# Chapter 1

# Theorem proving with Coq

## 1.1 Introduction

### 1.1.1 What is Coq?

Coq is a tool to help you write formal proofs, that are mechanically verifiable. This means that once you have proved something in Coq, you have very high assurance that it is true – more than what you usually have when doing a pen-and-paper proof. It can be used in an interactive style, thus we call it an interactive proof assistant. It is based on a very expressive logic, the Calculus of Inductive Constructions.

### 1.1.2 What Coq is not.

Coq is not a tool that will automatically prove theorems. So, you can't have Coq solve your logic or math homework for you, or prove the Goldbach conjecture. Still, it greatly simplifies the development of formal proofs, by automating some aspects of it.

### 1.1.3 Why you should learn to use it.

Many reasons: to understand formal mathematical logic better; to verify that a proof is indeed valid; to develop certified software.

### 1.1.4 How you could learn to use it.

- Sit through this section for the basics.
- Follow the standard Coq tutorial. (this presentation is based on it)
- Use the reference manual for things not covered there.
- For more detailed introductions, look at the Coq'Art book and Adam Chlipala's book.

- Treat it as a programming language: the only proper way to learn it is to use it!

### 1.1.5 How to use it.

Coq is installed in the zoo machines, in the directory

```
/c/cs430/bin
```

As a development environment for it, you can either use the CoqIDE or ProofGeneral, which is an Emacs-based environment. Both are installed in the same directory.

## 1.2 Propositional Logic

We will now see how to prove a simple theorem of propositional logic, namely the S axiom of the Hilbert calculus, in Coq. Its statement is
$(P \to Q \to R) \to (P \to Q) \to P \to R$

Coq commands are displayed using fixed-width font below. We start our Coq development by starting a new section. Sections are used in Coq developments for presentation reasons, in order to group definitions and theorems together, and also to introduce local assumptions and local variables.

`Section` PropositionalLogic.

In this section, for example, we will introduce three symbols that will serve as atomic propositions. This is done through the use of the following `Variables` command, where the type `Prop` is used to classify propositions.

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

As we said, Coq is an interactive tool. That means that instead of having to write a development in full and then pass it to Coq (the way we write programs and pass them to a compiler), we can feed a development little by little to Coq, which will then help us proceed. Environments like CoqIDE and ProofGeneral help us do that, and one should use the rest of this tutorial like that.

As a first step, we state the theorem we're trying to prove.

`Theorem` hilbert_axiom_S : $(P \to Q \to R) \to (P \to Q) \to P \to R$.

If we feed the above statement of the theorem to Coq, it replies with the following:
1 `subgoal`

$P$ : `Prop`
$Q$ : `Prop`
$R$ : `Prop`

$$(P \to Q \to R) \to (P \to Q) \to P \to R$$

What is presented here is the current proof state. Above the line lie the currently available variables and hypotheses; below the line lies the goal that we have to prove. Proving something in Coq involves using the right commands to transform this proof state into simpler forms or to satisfy the current goal using the current hypotheses. Such commands are called tactics.

The first tactic we will use is `intros`. This tactic transforms a proof state with a goal that involves logical implication, by moving the left-hand side of the implication into our set of hypotheses, giving it a name, and leaving the right-hand side of the goal.

`intros` *H1*.

The current proof state is the following:

1 `subgoal`

$P$ : `Prop`
$Q$ : `Prop`
$R$ : `Prop`
*H1* : $P \to Q \to R$

$$(P \to Q) \to P \to R$$

We proceed by doing the same thing for the next two implications.

`intros` *H2 H3*.

There are multiple ways to proceed; the way that is most natural for a Coq development to proceed at this point would be to state that the goal $R$ will be proved using the hypothesis $P \to Q \to R$. To prove $R$ using this hypothesis we need two extra requirements, namely $P$ and $Q$, which will become new subgoals that we have to prove. We use the `apply` tactic to tell Coq which hypothesis to use.

`apply` *H1*.

It is easy to prove the first subgoal $P$, since we already have it as a hypothesis. The `exact` tactic can be used in this case to take care of the goal, since the hypothesis *H3* is exactly the goal that we are trying to prove.

`exact` *H3*.

For proving the second subgoal $Q$, we can similarly use the hypothesis $P \to Q$. If we read the logical implication arrow as a functional arrow, we could view proofs of this proposition as functions that given a proof of proposition $P$, return a proof of proposition $Q$. This in fact is the reading of logical implication in constructive logic, which Coq adheres to. Therefore we can create a proof of $Q$ by applying the function *H2* : $P \to Q$ to the argument *H3* : $P$. Thus by using the tactic `exact` again we can solve the current goal.

`exact` (*H2 H3*).

Now Coq tells us

```
Proof completed.
```

which means that we're done. We use the command `Qed.` to close the proof of this theorem.

```
Qed.
```

Note that this is not the only way to prove the above fact, and perhaps not the best way. We could even have Coq prove it automatically for us, by using the `auto` tactic.

Let's see now how to handle logical connectives like $\wedge$ and $\vee$. Again we need to consider these in their constructive reading: a proof of $A \wedge B$ is composed of a proof of $A$ and a proof of $B$, while a proof of $A \vee B$ is a choice between the left-hand side or the right-hand side plus a proof of the associated proposition.

```
Theorem distr_and_or : (P ∧ Q) ∨ (P ∧ R) → P ∧ (Q ∨ R).
intros H.
```

Having the hypothesis $(P \wedge Q) \vee (P \wedge R)$, we want to proceed by cases on whether the left or the right proposition of the disjunction holds. Here the `destruct` tactic is useful: given a hypothesis, it will require that we prove the same goal for all of the possible ways that this hypothesis can be proven. Since in this case our hypothesis can only be proven by either a proof of $P \wedge Q$ or a proof of $P \wedge R$, it will generate two subgoals that we need to prove.

```
destruct H as [H1 | H2].
```

In the first case, we have that $P \wedge Q$, and we need to prove $P \wedge (Q \vee R)$. To prove a goal that involves a conjunction, we use the `split` tactic, that generates two subgoals, one for each operand of the conjunction.

```
split.
```

We can use `destruct` for the conjunctive hypothesis $P \wedge Q$, in order to get both $P$ and $Q$ as hypotheses.

```
destruct H1 as [HP HQ].
exact HP.
```

Now we need to prove $Q \vee R$; since we already have a proof for Q, we use the *left* tactic to denote that we will prove the left part of this disjunction.

```
destruct H1 as [HP HQ].
left.
exact HQ.
```

The case for $P \wedge R$ is similar; we first use `destruct` to get the proofs of $P$ and $R$ out of the proof of $P \wedge R$ and then we can just allude to `auto` here to complete the proof for us.

```
destruct H2 as [HP HR].
auto.
```

`Qed`.

Logical negation is represented as implication of False; so *not A* is equivalent to $A \rightarrow$ *False*. Still, the treatment of logical negation in Coq is uncommon, in that in its underlying logic the law of the excluded middle does not hold, unless we explicitly include it. We can still prove the law of noncontradiction:

`Theorem` noncontradiction : not $(P \wedge$ not $P)$.

We prove this by opening up the definition of logical negation using the `unfold` tactic. The rest is similar to the above.

`unfold` *not*.
`intros` *H*.
`destruct` *H* `as` [*H1 H2*].
`auto`.
`Qed`.

We can summarize some of the tactics we can use for propositional logic in the following table:

| | As a goal | As a hypothesis |
|---|---|---|
| $\rightarrow$ | `intros` | (various) |
| $\wedge$ | `split` | `destruct` |
| $\vee$ | *left* or *right* | `destruct` |
| *True* | `trivial` | – |
| *False* | – | `destruct` |

We conclude this section using the following command.

`End` PropositionalLogic.

## 1.3 Predicate logic

Universally quantified goals can be proven by introducing a new free variable through the `intros` tactic, while such hypotheses can be used by specifying instantiations for the quantified variables (implicitly or explicitly) when using tactics like `apply` and `destruct`. Existentially quantified goals can be proven by selecting a witness for which they hold; existentially quantified hypotheses can be opened up so as to give a name to their contained witness for the rest of the proof.

`Section` PredicateLogic.

We first assume a domain of discourse; the objects of this domain will be what we quantify over.

`Variable` *D* : `Set`.

We define the binary relation R, which can also be viewed as a two-place predicate over our domain of discourse.

`Variable` *R* : $D \rightarrow D \rightarrow$ `Prop`.

```
Theorem refl_if : (∀ x y : D, R x y → R y x) →
                  (∀ x y z : D, R x y → R y z → R x z) →
                  (∀ x : D, (∃ y : D, R x y) → R x x).
intros Rsym.
intros Rtrans.
```

When our goal begins with a universal quantifier, we can use intros (just like in the case of logical implication) to add a new free variable of the type that we quantify over to our list of hypotheses. In a pen-and-paper proof, we would say "assume an arbitrary x..."; this is what `intros` is equivalent to here.

```
intros x.
```

```
intros xR_.
```

The proof needs to proceed as follows; since we know that $\exists\ y : D,\ x\ R\ y$, we can show through the use of the transitive property that $R\ y\ x \rightarrow R\ x\ x$. Then, the proof can be completed through the use of the symmetric property. So the first step is to open up the existential so that we can use $y$ in the rest of the proof; we do this through the `destruct` tactic.

```
destruct xR_ as [y xRy].
```

Now we want to use the hypothesis that R is transitive; we do this through the `apply` tactic. The conclusion of this hypothesis is unified with the current goal; this instantiates its quantified variables $x$ and $z$, but the variable $y$ remains uninstantiated – Coq cannot magically guess what value we want for it. Thus we need to specify its value, or alternatively use the `eapply` tactic which would leave the variable uninstantiated, so that it can be unified later.

```
apply Rtrans with (y := y).
```

The current goal is proved directly by a hypothesis; we can use the tactic `assumption` to denote that without specifying the name of the hypothesis.

```
assumption.
```

To complete the proof we also need to show that $R(y,\mathrm{x})$, which is trivially proved using the hypothesis that $R$ is symmetric.

```
apply Rsym.
assumption.
Qed.
```

Let's try another theorem, that also involves functions and equality. We assume two functions, and also that our domain of discourse is inhabited (by requiring one element that belongs in it).

```
Variables f g : D → D.
Variable d : D.
```

```
Theorem example_pred_theorem : (∀ x : D, ∃ y : D, f x = g y) →
```

$$(\exists\ c : D, \forall\ x : D, f\ x = c) \rightarrow$$
$$(\exists\ y : D, \forall\ x : D, f\ x = g\ y).$$

It is obvious why this holds; the function $f$ is constant, so by instantiating our first premise with an arbitrary $x$ we can get a $y$ such that $g\ y = c$, which is exactly the witness we need for our existential goal.

`intros` *H1 H2*.

We open up the existential hypothesis that $f$ is constant, naming the constant value as $c$.

`destruct` *H2* `as` [*c H2'*].

Now we want to state the fact that there exists a $y$ such that $g\ y = c$. One way to do this is to use the `cut` tactic; it will introduce it as a new premise in our goal, and later we will have to show why this fact holds.

`cut` $(\exists\ y : D,\ c = g\ y)$.
`intros` *H3*.

Again we open up the just-introduced existential hypothesis to give an explicit name to its witness that we can use for the rest of the proof.

`destruct` *H3* `as` [*yc H3'*].

We will prove our goal exactly for this witness; note the use of the exists tactic to handle existential goals.

$\exists$ *yc*.

At this point we need to prove that $f\ x = g\ y$; we already know that $c = g\ y$ thus it suffices to show that $f\ x = c$ which is an assumption. We use the `rewrite` tactic to use the fact that $c = g\ y$ and replace $g\ y$ with $c$ in our goal.

`rewrite` $\leftarrow$ *H3'*.
`assumption`.

To prove that $\exists\ y : D,\ c = g\ y$, it suffices to change $c$ to $f\ x$ for an arbitrary $x$. That's what we do below, providing an explicit instantiation for the universally quantified variable of the hypothesis H2'.

`rewrite` $\leftarrow$ *H2'* `with` $(x := d)$.
`apply` *H1*.
`Qed`.

`End PredicateLogic`.

One more example of proofs for predicate logic propositions from elementary group theory: we show that in a group, there always is an x such that $a \times x = b$.

`Section GroupTheory`.

A group is defined by a set G and a closed operation on members of that set; we require that the operation is associative, that a unit element exists, and that every element has an inverse.

```
Variable G : Set.
Variable operation : G → G → G.
Variable e : G.
Variable inv : G → G.
Infix "×" := operation.

Hypothesis associativity : ∀ x y z : G, (x × y) × z = x × (y × z).
Hypothesis identity : ∀ x : G, x × e = e ∧ e × x = x.
Hypothesis inverse : ∀ x : G, x × inv x = e ∧ inv x × x = e.
```

The statement of the theorem follows.

```
Theorem latin_square_property : ∀ a b : G, ∃ x : G, a × x = b.
```

The proof proceeds by stating that such an $x$ as the theorem requires is $inv\ a \times b$; then we need to show why $a \times (inv\ a \times b) = b$, which is provable through associativity of $\times$, the fact that $a \times inv\ a = e$, and then that $e \times b = b$.

```
intros a b.
∃ (inv a × b).
rewrite ← associativity.
```

We want to isolate the left part of the conjunction of our hypothesis about the inverses, while instantiating its quantified variable to $a$.

```
destruct inverse with (x := a) as [H _].
rewrite H.
```

Same thing for the right part of the property of the identity element, instantiating its quantified variable to $b$.

```
destruct identity with (x := b) as [_ H'].
trivial.
Qed.

End GroupTheory.
```

We can thus augment the previous tactics table with the following:

| | As a goal | As a hypothesis |
|---|---|---|
| ∀ | `intros` | ... `with` $(x := ...)$ |
| ∃ | exists | `destruct` |
| = | `trivial` when $a = a$ | `rewrite` |

Yet another (still informal) explanation of the tactics is the following:

$$\frac{\Psi, H : P \vdash Q}{\Psi \vdash P \to Q}\texttt{intros H}$$

$$\frac{\Psi, H : P \to Q \vdash P}{\Psi, H : P \to Q \vdash Q}\texttt{apply H}$$

$$\frac{\Psi \vdash P \qquad \Psi \vdash Q}{\Psi \vdash P \wedge Q}\texttt{split}$$

$$\frac{\Psi, H1 : P, H2 : Q \vdash R}{\Psi, H : P \wedge Q \vdash R}\texttt{destruct H as [H1 H2]}$$

$$\frac{\Psi \vdash P}{\Psi \vdash P \vee Q}\texttt{left}$$

$$\frac{\Psi \vdash Q}{\Psi \vdash P \vee Q}\texttt{right}$$

$$\frac{\Psi, H1 : P \vdash R \qquad \Psi, H2 : Q \vdash R}{\Psi, H : P \vee Q \vdash R}\texttt{destruct H as [H1 | H2]}$$

$$\frac{\Psi, y : t \vdash P[y/x]}{\Psi \vdash \forall x : t.P}\texttt{intros y}$$

$$\frac{\begin{array}{c}\texttt{...\ \ tactic H' ...} \\ \Psi, H : \forall x : t.P, H' : P[a/x] \vdash Q\end{array}}{\Psi, H : \forall x : t.P \vdash Q}\texttt{tactic H' with (x := a)}$$

$$\frac{\Psi \vdash P[a/x]}{\Psi \vdash \exists x : t.P}\texttt{exists a}$$

$$\frac{\Psi, y : t, H' : P[y/x] \vdash Q}{\Psi, H : \exists x : t.P \vdash Q}\texttt{destruct H as [y H']}$$

$$\frac{\Psi \vdash P \to Q \qquad \Psi \vdash P}{\Psi \vdash Q}\texttt{cut P}$$

## 1.4 Inductive definitions

In Coq we specify the objects that we reason about by defining inductively how such objects can be constructed. For example, if we want to be able to reason about natural numbers, we would give the following definition:

```
Inductive nat : Set :=
    O : nat
  | S : nat → nat.
```

O is the constructor that corresponds to zero; S is the constructor that corresponds to the successor function.

Immediately from the previous definition, we get the principle of induction over natural numbers: `Print` *nat_ind*.

```
nat_ind : ∀ P : nat → Prop,
            P O →
            (∀ n : nat, P n → P (S n)) →
            ∀ n : nat, P n
```

In fact, inductive definitions play a much more central role in Coq than just being able to inductively define sets like natural numbers. Predicates like the $\leq$-relation for natural numbers are also be inductively defined.

```
Inductive le : nat → nat → Prop :=
    le_n : ∀ n : nat, le n n
  | le_S : ∀ n m : nat, le n m → le n (S m).
```

Even logical connectives like $\vee$ are defined inductively.

```
Inductive or (A B : Prop) : Prop :=
    or_introl : A → or A B
  | or_intror : B → or A B.
```

Let's now see some example proofs involving induction. We'll prove the theorems:
$\forall\ n : nat,\ O \leq n$
$\forall\ n\ m : nat,\ n \leq m \rightarrow S\ n \leq S\ m$

```
Theorem le_O_n : ∀ n : nat, O ≤ n.
```

We want to proceed by performing natural number induction. We specify this using the `induction` tactic, which gives us the two expected subgoals – the base case and the step case of the induction.

```
induction n.
```

Both cases are provable by using the appropriate constructor from the definition of *le*; in the second case we also need to use the inductive hypothesis. Using the tactic *constructor* we choose the constructor whose conclusion matches the current goal.

*constructor*.
*constructor*.
`trivial`.
`Qed`.

We could have written the above proof using the tactics combinator ;. This composes two tactics together, applying the second one to all the subgoals that the first one generates. Thus an equivalent way to write the above proof would be:
`induction` $n$; *constructor*; `trivial`.

Let's now move on to the next theorem.

`Theorem` le_S_S : $\forall\ n\ m$ : **nat**, $n \leq m \rightarrow \mathsf{S}\ n \leq \mathsf{S}\ m$.

What is interesting is that it is easier to prove this by performing induction on the derivation of $n \leq m$ instead of natural number induction on $n$ or $m$.

The induction principle that we get from the definition of $\leq$ is the following:
$(\forall\ n.P\ n\ n) \rightarrow (\forall\ n,\ m.P\ n\ m \rightarrow P\ n\ (S\ m)) \rightarrow$
$(\forall\ n,\ m.n \leq m \rightarrow P\ n\ m)$
By instantiating P to `fun` $a\ b \Rightarrow S\ a \leq S\ b$ we get:
$(\forall\ n.S\ n \leq S\ n) \rightarrow (\forall\ n,\ m.S\ n \leq S\ m \rightarrow S\ n \leq S\ (S\ m)) \rightarrow$
$(\forall\ n,\ m.n \leq m \rightarrow S\ n \leq S\ m)$

We specify that we want to perform induction on the derivation of $n \leq m$ again through the `induction` tactic. We give it the number of the hypothesis after all the universal quantifiers that we want to perform induction on. The rest is proved trivially by the definition of $\leq$ and the inductive hypothesis.

`induction` 1; *constructor*; `trivial`.

`Qed`.

As a last example, let's see how recursive functions are handled. We can use the `Fixpoint` command to define such functions, e.g. the addition of natural numbers:

```
Fixpoint plus (a b : nat) : nat :=
  match a with
      O ⇒ b
    |S a ⇒ S (plus a b)
  end.
```

These functions need to be terminating, in order to ensure logical consistency.

We can now prove theorems like the following:

`Theorem` le_plus : $\forall\ n\ m\ p,\ n \leq m \rightarrow n \leq \mathsf{plus}\ p\ m$.

The `simpl` tactic can be used to perform evaluation of a function.

We proceed by induction on $p$.

`induction` $p$.

Our goal is $n \leq m \rightarrow n \leq plus\ O\ m$; we can use `simpl` to see that it is equivalent to $n \leq m \rightarrow n \leq m$ – after the evaluation of $plus\ O\ m$.

```
simpl.
trivial.
```

Similarly in the inductive step case, we use `simpl` to perform the evaluation of $plus\ (S\ p)\ m$ in the goal.

```
intros H.
simpl.
constructor.
apply IHp.
trivial.
Qed.
```

## 1.5 Declarative proof style

The proof scripts we've been writing are very hard to read, and also perform too many low-level steps. There are a number of ways we can mitigate these problems, and arrive at scripts that are better at conveying the essential ideas of proofs, and easier to write. One way to do this is to use Coq's new declarative proof mode, where we specify intermediate steps rather than the tactics required to reach them; another way is through the careful use of automation. Here we'll briefly see the declarative style.

Let's revisit a theorem we proved earlier and see its declarative proof.

```
Theorem latin_square_property_decl : ∀ a b : G, ∃ x : G, a × x = b.
proof.
  let a : G, b : G.
  take (inv a × b).
  have H1:(a × (inv a × b) = (a × inv a) × b) by associativity.
  have H2:(a × inv a = e) by inverse.
  have (a × (inv a × b) = e × b) by H1, H2.
                              ~= b by identity.
  hence thesis.
end proof.
Qed.
```

Another example we saw before.

```
Theorem le_plus_decl : ∀ n m p : nat, n ≤ m → n ≤ p + m.
proof.
  let n, m, p : nat be such that H:(n ≤ m).
  per induction on p.
  suppose it is O.
```

```
    reconsider thesis as (n ≤ m).
    thus thesis by H.
  suppose it is (S p') and IH:(n ≤ p' + m).
    reconsider thesis as (n ≤ S (p' + m)).
    thus thesis by IH.
  end induction.
end proof.
Qed.
```

The basic way of doing proofs in the declarative style is the following:

- We use `let` and *assume* instead of intros.
- We use *have* to state one fact and explicitly provide justification for it by listing the hypotheses we use. `then` is similar but includes the previous fact in the justifications.
- We use *thus* to show (part of) the thesis, including justification if needed. *hence* includes the previous fact.
- When doing induction, we need to explicitly specify the different cases. Dependent induction is still not supported though...

For more info, look at the documentation for C-zar in the Coq reference manual.