Home About Coq Get Coq Documentation Community Consort News
Fork me on GitHub

The Coq Proof Assistant

Home

A tutorial by Mike Nahas

Mike Nahas's Coq Tutorial

Started 2012-Nov-06 Version 1.2, 2019-Jan-22 Tested with Coq version 8.10+alpha

Dedicated to Kernighan and Ritchie, who wrote a magnificent introduction to a programming language.

Introduction

Coq is a proof assistant. Coq helps you write formal proofs.

A "formal proof" is a mathematical proof that is written in a language similar to a programming language. (Actually, Coq's language *is* a programming language, but we'll get to that later.) Formal proofs are harder for a human to read, but easier for a program to read. The advantage is that a formal proof can be verified by a program, eliminating human error.

(NOTE: Yes, a human error could exist in the verifying program, the operating system, the computer, etc.. The risk in the verifying program is kept small by keeping the program very simple and small. As for the other parts, checking a proof on multiple systems make it much more likely that any errors are in the proofs rather than in the verification of the proof.)

This tutorial will teach you the basics of using Coq to write formal proofs. My aim is to teach you a powerful subset of Coq's commands by showing you lots of actual proofs. This tutorial will not teach you all of the Coq commands, but you will learn enough to get started using Coq.

Prerequisites

This tutorial will teach you how to use Coq to write formal proofs. I assume you already know how to write a proof and know about formal logic. I tried to make this tutorial easy to read, so if you don't know those things, keep reading and you *might* be able to pick them up.

I also assume you understand a programming language. It doesn't matter which programming language.

If you feel unprepared, the "further reading" section at the end has useful links.

Installing Coq

The easiest thing is to install "CoqIDE", the graphical version of Coq. Windows and MacOS installers and the latest source code are available at http://coq.inria.fr/download If you're running Linux, your package manager probably has a recent-ish version. (Debian/Ubuntu/Mint: "sudo apt install coqide", Fedora/CentOS might be 'su -c "yum install coq-coqide"'. Arch might be "sudo pacman -S coqide")

If you like the Emacs text editor, the alternative is to run "coqtop", the command-line version of Coq. It is also available at http://coq.inria.fr/download The Linux package is usually called "coq". To use it in Emacs, you have to also download and install the "Proof General" Emacs mode. Is is available at https://proofgeneral.github.io/ Linux package managers include some version of it. (Debian/Ubuntu/Mint calls it "proofgeneral". Fedora/CentOS might call it "emacs-common-proofgeneral". Arch might call it "proofgeneral".)

Loading this file

This file (yes, the one you're reading right now) is a Coq file. Coq file's names usually end in ".v". If this file's name currently ends in ".txt", you'll want to change it to ".v" before loading it into CoqIDE or Emacs. (If this file's name ends in ".html" or ".pdf", someone used Coq to generate this document, in which case you need to find the original ".v" file they used. If you can't find it, the author hosts a recent version at https://mdnahas.github.io/doc/nahas_tutorial.v)

Recent news

- Coq 8.9.0 is out
- Coq 8.9+beta1 is out
- Coq 8.8.2 is out

more

Syndicate

Once you've made sure the file's name ends in ".v", start CoqIDE or Emacs and load the file.

You need to know that this file was designed to work in a specific version of Coq. Coq is a research tool and the developers occasionally make small changes to its file format, so it is possible that this file is for a different version of Coq. As you progress through the tutorial, you may find a proof that your version of Coq doesn't like. You can probably make it work, but if you can't, you can try installing the latest version of CoqIDE and downloading the latest version of the tutorial from the author's website. (https://mdnahas.github.io/doc/nahas_tutorial.v))

Comments

Coq will ignore everything that starts with a '(' followed by a '*' and end with a '*' followed by a ')'. These are called comments.

Your First Proof!

We'll begin by proving the proposition:

```
  "for all things you could prove,
   if you have a proof of it, then you have a proof of it."
```

Okay, that's not that exciting but we can't print "Hello, World" in Coq...

```
Theorem my_first_proof : (forall A : Prop, A -> A).
Proof.
  intros A.
  intros proof_of_A.
  exact proof_of_A.
Qed.
```

Dissection of your first proof

A Coq proof starts by stating what you're trying to prove. That is done by the command "Theorem". After the word "Theorem" comes the name of the theorem: "my_first_proof". If you want use this theorem later, you'll refer to it by that name. Then comes a colon, the statement of what you're trying to prove, and a period ('.').

As you can see, every Coq command ends with a period.

Let's skip over how we express what you want to prove and go on to the actual proof itself. That starts, unsurprisingly, with the command "Proof" (with its command-terminating period). Then comes the actual proof, which takes 3 steps. Lastly, the "Qed" command ends the proof.

(NOTE: Instead of "Theorem", you may also see proofs that start with "Lemma", "Remark", "Fact", "Corollary", and "Proposition", which all mean the *SAME* thing. In my proofs, I only use "Theorem". Instead of "Qed", you may also see proofs that end with "Admitted" or "Defined", but these mean *DIFFERENT* things. Use "Qed" for now.)

Coq uses 3 different "languages" and you can see them all here in this proof.

- The "vernacular" language manages definitions and top-level interactions. Each of its commands starts with a capital letter: "Theorem", "Proof", and "Qed".
- The "tactics" language is used to write proofs. Its commands start with a lower-case letter: "intros" and "exact".
- The unnamed language of Coq terms is used to express what you want to prove. Its expressions use lots of operators and parentheses: "(forall A : Prop, A -> A)". (Technically, this language is a subset of the vernacular language, but it is useful to think of it as its own thing.)

Now, let's take a look inside this proof! Since it probably has scrolled off the screen, I'll reproduce it.

```
Theorem my_first_proof__again : (forall A : Prop, A -> A).
Proof.
  intros A.
  intros proof_of_A.
  exact proof_of_A.
Qed.
```

Seeing where you are in a proof

CoqIDE and Proof General are valuable because they show you the state in the middle of your proof. They show what you have proven and what you still need to prove.

Let's see the different states inside your first proof. Move your cursor (by clicking your mouse or using the arrow keys) over any line between "Proof." and "Qed.". Now let's see the state at that point.

In CoqIDE, there are three ways to do it: 1. From the menu bar, open the "Navigation" menu and select "go to" 2. In the tool bar, click on the 5th icon (a green arrow pointing at a yellow ball) 3. Use a keyboard combo. On my Mac, it's control-option-rightarrow.

In Proof General: Do "C-c C-Enter" (press control-c and then control-Enter)

In a different part of the screen you should see something like:

```
A : Prop
proof_of_A : A
============================
 A
```

Everything above the bar is what you know to exist or have assumed to exist. These are called "hypotheses" and we refer to everything above the bar as "the context". Below the bar is what we are trying to prove. It is called "the current subgoal".

"The goal" is the theorem we're trying to prove. A "subgoal" is what we are trying to prove at any point during the proof. We say "current subgoal" because during a proof we may have multiple things remaining to prove. For example, during a proof by induction, we will have one subgoal for the "base case" and another subgoal for the "inductive case". But we (usually) only work on one subgoal at a time and the one we're working on is called "the current subgoal".

Now I want to explain each tactic and how it helps prove this theorem. The proof has probably scrolled off your screen again, so here it is a third time.

```
Theorem my_first_proof__again__again : (forall A : Prop, A -> A).
Proof.
  intros A.
  intros proof_of_A.
  exact proof_of_A.
Qed.
```

Your first Tactic!

Our starting state is:

```

============================
 forall A : Prop, A -> A
```

Our goal (and current subgoal) starts with "forall A : Prop,...". In English, this would be "For all A such that A is a Prop, ...". One way to prove a statement like "For all x such that x is an integer, ..." is to assume we have an arbitrary integer x and show that the rest of the statement holds for x. The first tactic "intros" does exactly that.

Thus, every time I see "intros A.", I think "assume A".

The tactic "intros" takes "forall" off the front of the subgoal, changes its variable into a hypothesis in the context, and names the hypothesis. Recall, the context holds all things we've proven or, as in this case, assumed to be proven. We named the new hypothesis "A", which is the same as the name of the variable removed from the subgoal. You should always keep the same name, if possible.

RULE: If the subgoal starts with "(forall <name> : <type>, ..." Then use tactic "intros <name>.".

The state after "intros A." is:

```
A : Prop
============================
 A -> A
```

In Coq, "A : Prop" means you have something named "A" of type "Prop". In the future you will see "0 : nat" which means "0" of type "nat" (natural numbers) and you will see "true :

bool" which means "true" of type "bool" (boolean or true-false values). In some places, you will see "A B C : Prop", which means "A", "B", and "C" all have type "Prop".

The type "Prop" is easier to explain after we've executed the next command. The next tactic is "intros", which we said works when "forall" is at the front of the subgoal. It works here because "->" is actually shorthand for "forall". "B -> C" really means "(forall something_of_type_B : B, C)". So, "A->A" is "(forall something_of_type_A : A, A)". The result is that "intros proof_of_A." removes that hidden "forall" and moves the unnamed variable of type A to the hypothesis named "proof_of_A".

RULE: If the subgoal starts with "<type> -> ..." Then use tactic "intros <name>.".

The state after the second "intros" command, looks like:

```
A : Prop
proof_of_A : A
============================
 A
```

Now, we can talk about "Prop". "proof_of_A" is a proof. It has type "A", which means that "A" is something that could have a proof. In logic, that is called a "proposition". Since the proposition "A" has type "Prop", "Prop" must be the type of propositions.

Prop is an important concept, so let's see some propositions:

- (forall x : nat, (x < 5) -> (x < 6))
- (forall x y : nat, x + y = y + x)
- (forall A : Prop, A -> A)

All of these have type "Prop". They can all can have proofs. That last Prop should look familiar - it's what we're trying to prove right now!

IT IS VITALLY IMPORTANT THAT YOU DO NOT THINK OF A Prop AS BEING EITHER "TRUE" OR "FALSE". A Prop either has a proof or it does not have a proof. Godel shattered mathematics by showing that some true propositions can never proven. Tarski went further and showed that some propositions cannot even be said to be true or false!!! Coq deals with these obstacles in modern mathematics by restricting Prop to being either proven or unproven, rather than true or false.

Now that I've got that across, let's go and finish our first proof. After the second "intros" tactic, our subgoal is "A", which means "we need something of type A" or, because A is a proposition, "we need a proof of A". Now, the previous command moved a proof of A into the context and called it "proof_of_A". So, an hypothesis in our context (which are all things we know to exist) has a type that matches our subgoal (which is what we want to create), so we have an exact match. The tactic "exact proof_of_A" solves the subgoal (and the proof).

Ta-da! Your first proof!

RULE: If the subgoal matches an hypothesis, Then use tactic "exact <hyp_name>.".

Okay, let's try something more complicated!

Proofs with ->

Proof going forward

```
Theorem forward_small : (forall A B : Prop, A -> (A->B) -> B).
Proof.
 intros A.
 intros B.
 intros proof_of_A.
 intros A_implies_B.
 pose (proof_of_B := A_implies_B proof_of_A).
 exact proof_of_B.
Qed.
```

Look at the tactics used in this proof. (The tactics are the commands between "Proof." and "Qed.") You should be familiar with "intros" and "exact". The new one is "pose". We're going to use "pose" in a forward proof. A "forward proof" creates larger and more complex hypotheses in the context until one matches the goal. Later, we'll see a "backward proof", which breaks the goal into smaller and simpler subgoals until they're trivial.

In our current proof, the state right before the "pose" command is:

```
A : Prop
B : Prop
proof_of_A : A
A_implies_B : A -> B
============================
 B
```

The subgoal is "B", so we're trying to construct a proof of B.

In the context we have "A_implies_B : A->B". "A->B", if you recall, is equivalent to "(forall proof_of_A : A, B)", that is, for every proof of A, we have a proof of B.

It happens that we have a proof of A in our context and it's called (unoriginally) "proof_of_A". The expression "A_implies_B proof_of_A" computes the proof of B that is associated with that particular proof of A.

The command "pose" assigns the result of "A_implies_B proof_of_A" to the new hypothesis "proof_of_B". (Note the annoying extra set of parentheses that are necessary around the arguments to the "pose" command.)

RULE: If you have an hypothesis "<hyp_name>: <type1> -> <type2> -> ... -> <result_type>" OR an hypothesis "<hyp_name>: (forall <obj1>:<type1>, (forall <obj2>: <type2>, ... <result_type> ...))" OR any combination of "->" and "forall", AND you have hypotheses of type "type1", "type2"..., Then use tactic "pose" to create something of type "result_type".

The proof ends with the "exact" command. We could have ended with "exact (A_implies_B proof_of_A)", but I think it is easier to read when it ends with with "exact proof_of_B."

This was a forward proof. Let's see a backward one.

Proof going backward

```
Theorem backward_small : (forall A B : Prop, A -> (A->B)->B).
Proof.
 intros A B.
 intros proof_of_A A_implies_B.
 refine (A_implies_B _).
   exact proof_of_A.
Qed.
```

Notice that we're trying to prove the exact same theorem as before. However, I'm going to show a "backward proof" that breaks the goal into smaller and simpler subgoals. We will start this proof trying to find a proof of B, but then change that to find the simpler proof of A.

First, notice that the 4 "intros" commands from our last proof have changed to just 2. The "intros" command can take any number of arguments, each argument stripping a forall (or ->) off the front of the current subgoal. (WARNING: Don't use "intros" with no arguments at all - it doesn't do what you'd expect!) We could have combined all the "intros" into a single command, but I think it is cleaner to introduce the Props and the derived values separately.

The state right before the "refine" command is:

```
A : Prop
B : Prop
proof_of_A : A
A_implies_B : A -> B
============================
 B
```

The subgoal is the Prop "B", so we're trying to construct a proof of B.

We know that "A_implies_B" can create a proof of B, given a proof of A. We saw the syntax for it was "A_implies_B something_of_type_A". The command "refine (A_implies_B _)." let's us create the proof of B without specifying the argument of type A. (The parentheses are necessary for it to parse correctly.) This solves our current subgoal of type B and the unspecified argument - represented by the underscore ("_") - become a new "child" subgoal.

```
A : Prop
B : Prop
proof_of_A : A
A_implies_B : A -> B
============================
 A
```

In this case, the child subgoal has us trying to find a proof of A. Since it's a child subgoal, we indent the tactics used to solve it. And the tactic to solve it is our well worn "exact".

RULE: If you have subgoal "<goal_type>" AND have hypothesis "<hyp_name>: <type1> -> <type2> -> ... -> <typeN> -> <goal_type>", Then use tactic "refine (<hyp_name> _ ...)."

with N underscores.

The important thing to take away from this proof is that we changed the subgoal. That's what happens in a backward proof. You keep changing the subgoal to make it smaller and simpler. "A" doesn't seem much smaller or simpler than "B", but it was.

Now, let's rev this up...

Proof going backward (large)

```
Theorem backward_large : (forall A B C : Prop, A -> (A->B) -> (B->C) -> C).
Proof.
 intros A B C.
 intros proof_of_A A_implies_B B_implies_C.
 refine (B_implies_C _).
   refine (A_implies_B _).
     exact proof_of_A.
Qed.
```

Look at the sequence of tactics. It starts with a pair of "intros"s. That's followed by the new kid on the block "refine". We finish with "exact". That pattern will start to look familiar very soon.

The state before the first "refine" is:

```
A : Prop
B : Prop
C : Prop
proof_of_A : A
A_implies_B : A -> B
B_implies_C : B -> C
============================
 C
```

Our current subgoal is "C" and "C" is at the end of "B -> C", so we can do "refine (B_implies_C _)". That command creates a new child subgoal "B".

Then, our subgoal is "B" and "B" is at the end of "A -> B", so "refine (A_implies_B _)" creates a new child subgoal of "A".

Then we finish with "exact proof_of_A.". Easy as pie.

Let's do a really big example!

Proof going backward (huge)

```
Theorem backward_huge : (forall A B C : Prop, A -> (A->B) -> (A->B->C) -> C).
Proof.
 intros A B C.
 intros proof_of_A A_implies_B A_imp_B_imp_C.
 refine (A_imp_B_imp_C _ _).
   exact proof_of_A.

   refine (A_implies_B _).
     exact proof_of_A.
Qed.
```

Yes, something is different here! We have our starting "intros"s. We have a "refine". But then the commands are indented and we have two "exact"s!

The state before the first refine is:

```
A : Prop
B : Prop
C : Prop
proof_of_A : A
A_implies_B : A -> B
A_imp_B_imp_C : A -> B -> C
============================
 C
```

Now, we have a subgoal "C" and "C" is at the end of "A->B->C", so we can do "refine (A_imp_B_imp_C _ _)". Notice that "A_imp_B_imp_C" has two implication arrows (the "->"s), so refine requires two underscores and creates two subgoals - one for something of type A and another for something of type B.

I hinted that a formal proof is actually written in a programming language. We see that very clearly here. "A_imp_B_imp_C" is a function that takes two arguments, one of type A and one of type B, and returns a value of type C. The type of the function is "A->B->C" and a call to it looks like "A_imp_B_imp_C something_of_type_A something_of_type_B". Notice that there are no parentheses needed - you just put the arguments next to the name of the function. For example, "function arg1 arg2". This is a style common in functional programming languages. For those familiar with imperative languages, like C, C++, and Java, it will feel odd not to have parentheses and commas to denote the arguments.

The first "refine" created two subgoals. CoqIDE and Proof General will tell you that two subgoals exist, but they will only show you the context for the current subgoal.

In the text of the proof, we represent that "refine" created multiple subgoals by formatting them like an "if-then-else" or "switch/match" statement in a programming language. The proofs for each subgoal get indented and we use a blank line to separate them.

The proof of the first subgoal is trivial. We need an "A" and we have "proof_of_A : A". "exact proof_of_A" completes the subgoal. We now put the blank line in the proof to show we're moving on to the next subgoal.

The rest of the proof we've seen before. The proof for it is indented to show it's the result of a "branch" in the proof created by "refine (A_imp_B_impC _)".

Having seen this complex theorem proved with a backward proof, let's see what it would look like with a forward one.

Proof going forward (huge)

```
Theorem forward_huge : (forall A B C : Prop, A -> (A->B) -> (A->B->C) -> C).
Proof.
 intros A B C.
 intros proof_of_A A_implies_B A_imp_B_imp_C.
 pose (proof_of_B := A_implies_B proof_of_A).
 pose (proof_of_C := A_imp_B_imp_C proof_of_A proof_of_B).
 exact proof_of_C.
Show Proof.
Qed.
```

This is the same theorem as before, except it has a forward proof instead of a backward one.

In this proof, we can see the programming language underneath the proof. "A_implies_B" is a function with type "A->B", so when it is call with argument "proof_of_A", it generates a proof of B. The "pose" command assigns the resulting value to the (unimaginative) name "proof_of_B".

Likewise, "A_imp_B_imp_C" is a function with type "A->B->C". Called with "proof_of_A" and "proof_of_B", it produces a proof of C which becomes the hypothesis "proof_of_C".

There is a new vernacular command at the end: "Show Proof". If you put your cursor right after it and press "control-option-right_arrow" (in CoqIDE) or "C-C C-Enter" (in Proof General), you'll see the actual code for the proof. It looks like this:

```
(fun (A B C : Prop)
     (proof_of_A : A) (A_implies_B : A -> B) (A_imp_B_imp_C : A -> B -> C)
 =>
   let proof_of_B := A_implies_B proof_of_A in
   let proof_of_C := A_imp_B_imp_C proof_of_A proof_of_B in
     proof_of_C)
```

I formatted this version so that the correspondence to the proof is clearer. The "intros" commands declare the parameters for the function. "pose" declares constant values in the function. Lastly, the "exact" command is used to return the result of the function. As we go, you'll see the tight relationship in Coq between proofs and code.

At this point, I want to emphasize that Coq proofs are not normally this tedious or verbose. The proofs I've shown and the proofs I will show are demonstrating the mechanics of how Coq works. I'm using simple familiar types to make the mechanics' operations clear to you. Coq's tactic language contains commands for automatic theorem proving and for defining macros. Almost all the proofs in this tutorial are simple enough to dispatch with a single Coq tactic. But when you prove more complicated statements, you'll need all the commands I'm teaching in this tutorial.

This helps explain why most Coq proofs are backwards. Once a goal has been transformed into simple enough subgoals, those subgoals can each be proved by automation.

So far, we've only worked with proofs, propositions, and Prop. Let's add some more types!

true and false vs. True and False

The vernacular command "Inductive" lets you create a new type. I wanted to introduce the boolean type, which has two values: "true" and "false". The problem is that in addition to "true" and "false", Coq has two other entities named "True" and "False", where the first letter is capitalized. To keep the differences straight in your mind, I'm going to introduce them to you all at once.

```
Inductive False : Prop := .

Inductive True : Prop :=
  | I : True.

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

- Capital-F "False" is a Prop with no proofs.
- Capital-T "True" is a Prop that has a single proof called "I". (That's a capital-I, not a number 1.)
- Lastly, "bool" is a Set and "bool" has two elements: lower-case-t "true" and lower-case-f "false".

I find these names confusing. Recall that "Prop"s are things that may have a proof. So I think capital-T "True" and capital-F "False" should have been named "Provable" and "Unprovable" (or "AlwaysProvable" and "NeverProvable"). The lower-case ones act like what you're accustomed to.

Since we've been playing with Props, let's do some proofs with (the badly named) "True" and "False" and we'll come back to the lower-case "true" and "false" later.

Capital-T True and Capital-F False

True is provable

```
Theorem True_can_be_proven : True.
  exact I.
Qed.
```

If we look at the state before the first (and only) line of the proof, we see:

```

  ============================
   True
```

There are no hypotheses; the context is empty. We are trying to find a proof of True. By its definition, True has a single proof called "I". So, "exact I." solves this proof.

RULE: If your subgoal is "True", Then use tactic "exact I.".

Now, we turn to False.

Unprovability

I wrote earlier that a proposition either has a proof or it does not (yet) have a proof. In some cases, we can prove that a proposition can never have a proof. We do that by showing that for every possible proof of a proposition, we could generate a proof of False. Because False has no proofs (by its definition), we know that the proposition has no proofs.

So, to show that "A : Prop" has no proofs, we need to show

- (forall proof_of_A: A, False)

Or, equivalently

- A -> False

We do this so often that there is an operator "~" to represent it.

```
Definition not (A:Prop) := A -> False.

Notation "~ x" := (not x) : type_scope.
```

"Definition" is a vernacular command that says two things are interchangeable. So, "(not A)" and "A -> False" are interchangeable.

"Notation" is a vernacular command that creates an operator, in this case "~", and defines it as an alternate notation for an expression, in this case "(not _)". Since "not" takes a Prop, "~" can only be applied to a Prop.

(NOTE: The Notation command is how the operator "->" was created for "(forall ...)".)

Let's try to prove some things are unprovable!

False is unprovable

```
Theorem False_cannot_be_proven : ~False.
Proof.
  unfold not.
  intros proof_of_False.
  exact proof_of_False.
Qed.
```

The only new tactic here is "unfold". We said "Definition" says two expressions are interchangeable. Well, "unfold" and "fold" will interchange them. After the "unfold", we have

```
  ============================
   False -> False
```

The "unfold" exposes that "~" is really an "->". And we're very familiar with using the tactic "intros" to remove "->" at the front of the subgoal. The command "intros proof_of_False" does just that.

After that, it goes as usual. "intros"s at the start and "exact" at the end. It feels weird to have an hypothesis labeled "proof_of_False" doesn't it? It's weird because we know False has no proofs, so that hypothesis can never exist. Wouldn't it be better if we could say that directly?...

```
Theorem False_cannot_be_proven__again : ~False.
Proof.
  intros proof_of_False.
  case proof_of_False.
Qed.
```

This is the same theorem as before, but two things have changed in this proof.

First, it doesn't have "unfold not.". Since we know "~" is shorthand for "->", we can skip over the unfold tactic and go straight to using "intros".

RULE: If your subgoal is "~<type>" or "~(<term>)" or "(not <term>)", Then use tactic "intros".

The second change is that we found a new way to finish a proof! Instead of "exact", we use the "case" tactic. "case" is powerful: it creates subgoals for every possible construction of its argument. Since there is no way to construct a False, "case" creates no subgoals! Without a subgoal, we're done!

RULE: If any hypothesis is "<name> : False", Then use tactic "case <name>.".

-> Examples

We can use True and False to see that Coq's "->" operator really does act like "implication" from logic.

```
Theorem thm_true_imp_true : True -> True.
Proof.
  intros proof_of_True.
  exact I.
```

"exact proof_of_True." also works.

```
Qed.
```

```
Theorem thm_false_imp_true : False -> True.
Proof.
  intros proof_of_False.
  exact I.
```

"case proof_of_False." also works.

```
Qed.
```

```
Theorem thm_false_imp_false : False -> False.
Proof.
  intros proof_of_False.
  case proof_of_False.
```

"exact proof_of_False." works, but is not recommended.

```
Qed.
```

True->False can never be proven. We demonstrate that by proving ~(True->False).

```
Theorem thm_true_imp_false : ~(True -> False).
Proof.
  intros T_implies_F.
  refine (T_implies_F _).
    exact I.
Qed.
```

All of the above proofs should be obvious to you by now.

Reducto ad absurdium

Below is another staple of logic: reduction to absurdity. If a proposition has a proof and you prove that it cannot have a proof, then you can conclude anything.

```
Theorem absurd2 : forall A C : Prop, A -> ~ A -> C.
Proof.
  intros A C.
  intros proof_of_A proof_that_A_cannot_be_proven.
  unfold not in proof_that_A_cannot_be_proven.
  pose (proof_of_False := proof_that_A_cannot_be_proven proof_of_A).
  case proof_of_False.
Qed.
```

This is a tricky proof. Since our subgoal "C" doesn't appear in our hypotheses, we cannot end the proof with "exact something_of_type_C". The only other option we know (so far) is "case" on a proof of False.

The tactic "unfold ... in" is used to interchange the definition of "not" in a hypothesis. That exposes that the type "~A" is really "A -> False", that is, a function from a proof of A to a proof of False.

With that knowledge, we can call the function with "proof_of_A" to get a proof of False. And we end the proof with "case" on a proof of False!

That was a serious proof! But we've done about as much as we can do with Props, so let's get back to bools!

The return of lower-case true and lower-case false

Now, remember when I introduced capital-T True and capital-F False of type Prop, I also introduced the type "bool". Type "bool" has two different constructors: lower-case-t true and lower-case-f false.

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

"true" and "false" have type "bool" and "bool" has type "Set". "Set" is the type of normal datatypes, like "bool" and natural numbers and most of the other types you'll see. The exception is propositions, which have type "Prop".

Let's load some helper functions for the type "bool".

```
Require Import Bool.
```

"Require Import" is a vernacular command that loads definitions from a library. In this case, the library is named "Bool" and there is a file named "Bool.v" that contains its definitions, proofs, etc..

Two of the functions are:

```
Definition eqb (b1 b2:bool) : bool :=
  match b1, b2 with
    | true, true => true
    | true, false => false
    | false, true => false
    | false, false => true
  end.

Definition Is_true (b:bool) :=
  match b with
    | true => True
    | false => False
  end.
```

The first function "eqb" returns true if the two arguments match. ("eqb" is short for "equal for type bool".)

The second function "Is_true" converts a bool into a Prop. In the future, you can use "(<name> = true)", but we haven't described "=" yet. The operator "=" is REALLY cool, but you have to understand more basic types, like bool, first.

Let's do some proofs with these functions.

Is_true true is True

```
Theorem true_is_True: Is_true true.
Proof.
  simpl.
  exact I.
Qed.
```

"Is_true" is a function, so "Is_true true" is that function called with the argument "true". Since "Is_true"'s type is "bool->Prop", we know that the function call returns a "Prop", which is something that can be proven. So, this proof demonstrates that there exists a proof for "Is_true true".

Admittedly, "Is_true true" seems like a dumb thing to prove. Later, we'll replace the argument "true" with more useful boolean expressions, like "4 < 5" and other comparisons.

The proof contains a new tactic: "simpl". It is short for "simplify". If you have a function call and you have the definition of the function, the tactic "simpl" will execute the function on the arguments. In this case, the function returns something of type Prop and that Prop happens to be "True". (Yes, the function returned a type. This is common in Coq.)

"True" becomes our new subgoal. And we've seen how to prove a subgoal of "True" and that's "exact I" because "True" was defined to have a single proof named "I".

RULE: If the current subgoal contains a function call with all its arguments, Then use the tactic "simpl.".

I promised that you would see "Is_true" with a more complex argument. So, let me show you that.

Is_true called with a complex constant.

```
Theorem not_eqb_true_false: ~(Is_true (eqb true false)).
Proof.
  simpl.
  exact False_cannot_be_proven.
Qed.
```

The tactic "simpl" executes the functions to produce a subgoal of "~False". That should look familiar, because we've proved it before! We could copy-and-paste the proof *OR* we could just say the proof already exists! We gave the proof of "~False" the name "False_cannot_be_proven", so the tactic "exact False_cannot_be_proven" finishes the proof immediately! Sweet, isn't it?

Now, let's look at a more complex call to Is_true, where we'll get to see the tactic "case" show its power!

case with bools

```
Theorem eqb_a_a : (forall a : bool, Is_true (eqb a a)).
Proof.
  intros a.
  case a.
```

suppose a is true

```
    simpl.
    exact I.
```

suppose a is false

```
    simpl.
    exact I.
Qed.
```

Take a look at the state after "case a.". There's two subgoals! I said before that "case" creates subgoals for every possible construction of its argument. So far, we've only used "case" with hypotheses of type "False". False has no constructors, so it generates no subgoals and ends our proofs.

Hypotheses of type "bool" have two possible constructors "true" and "false". Thus, "case a" creates two new subgoals - one where "a" has been replaced by "true" and a second where "a" has been replaced by "false".

NOTE: The replacement of "a" by "true" (or "false") only happens inside the subgoal. It does NOT happen in the hypotheses. It is sometimes important to control where "a" is before calling "case"!

I believe it is good style to label the two different cases. I do that with the comment "suppose <hyp> is <constructor>". Since the definition of "bool" listed "true" before "false", the subgoal where "a" is "true" becomes the current subgoal. Once we prove it, the case where "a" is "false" will become the current subgoal.

RULE: If there is a hypothesis "<name>" of a created type AND that hypothesis is used in the subgoal, Then you can try the tactic "case <name>.".

Let's do one more example.

```
Theorem thm_eqb_a_t: (forall a:bool, (Is_true (eqb a true)) -> (Is_true a)).
Proof.
  intros a.
  case a.
```

suppose a is true

```
    simpl.
    intros proof_of_True.
    exact I.
```

suppose a is false

```
    simpl.
    intros proof_of_False.
    case proof_of_False.
Qed.
```

In this example, I had to control the location of "a" when using the tactic "case".

Usually, we would do "intros a" followed by another "intros" to move everything before the "->" into a hypothesis. If we did this, one usage of "a" would move from the subgoal into a hypothesis and "case a" would NOT replace it with "true" or "false". (The "case" tactic only changes the subgoal.) And we'd be unable to prove the theorem.

Instead, I left "a" in the subgoal and delayed the second "intros" until after the "case" command. (In fact, after the "simpl" too.) As a result, all usages of "a" were replaced by "true" or "false" and the theorem was provable.

And, Or

One of the most amazing things about Coq is its fundamental rules are so simple that even things like "and" and "or" can be defined in terms of them. I'll start with "or" because it shows some special features.

Or

Before showing you the definition for "or", I want you to see some examples so that the definition will make sense. In the examples below, imagine you have a hypothesis with a natural number "x".

- (or (x < 5) (x = 7))
- (or True False)
- (or (x = 0) (x = 1))

As you can see, "or" is a function. Each of its arguments are propositions - each of them are things that could be proved. And we know that the "or" itself can be proved, so the result of call to "or" must be a proposition. Since a proposition has type "Prop", the type of "or" is:

- or (A B:Prop) : Prop

That is, a function that it takes two propositions, A and B, and returns a proposition.

I said the result of "or" is a proposition - something that might have a proof. So how do we create a proof of "or"? For that, we need to see the definition...

```
Inductive or (A B:Prop) : Prop :=
  | or_introl : A -> A \/ B
  | or_intror : B -> A \/ B
where "A \/ B" := (or A B) : type_scope.
```

This vernacular command does four things.

- declares "or", a function that takes two Props and produces a Prop
- declares "or_introl", a constructor that takes a proof of "A" and returns a proof of "(or A B)"
- declares "or_intror", a constructor that takes a proof of "B" and returns a

proof of "(or A B)"
- declares "\/", an operator that is interchangeable with "or"

One way to think of this is to say "(or A B)" creates a type and "(or_introl proof_of_A)" and "(or_intror proof_of_B)" are instances of that type. In fact, the ONLY way to create things of type "or" is by using the constructors "or_introl" and "or_intror".

It is important to note that constructors, like "or_introl" and "or_intror", are functions that can be called, but do not have any definitions. They are not made of calls to other functions. They cannot be executed and the result looked at. The tactic "simpl" will not remove them. They are opaque constants.

The vernacular command "Inductive" creates new types. It's called "Inductive" since it lets you create inductive types, although none of our types so far has been inductive. (That will change!) It is the same command we saw earlier to declare other new types with their constants.

```
Inductive False : Prop := .

Inductive True : Prop :=
  I : True.

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

Right now, we want to play with "or", so let's get proving!

```
Theorem left_or : (forall A B : Prop, A -> A \/ B).
Proof.
  intros A B.
  intros proof_of_A.
  pose (proof_of_A_or_B := or_introl proof_of_A : A \/ B).
  exact proof_of_A_or_B.
Qed.
```

This proof should have gone pretty much as you expected. The only complication is that the "pose" command could not infer the type of B from "or_introl proof_of_A", so the type had to be given explicitly by putting ": A \/ B" at the end. "or_introl proof_of_A" by itself is a proof of "A \/ anything".

The proof is shorter without the "pose", since Coq knows the type from the subgoal we're trying to prove. You can see this in a similar proof using "or_intror" below.

```
Theorem right_or : (forall A B : Prop, B -> A \/ B).
Proof.
  intros A B.
  intros proof_of_B.
  refine (or_intror _).
    exact proof_of_B.
Qed.
```

An even shorter proof would have just "exact (or_intror proof_of_B)". But if you don't know all the constructor's arguments, using "refine" is a fine approach.

RULE: If the subgoal's top-most term is a created type, Then use "refine (<name_of_constructor> _ ...).".

Let's prove something a little more complicated...

Or commutes

```
Theorem or_commutes : (forall A B, A \/ B -> B \/ A).
Proof.
  intros A B.
  intros A_or_B.
  case A_or_B.
```

suppose A_or_B is (or_introl proof_of_A)

```
    intros proof_of_A.
    refine (or_intror _).
      exact proof_of_A.
```

suppose A_or_B is (or_intror proof_of_B)

```
    intros proof_of_B.
    refine (or_introl _).
      exact proof_of_B.
Qed.
```

This proof uses the tactic "case" to consider the two different ways we might have constructed the hypothesis "A \/ B". One possibility was using "(or_introl proof_of_A)", which would have meant there was a proof of "A". For that case, the tactic puts "A ->" at the front of the subgoal and we use "intros proof_of_A." to move the proof into the context. Notice that "case" changed the subgoal and not the context. "case" never changes the context - that's a useful property to know (and look out for) when working with it.

The other possibility for creating the hypothesis "A \/ B" was that the constructor "or_intror" was used. For it to be used, there had to exist a proof of "B". So, for the second case created by "case", the subgoal has "B ->" at the front and we use "intros proof_of_B." to move it into the context.

Once the "proof_of_A" or "proof_of_B" is in the context, the proofs of each subgoal follow the pattern we've seen before.

Now that we've seen how "or" works, let's take a look at "and".

And

I think we can go straight to the definition.

```
Inductive and (A B:Prop) : Prop :=
  conj : A -> B -> A /\ B

where "A /\ B" := (and A B) : type_scope.
```

As you could expect, "and" is a function that takes two Props and returns a Prop. That's the exact same as we saw with "or". However, while "or" had two constructors that each took one argument, "and" has a single constructor that takes two arguments. So, the one and only way to create something of type "(and A B)" is to use "(conj proof_of_A proof_of_B)".

Let's go straight to a proof!

```
Theorem both_and : (forall A B : Prop, A -> B -> A /\ B).
Proof.
  intros A B.
  intros proof_of_A proof_of_B.
  refine (conj _ _).
    exact proof_of_A.

    exact proof_of_B.
Qed.
```

Pretty simple proof. There is only one constructor for things of type "(and A B)", which is "conj". It takes two arguments, which after "refine", become two subgoals. The proofs of each are trivial.

After seeing "or", "and" should be easy, so let's jump to the complex proof.

And commutes

```
Theorem and_commutes : (forall A B, A /\ B -> B /\ A).
Proof.
  intros A B.
  intros A_and_B.
  case A_and_B.
```

suppose A_and_B is (conj proof_of_A proof_of_B)

```
    intros proof_of_A proof_of_B.
    refine (conj _ _).
      exact proof_of_B.

      exact proof_of_A.
Qed.
```

The "case" tactic looked at all the different ways to construct an "and" in the hypothesis and found there was only one: "(conj proof_of_A proof_of_B)". So, the tactic created one subgoal with both of those required values at the front of the subgoal.

Similarly, when we need to build an "and" to satisfy the subgoal, there is only one way to construct an "and": "conj" with two arguments. Thus, after "refine" two subgoals were created, one requiring something of type "B" and the other requiring something of type "A".

destruct tactic

The previous proof used "case" to create subgoals for all constructors for something of type "A /\ B". But there was only one constructor. So, we really didn't need to indent. Nor should we waste a line with a comment saying "suppose". And it'd be really nice if we didn't have to write the "intros" command.

Luckily, Coq provides a tactic called "destruct" that is a little more versatile than "case". I recommend using it for types that have a single constructor. The format is:

```
destruct <hyp> as [ <arg1> <arg2> ... ].
```

I put a comment with the name of the constructor inside the square braces.

The result is a much shorter and cleaner proof.

```
Theorem and_commutes__again : (forall A B, A /\ B -> B /\ A).
Proof.
  intros A B.
  intros A_and_B.
  destruct A_and_B as [ proof_of_A proof_of_B].
  refine (conj _ _).
    exact proof_of_B.

    exact proof_of_A.
Qed.
```

RULE: If a hypothesis "<name>" is a created type with only one constructor, Then use "destruct <name> as `<arg1> <arg2> ...` " to extract its arguments.

functions and inductive types

We just saw that "and" and "or" work with Props and are inductive types. In this section, I'll introduce "andb" and "orb" for working with "bool"s. Rather than inductive types, "andb" and "orb" are functions.

To be clear, "and" and "or" are "inductive types" - that is, types we defined where instances can only be produced by calling opaque functions called constructors. Those constructors are things like "or_introl", "or_intror", and "conj".

In fact, the type "bool" is also inductive type. It has two constructors: "true" and "false". These constructors take no arguments, so they are closer to constants than "opaque functions".

To manipulate "bool"s, we could use an inductive type, but it's much easier just to define a function. We've already seen one function on bools: "eqb", which did equality. Some other functions that Coq defines are:

- andb (b1 b2:bool) : bool
- orb (b1 b2:bool) : bool
- negb (b:bool) : bool

These are "and", "or", and "not" (but called "negation" for some reason). Like with "eqb", the names all ended in "b" to indicate they work with "bool"s. Two of these functions also come with operators:

```
Infix "&&" := andb : bool_scope.
Infix "" := orb : bool_scope.
```

We'll get some good practice proving that these functions on bools are equivalent to our inductive types on Props. As part of this, we'll need a form of equality for propositions that is called "if and only if" or "double implication".

```
Definition iff (A B:Prop) := (A -> B) /\ (B -> A).

Notation "A <-> B" := (iff A B) : type_scope.
```

You should know everything you see in the proofs below. Remember, "simpl" is used to execute a function and "case" and "destruct" are used on inductive types.

```
Theorem orb_is_or : (forall a b, Is_true (orb a b) <-> Is_true a \/ Is_true
b).
Proof.
  intros a b.
  unfold iff.
  refine (conj _ _).
```

orb -> \/

```
    intros H.
    case a, b.
```

suppose a,b is true, true

```
      simpl.
      refine (or_introl _).
```

```
        exact I.
```

suppose a,b is true, false

```
      simpl.
      exact (or_introl I).
```

suppose a,b is false, true

```
      exact (or_intror I).
```

suppose a,b is false, false

```
      simpl in H.
      case H.
```

\/ -> orb

```
    intros H.
    case a, b.
```

suppose a,b is true, true

```
      simpl.
      exact I.
```

suppose a,b is true, false

```
      exact I.
```

suppose a,b is false, true

```
      exact I.
```

suppose a,b is false, false

```
      case H.
```

suppose H is (or_introl A)

```
         intros A.
         simpl in A.
         case A.
```

suppose H is (or_intror B)

```
         intros B.
         simpl in B.
         case B.
Qed.
```

Wow! We're not in Kansas any more!

First, I used "case" to split the "\/" hidden inside the "<->".

Second, inside each of those branches, I did "case a,b", which created 4 subgoals, for every possible combination of the two constructors. I thought the proof was clearer that way, even though doing "case a" followed by "case b" might have been shorter.

The proofs for when "a" or "b" were true were easy. They were all the same proof. You can see that I make the proof shorter as I went along

- combining "refine" and "exact" and skipping right over "simpl".

When both "a" and "b" were false, I decided it was best to do "case" on an instance of False. For "orb", that was pretty easy. I did "simpl in H", which executed functions calls inside a hypothesis. That gave me an hypothesis of type False, and "case H" ended the subgoal.

However, for the inductive type "or", it was more complicated. I used "case H" to consider the different ways of constructing the "or". Both constructors - "or_introl" and "or_intror" - lead to a False hypothesis.

That was a real proof. Welcome to the big leagues.

RULE If a hypothesis "<name>" contain a function call with all its arguments, Then use the tactic "simpl in <name>.".

Let's try it with "andb" and "/\".

```
Theorem andb_is_and : (forall a b, Is_true (andb a b) <-> Is_true a /\
Is_true b).
Proof.
  intros a b.
  unfold iff.
  refine (conj _ _).
```

andb -> /\

```
    intros H.
    case a, b.
```

suppose a,b is true,true

```
      simpl.
      exact (conj I I).
```

suppose a,b is true,false

```
      simpl in H.
      case H.
```

suppose a,b is false,true

```
      simpl in H.
      case H.
```

suppose a,b is false,false

```
      simpl in H.
      case H.
```

/\ -> andb

```
    intros H.
    case a,b.
```

suppose a,b is true,true

```
      simpl.
      exact I.
```

suppose a,b is true,false

```
      simpl in H.
      destruct H as [ A B].
      case B.
```

suppose a,b is false,true

```
      simpl in H.
      destruct H as [ A B].
      case A.
```

suppose a,b is false,false

```
      simpl in H.
      destruct H as [ A B].
      case A.
Qed.
```

admit tactic

It should be easy for you to proof that "not" and "notb" are equivalent. Below is a statement of the theorem; fill it in if you think you can.

I did not include a proof, so where you would normally see a proof, you'll see the tactic "admit" and the vernacular command "Admitted". The tactic "admit" is a cheat. It ends a subgoal without solving it. A proof containing an "admit" is not a real proof, so Coq forces you to end it with "Admitted" instead of "Qed". I use these commands below so that Coq's parser won't get hung up because I didn't include a proof of the theorem.

The "admit" tactic has real uses. When there are multiple subgoals and you want to skip over the easy ones to work on the hard one first, the "admit" tactic can be used to get the easy subgoals out of the way. Or, if you are only part way done a proof but you want send someone a Coq file that parses, "admit" can be used to fill in your blanks.

So, take a swing at proving this theorem.

- HINT: Remember "~A" is "A -> False".
- HINT: You can make use of the proof "False_cannot_be_proven"
- HINT: When you get stuck, look at "thm_true_imp_false".

```
Theorem negb_is_not : (forall a, Is_true (negb a) <-> (~(Is_true a))).
Proof.
  admit.
```

delete "admit" and put your proof here.

```
Admitted.
```

when done, replace "Admitted." with "Qed."

RULE: If you have a subgoal that you want to ignore for a while, Then use the tactic "admit.".

Existence and Equality

Like "and" and "or", the concepts of "there exists" and "equality" are not fundamental to Coq. They are concepts defined inside of it. Only "forall" and the notion of creating a type with constructors is fundamental to Coq.

Existence

In Coq, you cannot just declare that something exists. You must prove it.

For example, we might want to prove that "there exists a bool 'a' such that (Is_true (andb a true))". We cannot just state that the "bool" exists. You need to produce a value for "a" - called the witness - and then prove that the statement holds for the witness.

The definition and operator are:

```
Inductive ex (A:Type) (P:A -> Prop) : Prop :=
  ex_intro : forall x:A, P x -> ex (A:=A) P.

Notation "'exists' x .. y , p" := (ex (fun x => .. (ex (fun y => p)) ..))
  (at level 200, x binder, right associativity,
   format "'[' 'exists'  '/  ' x  ..  y ,  '/  ' p ']'")
  : type_scope.
```

The proposition "ex P" should be read: "P is a function returning a Prop and there exists an argument to that function such that (P arg) has been proven". The function "P" is known as "the predicate". The constructor for "ex P" takes the predicate "P" , the witness (called "x" here) and a proof of "P x" in order to return something of type "ex P".

"exists ..., ..." is an operator to provide a friendly notation. For "and" and "or", we did the same with "/\" and "/\". For existence, the usually operator is a capital E written backwards, but that's difficult to type, so Coq uses the word "exists".

Let's test this out with the easy theorem I already mentioned.

```
Definition basic_predicate
:=
  (fun a => Is_true (andb a true))
.

Theorem thm_exists_basics : (ex basic_predicate).
Proof.
  pose (witness := true).
  refine (ex_intro basic_predicate witness _).
    simpl.
    exact I.
Qed.
```

I start by defining the predicate: a function that takes a single argument "a" (which Coq can determine is a "bool"). The key part of the proof is the tactic "refine (ex_intro ...)". The arguments to "ex_intro" are:

- the predicate
- the witness
- a proof of the predicated called with the witness

In this usage, I passed "ex_intro" the predicate, the witness, and use an "_" to create a new subgoal for the proof. The tactics "simpl" and "exist" are used to solve the proof.

I'll prove the same theorem, but this time I'll use "exists" operator to show you how much cleaner it looks.

```
Theorem thm_exists_basics__again : (exists a, Is_true (andb a true)).
Proof.
  pose (witness := true).
  refine (ex_intro _ witness _).
```

Coq automatically determines the predicate! We're left to prove that the witness satisfies the function.

```
    simpl.
    exact I.
Qed.
```

RULE: If the current subgoal starts "exists <name>, ..." Then create a witness and use "refine (ex_intro _ witness _)"

More existence

We often use "exists" and "forall" at the same time. Thus, we end up with proofs like the following.

```
Theorem thm_forall_exists : (forall b, (exists a, Is_true(eqb a b))).
Proof.
  intros b.
  case b.
```

b is true

```
    pose (witness := true).
    refine (ex_intro _ witness _).
      simpl.
      exact I.
```

b is false

```
    pose (witness := false).
    refine (ex_intro _ witness _).
      simpl.
      exact I.
Qed.
```

If you look at the proof above, the witness was always equal to "b". So, let's try simplifying the proof.

```
Theorem thm_forall_exists__again : (forall b, (exists a, Is_true(eqb a b))).
Proof.
  intros b.
  refine (ex_intro _ b _).
```

witness is b

```
  exact (eqb_a_a b).
Qed.
```

We used "b" as the witness and ended up with the state:

```
  b : bool
  ============================
   Is_true (eqb b b)
```

Now, we have already proved:

```
Theorem eqb_a_a : (forall a : bool, Is_true (eqb a a)).
```

I've told you that "->" is the type of a function and that "->" is shorthand for "(forall ...)", so it's not a far leap to see that our theorem "eqb_a_a" is a function from any bool (called "a" in the statement) to a proof of "Is_true (eqb a a)".

In our current proof, we needed that statement to hold for our particular hypothesis of "b". That is done by the function call "eqb_a_a b", which substitutes the specific "b" into the place of the generic "a" in the body of the "forall". The result is "Is_true (eqb b b)", which solves our proof.

Exists and Forall

Here is a classic theorem of logic, showing the relationship between "forall" and "exists". It gives us a chance to "destruct" an instance of type "ex".

```
Theorem forall_exists : (forall P : Set->Prop, (forall x, ~(P x)) -> ~(exists
x, P x)).
Proof.
  intros P.
  intros forall_x_not_Px.
  unfold not.
  intros exists_x_Px.
  destruct exists_x_Px as [ witness proof_of_Pwitness].
  pose (not_Pwitness := forall_x_not_Px witness).
  unfold not in not_Pwitness.
  pose (proof_of_False := not_Pwitness proof_of_Pwitness).
  case proof_of_False.
Qed.
```

The proof requires some explanation.

- we use intros and unfold until we have everything in the context.
- we use "destruct" to extract the witness and the proof of "P witness".
- we call "(forall x, ~(P x))" with the witness, to generate "(P witness) -> False"
- we call "P witness -> False" with "P witness" to get a proof of "False".
- we use the tactic "case" on "proof_of_False"

This proof is hard to read. If you have difficulty reading a proof, you can always use CoqIDE or Proof General to step through the proof and look at the state after each tactic.

Another good example of "exists" is proving that the implication goes the other way too.

```
Theorem exists_forall : (forall P : Set->Prop, ~(exists x, P x) -> (forall x,
~(P x))).
```

```
Proof.
  intros P.
  intros not_exists_x_Px.
  intros x.
  unfold not.
  intros P_x.
  unfold not in not_exists_x_Px.
  refine (not_exists_x_Px _).
    exact (ex_intro P x P_x).
Qed.
```

Again, this isn't a very readable proof. It goes by:

- we use intros and unfold until our subgoal is "False".
- we use "refine" to call a function whose type ends in "-> False"
- we create something of type "exists" by calling "ex_intro"

Calculating witnesses

For some predicates, the witness is a single value or is easy to generate. Other times, it isn't. In those cases, we can write a function to calculate the witness.

Right now, we can't write very complicated functions, since we only know a few types. When we have lists and natural numbers, you'll see some examples.

Equality

Now we come to the big prize: equality! Equality is a derived concept in Coq. It's an inductive type, just like "and", "or", and "ex" ("exists"). When I found that out, I was shocked and fascinated!

It's defined as:

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
    eq_refl : x = x :>A

where "x = y :> A" := (@eq A x y) : type_scope.

Notation "x = y" := (x = y :>_) : type_scope.
```

The "Inductive" statement creates a new type "eq" which is a function of a type A and 2 values of type A to Prop. (NOTE: One value of type A is written (x:A) before the ":" and the other is written "A ->" after. This is done so Coq infers the type "A" from the first value and not the second.) Calling "eq" with all its arguments returns a proposition (with type Prop). A proof of "eq x y" means that "x" and "y" both have the same type and that "x" equals "y".

The only way to create a proof of type "eq" is to use the only constructor "eq_refl". It takes a value of "x" of type "A" and returns "@eq A x x", that is, that "x" is equal to itself. (The "@" prevents Coq from inferring values, like the type "A".) The name "eq_refl" comes from the reflexive property of equality.

Lastly, comes two operators. The less commonly used one is "x = y :> A" which let's you say that "x" and "y" are equal and both have type "A". The one you'll use most of the time, "x = y", does the same but let's Coq infer the type "A" instead of forcing you to type it.

Now, if you we paying attention, you saw that "eq_refl" is the only constructor. We can only create proofs of "x = x"! That doesn't seem useful at all!

What you don't see is that Coq allows you to execute a function call and substitute the result for the function call. For example, if we had a function "plus" that added natural numbers (which have type "nat"), we could use "eq_refl (plus 1 1)" to create a proof of "eq nat (plus 1 1) (plus 1 1)". Then, if we execute the second function call, we get "eq nat (plus 1 1) 2", that is, "1 + 1 = 2"!

The concept of substituting a function call with its result or substituting the result with the function call is called "convertibility". One tactic we've seen, "simpl", replaces convertible values. We'll see more tactics in the future.

Now that we have a concept of what "=" means in Coq, let's use it!

Equality is symmetric

```
Theorem thm_eq_sym : (forall x y : Set, x = y -> y = x).
Proof.
  intros x y.
  intros x_y.
  destruct x_y as [].
  exact (eq_refl x).
Qed.
```

I use the "destruct" tactic on the type "eq" because it only has one constructor. Let's look at the state before and after that call.

Before, it is:

```
x : Set
y : Set
x_y : x = y
============================
 y = x
```

And after:

```
x : Set
============================
 x = x
```

By destructing the "eq_refl", Coq realizes that "x" and "y" are convertible and wherever the second name is, it can be replaced by the first. So, "y" disappears and is replaced by "x". (NOTE: "destruct" unlike "case", does change the context, so the hypotheses "y" and "x_y" also disappear.)

After "destruct", we're left with a subgoal of "x = x", which is solved by calling "eq_refl".

Just to become familiar, let's prove the other big property of equality: transitivity.

Equality is transitive

```
Theorem thm_eq_trans : (forall x y z: Set, x = y -> y = z -> x = z).
Proof.
  intros x y z.
  intros x_y y_z.
  destruct x_y as [].
  destruct y_z as [].
  exact (eq_refl x).
Qed.
```

Seems pretty easy, doesn't it?

Rather than using "destruct", most proofs using equality use the tactics "rewrite" and "rewrite <-". If "x_y" has type "x = y", then "rewrite x_y" will replace "x" with "y" in the subgoal and "rewrite <- x_y" will go the other way, replacing "y" with "x".

Let's see these tactics by proving transitivity again.

```
Theorem thm_eq_trans__again : (forall x y z: Set, x = y -> y = z -> x = z).
Proof.
  intros x y z.
  intros x_y y_z.
  rewrite x_y.
  rewrite <- y_z.
  exact (eq_refl y).
Qed.
```

That is much cleaner.

RULE: If you have a hypothesis "<name> : <a> = <b>" AND "<a>" in your current subgoal Then use the tactic "rewrite <name>.".

RULE: If you have a hypothesis "<name> : <a> = <b>" AND "<b>" in your current subgoal Then use the tactic "rewrite <- <name>.".

Let's try something that explicitly relies on convertibility. Recall, "andb" is and for "bools" and that it is represented by the operator "&&".

```
Theorem andb_sym : (forall a b, a && b = b && a).
Proof.
  intros a b.
  case a, b.
```

suppose a,b is true,true

```
    simpl.
    exact (eq_refl true).
```

suppose a,b is true,false

```
    simpl.
    exact (eq_refl false).
```

suppose a,b is false,true

```
    simpl.
    exact (eq_refl false).
```

suppose a,b is false,false

```
    simpl.
    exact (eq_refl false).
Qed.
```

So, for this proof, we divided it into 4 cases and used convertibility to show that equality held in each. Pretty simple.

I should do an example with inequality too.

Inequality with discriminate

Coq uses the operator "<>" for inequality, which really means "equality is unprovable" or "equality implies False".

```
Notation "x <> y  :> T" := (~ x = y :>T) : type_scope.
Notation "x <> y" := (x <> y :>_) : type_scope.
```

Let's start with a simple example. Recall that "negb" is the "not" operation for bools.

```
Theorem neq_nega: (forall a, a <> (negb a)).
Proof.
  intros a.
  unfold not.
  case a.
    intros a_eq_neg_a.
    simpl in a_eq_neg_a.
    discriminate a_eq_neg_a.

    intros a_eq_neg_a.
    simpl in a_eq_neg_a.
    discriminate a_eq_neg_a.
Qed.
```

To prove this, once again I had to delay calling "intros" until after "case". If I did not, "a" in "a_eq_neg_a" would not get replaced "true" or "false". I also used a new tactic: "discriminate".

"discriminate" operates on a hypothesis where values of inductive type are compared using equality. If the constructors used to generate the type are the different, like here where we have "true = false", then Coq knows that situation can never happen. It's like a proof of False. In that case, "discriminate <hypname>." ends the subgoal.

When working with inductive types, you will use "discriminate" to eliminate a lot of cases that can never happen.

RULE: If you have a hypothesis "<name> : (<constructor1> ...) = (<constructor2> ...) OR "<name> : <constant1> = <constant2> Then use the tactic "discriminate <name>.".

To really show off equality, we'll need something more numerous than just 2 booleans. Next up is natural numbers - an infinite set of objects - and induction - which allows us to prove properties about infinite sets of objects!

Natural Numbers and Induction

Peano Arithmetic

The natural numbers are 0, 1, 2, 3, 4, ... They are the simplest mathematical concept that contain an unbounded number of values.

In 1889, Peano published a simple set of axioms (fundamental rules) that defined the natural numbers and how to prove things about them. Peano's axioms have become the standard way in logic of talking about and manipulating the natural numbers. More complex schemes - like the decimal numbers we usually use - are justified by proving them equivalent to Peano's axioms.

Peano's definition of natural numbers is:

- 0 is a natural number.
- For every natural number n, (S n) is a natural number.

"(S n)" is spoken as "the successor of n" but you should think about it as "1 + n". If we were to list the first few of Peano's natural numbers, they would be:

- 0 is 0
- 1 is (S 0) = (1 + 0)
- 2 is (S (S 0)) = (1+ (1 + 0))
- 3 is (S (S (S 0)) = (1 + (1 + (1 + 0)))
- ...

Peano's natural numbers are represented in Coq by the type "nat".

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

WARNING: The "O" here is the capital-letter O, *not* the number zero.

So, Coq's natural numbers exactly matches the definition of Peano.

If we want to do something with "nat"s, we need some functions. Here is how Peano's definition of addition looks in Coq.

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O => m
  | S p => S (p + m)
  end

where "n + m" := (plus n m) : nat_scope.
```

Previously, we used the vernacular command "Definition" to define a function. The command here is "Fixpoint". "Fixpoint" must be used any time you define a recursive function. It's hard to see that "plus" calls itself because the recursive call is hidden by the "+" in "p + m".

We know that (2 + 3) = 5, so let's see how that property holds in Coq. Below, the difference between lines is one execution of a function call to "plus".

- (plus (S (S O)) (S (S (S O))))
- (S (plus (S O) (S (S (S O)))))
- (S (S (plus O (S (S (S O))))))
- (S (S (S (S (S O))))))

Each execution of "plus" strips an "S" off the front of the first number and puts it in front of the recursive call to "plus". When the first parameter is "O", "plus" returns the second argument.

Remember that "O" and "S" are constructors - they're opaque functions that were created when we declared "nat". There is no way to execute them. Therefore, "(S (S (S (S (S O))))))" does not have any function calls that can be executed and, therefore, is the canonical form of the number 5.

The proof of "2 + 3 = 5" only need one call to "simpl" to get it to the canonical form.

```
Theorem plus_2_3 : (S (S O)) + (S (S (S O))) = (S (S (S (S (S O))))).
Proof.
  simpl.
  exact (eq_refl 5).
Qed.
```

As you can see by "eq_refl 5", Coq will translate decimal numbers into terms of "nat" automatically. This means capital-letter "O" and the digit "0" are really the same term, which prevents a lot of mistakes. There are ways to instruct Coq to treat decimals as other types, such as integers, rationals, etc.

We can convince ourselves that the function "plus" has all the properties associated with addition, by proving those properties. So, let's prove them!

```
Theorem plus_O_n : (forall n, O + n = n).
Proof.
  intros n.
  simpl.
  exact (eq_refl n).
Qed.
```

Induction

It was easy to prove "(forall n, O + n = n)", but it's much harder to prove "(forall n, n + O = n)". The reason is that the function "plus" executes "match" on its first parameter. If the first parameter is a constant, such as "O", we can use "simpl" to execute "plus" and get the answer. But, if the first argument is a variable, like "n", we can't.

To solve this, we need to do the proof by induction on "n". Peano's axiom for induction is:

```
  If "P" is a predicate such that
     (P 0) is proven, and
     for all n, (P n) -> (P (S n)) is proven,
   Then (P n) is proven for all n.
```

The proof of "(P O)" is known as the "base case". The proof of "(forall n:nat, P n -> P (S n))" is known as the "inductive case" and, inside that case, "P n" is known as the "inductive hypothesis".

When "nat" was declared, Coq created a function that exactly represents induction on "nat"s. It is called "nat_ind" and you can see its type with the vernacular command "Check".

```
Check nat_ind.
```

Its output should look like this:

```
nat_ind
     : forall P : nat -> Type,
         P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Now, we *could* - define a function "P n" that represents "O + n = n"

- prove that the function holds for "P O"
- prove that the function holds for "P (S n)" assuming "P n"
- and then call the function nat_ind with those 3 values.

But that's a lot of work. Luckily, Coq defines tactics that do most of the work for us! These tactics take a subgoal and infer the predicate P, and then generates child subgoals for the other pieces.

One tactic, "elim" works very similarly to "case".

```
Theorem plus_n_O : (forall n, n + O = n).
Proof.
  intros n.
  elim n.
```

base case

```
    simpl.
    exact (eq_refl O).
```

inductive case

```
    intros n'.
    intros inductive_hypothesis.
    simpl.
    rewrite inductive_hypothesis.
    exact (eq_refl (S n')).
Qed.
```

It's obvious how the tactic "elim" generates the base case and inductive case. It was important to me to show that the "n" in the inductive case was different from the original "n" in our goal. I chose to name it "n'" which is spoken "n prime".

I used the "rewrite" tactic with the inductive hypothesis to substitute "n' + O" in the subgoal. If you're using the "elim" tactic, you should always use the inductive hypothesis. If you don't, you should use the "case" tactic instead of "elim".

RULE: If there is a hypothesis "<name>" of a created type AND that hypothesis is used in the subgoal, AND the type has a recursive definition, Then you can try the tactic "elim <name>.".

```
*)
```

Induction tactic

Just as the "case" tactic has a similar "destruct" tactic, the "elim" tactic has a similar "induction" tactic. Let's look at the previous proof, but using this new tactic.

```
Theorem plus_n_O__again : (forall n, n + O = n).
Proof.
  intros n.
  induction n as [|n' inductive_hypothesis].
```

base case

```
    simpl.
    exact (eq_refl O).
```

inductive case

```
    simpl.
    rewrite inductive_hypothesis.
    exact (eq_refl (S n')).
Qed.
```

So, the "induction" tactic does the same thing as "elim.", except the names of the created variables are listed in the tactic, rather than being assigned later using "intros" in the inductive case of the proof.

The "induction" command creates 2 subgoals, one for the base case and another for the inductive case, and after the "as" keyword, there are 2 lists of variable names, one for the base case and one for the inductive case. Those lists are separated by verical bars ('|') inside the square brackets. The base case doesn't create any variables, so its list is empty. The inductive case creates two variables, and they are named "n'" and "inductive_hypothesis".

I said the "induction" command was similar to "destruct" and, if the type "destruct"ed has more than one constructor, the "destruct" command will create a subgoals for each constructor and the command needs a list of variable names for each constructor. For example, the type "or" has two constructors. Recall that something of type "or A B" can be created by "or_introl proof_of_A" or "or_intror proof_of_B". If I "destruct" an "or A B", it will create two subgoals and the "destruct" needs to have a list of variables for each. To demonstrate this, I'll redo the proof that or commutes.

```
Theorem or_commutes__again : (forall A B, A \/ B -> B \/ A).
Proof.
  intros A B.
  intros A_or_B.
  destruct A_or_B as [proof_of_A | proof_of_B].
```

suppose A_or_B is (or_introl proof_of_A)

```
    refine (or_intror _).
      exact proof_of_A.
```

suppose A_or_B is (or_intror proof_of_B)

```
    refine (or_introl _).
      exact proof_of_B.
Qed.
```

In my proofs, I like to use "case" and "elim" and only use "destruct" for types with a single constructor. However, some people prefer to use "destruct" and "induction" for every proof. The writers of Coq are talking about removing this duplication and may remove "case" and "elim" in the future.

If you want to use "destruct" and "induction", it is helpful to use the Vernacular command "Print", which prints out a definition and shows you how many constructors there are and which variables you need to name for each one.

```
Print or.
Print nat_ind.
```

Now we return to proofs on induction using "nat". And let's do a difficult proof. Let's prove that "n + m = m + n".

Addition is Symmetric

```
Theorem plus_sym: (forall n m, n + m = m + n).
Proof.
  intros n m.
  elim n.
```

base case for n

```
    elim m.
```

base case for m

```
      exact (eq_refl (O + O)).
```

inductive case for m

```
      intros m'.
      intros inductive_hyp_m.
      simpl.
      rewrite <- inductive_hyp_m.
      simpl.
      exact (eq_refl (S m')).
```

inductive case for n

```
    intros n'.
    intros inductive_hyp_n.
    simpl.
    rewrite inductive_hyp_n.
    elim m.
```

base case for m

```
      simpl.
      exact (eq_refl (S n')).

      intros m'.
      intros inductive_hyp_m.
      simpl.
      rewrite inductive_hyp_m.
      simpl.
      exact (eq_refl (S (m' + S n'))).
Qed.
```

That is a *hard* proof. I've seen an expert in Coq get stuck on it for 10 minutes. It uses a lot of the features that I've demonstrated in this tutorial. I encourage you to open a new file and try to prove it yourself. If you get stuck on part of it, you should use the "admit" tactic and move onto another part. If you get really stuck, open up this file and look at how I did it in my proof. You should keep practicing on that proof until you can do it automatically.

Common nat operators

"nat" is a commonly used type in Coq and there are a lot of operators defined for it. Having seen how "plus" was defined, you can probably guess the definition for "mult".

```
Fixpoint mult (n m:nat) : nat :=
  match n with
  | O => 0
  | S p => m + p * m
  end

where "n * m" := (mult n m) : nat_scope.
```

Also defined are:

- "n - m" := (minus n m)
- "n <= m" := (le n m)
- "n < m" := (lt n m)
- "n >= m" := (ge n m)
- "n > m" := (gt n m)

Along with:

- "x <= y <= z" := (x <= y /\ y <= z)
- "x <= y < z" := (x <= y /\ y < z)
- "x < y < z" := (x < y /\ y < z)
- "x < y <= z" := (x < y /\ y <= z)
- "max n m"
- "min n m"

The dangerous one in this group is "minus". The natural numbers don't go lower than 0, so "(1 - 2)" cannot return a negative number. Since Coq must return something of type "nat", it returns "O" instead. So you can do crazy proofs like this without intending!

```
Theorem minus_is_funny : (0 - 1) = (0 - 2).
Proof.
  simpl.
  exact (eq_refl 0).
Qed.
```

Later, with lists, I'll show you different ways to implement "minus" without this flaw.

Data types

You've learned the basics of Coq. This section introduces some commonly used data types, so that you get to see them and get some more experience reading proofs.

```
Require Import List.
```

Lists and Option

Lists are a common data structure in functional programming. (Imperative programs use more arrays.) The definition of a singly-linked list in Coq is:

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.
```

I'll address the type "Type" in a second, but for the moment know that a list takes a type called "A" and is either empty - constructed using "nil" - or a node containing a value of type "A" and a link to another list.

In a number of earlier places, I've ignored the type "Type". It hides some magic in Coq. We saw early on that "proof_of_A" was a proof and had type "A", which was a proposition. "A", since it was a proposition, had type "Prop". But if types can have types, what type does "Prop" have? The answer is "Type(1)". The answer to your next N questions is that "Type(1)" has type "Type(2)". "Type(2)" has type "Type(3)". Etc.

Similarly, "true" had type "bool". "bool" had type "Set". And "Set" has type "Type(1)" (just like "Prop").

Coq hides this infinite hierarchy from the user with the magic type "Type". When you use "Type", Coq will determine if the value lies in "Prop", "Set", or any of the the "Type(...)" types.

Going back to lists, there is an operator for building a list.

```
Infix "::" := cons (at level 60, right associativity) : list_scope.
```

So, the value "5 :: 4 :: nil" would be a "list nat" containing the values 5 and 4.

A simple function that works on all types of lists is "length":

```
Definition length (A : Type) : list A -> nat :=
  fix length l :=
  match l with
   | nil => O
   | _ :: l' => S (length l')
  end.
```

Just to get started, let's prove that adding an element to a list increases it's length by 1.

```
Theorem cons_adds_one_to_length :
   (forall A:Type,
   (forall (x : A) (lst : list A),
   length (x :: lst) = (S (length lst)))).
Proof.
  intros A.
  intros x lst.
  simpl.
  exact (eq_refl (S (length lst))).
Qed.
```

The proof is pretty simple, but the statement of what we want to prove is not! First, we need "A", which is the type stored in our lists. Then we need an instance of A and a instance of a list of A. Only then can we state that putting an element on the front of a list increases its length by 1.

The three forms of hd

Now, the simplest function you can imagine for a linked list is "hd", which returns the first value in the list. But there's a problem: what should we return if the list is empty?

In some programming languages, you might throw an exception - either explicitly or through a memory violation. In others, you might assume the program crashes. But if you to prove a program correct, these aren't choices you can make. The program needs to be predictable.

There are choices you can make:

- have "hd" take an additional parameter, which it returns if the list is empty
- have "hd" return a new data structure that may or may not contain the value
- pass "hd" a proof that the list is not empty

You didn't think of that last one, did you?! I'll show you each of these approaches so that you can see what they look like.

hd with additional parameter

```
Definition hd (A : Type) (default : A) (l : list A)
:=
  match l with
    | nil => default
    | x :: _ => x
  end.
```

This is the default version of hd in Coq and it looks like you would expect. This version is more useful than you might think thanks to "partial evaluation".

"partial evaluation" is passing only *some* of the arguments to a function. In Coq, and many functional languages, the result is a new function where the supplied arguments are now treated like constants in the function and the remaining parameters are can still be passed in.

Thus, we can do:

```
Definition my_hd_for_nat_lists
:=
  hd nat 0.
```

The remaining parameter - the list - can still be passed to "my_hd_for_nat_lists", but the "Type" parameter has been bound to "nat" and the parameter "default" has been bound to zero.

We can confirm that it works correctly by using the vernacular command "Compute", which executes all the function calls and prints the result.

```
Compute my_hd_for_nat_lists nil.
```

Prints "= 0 : nat".

```
Compute my_hd_for_nat_lists (5 :: 4 :: nil).
```

Prints "= 5 : nat".

We can also prove it correct.

```
Theorem correctness_of_hd :
   (forall A:Type,
   (forall (default : A) (x : A) (lst : list A),
   (hd A default nil) = default /\ (hd A default (x :: lst)) = x)).
Proof.
  intros A.
  intros default x lst.
  refine (conj _ _).
    simpl.
    exact (eq_refl default).

    simpl.
    exact (eq_refl x).
Qed.
```

hd returns option

The second approach is to have "hd" return another data structure that either contains the first value in the list or contains nothing. Coq defines a generic data structure for that, called "option".

```
Inductive option (A : Type) : Type :=
  | Some : A -> option A
  | None : option A.
```

"option" is a type with two constructors. "Some" takes a value and "None" does not. To get the value out of an "option", a proof will have to branch using "case" and look at the branch for the constructor "Some". Likewise, a program will have to use "match" and look at the branch for "Some".

Coq includes a version of "hd" that returns an "option". It's called "hd_error".

```
Definition hd_error (A : Type) (l : list A)
:=
  match l with
    | nil => None
    | x :: _ => Some x
  end.
```

Again, using "Compute" we can examine what it returns.

```
Compute hd_error nat nil.
```

Prints "= None : option nat".

```
Compute hd_error nat (5 :: 4 :: nil).
```

Prints "= Some 5 : option nat".

We can also prove it correct.

```
Theorem correctness_of_hd_error :
   (forall A:Type,
   (forall (x : A) (lst : list A),
   (hd_error A nil) = None /\ (hd_error A (x :: lst)) = Some x)).
Proof.
  intros A.
  intros x lst.
  refine (conj _ _).
    simpl.
    exact (eq_refl None).

    simpl.
    exact (eq_refl (Some x)).
Qed.
```

hd with a proof that the list is non-empty

Here's the option that's not available to any other programming language you've seen: we can pass to "hd" a proof that the list is not empty, so "hd" can never fail.

```
Definition hd_never_fail (A : Type) (lst : list A) (safety_proof : lst <>
nil)
  : A
:=
  (match lst as b return (lst = b -> A) with
    | nil => (fun foo : lst = nil =>
                   match (safety_proof foo) return A with
                   end
                )
    | x :: _ => (fun foo : lst = x :: _ =>
                   x
                )
  end) eq_refl.
```

I don't expect you to understand this function, let alone write it. It took me 30 minutes to write it, mostly by copying from code printed using "Show Proof." from the complicated proofs above.

But I do expect you to understand that it *can* be written in Coq. And, I expect you to be able to read a proof that the function does what I said it does.

```
Theorem correctness_of_hd_never_fail :
   (forall A:Type,
   (forall (x : A) (rest : list A),
   (exists safety_proof : ((x :: rest) <> nil),
      (hd_never_fail A (x :: rest) safety_proof) = x))).
Proof.
  intros A.
  intros x rest.
  assert (witness : ((x :: rest) <> nil)).
    unfold not.
    intros cons_eq_nil.
    discriminate cons_eq_nil.
  refine (ex_intro _ witness _).
    simpl.
    exact (eq_refl x).
Qed.
```

The proof is pretty simple. We create a witness for the exists. Since the witness involves an inequality, we end its proof with "discriminate". Then we use "simpl" to show that the function returns what the right value.

In this proof, I used the tactic "assert" instead of "pose". I wanted to write:

```
pose (witness := _ : ((x :: rest) <> nil)).
```

But Coq does not allow that for some reason. "pose" cannot have "_" for the complete value. If you want to use "_" for the whole value of pose, you have to use the tactic "assert".

Tail of lists

"hd" extracts the first value in a list and its partner "tl" returns the list after that first element. In Coq, it's definition is:

```
Definition tl (A : Type) (l:list A) :=
  match l with
    | nil => nil
    | a :: m => m
  end.
```

If you want to test your skills, it would be good practice to write three different versions of "tl", just like I did for "hd", and prove them correct.

```
Definition tl_error (A : Type) (l : list A)
  : option list A
:=
  ...
Definition tl_never_fail (A : Type) (lst : list A) (safety_proof : lst <> nil)
  : list A
:=
  ...
```

Here's just a simple proof that the "hd" and "tl" of a non-empty list can be used to reconstruct the list.

```
Theorem hd_tl :
   (forall A:Type,
   (forall (default : A) (x : A) (lst : list A),
   (hd A default (x::lst)) :: (tl A (x::lst)) = (x :: lst))).
Proof.
  intros A.
  intros default x lst.
  simpl.
  exact (eq_refl (x::lst)).
Qed.
```

Appending lists

We often want to do more than just add one element at the front of a list. Coq includes the function "app", short for "append", to concatenate two lists.

```
Definition app (A : Type) : list A -> list A -> list A :=
  fix app l m :=
  match l with
   | nil => m
   | a :: l1 => a :: app l1 m
  end.

Infix "++" := app (right associativity, at level 60) : list_scope.
```

Here are some nice theorems you can try to prove about appending lists. Since append is defined recursively, many of these theorems must be proved with induction using the "elim" tactic.

```
Theorem app_nil_l : (forall A:Type, (forall l:list A, nil ++ l = l)).
Proof.
  admit.
```

delete "admit" and put your proof here.

```
Admitted.
```

when done, replace "Admitted." with "Qed."

```
Theorem app_nil_r : (forall A:Type, (forall l:list A, forall l:list A, l ++
nil = l)).
Proof.
  admit.
```

delete "admit" and put your proof here.

```
Admitted.
```

when done, replace "Admitted." with "Qed."

```
Theorem app_comm_cons : forall A (x y:list A) (a:A), a :: (x ++ y) = (a :: x)
++ y.
Proof.
  admit.
```

delete "admit" and put your proof here.

```
Admitted.
```

when done, replace "Admitted." with "Qed."

```
Theorem app_assoc : forall A (l m n:list A), l ++ m ++ n = (l ++ m) ++ n.
Proof.
  admit.
```

delete "admit" and put your proof here.

```
Admitted.
```

when done, replace "Admitted." with "Qed."

```
Theorem app_cons_not_nil : forall A (x y:list A) (a:A), nil <> x ++ a :: y.
Proof.
  admit.
```

delete "admit" and put your proof here.

```
Admitted.
```

when done, replace "Admitted." with "Qed."

Conclusion

I designed this tutorial to give you just enough to get started in Coq. You know enough concepts to have an idea of how things work in Coq and how things relate. You may only know a small number of simple tactics, but they're powerful enough that you can prove real theorems.

As you get more experience, you'll see more complex concepts and use more tactics. You may even get to use SSReflect, which is a separate tactics language! As you read more Coq proofs, you'll also see different styles for writing proofs. (Any expert will tell you that this tutorial's simple style is strikingly different from that used in their proofs.)

I hope you've seen some of the power and possibilities of Coq. Many expert provers consider Coq proofs a puzzle and I wish you luck in solving your puzzles!

Postscripts

Contact

HTML and Coq versions of the tutorial are hosted on the author's professional website.

https://mdnahas.github.io/doc/nahas_tutorial.html https://mdnahas.github.io/doc/nahas_tutorial.v

The author, Michael Nahas, can be reached at michael@nahas.com

Further Reading

Coq uses a lot of concepts that I didn't explain. Below are some good sources on those topic.

For learning about proofs, I recommend the Pulitzer Prize winning book "Godel, Escher, Bach", Chapters 7 and 8. (Pages 181 to 230 in my copy.)

For learning about some of the ideas used in formal proofs, you can read about Intuitionistic logic, the Curry-Howard correspondence, and the BHK-Interpretation.

http://en.wikipedia.org/wiki/Intuitionistic_logic

http://en.wikipedia.org/wiki/Curry-Howard_correspondence

http://en.wikipedia.org/wiki/BHK_interpretation

If you have a deep interest in logic, I highly recommend Gentzen's "Investigations into Logical Deductions". It's not necessary at all, but it is a beautiful paper.

If you need to learn a programming language, OCaml is the one that would help you with Coq the most. Coq is written in OCaml and it borrows a lot of OCaml's syntax and style.

https://ocaml.org

To learn more about Coq, I found the textbook "Software Foundations" readable. It focuses on proving programs correct. You can also look at Coq's documentation webpage.

https://softwarefoundations.cis.upenn.edu/

https://coq.inria.fr/documentation

Vernacular commands

- "Theorem" starts a proof.
- "Qed" ends a proof.
- "Admitted" ends an incomplete proof.
- "Definition" declares a function.
- "Fixpoint" declares a recursive function.
- "Inductive" declares data types.
- "Notation" creates new operators.
- "Infix" also creates new operators.
- "Show Proof" prints out the current state of the proof.
- "Require Import" reads in definitions from a file.
- "Check" prints out a description of a type.
- "Compute" prints out the result of a function call.

The tactic guide

RULE: If the subgoal starts with "(forall <name> : <type>, ..." Then use tactic "intros <name>.".

RULE: If the subgoal starts with "<type> -> ..." Then use tactic "intros <name>.".

RULE: If the subgoal matches an hypothesis, Then use tactic "exact <hyp_name>.".

RULE: If you have an hypothesis "<hyp_name>: <type1> -> <type2> -> ... -> <result_type>" OR an hypothesis "<hyp_name>: (forall <obj1>:<type1>, (forall <obj2>:<type2>, ... <result_type> ...))" OR any combination of "->" and "forall", AND you have hypotheses of type "type1", "type2"..., Then use tactic "pose" to create something of type "result_type".

RULE: If you have subgoal "<goal_type>" AND have hypothesis "<hyp_name>: <type1> -> <type2> -> ... -> <typeN> -> <goal_type>", Then use tactic "refine (<hyp_name> _ ...)." with N underscores.

RULE: If your subgoal is "True", Then use tactic "exact I.".

RULE: If your subgoal is "~<type>" or "~(<term>)" or "(not <term>)", Then use tactic "intros".

RULE: If any hypothesis is "<name> : False", Then use tactic "case <name>.".

RULE: If the current subgoal contains a function call with all its arguments, Then use the tactic "simpl.".

RULE: If there is a hypothesis "<name>" of a created type AND that hypothesis is used in the subgoal, Then you can try the tactic "case <name>.".

RULE: If the subgoal's top-most term is a created type, Then use "refine (<name_of_constructor> _ ...).".

RULE: If a hypothesis "<name>" is a created type with only one constructor, Then use "destruct <name> as `<arg1> <arg2> ...` ." to extract its arguments.

RULE: If a hypothesis "<name>" contain a function call with all its arguments, Then use the tactic "simpl in <name>.".

RULE: If you have a subgoal that you want to ignore for a while, Then use the tactic "admit.".

RULE: If the current subgoal starts "exists <name>, ..." Then create a witness and use "refine (ex_intro _ witness _).".

RULE: If you have a hypothesis "<name> : <a> = <b>" AND "<a>" in your current subgoal Then use the tactic "rewrite <name>.".

RULE: If you have a hypothesis "<name> : <a> = <b>" AND "<b>" in your current subgoal Then use the tactic "rewrite <- <name>.".

RULE: If you have a hypothesis "<name> : (<constructor1> ...) = (<constructor2> ...) OR "<name> : <constant1> = <constant2> Then use the tactic "discriminate <name>.".

RULE: If there is a hypothesis "<name>" of a created type AND that hypothesis is used in the subgoal, AND the type has a recursive definition Then you can try the tactic "elim <name>.".

This page has been generated by coqdoc