505 Community Notes on Coq

Coq is a radically different programming environment than most of us are used to. This document's goal is to facilitate students in CSE 505 helping each other smooth out the learning curve by documenting the relevant features of Coq in plain language, assuming an audience with a traditional software engineering background.

Please edit this document and improve it! You can also leave comments about what needs improving, or post in Mattermost for faster help.

Table of Contents

Overview of Coq workflow

Using Coq is an interactive experience. Instead of a typical edit-compile-debug cycle, we typically use an IDE to “step through” a Coq file line by line. The IDE highlights the portion of the file that has been processed “so far”, and displays any relevant contextual information. Especially when proving theorems, this interactive experience is fundamental: proofs in Coq can usually only be understood by stepping through them line by line.

Coq is not itself a programming language, but a system consisting of *three* languages:

1. `Gallina`, the language of programs.
2. `Ltac`, the tactic language, the language of proofs.
3. `Vernacular`, the command language.

A Coq file is really a sequence of commands, written in the `Vernacular` language. (This is why the usual extension for Coq files is `.v` — 'v' for `Vernacular`.) Some commands take arguments that are `Gallina` programs or `Ltac` proofs, so in the end, the file will be a mix of all three languages

Commands (`Vernacular`)

Inductive

Inductive declares a new datatype by specifying its constructors.

Example:

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

Define bool to be of type Type. The constructors true and false are the only ways to make a value of type bool.

Definition

Define a (non-recursive) function or constant.

Example:

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

Define andb to take two bool arguments (b1 and b2) and return a bool. andb matches b1 against true and false (the constructors of the bool type). If b1 is true, then return b2. If b1 is false, return false.

Fixpoint

Define a recursive function. Functions must terminate on all arguments.

Example:

```
Fixpoint add (n1 : nat) (n2 : nat) : nat :=
  match n1 with
  | O => n2
  | S m1 => S (add m1 n2)
  end.
```

Define add to take two nats (n1 and n2). If the first nat is 0, just return the second nat (since 0 + anything = anything). Otherwise, add 1 to the recursive addition of the first nat - 1 and the second nat.

Lemma

Declare a theorem which you try to prove.

Proof

Start a proof (after a Lemma/Theorem).

Qed

Ends a proof and “double checks” it; fails if there is anything left to prove.

Theorem

Synonym for Lemma, just communicates to the reader that it’s important!

Check

Show the type of an expression.

Print

Print the definition of a function or constant (can be ugly!)

Compute

Run an expression to its final value.

Abort

Give up on the current lemma or theorem.

Search

Search for a stdlib lemma with a shape (e.g., `_ + _`).

SearchAbout

Search for all proofs and definitions about a particular name.

Programs (`Gallina`)

- match — pattern matching, kind of like `switch` in other languages, choosing among cases
- patterns
- “don’t care” pattern — a.k.a. wildcard, denoted by _, matches anything in a pattern match
- variables
- Constructors - explain?
- function application (aka function call)

Proofs and Tactics (`Ltac`)

apply

Coq's `apply` tactic performs 'backwards reasoning' on the conclusion to be proved by applying a known fact (a hypothesis or a previously proven theorem) whose conclusion matches the conclusion. There are multiple forms:

- `apply H`—if `H`'s conclusion matches the form of the conclusion to be proved, replace the current conclusion with `H`'s hypothesis.
- If a hypothesis has a `forall` you want to get rid of, you can combine it with another hypothesis to get a simpler prop. E.g. `apply IH1 in H`.

assumption (Coq)

Coq's `assumption` tactic inspects the local context for a hypothesis whose type is convertible to the goal. If such a hypothesis is found, the subgoal is proved. Otherwise it fails.

auto (Coq)

Coq's `auto` tactic tries to guess what the next right step is in the proof. It first tries to use the `assumption` tactic, followed by using intros.

cases (FRAP)

[Link to Ltac]
FRAP's `cases` tactic is similar to Coq's `destruct` tactic. Replace the goal with an expression.

Example

```
(* `foo` is a truly dumb function... *)
Definition foo(a : arith): nat :=
  match a with
  | Const _ => 0
  | Plus _ _ => 0
  | Times _ _ => 0
  end.

(* ...and we can prove it! *)
Lemma foo_is_a_dumb_function:
  forall (a : arith),
      foo(a) = 0.
Proof.
  intros.
  cases a.

  (* Here is our use of cases. It transforms our goal from:
   *
   *   a : arith
   *   ============================
   *   foo a = 0
   *
   * to
   *
```

```
    *   n : nat
    *   ============================
    *   foo (Const n) = 0
    *   subgoal 2 is: foo (Plus a1 a2) = 0
    *   subgoal 3 is: foo (Times a1 a2) = 0
    *)
Abort.
```

destruct (Coq)

Coq's `destruct` tactic should be avoided in favor of FRAP's `cases` tactic.

equality

FRAP's `equality` tactic uses an E-graph to implement a complete decision procedure for the theory of equality and uninterpreted functions. This allows Coq to conclude that things like

```
Lemma eq_and_uninterpreted_funcs:
    forall {A} (f : A -> A) (a : A) (b : A),
      a = b -> f a = f b.
Proof.
  equality.
Qed.
```

In particular, the theory of equality and uninterpreted function allows Coq to prove terms equal (`a = b`) by use of any of symmetry, transitivity, reflexivity, and congruence of equality.

exists (Coq)

Coq's `exists` tactic is used to show that an inductive datatype `I` exists. `exists` can be used only when `I` has a single construtor.

eexists (Coq)

Coq's `eexists` tactic operates the same as `exists` except in the case where it cannot instantiate a variable. While `exists` will fail, `eexists` inserts a placeholder variable.

f_equal

reduce function and parameter equality to only different things

- E.g. to prove `(func1 a1 a2) = (func2 b1 b2)`. If `func1 = func2` and `a1 = b1`, then the `f_equal` will reduce the goal to `a2 = b2`.
- Won't work if you have a `forall` in your goal. In that case, try to use `congruence`, or `intro` all variables first.

induction

Coq's `induction` tactic takes a single argument `term` that is of an Inductive type. `induction` creates two subgoals for term: the base case and the inductive hypothesis, each being used to prove the current goal.

intro

Among other things, the `intro` tactic applies to a goal that starts with a dependent product (i.e., `forall x: T, U`). `intro` moves `x` into the local context (i.e., "above the line"), and the new subgoal is `U`.

intros

The `intros` tactic repeats the intro tactic until it meets the head-constant.

propositional (FRAP)

FRAP's `propositional` tactic simplifies a goal into zero or more subgoals based on propositional logic alone.

reflexivity

prove equality---you should prefer to use FRAP's `equality`

rewrite

Coq's `rewrite` tactic replaces a match in the current subgoal with either a previous lemma or the inductive hypothesis

- If your current subgoal has a `forall` then rewrite will not work, instead use `intro x` to remove the `forall` and then use `rewrite some_lemma`
- If you would like to only replace a match that is nested in another match, you can specify the variable(s) that you want it to replace. For example, `add (add a b) c` will be turned into `add c (add a b)` if you do `rewrite add_comm`. To turn `add (add a b) c` into `add (add b a) c`, you can use `rewrite (add_comm a b)`.

- `rewrite <- l`: rewrite with Lemma l from LHS to RHS.

simpl

This tactic tries to compute as much as possible of a function call. There is similar to FRAP's more powerful `simplify` tactic

subst

Coq's `subst` tactic performs substitution on variables bound in a hypothesis. So if the variable `a` is bound to an expression in the hypothesis, say via `H: a = f x`, and you want to prove something relating a and `f x`, for instance, that `g a = g (f x)`, you can run the `subst` tactic to substitute `f x` in for `a`.

symmetry

The `symmetry` tactic replaces a goal `x = y` with `y = x`. E.g. if you have a lemma

```
lemma_1 : forall x y, f x = g y
```

and need to prove

```
g y = f x
```

use `symmetry. apply lemma_1.`

try

The `try` tactic squashes error from an incorrectly applied tactic. This can be useful if you have cases with very similar tactics and want to be able to chain them.

Instead of

```
cases x.
  - simplify. reflexivity. rewrite IHn. f_equal.
  - simplify. rewrite IHn. f_equal.
```

You can write

```
cases x; simplify; try reflexivity; rewrite IHn; f_equal.
```

semicolon (`;`)

Sequence tactics `a` and `b` into a new tactic `a; b`. `a; b` first runs `a` on the current subgoal, and then runs `b` on each subgoal generated by `a`.
If you know how many subgoals there are after one step (say `3`), and you don't want to apply the same next step to them, you can use a vertical bar to separate subgoals: `a; [b | | ]`.

period (`.`)

bullets (`-`, `+`, `*`)

Coq allows an optional syntax to denote a subgoal of the previous destruct/induction command. Nested subgoals should alternate syntax

Points of Confusion and General Advice

- constructor — not OOP constructors - maybe explain it?
- induction before intro!
- know when you're stuck
- Don't use more than one induction per proof.
    - prove a helper lemma
- "generalize the induction hypothesis"
- ";" vs "."
    - tactic1; tactic2. means "run tactic1 on the current goal, then run tactic2 on every subgoal generated by tactic1"
    - tactic1. tactic2. means "run tactic2 on the *first* subgoal generated by tactic1". *(So they're the same if tactic1 generates only one subgoal, but when it generates more than one, semicolon runs on all of them)*

FRAP and Coq tactics translation

At a high level, you can map several default Coq tactics to their nicer FRAP analogs:

- Coq's `cases` is kind of like FRAP's `destruct`
- Coq's `simpl` is kind of like FRAP's `simplify`
- Coq's `reflexivity` is kind of like FRAP's `equality`

- Coq's `lia` (or [deprecated] `omega`) is kind of like FRAP's `linear_arithmetic`
- Coq's `induction` is kind of like FRAP's `induct`

Questions:

What does “induct 1”. Mean?
Induct 1 means to induct on the first hypothesis.

Constructor v econstructor?
https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacv.econstructor

What does this mean in plain english? Example? The Coq documentation needs some improvements if is supposed to be a usable system.

eapply

Apply replaces forward the premise with the conclusion

Econstruct

It would be nice to get a better explanation of the syntax rather than looking through random files.

{| variable := value|} translates to a record or struct of key values

Propositional splits and /\ into two subgoals

Invert when recursive and inductive
Unfold when not inductive

FAQs

How do I prove by contradiction?

todo