

Mathematical formalization using Coq

Daniel Schepler

November 5, 2011

Contents

1	First steps	2
1.1	Declarations and types	2
1.2	Proof mode	7
1.3	Definitions	15
1.4	Exercises	19

Introduction

Coq is a proof assistant; its main applications are:

1. Formal program verification. For example, Coq has been used to write and certify a complete compiler for a large subset of the C language (and what's left out is largely misfeatures of C which shouldn't normally be used in any case).
2. Formalization of mathematical proofs. One notable achievement of Coq in this area is a full verified proof of the Four Color Theorem.

Most existing Coq tutorials focus mainly on the first application above. The purpose of this tutorial is to provide an introduction to the second application: mathematical formalization.

First of all, Coq is a proof assistant, and not an automated theorem prover. This means that you'll still have to convince Coq that your theorems are true. However, Coq does provide for a limited amount of automation in the search for proofs of substeps. Moreover, Coq features a *tactic language* which allows you to add domain-specific knowledge to aid in your proofs. For example, Coq comes with built-in tactics to prove trivial ring and semiring identities (e.g. $x(x + y) + y^2 = y(x + y) + x^2$).

This tutorial focuses on the use of CoqIde, a specialized environment for interactive development of Coq definitions and proofs. Another popular interface to Coq is Proof General, an Emacs mode which interfaces with the backend. Most of this tutorial should work fine with Proof General, except that you'll need to substitute any CoqIde specific instructions with the appropriate equivalents in Proof General.

1 First steps

First, you will of course need to install Coq on your machine. This tutorial will not cover the details of how to do that. You should find instructions at <http://coq.inria.fr/>. This version of the manual uses Coq version 8.3pl2, although it should presumably mostly work fine with other recent versions of Coq.

Start up CoqIde, and you'll be presented with a workspace divided into several panels. The main left panel is where you enter instructions to Coq. The top right panel will show the current proof state when you're in proof mode, and the bottom right panel will output Coq's responses to queries or proof tactics.

1.1 Declarations and types

In this chapter, to illustrate the basic operation of Coq for constructing formal proofs, we'll be writing some basic proofs on the foundations of group theory. To start off, enter:

```
Parameter G : Type.
```

Then submit this directive to Coq using the shortcut Alt+Ctrl+↓ (Alt+Ctrl+Cmd+↓ on Mac). Note that the terminating `.` is important: it tells Coq where a directive ends. If you forget it, the shortcut won't do anything.

For the rest of the manual, we'll use the following convention to indicate input to Coq and Coq's responses:

```
Coq < Parameter G : Type.  
G is assumed
```

Here a line prefixed with “Coq <” indicates what should be input to Coq, and (optionally) following text in italics indicates Coq's response to that input.

This directive tells Coq that we want to assume that `G` is some (unspecified) element of the type `Type`. In contrast to the foundations of Zermelo-Fraenkel set theory (ZF or ZFC) which you may be familiar with, Coq's foundation is a type-based theory. This means that every object must have a type, and any expression you enter must “type check” in some way. For example, try entering and executing this (we'll describe the `Check` directive in more detail a little bit later):

```
Coq < Check (0 = false).  
Toplevel input, characters 11-16:  
> Check (0 = false).  
> ~~~~~  
Error: The term "false" has type "bool"  
while it is expected to have type "nat".
```

This means that `0` is of type `nat`, while `false` is of type `bool`, and the two types are incompatible. Therefore, it doesn't make sense to ask whether the two objects are equal. This matches more closely how most mathematicians think in day-to-day work than the model of Zermelo-Fraenkel set theory: in ZF, it makes sense for any two objects x and y to ask whether $x = y$. However, in most mathematical work, for example if G and H are two groups and $g \in G$, $h \in H$, we wouldn't write expressions like $g = h$, and we'd be likely to set up functions in such a way that we wouldn't have to worry about whether or not G and H might “overlap.”

In this case, because we're specifying that `G` is of type `Type`, Coq will know that `G` is something that can contain elements.

Now we'll declare the identity element of our group (be sure to remove the faulty `Check` directive before continuing):

```
Coq < Parameter e : G.
e is assumed
```

This is similar to the previous `Parameter` directive, except that this time we're telling Coq that `e` is an unspecified element of `G`. Note, however, that for most types, the elements of that type cannot serve as types themselves. For example, try entering:

```
Coq < Parameter x : e.
Toplevel input, characters 14-15:
> Parameter x : e.
> ~
Error: The term "e" has type "G" which should be Set, Prop or Type.
```

We'll describe `Set` and `Prop` later. For now, we'll interpret this error message as saying: because `e` is an element of `G` which is not the same as `Type`, it doesn't make sense to use `e` as a type. Again, this matches more closely how most mathematicians think in day-to-day work than the model of Zermelo-Fraenkel set theory: in ZF, every object is a set, and it makes sense for any two objects x and y to ask whether $x \in y$. In contrast, if G is an arbitrary group, most mathematicians would think of the elements of G as atoms instead of as sets themselves, which is exactly the way Coq treats them. (In case you're wondering how Coq would handle sets of sets, sets of sets of sets, etc., we'll cover that later.)

Now, once you've removed the faulty `Parameter` directive, we move on to declaring the inverse operation:

```
Coq < Parameter inv : G -> G.
inv is assumed
```

Here `G -> G` represents the type of all functions from `G` to `G`, so we're specifying that `inv` is some function which is an element of this type.

```
Coq < Parameter mult : G -> G -> G.
mult is assumed
```

In this declaration, Coq's convention diverges slightly from the usual mathematical convention that the binary group operator would be declared as a function $\cdot : G \times G \rightarrow G$. Let us dissect the type of this declaration a bit. The main point to keep in mind is that in Coq, the `->` operator has "right associativity", which means that `G -> G -> G` is interpreted as `G -> (G -> G)`. Therefore, `mult` is a function which takes in an element of G , and returns a function which takes in a second element of G and returns the product in G .

Now if we want to write an application of one of these functions, we do it by writing first the function name, and then the value we wish to input. We'll present a few examples using the `Check` directive. What this directive does is take in an expression, and tell you what type the expression has; or if the expression doesn't type check, it outputs an error indicating where it sees a problem. For example, try the following `Check` directives:

```

Coq < Check (inv e).
inv e
    : G

Coq < Check (mult e).
mult e
    : G -> G

Coq < Check ((mult e) e).
mult e e
    : G

Coq < Check (mult e e).
mult e e
    : G

```

The last example here shows an interesting point: in Coq, function application has “left associativity,” so that if we wish to apply a function with multiple arguments, we can just write the function name followed by a list of the arguments. On the other hand, as the second example above shows, Coq’s convention automatically allows for “partial application” of a function: in this example, we specify the first argument of `mult` while leaving the second argument unspecified, resulting in a function from `G` to `G`.

However, if any of the arguments of a function is a complex expression itself, we must enclose it in parentheses. For example, if we try:

```

Coq < Check (mult inv e e).
Toplevel input, characters 12-15:
> Check (mult inv e e).
>
Error: The term "inv" has type "G -> G"
while it is expected to have type "G".

```

Here, Coq thinks we mean to feed the `inv` function itself as an argument to `mult`, which doesn’t match the expected type `G` of the argument. By inserting parentheses around the call to `inv e`, we can correct Coq’s interpretation so that the expression makes sense. So replace the faulty `Check` directive with:

```

Coq < Check (mult (inv e) e).
mult (inv e) e
    : G

```

Likewise:

```

Coq < Check (inv mult e e).
Toplevel input, characters 11-15:
> Check (inv mult e e).
>
Error: The term "mult" has type "G -> G -> G"
while it is expected to have type "G".

```

```
Coq < Check (inv (mult e e)).
inv (mult e e)
  : G
```

Now let's move on to telling Coq about the group axioms we'll be using. But before we do that, note that it's bad form in general to leave **Check** directives in a finished Coq script: they will cause extraneous output to be printed if you need to go back and recompile the script. However, CoqIde will not allow you to modify green highlighted text, as this could very likely invalidate what's already been verified. So first rewind the execution point before the **Check** directives you've entered. You can do this either by pressing Ctrl+Alt+↑ (Ctrl+Alt+Cmd+↑ on Mac) repeatedly until the green highlight does not include any of them, or by positioning the cursor just before the first **Check** directive and then clicking on the "Go to cursor" button (it looks like a green arrow pointing to a yellow sphere). Then go ahead and delete any **Check** directives you've entered.

Note that CoqIde has a feature to execute **Check** directives without inserting them into the main script: select Queries → Check from the menu, and you'll get a "Command Pane" at the bottom. Then you can enter the expression to check in the edit box to the right, and press enter or click "Ok" to execute the check. Once you're done with this command pane, you can click on the X button on the left to hide it. However, personally, I find this pane more cumbersome to work with than just entering the **Check** directives directly and then cleaning them up.

Now go on to enter:

```
Coq < Axiom left_id : forall x:G, mult e x = x.
left_id is assumed

Coq < Axiom left_inv : forall x:G, mult (inv x) x = e.
left_inv is assumed

Coq < Axiom assoc : forall x y z:G,
Coq <   mult x (mult y z) = mult (mult x y) z.
assoc is assumed
```

These declarations are fairly straightforward to read. One thing to note is that what comes after the colon should be a proposition of type Prop:

```
Coq < Check (forall x:G, mult e x = x).
forall x : G, mult e x = x
  : Prop
```

Also note that Coq makes a distinction between a proposition and the truth or falsehood of that proposition. This is because in general, for example if G is an infinite group, it would be impossible for Coq to compute whether a proposition involving **forall** or **exists** is true or not.

Note that already, in the declaration of the axiom **assoc**, writing the two expressions is slightly cumbersome, and you might imagine that things will only get worse as we move to more and more complex expressions. Fortunately, Coq allows us to declare that we want to write our group multiplication operator using the built-in symbol *****. So let's rewind execution above the axiom declarations and replace them with:

```
Coq < Infix "*" := mult.
Coq <
```

```

Coq < Axiom left_id : forall x:G, e * x = x.
left_id is assumed

Coq < Axiom left_inv : forall x:G, inv x * x = e.
left_inv is assumed

Coq < Axiom assoc : forall x y z:G, x * (y * z) = (x * y) * z.
assoc is assumed

```

You can now review the axioms if you wish using the **Check** directive. This can be useful if you forgot the exact form you used to write the axiom, and you don't want to go back and search for the declaration:

```

Coq < Check left_id.
left_id
      : forall x : G, e * x = x

Coq < Check left_inv.
left_inv
      : forall x : G, inv x * x = e

Coq < Check assoc.
assoc
      : forall x y z : G, x * (y * z) = x * y * z

```

In the last example, we see that `*` has left associativity, so Coq was able to leave out the parentheses on the right hand side. This will allow us to write long products like `a * b * c * d * e` without having to fully parenthesize the expression as `((a * b) * c) * d * e`.

One thing to note is that `a * b` is just shorthand notation for `mult a b`, and the two mean exactly the same thing. In fact, Coq uses `mult a b` as its internal representation, and only uses the notation when displaying expressions. If you want to see the internal representation, you can do so by selecting “Display → Deactivate notations display” from the menu. If you do this, and then retry the **Check** directives above, you'll see:

```

Coq < Check left_id.
left_id
      : forall x : G, eq (mult e x) x

Coq < Check left_inv.
left_inv
      : forall x : G, eq (mult (inv x) x) e

Coq < Check assoc.
assoc
      : forall x y z : G, eq (mult x (mult y z)) (mult (mult x y) z)

```

(This reveals that `=` is just another predefined notation for application of a built-in function `eq`.)

1.2 Proof mode

Now that we've declared the objects and axioms of the theory of a group, we're ready to move on to formalizing the bootstrapping of some of the other identities commonly used in groups. First, clean up your **Checks**, and reenable notation display by selecting "Display \rightarrow Deactivate notations display" again.

Our first step is to prove the left cancellation property:

```
Coq < Proposition left_cancel : forall x y z:G,  
Coq <   x * y = x * z -> y = z.  
1 subgoal
```

```
=====
```

$$\text{forall } x \ y \ z : G, \ x * y = x * z \rightarrow y = z$$

```
Coq < Proof.
```

Note the use of the logical implication connective \rightarrow , which takes in two propositions P and Q and returns the proposition $P \rightarrow Q$ that P implies Q .

Once you execute the **Proposition** directive, CoqIde enters proof mode, and in the upper right pane displays:

```
1 subgoal  
-----(1/1)  
forall x y z : G, x * y = x * z -> y = z
```

At any point in constructing a proof, you can have some context for the current subgoal, and a list of subgoals. In this example, the context is empty, and there's one subgoal. What you need to do is apply "proof tactics" to manipulate the current context and subgoal until you've discharged all subgoals and there are no subgoals remaining.

The first thing we did was to execute the **Proof** directive. This directive doesn't actually do anything and is strictly speaking optional. However, it's customary to use this to delimit the start of the body of a "proof script" (which refers to the list of tactics which together resolve the initial proof state to a completed proof state).

In many proofs, the first step will be to move all parameters and hypotheses from the statement to be proved into the context. This is done using the **intros** tactic:

```
Coq < intros x y z Htancel.  
1 subgoal
```

```
x : G  
y : G  
z : G  
Htancel : x * y = x * z  
=====
```

$$y = z$$

Note that if you just use "**intros.**" without any names, then Coq will autogenerate names such as H , $H0$, $H1$, etc. for any hypotheses. However, giving explicit names is useful in longer proofs to

improve the readability of proof scripts, as well as aiding in updating proof scripts for changes in definitions or the statements of previous results, so it's a good habit to get into. (I have to admit that I personally am not necessarily in this habit, but it often happens that I wish I were.)

Our proof will be to multiply each side of the hypothesis on the left by x^{-1} . So our first step will be to generate a new subgoal representing the result using the **assert** tactic:

```
Coq < assert (inv x * (x * y) = inv x * (x * z))
Coq <   as Hinvx_times_tocancel.
2 subgoals
```

```

x : G
y : G
z : G
Htocancel : x * y = x * z
=====
inv x * (x * y) = inv x * (x * z)
subgoal 2 is:
y = z
```

Now we use the hypothesis Htocancel to replace $x * y$ with $x * z$ using the **rewrite** tactic:

```
Coq <   rewrite Htocancel.
2 subgoals
```

```

x : G
y : G
z : G
Htocancel : x * y = x * z
=====
inv x * (x * z) = inv x * (x * z)
subgoal 2 is:
y = z
```

Since both sides of the equality to be proved are the same, we can finish this subgoal using the reflexivity of equality:

```
Coq <   reflexivity.
1 subgoal
```

```

x : G
y : G
z : G
Htocancel : x * y = x * z
Hinvx_times_tocancel : inv x * (x * y) = inv x * (x * z)
=====
y = z
```


Now that we've proved the assertion, it's now available in the context of the original statement to be proved, under the name we gave it.

Next, what we want to do is to use the group axioms to manipulate the hypothesis `Hinvx_times_tocancel` until it reads the way we want:

```
Coq < rewrite assoc in Hinvx_times_tocancel.
Coq < rewrite assoc in Hinvx_times_tocancel.
Coq < rewrite left_inv in Hinvx_times_tocancel.
Coq < rewrite left_id in Hinvx_times_tocancel.
Coq < rewrite left_id in Hinvx_times_tocancel.
1 subgoal
```

```

x : G
y : G
z : G
Htocancel : x * y = x * z
Hinvx_times_tocancel : y = z
=====
y = z
```

If you were paying close attention while executing these rewrites, you might have noticed that rewriting using `left_inv` rewrote in two places, while for the rewriting using `assoc` and `left_id`, each time it only did a rewrite in one place and we had to repeat it for the other place. The reason for this is that `rewrite` searches for one special case of the proposition where the left hand side occurs, and then for that particular left hand side, it replaces each occurrence in the target with the right hand side. In this case, the two rewrites using `assoc` were using two different cases: $x^{-1}(xy) = (x^{-1}x)y$ and $x^{-1}(xz) = (x^{-1}x)z$. On the other hand, when we were rewriting using `left_inv`, in both places we were using the special case $x^{-1}x = e$.

Now we have exactly what we wanted to prove as one of the hypotheses. So we finish off the proof with:

```
Coq < assumption.
Proof completed.
```

Finally, there's one step left: to finalize the proof and register the new result so that it can be used in future proofs.

```
Coq < Qed.
intros x y z Htocancel.
assert (inv x * (x * y) = inv x * (x * z)) as Hinvx_times_tocancel.
rewrite Htocancel.
reflexivity.

rewrite assoc in Hinvx_times_tocancel.
rewrite assoc in Hinvx_times_tocancel.
rewrite left_inv in Hinvx_times_tocancel.
```

```

rewrite left_id in Hinvx_times_tocancel.
rewrite left_id in Hinvx_times_tocancel.
assumption.

```

left_cancel is defined

Overall, the proof script should look something like:

```

Proposition left_cancel :
  forall x y z : G, x * y = x * z -> y = z.
Proof.
intros x y z Htocal.
assert (inv x * (x * y) = inv x * (x * z))
  as Hinvx_times_tocancel.
  rewrite Htocal.
  reflexivity.

rewrite assoc in Hinvx_times_tocancel.
rewrite assoc in Hinvx_times_tocancel.
rewrite left_inv in Hinvx_times_tocancel.
rewrite left_id in Hinvx_times_tocancel.
rewrite left_id in Hinvx_times_tocancel.
assumption.
Qed.

```

Note the indentation for the proof of the assertion, and the separation between proofs for subgoals. In general, there's not necessarily any one best way to format proofs for readability, so whatever format works for you should be fine.

At this point, it's perhaps appropriate to explain a couple alternatives for the overall structure of a Coq proof. The most straightforward way is usually to “work backwards” from the goal, using tactics like `apply`, `rewrite`, `assumption`, or `reflexivity`. On the other hand, it is also possible to “work forwards” from the hypotheses, using tactics like `rewrite ... in H...`, or `apply ... in H...` (we'll see an example of `apply in` later in this section).

In the proof above, we used a mixture of these two strategies. First, we used `assert` to establish a “cut point” of the proof. Then we used argument backwards from the cut point to establish this intermediate result. From there, we manipulated this cut point “forwards” from its original statement until it read as we wanted.

Now that we've finished that proof, we'll be able to use it in future proofs exactly as we'd use the axioms. For example, you can `Check` its statement just as you would for an axiom:

```

Coq < Check left_cancel.
left_cancel
  : forall x y z : G, x * y = x * z -> y = z

```

Next, let's try another proof using this result:

```

Coq < Proposition right_id :
Coq <   forall x : G, x * e = x.

```

```
Coq < Proof.
```

```
Coq < intros.
```

```
1 subgoal
```

```

x : G
=====
x * e = x

```

The informal proof that we'll formalize for Coq is: we will use left cancellation by x^{-1} . To apply this, we need to prove that $x^{-1}(x \cdot e) = x^{-1}(x)$. But both sides are equal to e , so the conclusion follows.

The formalization for “use left cancellation by x^{-1} ” reads:

```
Coq < apply left_cancel with (x:=inv x).
```

```
1 subgoal
```

```

x : G
=====
inv x * (x * e) = inv x * x

```

Here the x on the left hand side of the $:=$ refers to the variable name in the statement of `left_cancel`, while the right hand side `inv x` is an expression evaluated in the current context to use for special casing `left_cancel`. Note that Coq was able to infer the values of y and z to use in `left_cancel`; however, there's no way to infer what value of x we want, so we had to specify that variable.

In general, the `apply` tactic will generate one subgoal for each hypothesis of the result being applied. In this case, since `left_cancel` has only one hypothesis, we're left with one subgoal. We now manipulate this term as usual:

```
Coq < rewrite assoc.
```

```
Coq < rewrite left_inv.
```

```
1 subgoal
```

```

x : G
=====
e * e = e

```

At this point, we could proceed by rewriting using `left_id` and then using `reflexivity`. However, in this case, we observe that the goal is precisely a special case of `left_id`:

```
Coq < apply left_id.
```

```
Coq < Qed.
```

Here, because `left_id` has no hypotheses, there are no new subgoals to be proved.

By now, you should have the basic tools to be able to prove the rest. As practice, you might want to try proving each proposition below yourself, before you enter the proofs presented in the text (each of which will introduce new tactics to add to your repertoire).

```

Coq < Proposition right_inv :
Coq <   forall x:G, x * inv x = e.
Coq < Proof.
Coq < intros.
Coq < apply left_cancel with (x := inv x).
Coq < rewrite assoc.
Coq < rewrite left_inv.
Coq < rewrite left_id.
1 subgoal

```

```

  x : G
  =====
  inv x = inv x * e

```

This isn't quite in the form of `right_id`: so if you try to apply `right_id` now, Coq will give an error like

```

Coq < apply right_id.
Toplevel input, characters 6-14:
> apply right_id.
> ~~~~~
Error: Impossible to unify "?96 * e = ?96" with "inv x = inv x * e".

```

So, to manipulate the goal into the required form, we will first use the `symmetry` tactic to switch the two sides of the equality:

```

Coq < symmetry.
1 subgoal

  x : G
  =====
  inv x * e = inv x

```

```

Coq < apply right_id.
Proof completed.
Coq < Qed.

```

Next, we prove an auxiliary result which we'll use for proving properties of the inverse function:

```

Coq < Proposition right_inv_unique :
Coq <   forall x y:G, x * y = e -> inv x = y.
Coq < Proof.
Coq < intros.
Coq < apply left_cancel with (x := x).

```

1 *subgoal*

```
x : G
y : G
H : x * y = e
=====
x * inv x = x * y
```

At this point, we choose to use the `transitivity` tactic. Given a goal of the form `LHS = RHS`, using `transitivity intermed.` will generate two subgoals of the form `LHS = intermed` and `intermed = RHS`. So in this case:

```
Coq < transitivity e.
```

2 *subgoals*

```
x : G
y : G
H : x * y = e
=====
x * inv x = e
subgoal 2 is:
e = x * y
```

```
Coq < apply right_inv.
```

```
Coq <
```

```
Coq < symmetry.
```

```
Coq < assumption.
```

```
Coq < Qed.
```

Using this result, proving that $(x^{-1})^{-1} = x$ is straightforward:

```
Coq < Proposition inv_involution :
```

```
Coq < forall x:G, inv (inv x) = x.
```

```
Coq < Proof.
```

```
Coq < intros.
```

```
Coq < apply right_inv_unique.
```

```
Coq < apply left_inv.
```

```
Coq < Qed.
```

Similarly:

```
Coq < Proposition inv_prod :
```

```
Coq < forall x y:G, inv (x*y) = inv y * inv x.
```

```
Coq < Proof.
```

```

Coq < intros.
Coq < apply right_inv_unique.
Coq < rewrite assoc.
1 subgoal

  x : G
  y : G
  =====
  x * y * inv y * inv x = e
Coq < rewrite right_inv.
Toplevel input, characters 0-17:
> rewrite right_inv.
> ~~~~~
Error: Found no subterm matching "?133 * inv ?133" in the current goal.

```

But the goal, $x * y * \text{inv } y * \text{inv } x = e$, has “ $y * \text{inv } y$ ” right there! What’s the problem here?

To answer this question, you need to remember that $*$ is interpreted as a left associative operator. Thus, the goal actually means the same thing as $((x * y) * \text{inv } y) * \text{inv } x = e$, which doesn’t literally contain $y * \text{inv } y$ as a subterm. To fix this, we’ll need to rewrite from the right hand side of the axiom `assoc` to the left hand side; so replace the last line with:

```

Coq < rewrite <- assoc.
1 subgoal

  x : G
  y : G
  =====
  x * y * (inv y * inv x) = e

```

Well, this wasn’t what we wanted. We’ll have to help Coq by being a bit more specific about which instance of the axiom to use; so rewind and replace the last line again with:

```

Coq < rewrite <- assoc with (z := inv y).
1 subgoal

  x : G
  y : G
  =====
  x * (y * inv y) * inv x = e

```

Now the goal has the term we want to rewrite using `right_inv`, and the rest of the proof is straightforward:

```

Coq < rewrite right_inv.
Coq < rewrite right_id.

```

(On the last line, `rewrite left_id` would not have worked. Why not?)

```
Coq < apply right_inv.
```

```
Coq < Qed.
```

1.3 Definitions

Coq also lets you write definitions based on the group operations:

```
Coq < Definition commutator : G -> G -> G :=
```

```
Coq <   fun x y:G => x * y * inv x * inv y.
```

```
commutator is defined
```

We can also define a notation for the commutator operation:

```
Coq < Notation "[ x , y ]" := (commutator x y).
```

```
Setting notation at level 0.
```

Here the spacing inside the quotes is important: each name of a “parameter” of the notation must be in a word by itself.

Now let us see how to work with definitions in proofs:

```
Coq < Proposition commutator_inv :
```

```
Coq <   forall x y:G, inv [x, y] = [y, x].
```

```
Coq < Proof.
```

```
Coq < intros.
```

```
1 subgoal
```

```
x : G
```

```
y : G
```

```
=====
```

```
inv [x, y] = [y, x]
```

In order to proceed, we need to expand the definition of the commutator using the `unfold` tactic:

```
Coq < unfold commutator.
```

```
1 subgoal
```

```
x : G
```

```
y : G
```

```
=====
```

```
inv (x * y * inv x * inv y) = y * x * inv y * inv x
```

From here, the manipulations to complete the proof are straightforward:

```
Coq < rewrite inv_prod.
```

```
Coq < rewrite inv_prod.
```

```

Coq < rewrite inv_prod.
Coq < rewrite inv_involution.
Coq < rewrite inv_involution.
Coq < rewrite assoc.
Coq < rewrite assoc.
Coq < reflexivity.
Coq < Qed.

```

Note that in this case, it was starting to get tedious having to repeat the same tactic over and over. We can reduce this tedium by using **repeat**: if **T** is a proof tactic, then **repeat T** will execute **T** repeatedly until it fails. So, for example, we can rewrite the proof script above as:

```

Coq < Proposition commutator_inv:
Coq <   forall x y:G, inv [x, y] = [y, x].
Coq < Proof.
Coq < intros.
Coq < unfold commutator.
Coq < repeat rewrite inv_prod.
Coq < repeat rewrite inv_involution.
Coq < repeat rewrite assoc.
Coq < reflexivity.
Coq < Qed.

```

Similarly to definitions of functions, we can write definitions of propositions:

```

Coq < Definition commutes_with : G -> G -> Prop :=
Coq <   fun x y:G => x * y = y * x.
Coq < commutes_with is defined

```

This is a function which takes two arguments $x, y \in G$ and returns the proposition that x commutes with y . Now, for an example of working with this definition:

```

Coq < Proposition product_commutes :
Coq <   forall x y z:G, commutes_with x z -> commutes_with y z ->
Coq <     commutes_with (x * y) z.
Coq < Proof.
Coq < intros x y z Hxz_comm Hyz_comm.
Coq < 1 subgoal

    x : G

```



```

y : G
z : G
Hxz_comm : commutes_with x z
Hyz_comm : commutes_with y z
=====
commutes_with (x * y) z

```

(Note the syntax for multiple hypotheses. The `->` implication operator is interpreted using right associativity, just as the `->` function type operator is. Thus, the statement of the proposition is literally interpreted as: “for any $x, y, z \in G$, x commuting with z implies the proposition that y commuting with z implies that $x \cdot y$ commutes with z .”)

Now we unfold what we need to prove using:

```

Coq < red.
1 subgoal

x : G
y : G
z : G
Hxz_comm : commutes_with x z
Hyz_comm : commutes_with y z
=====
x * y * z = z * (x * y)

```

This tactic unfolds the top-level function application in its target (the current subgoal by default — you could also use, for example, “`red in Hxz_comm`.”) Therefore, in this case, it would have been equivalent to write “`unfold commutes_with`.”

From here, the necessary manipulations are straightforward:

```

Coq < rewrite <- assoc.
Coq < rewrite Hyz_comm.

```

Note that even though we haven’t unfolded the definition of `commutes_with` in `Hyz_comm`, we were still able to reuse it in `rewrite`. In general, Coq is often able to unfold definitions internally in tactics like `rewrite`, `apply` or even `reflexivity`, so that you do not need to unfold them manually and unnecessarily clutter the proof context.

```

Coq < rewrite assoc.
Coq < rewrite Hxz_comm.
Coq < rewrite assoc.
Coq < reflexivity.
Coq < Qed.

```

Note that Coq also provides a shorthand notation for defining functions. For example, we could have written the definitions above as:

```

Coq < Definition commutator2 (x y:G) : G :=
Coq <   x * y * inv x * inv y.
commutator2 is defined

Coq < Definition commutes_with2 (x y:G) : Prop :=
Coq <   x * y = y * x.
commutes_with2 is defined

```

Internally, these definitions mean exactly the same thing as the previous definitions:

```

Coq < Print commutator2.
commutator2 = fun x y : G => x * y * inv x * inv y
      : G -> G -> G

Coq < Print commutes_with2.
commutes_with2 = fun x y : G => x * y = y * x
      : G -> G -> Prop

```

On the other hand, the `fun` notation can be used anywhere a function needs to be input as an argument of another function or a parameter of a previously proved result, without necessarily needing to give that function a name. For example, in the first proof of `left_cancel` above, we could have used the built-in result `f_equal`, which states that if $x = y$, then $f(x) = f(y)$:

```

Coq < Proposition left_cancel_alt_proof : forall x y z:G,
Coq <   x * y = x * z -> y = z.
Coq < Proof.
Coq < intros x y z Htocalcel.
1 subgoal

```

```

x : G
y : G
z : G
Htocalcel : x * y = x * z
=====
y = z

```

```

Coq < apply f_equal with (f := fun g:G => inv x * g) in Htocalcel.
1 subgoal

```

```

x : G
y : G
z : G
Htocalcel : inv x * (x * y) = inv x * (x * z)
=====
y = z

```

...and so on, with the same `rewrites` as before. (Note that in this particular case, we could have used a “partially applied” function, so that

```
Coq < apply f_equal with (f := mult (inv x)).
```

would have exactly the same effect.)

1.4 Exercises

1. Prove:
Proposition `right_cancel` : forall x y z:G,
 $x * z = y * z \rightarrow x = y$.
Your proof should use `f_equal` as above, instead of `assert` as in the first proof of `left_cancel`.
2. Prove:
 - (a) Proposition `commutator_e_impl_commutes` :
forall x y:G, $[x, y] = e \rightarrow \text{commutes_with } x \ y$.
 - (b) Proposition `commutes_impl_commutator_e` :
forall x y:G, $\text{commutes_with } x \ y \rightarrow [x, y] = e$.
3. (a) Define the conjugation operation where $x^y := y^{-1}xy$, and then define a notation for conjugation using the built-in symbol \wedge .
(b) State and prove the following identities:
 - i. $x^z \cdot y^z = (xy)^z$.
 - ii. $e^y = e$.
 - iii. $x^e = x$.
 - iv. $(x^{-1})^y = (x^y)^{-1}$.
 - v. $(x^y)^z = x^{yz}$.