Coq Cheatsheet

Formal Foundations of Programming Languages, AS 2023

September 29, 2023

Contents

1 Proof structure and style

1.1 Lemma structure

A lemma and proof are stated as follows in Coq:

```
Lemma lemma_name vars :
  lemma statement.
Proof.
  tactic1.
  tactic2.
  ...
Qed.
```

This declares a lemma called `lemma_name` that says that `lemma statement` holds for all values of variables `vars`. The proof is composed of tactics, which correspond to the inference rules of logic. To develop or view a proof in Coq, you should use your IDE to step over each tactic one-by-one. This will show the intermediate proof state between each step. Note that a lemma can equivalently be written as:

```
Lemma lemma_name :
  forall vars, lemma statement.
```

In this course we favor the version where the top-most universally quantified variables are put in front of the colon. This makes lemma statements more concise and avoids us from having to introduce the variables explicitly in the proof.

1.2 Bullets

Some tactics create multiple subgoals, such as the `destruct` and `induction` tactic: it creates a subgoal for each constructor. We have to solve all the subgoals with a bulleted list of tactic scripts:

```
tactic1.
- tactic2.
- tactic3.
- tactic4.
```

Bullets can nested by using different bullets for different levels (`-`, `+`, `*`, `--`, `++`, `**`):

```
tactic1.
- tactic2.
  + tactic3
  + tactic4.
- tactic5.
```

By convention, we always use the same order for the bullets (`-` for the top level, `+` for the next, and so on).

We can also enter subgoals using brackets:

```
tactic1.
{ tactic2. }
{ tactic3. }
tactic4.
{ tactic5. }
tactic6.
```

This is most useful for solving side conditions or small trivial goals like the base case of an induction proof. With bullets, we get a deep level of nesting if we have a sequence of tactics with side conditions. With brackets, we do not need to enclose the last subgoal in brackets, thus preventing deep nesting.

If you use the mandatory library file from this course, use of bullets is enforced—Coq will give an error if you do not use bullets.

2 Unicode notation

While every term in the course material is written entirely using ASCII characters, Coq will use unicode characters to print them back to you, which is easier to read and more compact. The following table helps to match the unicode pretty-printing back to the ASCII syntax.

Unicode	ASCII	Unicode	ASCII
`P ∧ Q`	`P /\ Q`	`¬ P`	`~ P`
`P ∨ Q`	`P \/ Q`	`t ≠ u`	`~(t = u)`
`P → Q`	`P -> Q`		
`∀ x, P`	`forall x, P`		
`∃ x, P`	`exists x, P`		

3 Logical reasoning

3.1 Tactics that modify the goal

Goal	Tactic
`P -> Q`	`intros H`
`~P`	`intros H` (Coq defines `~P` as `P -> False`)
`forall x, P x`	`intros x`
`exists x, P x`	`exists x`
`P /\ Q`	`split` (also works for `P <-> Q`, which is defined as `(P -> Q) /\ (Q -> P)`)
`P \/ Q`	`left`, `right`
`Q`	`apply H` (where `H : (...) -> Q` is a lemma or hypothesis with conclusion `Q`)
`False`	`apply H` (where `H : (...) -> ~P` is a lemma or hypothesis with conclusion `~P`)
Any goal	`exfalso` (turns any goal into `False`)
Any goal	`assumption` (solves goal if it follows from a hypothesis)
Any goal	`admit` (skips goal so that you can work on other subgoals)

When using `apply H` with a lemma `H : P1 -> P2 -> (...) -> Q`, Coq will create subgoals for each assumption `P1`, `P2`, *etc.* If the lemma has no assumptions, then then `apply H` finishes the goal.

3.2 Tactics that modify a hypothesis

Hypothesis	Tactic
`H : False`	`destruct H`
`H : P /\ Q`	`destruct H as [H1 H2]`
`H : P \/ Q`	`destruct H as [H1|H2]`
`H : exists x, P x`	`destruct H as [x H]`
`H : forall x, P x`	`specialize (H y)`
`H : P -> Q`	`specialize (H G)` (where `G : P` is a lemma or hypothesis)
`H : P`	`apply G in H`, `eapply G in H` (where `G : P -> (...)` is a lemma or hypothesis)
`H : P`, `x : A`	`clear H`, `clear x` (remove hypothesis `H` or variable `x`)

4 Equality, rewriting, and computation rules

Tactic	Meaning
`reflexivity`	Solve goal of the form `x = x` or `P <-> P`
`symmetry`	Turn goal `x = y` into `y = x` (or `P <-> Q`)
`symmetry in H`	Turn hypothesis `H : x = y` into `H : y = x` (or `P <-> Q`)
`rewrite /f`	Replace constant `f` with its definition (only in the goal)
`rewrite /f in H`	Replace constant `f` with its definition (in hypothesis `H`)
`rewrite /f in *`	Replace constant `f` with its definition (everywhere)
`simpl`	Rewrite with computation rules (in the goal)
`simpl in H`	Rewrite with computation rules (in hypothesis `H`)
`simpl in *`	Rewrite with computation rules (everywhere)
`rewrite H`	Rewrite `H : x = y` or `H : P <-> Q` (in the goal)
`rewrite H in G`	Rewrite `H` (in hypothesis `G`)
`rewrite H in *`	Rewrite `H` (everywhere)
`rewrite -H`	Rewrite `H : x = y` backwards
`rewrite H G`	Rewrite using `H` and then `G`
`rewrite !H`	Repeatedly rewrite using `H`
`rewrite ?H`	Try rewriting using `H`
`injection H as H`	Use injectivity of `C` to turn `H : C x = C y` into `H : x = y`
`discriminate H`	Solve goal with inconsistent assumption `H : C x = D y`

Rewriting also works with quantified equalities. For example, if you have `H : forall n m, n + m = m + n` then you can do `rewrite H`. Coq will instantiate `n` and `m` based on what it finds in the goal. You can specify a particular instantiation `n:=3, m:=5` using `rewrite (H 3 5)`, or `m:=5` using `rewrite (H _ 5)`.

5 Inductive types and relations

5.1 Inductive types

Here, `foo` is `bool`, `nat`, `list`, `option`, *etc.*

Term	Tactic
`x : foo`	`destruct x as [a b|c d e|f]`
`x : foo`	`destruct x as [a b|c d e|f] eqn:Hx` (adds equation `Hx : x = (...)` to context)
`x : foo`	`induction x as [a b IH|c d e IH1 IH2|f IH]`

5.2 Getting the right induction hypothesis

The `revert` tactic is useful to obtain the correct induction hypothesis:

Hypothesis	Tactic	
`H : P`	`revert H`	(opposite of `intros H`: turn goal `Q` into `P -> Q`)
`x : A`	`revert x`	(opposite of `intros x`: turn goal `Q` into `forall x, Q`)

A common pattern is `revert x. induction n as ...; intros x; simpl.`

A good rule of thumb is that you should create a separate lemma for each inductive argument, so that `induction` is only ever used at the start of a lemma (possibly preceded by some `revert`).

6 Searching for lemmas and definitions

Command	Meaning
`Search nat`	Prints all lemmas and definitions about `nat`
`Search (0 + _ = _)`	Prints all lemmas containing the pattern `0 + _ = _`
`Search (_ + _ = _) 0`	Prints all lemmas containing `_ + _ = _` and `0`
`Search (list _ -> list _)`	Prints all definitions and lemmas containing the pattern
`Search Nat.add Nat.mul`	Prints all lemmas relating addition and multiplication
`Search "rev"`	Prints all definitions and lemmas containing substring `"rev"`
`Search "+" "*" "="`	Prints all definitions and lemmas containing both `+, *, =`
`Check (1 + 1)`	Prints the type of `1+1`
`Compute (1 + 1)`	Prints the normal form of `1+1`
`Print Nat.add`	Prints the definition of `Nat.add`
`About Nat.add`	Prints information about `Nat.add`
`Locate "+"`	Prints information about notation `+`