we really have to do is prove `x`. We can `apply` that hypothesis to a goal of `y` to transform it into `x`.

In this example we prove modus ponens. We know that `(p -> q)` and we want to prove `q` so we can use `apply` the hypothesis to transform the goal from `q` into `p`. Then we see that `p` is already an assumption so we are done!

```
Lemma modus_ponens:
  forall p q : Prop, (p -> q) -> p -> q.
Proof.
  intros.
  apply H.
  assumption.
Qed.
```

```
1 subgoal
p : Prop
q : Prop
H : p -> q
H0 : p
-----------(1/1)
q
```

Use it when: you have a hypothesis where the conclusion (on the right of the arrow) is the same as your goal.

Advanced usage: If we know that `x` implies `y` and we know that `x` is true, we can transform `x` into `y` in our context using `apply`.

In this example we prove modus ponens again. We still have our hypothesis,

`H: p -> q`
This time we `apply` it to a different hypothesis,
`H0: p`
to turn that hypothesis into `q`.

```
Lemma modus_ponens_again:
  forall p q : Prop, (p -> q) -> p -> q.
Proof.
  intros.
  apply H in H0.
  assumption.
Qed.
```

```
1 subgoal
p : Prop
q : Prop
H : p -> q
H0 : p
-----------(1/1)
q
```

subst

If you know that an identifier (name for something) is equal to something else, you can use `subst` to substitute the identifier for the other thing.

In this example we know that `a = b` and we want to show `b = a`. We can use `subst` to transform the `a` in the goal into a `b`, so our goal becomes `b = b`. Then we can finish the proof using `reflexivity`.

```
Inductive bool: Set :=
  | true
  | false.

Lemma equality_commutes:
  forall (a: bool) (b: bool), a = b -> b = a.
Proof.
  intros.
  subst.
  reflexivity.
Qed.
```

```
1 subgoal
a : bool
b : bool
H : a = b
-----------(1/1)
b = a
```

Use it when: you want to transform an identifier into an equivalent term.

rewrite

If we know two terms are equal we can transform one term into the other using `rewrite`.

While `rewrite` is similar to `subst`, it also works when both sides of the equality are terms. An identity is just a name like `x`, while a term can be more complex, like a function application: `(f x)`.

In this example we prove that if we have a function `f` and `(f x) = (f y)` then `(f y) = (f x)`. We use `rewrite` to transform `(f x)` in our goal into `(f y)` and finish the proof using `reflexivity`.

```
Inductive bool: Set :=
  | true
  | false.

Lemma equality_of_functions_commutes:
  forall (f: bool->bool) x y,
    (f x) = (f y) -> (f y) = (f x).
Proof.
  intros.
  rewrite H.
  reflexivity.
Qed.
```

```
1 subgoal
f : bool -> bool
x : bool
y : bool
H : f x = f y
-----------(1/1)
f y = f x
```

Use it when: you know two terms are equivalent and you want to transform one into the other.

Advanced usage: you can also apply `rewrite` backwards, and to terms in your context.

Backwards
If we have the hypothesis
`H : f x = f y`
we can change our goal from `f y` into `f x` using `rewrite` backwards:
`rewrite <- H`

In context
We can use `rewrite H1 in H2` to transform one hypothesis using a different hypothesis.

In this example we prove that equality of function application is transitive. We can use either an in-context `rewrite` or a backward `rewrite` on the goal.

```
Inductive bool: Set :=
  | true
  | false.

Lemma equality_of_functions_transits:
  forall (f: bool->bool) x y z,
    (f x) = (f y) ->
    (f y) = (f z) ->
    (f x) = (f z).
Proof.
  intros.
  rewrite H0 in H. (* or rewrite <- H0 *)
  assumption.
Qed.
```

```
1 subgoal
1 subgoal
f : bool -> bool
x : bool
y : bool
z : bool
H : f x = f y
H0 : f y = f z
-----------(1/1)
f x = f z
```

simpl

When we have a complex term we can use `simpl` to crunch it down.

In this example we prove that adding zero to any number returns the same number. We use `simpl` to "run" the `add` function in the goal. Since in the example the first argument to `add` is `O`, it simplifies the function application to just the result.

```
Inductive nat : Set :=
  | O
  | S : nat -> nat.

Fixpoint add (a: nat) (b: nat) : nat :=
  match a with
    | O => b
    | S x => S (add x b)
  end.

Lemma zero_plus_n_equals_n:
  forall n, (add O n) = n.
Proof.
  intros.
  simpl.
  reflexivity.
Qed.
```

```
1 subgoal
n : nat
-----------(1/1)
add O n = n
```

cut

Sometimes to prove a goal you need an extra hypothesis. In this case, you can add the hypothesis using `cut`. This allows you to first prove your goal using the new hypothesis, and then prove that the new hypothesis is also true.

In this example we will prove that if `x = y` and `y = z` then `f x = f z`, for any function `f`. This is related to transitivity. To prove the goal, we first add the intermediate proposition that `x = z`. Then we have to prove that `x = z` implies `f x = f z`, and that `x` is actually equal to `z`.

```
Inductive bool: Set :=
  | true
  | false.

Lemma xyz:
  forall (f: bool->bool) x y z,
    x  = y -> y = z -> f x = f z.
Proof.
  intros.
  cut (x = z).
  - intro. subst. reflexivity.
  - subst. reflexivity.
Qed.
```

```
2 subgoals
f : bool -> bool
x : bool
y : bool
z : bool
```

```
H : x = y
H0 : y = z
---------(1/2)
x = z -> f x = f z
---------(2/2)
x = z
```

Use it when: you want to add an intermediate hypothesis to your proof that will make the proof easier.

destruct

We use `destruct` to perform case analysis on a term.

If we have a term of some type but we don't know what the term actually is, we can use `destruct` to examine all the possible options. It generates subgoals for each possible constructor that could have been used to construct the term. Then we prove the goal for each possibility.

In this example we show that if we negate a boolean twice, we get the same boolean back. We cannot prove this for a general `b` but we use `destruct` to prove it for any possible value of `b` (`true` or `false`).

```
Inductive bool: Set :=
  | true
  | false.

Definition not (b: bool) : bool :=
  match b with
    | true => false
    | false => true
  end.

Lemma not_not_x_equals_x:
  forall b, not (not b) = b.
Proof.
  intro.
  destruct b.
  - reflexivity.
  - reflexivity.
Qed.
```

```
1 subgoal
b : bool
-----------(1/1)
not (not b) = b
```

inversion

Sometimes you have a hypothesis that can't be true unless other things are also true. We can use `inversion` to discover other necessary conditions for a hypothesis to be true.

In this example we prove that if the successors of `a` and `b` are equal then `a` and `b` are also equal. We assume that `S a = S b`. However, this can only be true if `a = b` because of how we construct `nat`s. We use `inversion` to make Coq analyze the ways it can construct `a` and `b` and it realizes that they must be equal and adds it to the context.

```
Inductive nat : Set :=
  | O
  | S : nat -> nat.

Lemma successors_equal_implies_equal:
```

```
  forall a b, S a = S b -> a = b.
Proof.
  intros.
  inversion H.
  reflexivity.
Qed.
```

```
1 subgoal
a : nat
b : nat
H : S a = S b
-----------(1/1)
a = b
```

induction

If we want to prove a theorem using induction, we use `induction`!

When we use `induction`, Coq generates subgoals for every possible constructor of the term, similar to `destruct`. However, for inductive constructors (like `S x` for `nat`s), you also get an inductive hypothesis to help you prove your goal.

In this example we prove that adding any number to zero gives you the same number. We perform induction on `n` and get two cases.

If `n` is `O` then we know that `(add O O)` is `O` so we can use reflexivity. This is the base case.

For the inductive case we assume that the property holds for all numbers up to `n` and we have to prove it for `(S n)` (read: `n+1`).

To prove this we run the `add` function for one step using `simpl`. This brings the `S` outside the `add` function and now we can `rewrite` the goal using our inductive hypothesis. Then we use `reflexivity` to finish the proof. Good ol' `reflexivity`.

```
Inductive nat : Set :=
  | O
  | S : nat -> nat.

Fixpoint add (a: nat) (b: nat) : nat :=
  match a with
    | O => b
    | S x => S (add x b)
  end.

Lemma n_plus_zero_equals_n:
  forall n, (add n O) = n.
Proof.
  induction n.
- reflexivity.
- simpl. rewrite IHn. reflexivity.
Qed.
```

```
2 subgoals
-----------(1/2)
add O O = O
-----------(2/2)
add (S n) O = S n
```

auto

Sometimes a goal looks easy but you may be feeling lazy. Why not try `auto`?

`auto` will intro variables and hypotheses and then try applying various other tactics to solve the goal. Which other tactics does it try? Who knows man.

The good thing is that `auto` can't fail. At worst it will leave your goal unchanged. So go wild!

In this example we'll prove modus tollens using just `auto`!

```
Lemma modus_tollens:
forall p q: Prop, (p->q) -> ~q -> ~p.
Proof.
  auto.
Qed.

No more subgoals.
```

Use it when: you think the goal is easy but you're feeling lazy.

`intuition`

If you thought `auto` was good, `intuition` is even better!

The `intuition` tactic also `intros` variables and hypotheses and applies tactics to them, including `auto`. Sometimes it works when `auto` doesn't.

In this example we'll prove that if we know the conjunction of `p` and `q`, we also know `p` by itself. `auto` can't solve the goal by itself but `intuition` can.

```
Lemma conjunction_elimination:
forall p q, p /\ q -> p.
Proof.
  intuition.
Qed.

No more subgoals.
```

Use it when: `auto` doesn't work but you think it should be easy to prove.

`omega`

If you are trying to prove something "mathy" you should try the `omega` tactic. It's good at reasoning about goals involving nats and integers.

In this example we'll prove that an odd number can never equal an even number using `omega`.

```
Require Import ZArith.
(* or Require Import Omega. *)

Lemma odds_arent_even:
forall a b: nat, 2*a + 1 <> 2*b.
Proof.
  intros.
  omega.
Qed.

No more subgoals.
```