Coq Cheatsheet

Formal Foundations of Programming Languages, AS 2023

October 30, 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`, `eapply H` (where `H : (...) -> Q` is a lemma or hypothesis with conclusion `Q`)
`False`	`apply H`, `eapply 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	`done` (simple solver for trivial goals or contradictory hypotheses)
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.

When using `apply H` with a quantified lemma `H : forall x, (...)`, Coq will try to automatically find the right `x` for you. The `apply` tactic will fail if Coq cannot determine `x`. For example, you can then explicitly choose the instantiation `4` for `x` using `apply (H 4)`, or you can use a `with` clause such as `apply H with (x := v)`.[1] You can also use `eapply H` to use an e-var `?x`, which means that the instantiation will be determined later. If there are multiple `forall`-quantifiers you can do `eapply (H _ _ 4)`, to let Coq determine the ones where you put `_`. Similarly, `eexists` will instantiate an existential quantifier with an e-var `?x`. For example, if your goal is `exists n, P n` and you have `H : P 3`, then you can type `eexists; apply H`. This automatically determines that `n` should be `3`.

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`)

3.3 Forward reasoning

Tactic	Meaning
`assert P as H`	Create new hypothesis `H : P` after proving subgoal `P`
`assert P as H by tac`	Create new hypothesis `H : P` after proving subgoal `P` using `tac`
`assert (G := H)`	Duplicate hypothesis
`cut P`	Split goal `Q` into two subgoals `P -> Q` and `P`

[1] The latter is more more flexible as it is insensitive to the order of the universal quantifiers.

Brackets are useful with the assert tactic:

```
assert P as H.
{ (* ... proof of P ... *) }
```

4 Equality, rewriting, and computation

Tactic	Meaning
`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 /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)
`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`)
`replace e1 with e2`	Replaces `e1` by `e2` and generates a new goal that `e1 = e2`
`replace e1 with e2 by ...`	Replaces `e1` by `e2` by proving `e1 = e2` via the tactic ...
`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 //`	Equivalent to `done`
`rewrite /=`	Equivalent to `simpl`
`rewrite H G /=`	Rewrite using `H` and then `G`, and then `simpl`
`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`
`subst x`	Needs a hypothesis `H : x = t` or `H : t = x`; replaces all `x` by `t` and removes `x` and `H`
`subst`	`subst x` for all variables `x` with a suitable equality hypothesis

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, predicates, and relations

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

<table>
<tr><th>Term</th><th>Tactic</th></tr>
<tr><td><code>x : foo</code></td><td><code>destruct x as [a b|c d e|f]</code></td></tr>
<tr><td><code>x : foo</code></td><td><code>destruct x as [a b|c d e|f] eqn:Hx</code> (adds equation <code>Hx : x = ...</code> to context)</td></tr>
<tr><td><code>x : foo x y</code></td><td><code>inversion x</code> (if any of the <code>x</code>, <code>y</code> is not a variable)</td></tr>
<tr><td><code>x : foo</code></td><td><code>induction x as [a b IH|c d e IH1 IH2|f IH]</code></td></tr>
</table>

5.1 Getting the right induction hypothesis

The `revert` and `remember` tactics are useful to obtain the correct induction hypothesis:

Tactic	Meaning
`revert H`	Opposite of `intros H`: turn goal `Q` into `P -> Q`
`revert x`	Opposite of `intros x`: turn goal `Q` into `forall x, Q`
`remember t as x`	Opposite of `subst x`: replaces `t` by a new variable `x` and introduces an equality `x = e`

A common pattern is `revert x; induction n as ...; intros x; simpl`. `remember` can be needed when doing induction over inductive predicates/relations such as `even x` and the `x` is not a variable (*i.e.,* the same situations where `inversion` would be used instead of `destruct`).

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 Introduction patterns

The `destruct x as pat` and `intros pat` tactics can unpack multiple levels at once using nested *intro patterns*. For example, if the goal is `(P /\ exists x : option A, Q1 \/ Q2) -> (...)` then `intros [H [[x|] [G|G]]]` eliminates the conjunction, unpacks the existential, case analyzes the `x : option A`, and case eliminates the disjunction (creating 4 subgoals). The `intros` tactic can also be chained to introduce multiple hypotheses: `intros x y` is equivalent to `intros x; intros y`

Data	Pattern
`exists x, P`	`[x H]`
`P /\ Q`	`[H1 H2]`
`P \/ Q`	`[H1|H2]`
`False`	`[]`
`A * B`	`[x y]`
`A + B`	`[x|y]`
`option A`	`[x|]`
`bool`	`[|]`
`nat`	`[|n]`
`list A`	`[|x xs]`
Inductive type	`[a b|c d e|f]`
Inductive type	`[]` (unpack with names chosen by Coq)
`x = y`	`->` (substitute `x` with `y`)
`x = y`	`<-` (substitute `y` with `x`)
`C x1 x2 = C y1 y2`	`[= H1 H2]` (for constructor `C`, gives `H1 : x1 = y1` and `H2 : x2 = y2`)
`C x1 x2 = C' y1 y2`	`[=]` (for different constructors `C` and `C'`, derives a contradiction)
Any	`?` (introduce variable/hypothesis with name chosen by Coq, **use with care!**)
Any	`pattern%lemma` (`apply lemma in` on the assumption, then continue with `pattern`)

Furthermore, `(x & y & z & ...)` is equivalent to `[x [y [z ...]]]`. This introduction pattern is useful when unpacking definitions with many nested existentials and conjunctions.

Because `exists x, P`, `P /\ Q`, `P \/ Q`, `False` are *defined* as inductive types, their intro patterns are special cases of the introduction pattern for inductive types, and you can also use the `[]` intro pattern for them.

Introduction patterns can be used with the `assert P as pat` tactic, *e.g.,* `assert (A = B) as ->` or `assert (exists x, P) as [x H]`. You can also use them with the `apply H in G as pat` tactic.

7 Automation

Tactic	Meaning
`done`	Simple solver for trivial goals or contradictory hypotheses
`simplify_eq`	Automated tactic that performs `subst`, `injection`, and `discriminate` repeatedly
`congruence`	Solver for equalities with uninterpreted symbols and inductive constructors
`lia`	Solver based on linear integer arithmetic for goals involving `nat`
`tauto`	Solver for propositional tautologies
`eauto`	Solver based on resolution/backwards-chaining

The `eauto` tactic tries to solve goals using `eapply`, `reflexivity`, `eexists`, `split`, `left`, `right`. You can specify the search depth using `eauto n` (the default is $n = 5$).

You can give `eauto` additional lemmas to use with `eauto using lemma1, lemma2`. You can also use `eauto using foo` where `foo` is an inductive type. This will use all the constructors of `foo` as lemmas.

You can permanently add lemmas to the `eauto` database, which will then be used for all future invocations of the tactic, using `Hint Resolve lemma : core`.

8 Composing tactics

Tactic	Meaning
`tac1; tac2`	Do `tac2` on all subgoals created by `tac1`
`tac1; first tac2`	Do `tac2` only on the first subgoal
`tac1; last tac2`	Do `tac2` only on the last subgoal
`tac1; [tac2|..|tac3|tac4]`	Do `tac2` on the first subgoal, `tac3` and `tac4` on the last two subgoals, and nothing on the rest
`tac1; [tac2|tac3..|tac4]`	Do `tac2` on first subgoal, `tac4` on the last subgoal, and `tac3` on the rest
`tac1 || tac2`	Try `tac1` and if it fails do `tac2`
`try tac1`	Try `tac1`, and do nothing if it fails
`repeat tac1`	Repeatedly do `tac1` until it fails
`progress tac1`	Do `tac1` and fail if it does nothing
`by tac`	Shorthand for `tac; done`

9 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 `+`