Library Prop

Propositional Logic

```
Section prop.
```

A proposition is a definitive statement which we may be able to prove. In Coq we write `P : Prop` to express that `P` is a proposition.

We will later introduce ways to construct interesting propositions, but in the moment we will use propositional variables instead. We declare in Coq:

```
Variables P Q R : Prop.
```

This means that the `P`,`Q`,`R` are atomic propositions which may be substituted by any concrete propositions. In the moment it is helpful to think of them as statements like "The sun is shining" or "We go to the zoo."

We are going to introduce a number of connectives and logical constants to construct propositions:

- Implication `->`, read `P -> Q` as if `P` then `Q`.
- Conjunction `/₩`, read `P /₩ Q` as `P` and `Q`.
- Disjunction `₩/`, read `P ₩/ Q` as `P` or `Q`.
- `False`, read `False` as "Pigs can fly".
- `True`, read `True` as "It sometimes rains in England."
- Negation `~`, read `~ P` as not `P`. We define `~ P` as `P -> False`.
- Equivalence, `<->`, read `P <-> Q` as `P` is equivalent to `Q`. We define `P <-> Q` as `(P -> Q) /₩ (Q -> P)`.

As in algebra we use parentheses to group logical expressions. To save parentheses there are a number of conventions:

- Implication is right associative, i.e. we read `P -> Q -> R` as `P -> (Q -> R)`.
- Implication and equivalence bind weaker than conjunction and disjunction. E.g. we read `P ₩/ Q -> R` as `(P ₩/ Q) -> R`.
- Conjunction binds stronger than disjunction. E.g. we read `P /₩ Q ₩/ R` as `(P /₩ Q) ₩/ R`.
- Negation binds stronger than all the other connectives, e.g. we read `~ P /₩ Q` as `(~ P) /₩ Q`.

This is not a complete specification. If in doubt use parentheses.

We will now discuss how to prove propositions in Coq. If we are proving a statement containing propositional variables then this means that the statement is true for all replacements of the variables with actual propositions. We say it is a tautology.

Our first proof

We start with a very simple tautology `P -> P`, i.e. if `P` then `P`. To start a proof we write:

```
Lemma I : P -> P.
```

It is useful to run the source of this document in Coq to see what happens. Coq enters a proof state and shows what we are going to prove under what assumptions. In the moment our assumptions are that `P`,`Q`,`R` are propositions and our goal is `P -> P`. To prove an implication we add the left hand side to the assumptions and continue to prove the right hand side - this is done using the `intro` tactic. We also choose a name for the assumption, let's call it `p`.

```
intro p.
```

This changes the proof state: we now have to prove `P` but we also have a new assumption `p : P`. We can finish the proof by using this assumption. In Coq this can done by using the `exact` tactic.

```
exact p.
```

This finishes the proof. We only have to instruct Coq to save the proof under the name we have indicated in the beginning, in this case `I`.

```
Qed.
```

Qed stands for "Quod erat demonstrandum". This is Latin for "What was to be shown."

Using assumptions.

Next we will prove another tautology, namely `(P -> Q) -> (Q -> R) -> P -> R`. Try to understand why this is intuitively true for any propositions `P`,`Q` and `R`.

To prove this in Coq we need to know how to use an implication which we have assumed. This can be done using the `apply` tactic: if we have assumed `P -> Q` and we want to prove `Q` then we can use the assumption to reduce (hopefully) the problem to proving `P`. Clearly, using this step is only sensible if `P` is actually easier to prove than `Q`. Step through the next proof to see how this works in practice!

```
Lemma C : (P -> Q) -> (Q -> R) -> P -> R.
```

We have to prove an implication, hence we will be using `intro`. Because `->` is right associative the proposition can be written as `(P -> Q) -> ((Q -> R) -> P -> R)`. Hence we are going to assume `P -> Q`.

```
intro pq.
```

we continue assuming...

```
intro qr.
intro p.
```

Now we have three assumptions `P -> Q`, `Q -> R` and `P`. It remains to prove `R`. We cannot use `intro` any more because our goal is not an implication. Instead we need to use our assumptions. The only assumption which could help us to prove `R` is `Q -> R`. We use the `apply` tactic.

```
apply qr.
```

Apply uses `Q -> R` to reduce the problem to prove `R` to the problem to prove `Q`. Which in turn can be further reduced to proving `P` using `P -> Q`.

```
apply pq.
```

And now it only remains to prove `P` which is one of our assumptions - hence we can use `exact` again.

```
exact p.
Qed.
```

Introduction and Elimination

We observe that there are two types of proof steps (tactics):

- introduction: How can we prove a proposition? In the case of an implication this is `intro`. To prove `P -> Q`, we assume `P` and prove `Q`.
- elimination: How can we use an assumption? In the case of implication this is `apply`. If we know `P -> Q` and we want to prove `Q` it is sufficient to prove `P`.

Actually `apply` is a bit more general: if we know `P1 -> P2 -> ... -> Pn -> Q` and we want to prove `Q` then it is sufficient to prove `P1`,P2,...,Pn. Indeed the distinction of introduction and elimination steps is applicable to all the connectives we are going to encounter. This is a fundamental symmetry in reasoning.

There is also a 3rd kind of steps: structural steps. An example is `exact` which we can use when we want to refer to an assumption. We can also use `assumption` then we don't even have to give the name of the assumption.

If we want to combine several `intro` steps we can use `intros`. We can also use `intros` without parameters in which case Coq does as many `intro` as possible and invents the names itself.

Conjunction

How to prove a conjunction? To prove `P /₩ Q` we need to prove `P` and `Q`. This is achieved using the `split` tactic. We look at a simple example.

```
Lemma pair : P -> Q -> P /₩ Q.
```

On the top level we have to prove an implication.

```
intros p q.
```

now to prove `P /₩ Q` we use `split`.

```
split.
```

This creates two subgoals. We do the first

```
exact p.
```

And then the 2nd

```
exact q.
Qed.
```

How do we use an assumption `P /₩ Q`. We use `destruct` to split it into two assumptions. As an example we prove that `P /₩ Q -> Q /₩ P`.

```
Lemma andCom : P /₩ Q -> Q /₩ P.
intro pq.
destruct pq as [p q].
split.
```

Now we need to use the assumption `P /₩ Q`. We destruct it into two assumptions: `P` and `Q`. `destruct` allows us to name the new assumptions.

```
exact q.
exact p.
```

```
Qed.
```

Can you see a shorter proof of the same theorem ?

To summarize for conjunction we have:

- introduction: `split`: to prove `P /₩ Q` we prove both `P` and `Q`.
- elimination: `destruct`: to prove something from `P /₩ Q` we prove it from assuming both `P` and `Q`.

The currying theorem

Maybe you have already noticed that a statement like `P -> Q -> R` basically means that `R` can be proved from assuming both `P` and `Q`. Indeed, it is equivalent to `P /₩ Q -> R`. We can show this formally by using `<->` for the first time.

All the steps we have already explained so I won't comment. It is a good idea to step through the proof using Coq.

```
Lemma curry : (P /₩ Q -> R) <-> (P -> Q -> R).
unfold iff.
split.
intros H p q.
apply H.
split.
exact p.
exact q.
intros pqr pq.
apply pqr.
destruct pq as [p q].
exact p.
destruct pq as [p q].
exact q.
Qed.
```

I call this the currying theorem, because this is the logical counterpart of currying in functional programming: i.e. that a function with several parameters can be reduced to a function which returns a function. So in Haskell addition has the type `Int -> Int -> Int`.

Disjunction

To prove a disjunction like `P ₩/ Q` we can either prove `P` or `Q`. This is done via the tactics `left` and `right`. As an example we prove `P -> P ₩/ Q`.

```
Lemma inl : P -> P ₩/ Q.
intros p.
```

Clearly, here we have to use `left`.

```
left.
exact p.
Qed.
```

To use a disjunction `P ₩/ Q` to prove something we have to prove it from both `P` and `Q`. The tactic we use is also called `destruct` but in this case `destruct` creates two subgoals. This can be compared to case analysis in functional programming. Indeed we can prove the following theorem.

```
Lemma case : P ₩/ Q -> (P -> R) -> (Q -> R) -> R.
intros pq pr qr.
destruct pq as [p | q].
```

The syntax for `destruct` for disjunction is different if we want to name the assumption we have to separate them with `|`. Indeed each of them will be visible in a different part of the proof. First we assume `P`.

```
apply pr.
exact p.
```

And then we assume `Q`

```
apply qr.
exact q.
Qed.
```

So again to summarize: For disjunction we have:

- introduction: there are two ways to prove a disjunction `P ₩/ Q`. We use `left` to prove it from `P` and `right` to prove it from `Q`.
- elimination: If we have assumed `P ₩/ Q` then we can use `destruct` to prove our current goal from assuming `P` and from assuming `Q`.

Distributivity

As an example of how to combine the proof steps for conjunction and disjunction we show that distributivity holds, i.e. `P /₩ (Q ₩/ R)` is logically equivalent to `(P /₩ Q) ₩/ (P /₩ R)`. This is reminiscent of the principle in algebra that `x * (y + z) = x * y + x * z`.

```
Lemma andOrDistr : P /₩ (Q ₩/ R)
```

```
              <-> (P /₩ Q) ₩/ (P /₩ R).
split.
intro pqr.
destruct pqr as [p qr].
destruct qr as [q | r].
left.
split.
exact p.
exact q.
right.
split.
exact p.
exact r.
intro pqpr.
destruct pqpr as [pq | pr].
split.
destruct pq as [p q].
exact p.
left.
destruct pq as [p q].
exact q.
destruct pr as [p r].
split.
exact p.
right.
exact r.
Qed.
```

As before: to understand the working of this script it is advisable to step through it using Coq.

True and False

`True` is just a conjunction with no arguments as opposed to `/₩` which has two. Similarity `False` is a disjunction with no arguments. As a consequence we already know the proof rules for `True` and `False`.

We can prove `True` without any assumptions.

```
Lemma triv : True.
split.
```

Here we split but instead of two subgoals we get none.

```
Qed.
```

On the other had we can prove anything from `False`. This is called "ex falso quod libet" in Latin.

```
Lemma exFalso : False -> P.
intro f.
destruct f.
```

Here instead of two subgoals we get none.

```
Qed.
```

In terms of introduction and elimination steps we may summarize:

- True: There is one introduction rule but no elimination.
- False: There is one elimination rule but no introduction.

Negation

`~ P` is defined as `P -> False`. Using this we can establish some basic theorems about negation. First we show that we cannot have both `P` and `~ P`, that is we prove `~ (P /₩ ~ P)`.

```
Lemma incons : ~ ( P /₩ ~ P).
unfold not.
intro h.
destruct h as [p np].
apply np.
exact p.
Qed.
```

Another example is to show that `P` implies `~ ~ P`.

```
Lemma p2nnp : P -> ~ ~ P.
unfold not.
intros p np.
apply np.
exact p.
Qed.
```

Classical Reasoning

You may expect that we can also prove the other direction `~ ~ P -> P` and that indeed `P <-> ~ ~ P`. We can reason that `P` is either `True` or `False` and in both cases `~ ~ P` will be the same. However, this reasoning is not possible using the principles we have introduced so far. The reason is that Coq is based on intuitionistic logic, and the above proposition is not provable intuitionistically.

However, we can use an additional axiom, which corresponds to the principle that every proposition is either `True` or `False`, this is the Principle of the Excluded Middle `P ₩/ ~ P`. In Coq this can be achieved by:

```
Require Import Coq.Logic.Classical.
```

This means we are now using Classical Logic instead of Intuitionistic Logic. The only difference is that we have an axiom `classic` which proves the principle of the excluded middle for any proposition. We can use this to prove `~ ~ P -> P`.

```
Lemma nnpp : ~~P -> P.
 intro nnp.
```

Here we use a particular instance of `classic` for `P`.

```
destruct (classic P) as [p | np].
```

First case `P` holds

```
exact p.
```

2nd case `~ P` holds. Here we appeal to `exFalso`.

```
apply exFalso.
```

Notice that we have shown `exFalso` only for `P`. We should have shown it for any proposition but this would involve quantification over all propositions and we haven't done this yet.

```
apply nnp.
exact np.
Qed.
```

Unless stated otherwise we will try to prove propositions intuitionsitically, that is without using `classic`. An intuitionistic proof provides a positive reason why something is true, while a classical proof may be quite indirect and not so easily acceptable intuitively. Another advantage of intuitionistic reasoning is that it is constructive, that is whenever we prove the existence of a certain object we can also explicitly construct it. This is not true in intuitionistic logic. Moreover, in intuitionistic logic we can make differences which disappear when using classical logic. For example we can explicit state when a property is decidable, i.e. can be computed by a computer program.

The cut rule

This is a good point to introduce another structural rule: the cut rule. Cutting a proof means to introduce an intermediate goal, then you prove your current goal from this intermediate goal, and you prove theintermediate goal. This is particularly useful when you use the intermediate goal several times.

In Coq this can be achieved by using assert. assert h : H introduces H as a new subgoal and after you have proven this you can use an assumption h : H to prove your original goal.

The following (artificial) example demonstrates the use of assert.

```
Lemma usecut : (P /₩ ~P) -> Q.
intro pnp.
```

If we had a generic version of exFalso we could use this. Instead we can introduce False as an intermediate goal.

```
assert (f : False).
```

which is easy to prove

```
destruct pnp as [p np].
apply np.
exact p.
```

and using False it is easy to prove Q.

```
destruct f.
Qed.
```

This example also shows that sometimes we have to cut (i.e. use assert) to prove something.

Index

This page has been generated by coqdoc

Library Pred

Predicate Logic

```
Section pred.
```

Predicate logic extends propositional logic: we can talk about sets of things, e.g. numbers and define properties, called predicates and relations. We will soon define some useful sets and ways to define sets but for the moment, we will use set variables as we have used propositional variables before.

In Coq we can declare set variables the same way as we have declared propositional variables:

```
Variables A B : Set.
```

Thus we have declared A and B to be variables for sets. For example think of A=the set of students and B= the set of modules. That is any tautology using set variable remains true if we substitute the set variables with any conrete set (e.g. natural numbers or booleans, etc).

Next we also assume some predicate variables, we let P and Q be properties of A (e.g. P x may mean P is clever and Q x means x is funny).

```
Variables P Q : A -> Prop.
```

Coq views these predicates as functions from `A` to `Prop`. That is if we have an element of `A`, e.g. `a : A`, we can apply `P` to a by writing `P a` to express that `a` has the property `P`.

We can also have properties relating several elements, possibly of different sets, these are usually called *relations*. We introduce a relation `R`, relating `A` and `B` by:

```
Variable R : A -> B -> Prop.
```

E.g. R could be the relation "attends" and we would write "R jim g52ifr" to express that Jim attends g52ifr.

Universal quantification

To say all elements of A have the property P, we write `forall x:A, P x` more general we can form `forall x:A, PP` where `PP` is a proposition possibly containing the variable `x`. Another example is `forall x:A,P x -> Q x` meaning that any element of `A` that has the property `P` will also have the property `Q`. In our example that would mean that any clever student is also funny.

As an example we show that if all elements of `A` have the property `P` and that if whenever an element of `A` has the property `P` has also the property `Q` then all alements of `A` have the property `Q`. That is if all students are clever, and every clever student is funny, then all students are funny. In predicate logic we write `forall( x:A,P x) -> forall( x:A,P x -> Q x) -> forall x:A, Q x`.

We introduce some new syntactic conventions: the scope of an forall always goes as far as possible. That is we read `forall x:A,P x /₩ Q` as `forall x:A, (P x /₩ Q)`. Given this could we have saved any parentheses in the example above without changing the meaning?

As before we use introduction and elimination steps. Maybe surprisingly the tactics for implication and universal quantification are the same. The reason is that in Coq's internal language implication and universal quantification are actually the same.

```
Lemma AllMono : (forall x:A,P x) -> (forall x:A,P x -> Q x) -> forall x:A, Q x.
 intros H1 H2.
```

To prove `forall x:A,Q x` assume that there is an element `a:A` and prove `Q a` We use `intro a` to do this.

```
 intro a.
```

If we know `H2 : forall x:A,P x -> Q x` and we want to prove `Q a` we can use `apply H2` to instantiate the assumption to `P a -> Q a` and at the same time eliminate the implication so that it is left to prove `P a`.

```
apply H2.
```

Now if we know `H1 : forall x:A,P x` and we want to show `P a`, we use `apply H1` to prove it. After this the goal is completed.

```
apply H1.
```

In the last step we only instantiated the universal quantifier.

```
Qed.
```

So to summarize:

- introduction for `forall`: To show `forall x:A,P x` we say `intro a` which introduces an assumption `a:A` and it is left to show `P` where each free occurence of `x` is replaced by `a`.
- elimination for `forall`: We only describe the simplest case: If we know `H : forall x:A,P` and we want to show P where `x` is replaced by `a` we use `apply H` to prove `P`

a.

When I say that each free occurence of x in the proposition P is replaced by a, I mean that occurences of x which are in the scope of another quantifier (these are called bound) are not affected. E.g. if P is `Q x /₩ forall x:A,R x x` then the only free occurence of x is the one in `Q x`. That is we obtain `Q a /₩ forall x:A,R x x`. The occurences of x in `forall x:A,R x x` are bound.

We can also use `intros` here. That is if the current goal is `forall x:A,P x -> Q x` then `intros x P` will introduce the assumptions `x:A` and `H:P x`.

The general case for `apply` is a bit hard to describe. Basically apply may introduce several subgoals if the assumption has a prefix of `forall` and `->`. E.g. if we have assumed `H : forall x:Aforall, y:B,P x -> Q y -> R x y` and our current goal is `R a b` then `apply H` will instantiate x with a and y with b and generate the new goals `Q b` and `R a b`.

Next we are going to show that `forall` *commutes with* `/₩`. That is we are going to show `forall( x:A,P x /₩ Q x) <-> forall( x:A, P x) /₩ forall( x:A, Q x)` that is "all students are clever and funny" is equivalent to "all students are clever" and "all students are funny".

```
Lemma AllAndCom : (forall x:A,P x /₩ Q x) <-> (forall x:A, P x) /₩ (forall x:A, Q x).
split.
```

Proving ->

```
intro H.
split.
intro a.
assert (pq : P a /₩ Q a).
apply H.
destruct pq as [p q].
exact p.
intro a.
assert (pq : P a /₩ Q a).
apply H.
destruct pq as [p q].
exact q.
```

Proving <-

```
intro H.
destruct H as [p q].
intro a.
split.
apply p.
apply q.
Qed.
```

This proof is quite lengthy and I even had to use `assert`. There is a shorter proof, if we use `edestruct` instead of `destruct`. The "e" version of tactics introduce metavariables (visible as ?x) which are instantiated when we are using them. See the Coq reference manual for details.

I only do the `->` direction using `edestruct`, the other one stays the same.

```
Lemma AllAndComE : (forall x:A,P x /₩ Q x) -> (forall x:A, P x) /₩ (forall x:A, Q x).
```

Proving `->`

```
intro H.
split.
intro a.
edestruct H as [p q].
apply p.
intro a.
edestruct H as [p q].
apply q.
Qed.
```

Question: Does `forall` also commute with `₩/`? That is does `forall( x:A,P x ₩/ Q x) <-> forall( x:A, P x) ₩/ forall( x:A, Q x)` hold? If not, how can you show that?

Existential quantification

To say that there is an element of A having the property P, we write `exists x:A, P x` more general we can form `exists x:A, PP` where `PP` is a proposition possibly containing the variable `x`. Another example is `exists x:A,P x /₩ Q x` meaning that there is an element of `A` that has the property `P` and the property `Q`. In our example that would mean that there is a student who is both clever and funny.

As an example we show that if there is an element of `A` having the property `P` and that if whenever an element of `A` has the property `P` has also the property `Q` then there is an elements of `A` having the property `Q`. That is if there is a clever student, and every clever student is funny, then there is a funny student. In predicate logic we write `(exists x:A,P x) -> forall( x:A,P x -> Q x) -> exists x:A, Q x`.

Btw, we are not changing the 2nd quantifier, it stays `forall`. What would happen if we would replace it by `exists`?

The syntactic conventions for `exists` are the same as for `forall`: the scope of an `exists` always goes as far as possible. That is we read `exists x:A,P x /₩ Q` as `exists x:A, (P x /₩ Q)`.

The tactics for existential quatification are similar to the ones for conjunction. To prove an existential statement `exists x:A,PP` we use `exists a` where `a : A` is our *witness*. We then have to prove `PP` where each free occurence of `x` is replaced by `a`. To use an assumption `H : exists x:A,PP` we employ `destruct H as [a p]` which destructs `H` into `a : A` and `p : PP'` where `PP'` is `PP` where all free occurences of `x` have been replaced by `a`.

```
Lemma ExistsMono : (exists x:A,P x) -> (forall x:A,P x -> Q x) -> exists x:A, Q x.
 intros H1 H2.
```

We first eliminate or assumption.

```
destruct H1 as [a p].
```

And now we introduce the existential.

```
exists a.
apply H2.
```

In the last step we instantiated a universal quantifier.

```
exact p.
Qed.
```

So to summarize:

- introduction for `exists` To show `exists x:A,P` we say `exists a` where `a : A` is any expression of type `a`. It remains to show `P` where any free occurence of `x` is replaced by `a`.
- elimination for `exists` If we know `H : exists x:A,P` we can use `destruct H as [a p]` which destructs `H` intwo two assumptions: `a : A` and `p : P'` where `P'` is obtained from `P` by replacing all free occurences of `x` in `P` by `a`.

Next we are going to show that `exists` *commutes with* `₩/`. That is we are going to show `(exists x:A,P x ₩/ Q x) <-> (exists x:A, P x) ₩/ (exits x:A, Q x)` that is "there is a student who is clever or funny" is equivalent to "there is a clever student or there is a funny student".

```
Lemma ExOrCom : (exists x:A,P x ₩/ Q x) <-> (exists x:A, P x) ₩/ (exists x:A, Q x).
split.
```

Proving `->`

```
 intro H.
```

It would be too early to use the introduction rules now. We first need to analyze the assumptions. This is a common situation.

```
destruct H as [a pq].
destruct pq as [p | q].
```

First case `P a`.

```
 left.
```

```
exists a.
exact p.
```

Second case `Q a`.

```
right.
exists a.
exact q.
```

Proving `<-`

```
intro H.
destruct H as [p | q].
```

First case `exists x:A,P x`

```
destruct p as [a p].
exists a.
left.
exact p.
```

Second case `exists x:A,Q x`

```
destruct q as [a q].
exists a.
right.
exact q.
Qed.
```

Another Currying Theorem

There is also a currying theorem in predicate logic which exploits the relation between `->` and `forall` on the one hand and `/₩` and exists on the other. That is we can show that `forall x:A,P x -> S` is equivalent to `(exists x:A,P x) -> S`. Intuitively, think of `S` to be "the lecturer is happy". Then the left hand side can be translated as "If there is any student who is clever, then the lecturer is happy" and the right hand side as "If there exists a student who is clever, then the lecturer is happy". The relation to the propositional currying theorem can be seen, when we replace `forall` by `->` and `exists` by `/₩`.

To prove this tautology we assume an additional proposition.

```
Variable S : Prop.

Lemma Curry : (forall x:A,P x -> S) <-> ((exists x:A,P x) -> S).
split.
```

proving `->`

```
intro H.
intro p.
destruct p as [a p].
```

With our limited knowledge of Coq's tactic language we need to instantiate `H` using `assert`. There are better ways to do this... We will see later.

```
assert (H' : P a -> S).
apply H.
apply H'.
exact p.
```

proving `<-`.

```
intro H.
intros a p.
apply H.
exists a.
exact p.
Qed.
```

As before the explicit instantiation using `assert` can be avoided by using the "e" version of a tactic. In this case it is `eapply`. Again, I refer to the Coq reference manual for details. I only do one direction, the other one stays the same.

```
Lemma CurryE : (forall x:A,P x -> S) -> ((exists x:A,P x) -> S).
```

proving `->`

```
intro H.
intro p.
destruct p as [a p].
eapply H.
apply p.
Qed.
```

Equality

Predicate logic comes with one generic relation which is defined for all sets: equality (`=`). Given two expressions `a,b : A` we write `a = b : Prop` for the proposition that `a` and `b` are equal, that is they describe the same object.

How can we prove an equality? That is what is the introduction rule for equality? We can prove that every expression is `a : A` is equal to itself `a = a` using the tactic `reflexivity`. How can we use an assumption `H : a = b`? That is how can we eliminate equality? If we want to prove a goal `P` which contains the expression `a` we can use `rewrite H` to *rewrite* all those `a`s into `b`s.

To demonstrate how to use these tactics we show that equality is an *equivalence relation* that is, it is:

- reflexive (`forall a:A, a = a`)
- symmetric (`forall a b:A, a=b -> b=a`)
- transitive (`forall a b c:A, a=b -> b=c -> a=c`.

```
Lemma eq_refl : forall a:A, a = a.
intro a.
```

Here we just invoke the reflexivity tactic.

```
reflexivity.
Qed.

Lemma eq_sym : forall a b:A, a=b -> b=a.
intros a b H.
```

Here we use rewrite to reduce the goal.

```
rewrite H.
reflexivity.
Qed.

Lemma eq_trans : forall a b c:A, a=b -> b=c -> a=c.
intros a b c ab bc.
rewrite ab.
exact bc.
Qed.
```

Do you know any other equivalence relations?

Classical Predicate Logic

The principle of the excluded middle `classic P : P ₩/ ~P` has many important applications in predicate logic. As an example we show that `exists x:A,P x` is equivalent to `~ forall x:A, ~ P x`.

Instead of using `classic` directly we use the derivable principle `NNPP : ~ ~ P -> P` which is also defined in `Coq.Logic.Classical`.

```
Require Import Coq.Logic.Classical.

Lemma ex_from_forall : (exists x:A, P x) <-> ~ forall x:A, ~ P x.
split.
```

proving `->`

```
intro ex.
intro H.
destruct ex as [a p].
assert (npa : ~ (P a)).
apply H.
apply npa.
exact p.
```

proving <-

```
intro H.
apply NNPP.
```

Instead of proving `exists x:A,P x` which is hard, we show `~~ exists x:A,P x` which is easier.

```
intro nex.
apply H.
intros a p.
apply nex.
exists a.
exact p.
Qed.
```

Index

This page has been generated by coqdoc

Library Bool

Bool

```
Section Bool.
```

Defining bool and operations

We define bool : Set as a finite set with two elements: true : bool and false : bool. In set theoretic notation we would write bool = { true , false }.

The function negb : bool -> bool (boolean negation) can be defined by pattern matching using the match construct.

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

This should be familiar from g51fun - in Haskell match is called case. Indeed Haskell offers a more convenient syntax for top-level pattern.

We can evaluate the function using the slightly lengthy phrase Eval compute in (...):

```
Eval compute in (negb true).
```

The evaluator replaces

negb true

with

match true with | true => false | false => true end.

which in turn evaluates to

false

```
Eval compute in negb (negb true).
```

We know already that `negb true` evaluates to `false` hence `negb (negb true)` evaluates to `negb false` which in turn evaluates to `true`.

Other boolean functions can be defined just as easily:

```
Definition andb(b c:bool) : bool :=
  if b then c else false.

Definition orb (b c : bool) : bool :=
  if b then true else c.
```

The Coq prelude also defines the infix operators && and || for andb and orb respectively, with && having higher precedence than ||. Note however, that you cannot use ! (for negb) since this is used for other purposes in Coq.

Reasoning about Bool

We can now use predicate logic to show properties of boolean functions. As a first example we show that the function `negb` is *idempotent*, that is

```
forall b :bool, negb (negb b) = b
```

To prove this, the only additional thing we have to know is that we can analyze a boolean variable `b : bool` using `destruct b` which creates a case for `b = true` and one for `b = false`.

```
Lemma negb_idem : forall b :bool, negb (negb b) = b.
intro b.
destruct b.
```

Case for `b = true`

Our goal is negb (negb true) = true. As we have already seen `negb (negb true)` evaluates to true. Hence this goal can be proven using `reflexivity`. Indeed, we can make this visible by using `simpl`.

```
simpl.
reflexivity.
```

Case for `b = false`

This case is exactly the same as before.

```
simpl.
reflexivity.
Qed.
```

There is a shorter way to write this proof by using `;` instead of `,` after `destruct`. We can also omit the `simpl` which we only use for cosmetic reasons.

```
Lemma negb_idem' : forall b :bool, negb (negb b) = b.
intro b.
destruct b;
  reflexivity.
Qed.
```

Indeed, proving equalities of boolean functions is very straightforward. All we need is to analyze all cases and then use refl. For example to prove that `andb` is commutative, i.e.

```
forall x y : bool, andb x y = andb y x
```

(we use the abbrevation: `forall x y : A,...` is the same as `forall x:Aforall, y:A, ...`.

```
Lemma andb_comm : forall x y : bool, andb x y = andb y x.
intros x y.
destruct x;
  (destruct y;
     reflexivity).
Qed.
```

We can also prove other properties of `bool` not directly related to the functions, for example, we know that every boolean is either true or false. That is

```
forall b : bool, b = true ₩/ b = false
```

This is easy to prove:

```
Lemma true_or_false : forall b : bool,
       b = true ₩/ b = false.
intro b.
destruct b.
```

b = true

```
left.
reflexivity.
```

b = false

```
right.
reflexivity.
Qed.
```

Next we want to prove something which doesn't involve any quantifiers, namely

```
~ (true = false)
```

This is not so easy, we need a little trick. We need to embed `bool` into `Prop`, mapping `true` to `True` and `false` to `False`. This is achieved via the function Istrue:

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

Lemma diff_true_false :
      ~ (true = false).
intro h.
```

We now need to use a new tactic to replace `False` by `IsTrue False`. This is possible because `IsTrue False` evaluates to `false`. We are using `fold` which is the inverse to `unfold` which we have seen earlier.

```
fold (Istrue false).
```

Now we can simply apply the equation `h` backwards.

```
rewrite<- h.
```

Now by unfolding we can replace `Istrue true` by `True`

```
unfold Istrue.
```

Which is easy to prove.

```
split.
Qed.
```

Actually there is a tactic `discriminate` which implements this proof and which allows us to prove directly that any two different constructors (like `true` and `false`) are different. We shall use `discriminate` in future.

Reflection

We notice that there is a logical operator `/₩` which acts on `Prop` and a boolean operator `andb` (or `&&`) which acts on `bool`. How are the two related?

We can use `/₩` to specify `andb`, namely we say that `andb x y = true` is equivalent to `x = true` and `y = true`. That is we prove:

```
Lemma and_ok : forall x y : bool,
  andb x y = true <-> x = true /₩ y = true.
intros x y.
split.
```

->

```
destruct x.
```

x=true

```
intro h.
split.
reflexivity.
simpl in h.
exact h.
```

Why did the last step work?

x = false

```
intro h.
simpl in h.
discriminate h.
```

<-

```
intro h.
destruct h as [hx hy].
rewrite hx.
exact hy.

Qed.

End Bool.
```

Index

This page has been generated by coqdoc