

SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

[TABLE OF CONTENTS](#)

[INDEX](#)

[ROADMAP](#)

LOGIC

LOGIC IN COQ

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

In previous chapters, we have seen many examples of **factual claims (propositions)** and ways of presenting evidence of their truth (*proofs*). In particular, we have worked extensively with **equality propositions** of the form $e_1 = e_2$, with **implications** ($P \rightarrow Q$), and with **quantified propositions** ($\forall x, P$). In this chapter, we will see how Coq can be used to carry out other familiar forms of logical reasoning.

Before diving into details, let's talk a bit about the status of mathematical statements in Coq. Recall that **Coq is a typed language**, which means that every sensible expression in its world has an associated type. Logical claims are no exception: any statement we might try to prove in Coq has a type, namely **Prop**, the type of *propositions*. We can see this with the `Check` command:

```
Check 3 = 3.
(* ==> Prop *)
```



```
Check ∀ n m : nat, n + m = m + n.
(* ==> Prop *)
```

Note that *all* syntactically well-formed propositions have type **Prop** in Coq, regardless of whether they are true.

Simply *being* a proposition is one thing; being **provable** is something else!

```
Check 2 = 2.
(* ==> Prop *)
```



```
Check ∀ n : nat, n = 2.
(* ==> Prop *)
```



```
Check 3 = 4.
(* ==> Prop *)
```

Indeed, propositions don't just have types: they are *first-class objects* that can be manipulated in the same ways as the other entities in Coq's world.

So far, we've seen one primary place that propositions can appear: in **Theorem** (and **Lemma** and **Example**) declarations.

```
Theorem plus_2_2_is_4 :
  2 + 2 = 4.
Proof. reflexivity. Qed.
```

But propositions can be used in many other ways. For example, we can give a name to a proposition using a **Definition**, just as we have given names to expressions of other sorts.

```
Definition plus_fact : Prop := 2 + 2 = 4.
Check plus_fact.
(* ==> plus_fact : Prop *)
```

We can later use this name in any situation where a proposition is expected — for example, as the claim in a **Theorem** declaration.

```
Theorem plus_fact_is_true :
  plus_fact.
Proof. reflexivity. Qed.
```

We can also write *parameterized propositions* — that is, functions that take arguments of some type and return a proposition.

For instance, the following function takes a number and returns a proposition asserting that this number is equal to three:

```
Definition is_three (n : nat) : Prop :=
  n = 3.
Check is_three.
(* ==> nat -> Prop *)
```

In Coq, functions that return propositions are said to define *properties* of their arguments.

For instance, here's a (polymorphic) property defining the familiar notion of an *injective function*.

```
Definition injective {A B} (f : A → B) :=
  ∀ x y : A, f x = f y → x = y.

Lemma succ_inj : injective S.
Proof.
  intros n m H. inversion H. reflexivity.
Qed. Inductive eq (A : Type) (x : A) : A -> Prop := eq_refl : x = x
```

how exactly?

The equality operator `=` is also a function that returns a `Prop`.

The expression `n = m` is syntactic sugar for `eq n m` (defined using Coq's Notation mechanism). Because `eq` can be used with elements of any type, it is also polymorphic:

```
Check @eq.
(* ==> forall A : Type, A -> A -> Prop *)
```

(Notice that we wrote `@eq` instead of `eq`: The type argument `A` to `eq` is declared as implicit, so we need to turn off implicit arguments to see the full type of `eq`.)

Logical Connectives

Conjunction

The *conjunction*, or *logical and*, of propositions `A` and `B` is written `A \wedge B`, representing the claim that both `A` and `B` are true.

```
Example and_example : 3 + 4 = 7  $\wedge$  2 * 2 = 4.
```

To prove a conjunction, use the `split` tactic. It will generate two subgoals, one for each part of the statement:

```
Proof.
(* WORKED IN CLASS *)
split.
- (* 3 + 4 = 7 *) reflexivity.
- (* 2 + 2 = 4 *) reflexivity.
Qed.
```

For any propositions `A` and `B`, if we assume that `A` is true and we assume that `B` is true, we can conclude that `A \wedge B` is also true.

```
Lemma and_intro :  $\forall A B : \text{Prop}, A \rightarrow B \rightarrow A \wedge B$ .
Proof.
intros A B HA HB. split.
- apply HA.
- apply HB.
Qed.
```

Since applying a theorem with hypotheses to some goal has the effect of generating as many subgoals as there are hypotheses for that theorem, we can apply `and_intro` to achieve the same effect as `split`.

```
Example and_example' : 3 + 4 = 7  $\wedge$  2 * 2 = 4.
Proof.
apply and_intro.
- (* 3 + 4 = 7 *) reflexivity.
- (* 2 + 2 = 4 *) reflexivity.
Qed.
```

Exercise: 2 stars (and exercise)

```
Example and_exercise :
 $\forall n m : \text{nat}, n + m = 0 \rightarrow n = 0 \wedge m = 0$ .
Proof.
(* FILL IN HERE *) Admitted.
```



why doesn't this work? rewrite `<- plus_comm, plus_n_Sm` in H.

So much for proving conjunctive statements. To go in the other direction — i.e., to use a conjunctive hypothesis to help prove something else — we employ the `destruct` tactic.

If the proof context contains a hypothesis H of the form $A \wedge B$, writing `destruct H` as `[HA HB]` will remove H from the context and add two new hypotheses: HA , stating that A is true, and HB , stating that B is true.

```
Lemma and_example2 :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  (* WORKED IN CLASS *)
  intros n m H.
  destruct H as [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

As usual, we can also destruct H right when we introduce it, instead of introducing and then destructing it:

```
Lemma and_example2' :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  intros n m [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

You may wonder why we bothered packing the two hypotheses $n = 0$ and $m = 0$ into a single conjunction, since we could have also stated the theorem with two separate premises:

```
Lemma and_example2'' :
  ∀ n m : nat, n = 0 → m = 0 → n + m = 0.
Proof.
  intros n m Hn Hm.
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

For this theorem, both formulations are fine. But it's important to understand how to work with conjunctive hypotheses because conjunctions often arise from intermediate steps in proofs, especially in bigger developments. Here's a simple example:

```
Lemma and_example3 :
  ∀ n m : nat, n + m = 0 → n * m = 0.
Proof.
  (* WORKED IN CLASS *)
  intros n m H.
  assert (H' : n = 0 ∧ m = 0).
  { apply and_exercise. apply H. }
  destruct H' as [Hn Hm].
  rewrite Hn. reflexivity.
Qed.
```

Another common situation with conjunctions is that we know $A \wedge B$ but in some context we need just A (or just B). The following lemmas are useful in such cases:

```
Lemma proj1 : ∀ P Q : Prop,
  P ∧ Q → P.
Proof.
  intros P Q [HP HQ].
  apply HP. Qed.
```

Exercise: 1 star, optional (proj2)

```
Lemma proj2 : ∀ P Q : Prop,
  P ∧ Q → Q.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Finally, we sometimes need to rearrange the order of conjunctions and/or the grouping of multi-way conjunctions. The following commutativity and associativity theorems are handy in such cases.

```
Theorem and_commut : ∀ P Q : Prop,
  P ∧ Q → Q ∧ P.
Proof.
  intros P Q [HP HQ].
  split.
  - (* left *) apply HQ.
  - (* right *) apply HP. Qed.
```

Exercise: 2 stars (and_assoc)

(In the following proof of associativity, notice how the *nested* `intros` pattern breaks the hypothesis $H : P \wedge (Q \wedge R)$ down into $HP : P$, $HQ : Q$, and $HR : R$. Finish the proof from there.)

```
Theorem and_assoc : ∀ P Q R : Prop,
  P ∧ (Q ∧ R) → (P ∧ Q) ∧ R.
Proof.
  intros P Q R [HP [HQ HR]].
  (* FILL IN HERE *) Admitted.
```

□

By the way, the `infix notation` `∧` is actually just syntactic sugar for `and A B`. That is, `and` is a Coq operator that takes two propositions as arguments and yields a proposition.

```
Check and.
(* ==> and : Prop → Prop → Prop *)
```

Disjunction

Another important connective is the *disjunction*, or `logical or`, of two propositions: $A \vee B$ is true when either A or B is. (Alternatively, we can write `or A B`, where `or : Prop → Prop → Prop`.)

To use a disjunctive hypothesis in a proof, we proceed by case analysis, which, as for nat or other data types, can be done with `destruct` or `intros`. Here is an example:

```
Lemma or_example : 
  ∀ n m : nat, n = 0 ∨ m = 0 → n * m = 0.
Proof.
  (* This pattern implicitly does case analysis on
   n = 0 ∨ m = 0 *)
  intros n m [Hn | Hm].
  - (* Here, n = 0 *)
    rewrite Hn. reflexivity.
  - (* Here, m = 0 *)
    rewrite Hm. rewrite <- mult_n_0.
    reflexivity.
Qed.
```

Conversely, to show that a disjunction holds, we need to show that one of its sides does. This is done via two tactics, `left` and `right`. As their names imply, the first one requires proving the left side of the disjunction, while the second requires proving its right side. Here is a trivial use...

```
Lemma or_intro : ∀ A B : Prop, A → A ∨ B.
Proof.
  intros A B HA.
  left.
  apply HA.
Qed.
```

... and a slightly more interesting example requiring both `left` and `right`:

```
Lemma zero_or_succ :
  ∀ n : nat, n = 0 ∨ n = S (pred n).
Proof.
  (* WORKED IN CLASS *)
  intros [|n].
  - left. reflexivity.
  - right. reflexivity.
Qed.
```

Falsehood and Negation

So far, we have mostly been concerned with proving that certain things are *true* — addition is commutative, appending lists is associative, etc. Of course, we may also be interested in *negative* results, showing that certain propositions are *not* true. In Coq, such negative statements are expressed with the negation operator \neg .

To see how negation works, recall the discussion of the *principle of explosion* from the `Tactics` chapter; it asserts that, if we assume a contradiction, then any other proposition can be derived. Following this intuition, we could define $\neg P$ ("not P ") as $\forall Q, P \rightarrow Q$. Coq actually makes a slightly different choice, defining $\neg P$ as $P \rightarrow \text{False}$, where `False` is a specific contradictory proposition defined in the standard library.

```
Module MyNot.
```

```
Definition not (P:Prop) := P → False.

Notation "¬ x" := (not x) : type_scope.

Check not.
(* ==> Prop -> Prop *)

End MyNot.
```

Since `False` is a contradictory proposition, the principle of explosion also applies to it. If we get `False` into the proof context, we can use `destruct` (or `inversion`) on it to complete any goal:

<https://stackoverflow.com/questions/50600673/using-destruct-instead-of-inversion>

```
Theorem ex_falso_quodlibet : ∀ (P:Prop),
  False → P.
Proof.
  (* WORKED IN CLASS *)
  intros P contra.
  destruct contra. Qed.
```

`False` is an empty inductive data type, i.e. it has no possible values, see here. `True` is an inductive data type with a single value `I`. When we `destruct` a value `X` of inductive data type we replace the current goal with multiple subgoals, one per each possible value of `X`.

When we `destruct` `False`, we end up with zero subgoals to prove (as it has zero values), therefore proof is completed.

The Latin *ex falso quodlibet* means, `Basically, destruct and inversion do more or less the same thing here.` this is another common name for the principle of explosion.

Exercise: 2 stars, optional (not implies our not)

`False` is the always false proposition
`Inductive False : Prop :=`

Show that Coq's definition of negation implies the int this def is basically similar to unit in haskell where you could have no data constructor!!!

```
Fact not_implies_our_not : ∀ (P:Prop),
  ¬ P → (∀ (Q:Prop), P → Q).
Proof.
  (* FILL IN HERE *) Admitted.
```

think of it in terms of curry-howard thing,
if you have a value for false you
basically say that you can
prove thiis proposition (type) and
that's why it doesn't have a value

This is how we use `not` to state that 0 and 1 are different elements of `nat`:

```
Theorem zero_not_one : ¬(0 = 1).
Proof.
  intros contra. inversion contra.
Qed.
```

Such inequality statements are frequent enough to warrant a special notation, $x \neq y$:

```
Check (0 ≠ 1).
(* ==> Prop *)

Theorem zero_not_one' : 0 ≠ 1.
Proof.
  intros H. inversion H.
Qed.
```

It takes a little practice to get used to working with negation in Coq. Even though you can see perfectly well why a statement involving negation is true, it can be a little tricky at first to get things into the right configuration so that Coq can understand it! Here are proofs of a few familiar facts to get you warmed up.

```
Theorem not_False :
  ¬ False.
```

Proof.unfold `not`. intros `H`. destruct `H`. Qed.

why not use apply `H` instead of `destruct H`?
since that works too!!
same reason as the prev page
DON'T ASK!!

Theorem contradiction_implies_anything : $\forall P Q : \text{Prop}, (P \wedge \neg P) \rightarrow Q$.

Proof.

```
(* WORKED IN CLASS *)
intros P Q [HP HNA]. unfold not in HNA.
apply HNA in HP. destruct HP. Qed.
```

Theorem double_neg : $\forall P : \text{Prop}, P \rightarrow \neg\neg P$.

Proof.

```
(* WORKED IN CLASS *)
intros P H. unfold not. intros G. apply G. apply H. Qed.
```

Maybe LATER!!!!

Exercise: 2 stars, advanced, recommended (double neg inf)Write an informal proof of `double_neg`:*Theorem: P implies $\neg\neg P$, for any proposition P .*`(* FILL IN HERE *)`

□

Exercise: 2 stars, recommended (contrapositive)

Theorem contrapositive : $\forall (P Q : \text{Prop}), (P \rightarrow Q) \rightarrow (\neg Q \rightarrow \neg P)$.

Proof.`(* FILL IN HERE *) Admitted.`

□

Exercise: 1 star (not both true and false)

Theorem not_both_true_and_false : $\forall P : \text{Prop}, \neg (P \wedge \neg P)$.

Proof.`(* FILL IN HERE *) Admitted.`

□

Exercise: 1 star, advanced (informal not PNP)

Maybe LATER!!!!

`Definition informal_not_PNP_TODO := 0.`Write an informal proof (in English) of the proposition $\forall P : \text{Prop}, \neg (P \wedge \neg P)$.`(* FILL IN HERE *)`

□

Similarly, since inequality involves a negation, it requires a little practice to be able to work with it fluently. Here is one useful trick. If you are trying to prove a goal that is **nonsensical** (e.g., the goal state is `false = true`), apply `ex_falso_quodlibet` to change the goal to `False`. This makes it easier to use assumptions of the form $\neg P$ that may be available in the context — in particular, assumptions of the form $x \neq y$.

```
Theorem not_true_is_false : ∀ b : bool,
  b ≠ true → b = false.

Proof.
  intros [] H.
  - (* b = true *)
    unfold not in H.
    apply ex_falso_quodlibet.
    apply H. reflexivity.
  - (* b = false *)
    reflexivity.

Qed.
```

Since reasoning with `ex_falso_quodlibet` is quite common, Coq provides a built-in tactic, `exfalso`, for applying it.

```
Theorem not_true_is_false' : ∀ b : bool,
  b ≠ true → b = false.

Proof.
  intros [] H.
  - (* b = false *)
    unfold not in H.
    exfalso. (* <==> *)
    apply H. reflexivity.
  - (* b = true *) reflexivity.

Qed.
```

Truth

Besides `False`, Coq's standard library also defines `True`, a proposition that is trivially true. To prove it, we use the predefined constant `I : True`:

```
Lemma True_is_true : True.

Proof. apply I. Qed.
```

Unlike `False`, which is used extensively, `True` is used quite rarely, since it is trivial (and therefore uninteresting) to prove as a goal, and it carries no useful information as a hypothesis. But it can be quite useful when defining complex `Props` using conditionals or as a parameter to higher-order `Props`. We will see examples of such uses of `True` later on.

Logical Equivalence

The handy "if and only if" connective, which asserts that two propositions have the same truth value, is just the conjunction of two implications.

```
Module MyIff.

Definition iff (P Q : Prop) := (P → Q) ∧ (Q → P).

Notation "P ↔ Q" := (iff P Q)
  (at level 95, no associativity)
  : type_scope.

End MyIff.
```

```
Theorem iff_sym : ∀ P Q : Prop,
  (P ↔ Q) → (Q ↔ P).
Proof.
  (* WORKED IN CLASS *)
  intros P Q [HAB HBA].
  split.
  - (* -> *) apply HBA.
  - (* <- *) apply HAB. Qed.

Lemma not_true_iff_false : ∀ b,
  b ≠ true ↔ b = false.
Proof.
  (* WORKED IN CLASS *)
  intros b. split.
  - (* -> *) apply not_true_is_false.
  - (* <- *)
    intros H. rewrite H. intros H'. inversion H'.
Qed.
```

Exercise: 3 stars (or distributes over and)

```
Theorem or_distributes_over_and : ∀ P Q R : Prop,
  P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ R).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Some of Coq's tactics treat `iff` statements specially, avoiding the need for some low-level proof-state manipulation. In particular, `rewrite` and `reflexivity` can be used with `iff` statements, not just equalities. To enable this behavior, we need to import a Coq library that supports it:

```
Require Import Coq.Setoids.Setoid.
```

Here is a simple example demonstrating how these tactics work with `iff`. First, let's prove a couple of basic `iff` equivalences...

```
Lemma mult_0 : ∀ n m, n * m = 0 ↔ n = 0 ∨ m = 0.
+
Lemma or_assoc :
  ∀ P Q R : Prop, P ∨ (Q ∨ R) ↔ (P ∨ Q) ∨ R.
+
```

We can now use these facts with `rewrite` and `reflexivity` to give smooth proofs of statements involving equivalences. Here is a ternary version of the previous `mult_0` result:

```
Lemma mult_0_3 :
  ∀ n m p, n * m * p = 0 ↔ n = 0 ∨ m = 0 ∨ p = 0.
Proof.
  intros n m p.                                     wow!!!
  rewrite mult_0. rewrite mult_0. rewrite or_assoc.
```

reflexivity.
Qed.

The apply tactic can also be used with \leftrightarrow . When given an equivalence as its argument, apply tries to guess which side of the equivalence to use.

```
Lemma apply_iff_example :  
   $\forall n m : \text{nat}, n * m = 0 \rightarrow n = 0 \vee m = 0.$   
Proof.  
  intros n m H. apply mult_0. apply H.  
Qed.
```

so you can apply \leftrightarrow when
you're proving an implication \rightarrow

Existential Quantification

Another important logical connective is *existential quantification*. To say that there is some x of type T such that some property P holds of x , we write $\exists x : T, P$. As with \forall , the type annotation $: T$ can be omitted if Coq is able to infer from the context what the type of x should be.

To prove a statement of the form $\exists x, P$, we must show that P holds for some specific choice of value for x , known as the *witness* of the existential. This is done in two steps: First, we explicitly tell Coq which witness t we have in mind by invoking the tactic $\exists t$. Then we prove that P holds after all occurrences of x are replaced by t .

```
Lemma four_is_even :  $\exists n : \text{nat}, 4 = n + n.$   
Proof.  
   $\exists 2.$  reflexivity.  
Qed.
```

Conversely, if we have an existential hypothesis $\exists x, P$ in the context, we can destruct it to obtain a witness x and a hypothesis stating that P holds of x .

```
Theorem exists_example_2 :  $\forall n,$   
   $(\exists m, n = 4 + m) \rightarrow$   
   $(\exists o, n = 2 + o).$   
Proof.  
  (* WORKED IN CLASS *)  
  intros n [m Hm]. (* note implicit destruct here *)  
   $\exists (2 + m).$   
  apply Hm. Qed.
```

Exercise: 1 star, recommended (dist not exists)

Prove that " P holds for all x " implies "there is no x for which P does not hold." (Hint: `destruct H as [x E]` works on existential assumptions!)

```
Theorem dist_not_exists :  $\forall (X:\text{Type}) (P : X \rightarrow \text{Prop}),$   
   $(\forall x, P x) \rightarrow \neg (\exists x, \neg P x).$   
Proof.  
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (dist exists or)

Prove that existential quantification distributes over disjunction.

```
Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
  (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Programming with Propositions

The logical connectives that we have seen provide a rich vocabulary for defining complex propositions from simpler ones. To illustrate, let's look at how to express the claim that an element x occurs in a list l . Notice that this property has a simple recursive structure:

- If l is the empty list, then x cannot occur on it, so the property " x appears in l " is simply false.
- Otherwise, l has the form $x' :: l'$. In this case, x occurs in l if either it is equal to x' or it occurs in l' .

We can translate this directly into a straightforward recursive function taking an element and a list and returning a proposition:

```
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] ⇒ False
  | x' :: l' ⇒ x' = x ∨ In x l'
  end.
```

When `In` is applied to a concrete list, it expands into a concrete sequence of nested disjunctions.

```
Example In_example_1 : In 4 [1; 2; 3; 4; 5].
```

`Proof.`

```
(* WORKED IN CLASS *)
simpl. right. right. right. left. reflexivity.
```

`Qed.`

```
Example In_example_2 :
```

```
  ∀ n, In n [2; 4] →
  ∃ n', n = 2 * n'.
```

`Proof.`

```
(* WORKED IN CLASS *)
simpl.
intros n [H | [H | []]].
- ∃ 1. rewrite <- H. reflexivity.
- ∃ 2. rewrite <- H. reflexivity.
```

`Qed.`

(Notice the use of the empty pattern to discharge the last case *en passant*.)

We can also prove more generic, higher-level lemmas about `In`.

Note, in the next, how `In` starts out applied to a variable and only gets expanded when we do case analysis on this variable:

```
Lemma In_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
  In x l →
  In (f x) (map f l).
Proof.
  intros A B f l x.
  induction l as [|x' l' IHl'].
  - (* l = nil, contradiction *)
    simpl. intros [].
  - (* l = x' :: l' *)
    simpl. intros [H | H].
    + rewrite H. left. reflexivity.
    + right. apply IHl'. apply H.
Qed.
```

This way of defining propositions recursively, though convenient in some cases, also has some drawbacks. In particular, it is subject to Coq's usual restrictions regarding the definition of recursive functions, e.g., the requirement that they be "obviously terminating." In the next chapter, we will see how to define propositions *inductively*, a different technique with its own set of strengths and limitations.

Martin: there are different formalization of set theory, it'd help to look at some of them to get a good idea of it!!

Exercise: 2 stars (`In_map` iff)

```
Lemma In_map_iff :
  ∀ (A B : Type) (f : A → B) (l : list A) (y : B),
  In y (map f l) ↔
  ∃ x, f x = y ∧ In x l.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (`In_app` iff)

```
Lemma In_app_iff : ∀ A l l' (a:A),
  In a (l++l') ↔ In a l ∨ In a l'.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, recommended (`All`)

Recall that functions returning propositions can be seen as *properties* of their arguments. For instance, if `P` has type `nat → Prop`, then `P n` states that property `P` holds of `n`.

Drawing inspiration from `In`, write a recursive function `All` stating that some property `P` holds of all elements of a list `l`. To make sure your definition is correct, prove the `All_In` lemma below. (Of course, your definition should *not* just restate the left-hand side of `All_In`.)

```
Fixpoint All {T : Type} (P : T → Prop) (l : list T) : Prop
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
```

Admitted.

```
Lemma All_In :
   $\forall T (P : T \rightarrow \text{Prop}) (l : \text{list } T),$ 
   $(\forall x, \text{In } x l \rightarrow P x) \leftrightarrow$ 
   $\text{All } P l.$ 
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (combine odd even)

Complete the definition of the `combine_odd_even` function below. It takes as arguments two properties of numbers, `Podd` and `Peven`, and it should return a property `P` such that `P n` is equivalent to `Podd n` when `n` is odd and equivalent to `Peven n` otherwise.

```
Definition combine_odd_even (Podd Peven : nat  $\rightarrow$  Prop) : nat  $\rightarrow$  Prop
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

To test your definition, prove the following facts:

```
Theorem combine_odd_even_intro :
   $\forall (Podd Peven : \text{nat} \rightarrow \text{Prop}) (n : \text{nat}),$ 
   $(\text{oddb } n = \text{true} \rightarrow Podd n) \rightarrow$ 
   $(\text{oddb } n = \text{false} \rightarrow Peven n) \rightarrow$ 
  combine_odd_even Podd Peven n.
Proof.
  (* FILL IN HERE *) Admitted.
```

```
Theorem combine_odd_even_elim_odd :
   $\forall (Podd Peven : \text{nat} \rightarrow \text{Prop}) (n : \text{nat}),$ 
  combine_odd_even Podd Peven n  $\rightarrow$ 
   $\text{oddb } n = \text{true} \rightarrow$ 
  Podd n.
Proof.
  (* FILL IN HERE *) Admitted.
```

```
Theorem combine_odd_even_elim_even :
   $\forall (Podd Peven : \text{nat} \rightarrow \text{Prop}) (n : \text{nat}),$ 
  combine_odd_even Podd Peven n  $\rightarrow$ 
   $\text{oddb } n = \text{false} \rightarrow$ 
  Peven n.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Applying Theorems to Arguments

One feature of Coq that distinguishes it from many other proof assistants is that it treats *proofs* as first-class objects.

There is a great deal to be said about this, but it is not necessary to understand it in detail in order to use Coq. This section gives just a taste, while a deeper exploration can be found in the optional chapters `ProofObjects` and `IndPrinciples`.

We have seen that we can use the `Check` command to ask Coq to print the type of an expression. We can also use `Check` to ask what theorem a particular identifier refers to.

```
Check plus_comm.
(* ==> forall n m : nat, n + m = m + n *)
```

Coq prints the *statement* of the `plus_comm` theorem in the same way that it prints the *type* of any term that we ask it to `Check`. Why?

The reason is that the identifier `plus_comm` actually refers to a *proof object* — a data structure that represents a logical derivation establishing of the truth of the statement $\forall n m : \text{nat}, n + m = m + n$. The type of this object *is* the statement of the theorem that it is a proof of.

Intuitively, this makes sense because the statement of a theorem tells us what we can use that theorem for, just as the type of a computational object tells us what we can do with that object — e.g., if we have a term of type `nat → nat → nat`, we can give it two nats as arguments and get a `nat` back. Similarly, if we have an object of type $n = m \rightarrow n + n = m + m$ and we provide it an "argument" of type $n = m$, we can derive $n + n = m + m$.

Operationally, this analogy goes even further: by applying a theorem, as if it were a function, to hypotheses with matching types, we can specialize its result without having to resort to intermediate assertions. For example, suppose we wanted to prove the following result:

```
Lemma plus_comm3 :
  ∀ x y z, x + (y + z) = (z + y) + x.
```

It appears at first sight that we ought to be able to prove this by rewriting with `plus_comm` twice to make the two sides match. The problem, however, is that the second `rewrite` will undo the effect of the first.

```
Proof.
intros x y z.
rewrite plus_comm.
rewrite plus_comm.
(* We are back where we started... *)
Abort.
```

One simple way of fixing this problem, using only tools that we already know, is to use `assert` to derive a specialized version of `plus_comm` that can be used to rewrite exactly where we want.

```
Lemma plus_comm3_take2 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
```

```

intros x y z.
rewrite plus_comm.
assert (H : y + z = z + y).
{ rewrite plus_comm. reflexivity. }
rewrite H.
reflexivity.
Qed.

```

A more elegant alternative is to apply `plus_comm` directly to the arguments we want to instantiate it with, in much the same way as we apply a polymorphic function to a type argument.

```

Lemma plus_comm3_take3 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite plus_comm.
  rewrite (plus_comm y z).
  reflexivity.
Qed.

```

You can "use theorems as functions" in this way with almost all tactics that take a theorem name as an argument. Note also that theorem application uses the same inference mechanisms as function application; thus, it is possible, for example, to supply wildcards as arguments to be inferred, or to declare some hypotheses to a theorem as implicit by default. These features are illustrated in the proof below.

```

Example lemma_application_ex :
  ∀ {n : nat} {ns : list nat},
    In n (map (fun m => m * 0) ns) →
    n = 0.
Proof.
  intros n ns H.
  destruct (proj1 _ _ (In_map_if _ _ _ _ _) H)
  as [m [Hm _]]. as [m [Hm _]].
  rewrite mult_0_r in Hm. rewrite <- Hm. reflexivity.
Qed.

```

We will see many more examples of the idioms from this section in later chapters.

Coq vs. Set Theory

Coq's logical core, the *Calculus of Inductive Constructions*, differs in some important ways from other formal systems that are used by mathematicians for writing down precise and rigorous proofs. For example, in the most popular foundation for mainstream paper-and-pencil mathematics, Zermelo-Fraenkel Set Theory (ZFC), a mathematical object can potentially be a member of many different sets; a term in Coq's logic, on the other hand, is a member of at most one type. This difference often leads to slightly different ways of capturing informal mathematical concepts, but these are, by and large, quite natural and easy to work with. For example, instead of saying

at most one?
how could it be of
no type?
Eric: a term that has
a type error
has no type
e.g.: 3 + False

that a natural number n belongs to the set of even numbers, we would say in Coq that $\text{ev } n$ holds, where $\text{ev} : \text{nat} \rightarrow \text{Prop}$ is a property describing even numbers.

However, there are some cases where translating standard mathematical reasoning into Coq can be either cumbersome or sometimes even impossible, unless we enrich the core logic with additional axioms. We conclude this chapter with a brief discussion of some of the most significant differences between the two worlds.

Functional Extensionality

The equality assertions that we have seen so far mostly have concerned elements of inductive types (`nat`, `bool`, etc.). But since Coq's equality operator is polymorphic, these are not the only possibilities — in particular, we can write propositions claiming that two *functions* are equal to each other:

```
Example function_equality_ex1 : plus 3 = plus (pred 4).  
+
```

In common mathematical practice, two functions f and g are considered equal if they produce the same outputs:

$$(\forall x, f x = g x) \rightarrow f = g$$

This is known as the principle of *functional extensionality*.

Informally speaking, an "extensional property" is one that pertains to an object's observable behavior. Thus, functional extensionality simply means that a function's identity is completely determined by what we can observe from it — i.e., in Coq terms, the results we obtain after applying it.

Functional extensionality is not part of Coq's basic axioms. This means that some "reasonable" propositions are not provable.

```
Example function_equality_ex2 :  
(fun x => plus x 1) = (fun x => plus 1 x).  
Proof.  
(* Stuck *)  
Abort.
```

However, we can add functional extensionality to Coq's core logic using the `Axiom` command.

```
Axiom functional_extensionality : ∀ {X Y : Type}  
{f g : X → Y},  
(∀ (x:X), f x = g x) → f = g.
```

Using `Axiom` has the same effect as stating a theorem and skipping its proof using `Admitted`, but it alerts the reader that this isn't just something we're going to come back and fill in later!

We can now invoke functional extensionality in proofs:

```
Example function_equality_ex2 :  
(fun x => plus x 1) = (fun x => plus 1 x).
```

Proof.

```
apply functional_extensionality. intros x.
apply plus_comm.
Qed.
```

Naturally, we must be careful when adding new axioms into Coq's logic, as they may render it *inconsistent* — that is, they may make it possible to prove every proposition, including `False`!

Unfortunately, there is no simple way of telling whether an axiom is safe to add: hard work is generally required to establish the consistency of any particular combination of axioms.

Fortunately, it is known that adding functional extensionality, in particular, is consistent.

To check whether a particular proof relies on any additional axioms, use the `Print Assumptions` command.

```
Print Assumptions function_equality_ex2.
(* ==>
  Axioms:
  functional_extensionality :
    forall (X Y : Type) (f g : X -> Y),
      (forall x : X, f x = g x) -> f = g *)
```

Exercise: 4 stars (tr_rev_correct)

One problem with the definition of the list-reversing function `rev` that we have is that it performs a call to `app` on each step; running `app` takes time asymptotically linear in the size of the list, which means that `rev` has quadratic running time. We can improve this with the following definition:

```
Fixpoint rev_append {X} (l1 l2 : list X) : list X :=
  match l1 with
  | [] => l2
  | x :: l1' => rev_append l1' (x :: l2)
  end.

Definition tr_rev {X} (l : list X) : list X :=
  rev_append l [].
```

This version is said to be *tail-recursive*, because the recursive call to the function is the last operation that needs to be performed (i.e., we don't have to execute `++` after the recursive call); a decent compiler will generate very efficient code in this case. Prove that the two definitions are indeed equivalent.

```
Lemma tr_rev_correct : ∀ X, @tr_rev X = @rev X.
(* FILL IN HERE *) Admitted.
```

□

Propositions and Booleans

We've seen two different ways of encoding logical facts in Coq: with *booleans* (of type `bool`), and with *propositions* (of type `Prop`).

For instance, to claim that a number `n` is even, we can say either

- (1) that `evenb n` returns `true`, or
- (2) that there exists some `k` such that `n = double k`. Indeed, these two notions of evenness are equivalent, as can easily be shown with a couple of auxiliary lemmas.

Of course, it would be very strange if these two characterizations of evenness did not describe the same set of natural numbers! Fortunately, we can prove that they do...

We first need two helper lemmas.

```
Theorem evenb_double : ∀ k, evenb (double k) = true.
Proof.
  intros k. induction k as [|k' IHk'].
  - reflexivity.
  - simpl. apply IHk'.
Qed.
```

Exercise: 3 stars (evenb_double_conv)

```
Theorem evenb_double_conv : ∀ n,
  ∃ k, n = if evenb n then double k
            else S (double k).
Proof.
  (* Hint: Use the evenb_S lemma from Induction.v. *)
  (* FILL IN HERE *) Admitted.
```

□

```
Theorem even_bool_prop : ∀ n,
  evenb n = true ↔ ∃ k, n = double k.
+
```

In view of this theorem, we say that the boolean computation `evenb n` reflects the logical proposition $\exists k, n = \text{double } k$.

Similarly, to state that two numbers `n` and `m` are equal, we can say either (1) that `beq_nat n m` returns `true` or (2) that `n = m`. Again, these two notions are equivalent.

```
Theorem beq_nat_true_iff : ∀ n1 n2 : nat,
  beq_nat n1 n2 = true ↔ n1 = n2.
+
```

However, even when the boolean and propositional formulations of a claim are equivalent from a purely logical perspective, they need not be equivalent *operationally*.

Equality provides an extreme example: knowing that `beq_nat n m = true` is generally of little direct help in the middle of a proof involving `n` and `m`; however, if we convert the statement to the equivalent form `n = m`, we can rewrite with it.

The case of even numbers is also interesting. Recall that, when proving the backwards direction of `even_bool_prop` (i.e., `evenb_double`, going from the propositional to the boolean claim), we used a simple induction on `k`. On the other hand, the converse (the `evenb_double_conv` exercise) required a clever generalization, since we can't directly prove $(\exists k, n = \text{double } k) \rightarrow \text{evenb } n = \text{true}$.

For these examples, the propositional claims are more useful than their boolean counterparts, but this is not always the case. For instance, we cannot test whether a general proposition is true or not in a function definition; as a consequence, the following code fragment is rejected:

```
Fail Definition is_even_prime n :=
  if n = 2 then true
  else false.
```

Coq complains that `n = 2` has type `Prop`, while it expects an elements of `bool` (or some other inductive type with two elements). The reason for this error message has to do with the *computational* nature of Coq's core language, which is designed so that every function that it can express is computable and total. One reason for this is to allow the extraction of executable programs from Coq developments. As a consequence, `Prop` in Coq does *not* have a universal case analysis operation telling whether any given proposition is true or false, since such an operation would allow us to write non-computable functions.

Although general non-computable properties cannot be phrased as boolean computations, it is worth noting that even many *computable* properties are easier to express using `Prop` than `bool`, since recursive function definitions are subject to significant restrictions in Coq. For instance, the next chapter shows how to define the property that a regular expression matches a given string using `Prop`. Doing the same with `bool` would amount to writing a regular expression matcher, which would be more complicated, harder to understand, and harder to reason about.

Conversely, an important side benefit of stating facts using booleans is enabling some proof automation through computation with Coq terms, a technique known as *proof by reflection*. Consider the following statement:

```
Example even_1000 :  $\exists k, 1000 = \text{double } k$ .
```

The most direct proof of this fact is to give the value of `k` explicitly.

```
Proof.  $\exists 500$ . reflexivity. Qed.
```

On the other hand, the proof of the corresponding boolean statement is even simpler:

```
Example even_1000' : evenb 1000 = true.
Proof. reflexivity. Qed.
```

What is interesting is that, since the two notions are equivalent, we can use the boolean formulation to prove the other one without mentioning the value 500 explicitly:

```
Example even_1000'':  $\exists k, 1000 = \text{double } k.$ 
Proof. apply even_bool_prop. reflexivity. Qed.
```

Although we haven't gained much in terms of proof size in this case, larger proofs can often be made considerably simpler by the use of reflection. As an extreme example, the Coq proof of the famous *4-color theorem* uses reflection to reduce the analysis of hundreds of different cases to a boolean computation. We won't cover reflection in great detail, but it serves as a good example showing the complementary strengths of booleans and general propositions.

Exercise: 2 stars (logical connectives)

The following lemmas relate the propositional connectives studied in this chapter to the corresponding boolean operations.

```
Lemma andb_true_iff :  $\forall b_1 b_2:\text{bool},$ 
 $b_1 \&& b_2 = \text{true} \leftrightarrow b_1 = \text{true} \wedge b_2 = \text{true}.$ 
Proof.
(* FILL IN HERE *) Admitted.

Lemma orb_true_iff :  $\forall b_1 b_2,$ 
 $b_1 || b_2 = \text{true} \leftrightarrow b_1 = \text{true} \vee b_2 = \text{true}.$ 
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 1 star (beq_nat_false iff)

The following theorem is an alternate "negative" formulation of `beq_nat_true_iff` that is more convenient in certain situations (we'll see examples in later chapters).

```
Theorem beq_nat_false_iff :  $\forall x y : \text{nat},$ 
 $\text{beq\_nat } x y = \text{false} \leftrightarrow x \neq y.$ 
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (beq_list)

Given a boolean operator `beq` for testing equality of elements of some type `A`, we can define a function `beq_list` `beq` for testing equality of lists with elements in `A`.

Complete the definition of the `beq_list` function below. To make sure that your definition is correct, prove the lemma `beq_list_true_iff`.

```
Fixpoint beq_list {A : Type} (beq : A  $\rightarrow$  A  $\rightarrow$  bool)
(l1 l2 : list A) : bool
(* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Lemma beq_list_true_iff :
 $\forall A (beq : A \rightarrow A \rightarrow \text{bool}),$ 
 $(\forall a_1 a_2, \text{beq } a_1 a_2 = \text{true} \leftrightarrow a_1 = a_2) \rightarrow$ 
 $\forall l_1 l_2, \text{beq\_list beq } l_1 l_2 = \text{true} \leftrightarrow l_1 = l_2.$ 
```

Proof.

(* FILL IN HERE *) **Admitted.**

□

Exercise: 2 stars, recommended (All forallb)

Recall the function `forallb`, from the exercise `forall_exists_challenge` in chapter Tactics:

```
Fixpoint forallb {x : Type} (test : x → bool) (l : list x) : 
  bool :=
  match l with
  | [] ⇒ true
  | x :: l' ⇒ andb (test x) (forallb test l')
```

Prove the theorem below, which relates `forallb` to the `All` property of the above exercise.

```
Theorem forallb_true_iff : ∀ x test (l : list x),
  forallb test l = true ↔ All (fun x ⇒ test x = true) l.
```

Proof.

(* FILL IN HERE *) **Admitted.**

Are there any important properties of the function `forallb` which are not captured by this specification?

(* FILL IN HERE *)

□

Classical vs. Constructive Logic

We have seen that it is not possible to test whether or not a proposition `P` holds while defining a Coq function. You may be surprised to learn that a similar restriction applies to *proofs*! In other words, the following intuitive reasoning principle is not derivable in Coq:

```
Definition excluded_middle := ∀ P : Prop,
  P ∨ ¬ P.
```

To understand operationally why this is the case, recall that, to prove a statement of the form `P ∨ Q`, we use the `left` and `right` tactics, which effectively require knowing which side of the disjunction holds. But the universally quantified `P` in `excluded_middle` is an *arbitrary* proposition, which we know nothing about. We don't have enough information to choose which of `left` or `right` to apply, just as Coq doesn't have enough information to mechanically decide whether `P` holds or not inside a function.

However, if we happen to know that `P` is reflected in some boolean term `b`, then knowing whether it holds or not is trivial: we just have to check the value of `b`.

```
Theorem restricted_excluded_middle : ∀ P b,
  (P ↔ b = true) → P ∨ ¬ P.
```

Proof.

```

intros P [] H.
- left. rewrite H. reflexivity.
- right. rewrite H. intros contra. inversion contra.
Qed.

```

In particular, the excluded middle is valid for equations $n = m$, between natural numbers n and m .

```

Theorem restricted_excluded_middle_eq : ∀ (n m : nat),
  n = m ∨ n ≠ m.

Proof.
  intros n m.
  apply (restricted_excluded_middle (n = m) (beq_nat n m)).
  symmetry.
  apply beq_nat_true_iff.
Qed.

```

It may seem strange that the general excluded middle is not available by default in Coq; after all, any given claim must be either true or false. Nonetheless, there is an advantage in not assuming the excluded middle: statements in Coq can make stronger claims than the analogous statements in standard mathematics. Notably, if there is a Coq proof of $\exists x, P x$, it is possible to explicitly exhibit a value of x for which we can prove $P x$ — in other words, every proof of existence is necessarily *constructive*.

Logics like Coq's, which do not assume the excluded middle, are referred to as *constructive logics*.

More conventional logical systems such as ZFC, in which the excluded middle does hold for arbitrary propositions, are referred to as *classical*.

The following example illustrates why assuming the excluded middle may lead to non-constructive proofs:

Claim: There exist irrational numbers a and b such that a^b is rational.

Proof: It is not difficult to show that $\sqrt{2}$ is irrational. If $\sqrt{2}^{\sqrt{2}}$ is rational, it suffices to take $a = b = \sqrt{2}$ and we are done. Otherwise, $\sqrt{2}^{\sqrt{2}}$ is irrational. In this case, we can take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$, since $a^b = \sqrt{2}^{\sqrt{2} * \sqrt{2}} = \sqrt{2}^2 = 2$. \square

Do you see what happened here? We used the excluded middle to consider separately the cases where $\sqrt{2}^{\sqrt{2}}$ is rational and where it is not, without knowing which one actually holds! Because of that, we wind up knowing that such a and b exist but we cannot determine what their actual values are (at least, using this line of argument).

As useful as constructive logic is, it does have its limitations: There are many statements that can easily be proven in classical logic but that have much more complicated constructive proofs, and there are some that are known to have no constructive proof at all! Fortunately, like functional extensionality, the excluded middle is known to be compatible with Coq's logic, allowing us to add it safely as an

axiom. However, we will not need to do so in this book: the results that we cover can be developed entirely within constructive logic at negligible extra cost.

It takes some practice to understand which proof techniques must be avoided in constructive reasoning, but arguments by contradiction, in particular, are infamous for leading to non-constructive proofs. Here's a typical example: suppose that we want to show that there exists x with some property P , i.e., such that $P x$. We start by assuming that our conclusion is false; that is, $\neg \exists x, P x$. From this premise, it is not hard to derive $\forall x, \neg P x$. If we manage to show that this intermediate fact results in a contradiction, we arrive at an existence proof without ever exhibiting a value of x for which $P x$ holds!

The technical flaw here, from a constructive standpoint, is that we claimed to prove $\exists x, P x$ using a proof of $\neg \neg (\exists x, P x)$. Allowing ourselves to remove double negations from arbitrary statements is equivalent to assuming the excluded middle, as shown in one of the exercises below. Thus, this line of reasoning cannot be encoded in Coq without assuming additional axioms.

Exercise: 3 stars (excluded middle irrefutable)

Proving the consistency of Coq with the general excluded middle axiom requires complicated reasoning that cannot be carried out within Coq itself. However, the following theorem implies that it is always safe to assume a decidability axiom (i.e., an instance of excluded middle) for any *particular* Prop P . Why? Because we cannot prove the negation of such an axiom. If we could, we would have both $\neg (P \vee \neg P)$ and $\neg \neg (P \vee \neg P)$ (since P implies $\neg \neg P$, by the exercise below), which would be a contradiction. But since we can't, it is safe to add $P \vee \neg P$ as an axiom.

```
Theorem excluded_middle_irrefutable: ∀ (P:Prop),
  ¬ ¬ (P ∨ ¬ P).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, advanced (not_exists_dist)

It is a theorem of classical logic that the following two assertions are equivalent:

$$\begin{aligned} &\neg (\exists x, \neg P x) \\ &\forall x, P x \end{aligned}$$

The `dist_not_exists` theorem above proves one side of this equivalence.

Interestingly, the other direction cannot be proved in constructive logic. Your job is to show that it is implied by the excluded middle.

```
Theorem not_exists_dist :
  excluded_middle →
  ∀ (X:Type) (P : X → Prop),
    ¬ (exists x, ¬ P x) → (forall x, P x).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 5 stars, optional (classical axioms)

For those who like a challenge, here is an exercise taken from the Coq'Art book by Bertot and Casteran (p. 123). Each of the following four statements, together with `excluded_middle`, can be considered as characterizing classical logic. We can't prove any of them in Coq, but we can consistently add any one of them as an axiom if we wish to work in classical logic.

Prove that all five propositions (these four plus `excluded_middle`) are equivalent.

```
Definition peirce :=  $\forall P Q : \text{Prop},$ 
 $((P \rightarrow Q) \rightarrow P) \rightarrow P.$ 
```

```
Definition double_negation_elimination :=  $\forall P : \text{Prop},$ 
 $\sim\sim P \rightarrow P.$ 
```

```
Definition de_morgan_not_and_not :=  $\forall P Q : \text{Prop},$ 
 $\sim(\sim P \wedge \sim Q) \rightarrow P \vee Q.$ 
```

```
Definition implies_to_or :=  $\forall P Q : \text{Prop},$ 
 $(P \rightarrow Q) \rightarrow (\neg P \vee Q).$ 
```

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

□