COQ CHEATSHEET

Jules Jacobs

October 5, 2022

1 INTRODUCTION

This is Coq code that proves the strong induction principle for natural numbers:

```
From Coq Require Import Lia.

Lemma strong_induction (P : nat -> Prop) :
  (forall n, (forall m, m < n -> P m) -> P n) -> forall n, P n.
Proof.
  intros H n. eapply H. induction n.
  - lia.
  - intros m Hm. eapply H.
    intros k Hk. eapply IHn. lia.
Qed.
```

Coq proofs manipulate the *proof state* by executing a sequence of *tactics* such as `intros`, `eapply`, `induction`. Coq calculates the proof state for you after executing each tactic. Here's what Coq displays after executing the second `intros m Hm.`:

```
P: nat -> Prop
H: forall n : nat, (forall m : nat, m < n -> P m) -> P n
n: nat
IHn: forall m : nat, m < n -> P m
m: nat
Hm: m < S n
----------------------------
P m
```

The proof state consists of a list of variables and hypotheses above the line, and a goal below the line. Executing a tactic may result in zero or more subgoals. A subgoal is solved if we succesfully apply a tactic that creates no new subgoals (such as the `lia` tactic, which solves simple numeric goals). Some tactics create multiple subgoals, such as the `induction` tactic: it creates one subgoal for the base case of the induction, and one subgoal for the inductive case. If a tactic creates multiple subgoals, we solve them using a bulleted list of tactic scripts, or using brackets:

```
(* Simple bullets *)
tac1.
+ tac2.
+ tac3.
+ tac4.
+ tac5.
+ tac6.
```

```
(* Nested bullets *)
tac1.
+ tac2.
  * tac3
  * tac4.
+ tac5.
 ++ tac6.
```

```
(* Brackets *)
tac1.
{ tac2. }
{ tac3. }
tac4.
{ tac5. }
tac6.
```

We usually use bullets if the subgoals are on equal footing, and we use brackets for simple side-conditions. We do not have to enclose the last subgoal in brackets, thus preventing deep nesting.

2 LOGICAL REASONING

2.1 *Tactics that modify the goal*

Goal	Tactic
$P \to Q$	`intros H`
$\neg P$	`intros H` (Coq defines $\neg P$ as $P \to \mathsf{False}$)
$\forall x, P(x)$	`intros x`
$\exists x, P(x)$	`exists x`, `eexists`
$P \wedge Q$	`split` (also works for $P \leftrightarrow Q$, which is defined as $(P \to Q) \wedge (Q \to P)$)
$P \vee Q$	`left`, `right`
Q	`apply H`, `eapply H` (where `H` : (...) $\to Q$ is a lemma with conclusion Q)
False	`apply H`, `eapply H` (where `H` : (...) $\to \neg P$ is a lemma with conclusion $\neg P$)
Any goal	`exfalso` (turns any goal into False)
Skip goal	`admit` (skips goal so that you can work on other subgoals)

When using `apply H` with a lemma `H` : $P_1 \to P_2 \to Q$, Coq will create subgoals for each assumption P_1 and P_2. 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. You can then explicitly choose an instantiation $x = 4$ using `apply (H 4)`. You can also use `eapply H` to use an E-var ?x, which means that the instanation 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. If your goal is $\exists n, P\ n$ and you have `H` : P 3, then you can type `eexists. apply H.` This automatically determines $n = 3$.

2.2 *Tactics that modify a hypothesis*

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

2.3 *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 $P \to Q$ and P

Brackets are useful with the assert tactic: `assert P as H. { ... proof of P ... }`

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`)
`unfold f`	Replace constant `f` with its definition (only in the goal)
`unfold f in H`	Replace constant `f` with its definition (in hypothesis `H`)
`unfold 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`.
`subst`	Substitute away all equations `H : x = A` with a variable on one side.
`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`.
`simplify_eq`	Automated tactic that does `subst`, `injection`, and `discriminate` automatically.

Rewriting also works with quantified equalities. 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)`.

The `simplify_eq` tactic is from stdpp. Although it is not a built-in tactic, I mention it because it is incredibly useful.

4 INDUCTIVE TYPES AND RELATIONS

4.1 *Inductive types* Foo

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:E` (adds equation `E : x = (...)` to context)
x : Foo	`induction x as [a b IH|c d e IH1 IH2|f IH]`

4.2 *Inductive relations* Foo x y

Goal/Hypothesis	Tactic
Foo x y	`constructor`, `econstructor` (tries applying all constructors of Foo)
H : Foo x y	`inversion H` (use when x,y are fixed terms)
H : Foo x y	`induction H` (use when x,y are variables)

It is often useful to define the tactic `Ltac inv H := inversion_clear H; subst.` and use this instead of `inversion`.

4.3 *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 $P \to Q$)
x : A	`revert x` (opposite of `intros x`: turn goal Q into $\forall x, Q$)

A common pattern is `revert x. induction n; 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`).

5 PROOF SEARCH WITH `eauto`

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.

6 INTRO PATTERNS

The `destruct x as pat` and `intros pat` tactics can unpack multiple levels at once using nested *intro patterns*: if the goal is $(P \land \exists x : \mathtt{option}\ A, Q_1 \lor Q_2) \to (\ldots)$ then `intros [H [[x|] [G|G]]]` splits the conjunction, unpacks the existential, case analyzes the $x : \mathtt{option}\ A$, and case analyzes the disjunction (creating 4 subgoals). The intros tactic can also be chained to introduce multiple hypotheses: `intros x y.` ≡ `intros x. intros y.`

Data	Pattern
$\exists x, P$	`[x H]`
$P \land Q$	`[H1 H2]`
$P \lor Q$	`[H1|H2]`
$\mathtt{False}$	`[]`
$A * B$	`[x y]`
$A + B$	`[x|y]`
$\mathtt{option}\ A$	`[x|]`
$\mathtt{bool}$	`[|]`
$\mathtt{nat}$	`[|n]`
$\mathtt{list}\ A$	`[x xs|]`
Inductive type	`[a b|c d e|f]`
Inductive type	`[]` (unpack with names chosen by Coq)
$x = y$	`->` (substitute the equality $x \mapsto y$)
$x = y$	`<-` (substitute the equality $y \mapsto x$)
Any	`?` (introduce variable/hypothesis with name chosen by Coq)

Furthermore, `(x & y & z & ...)` is equivalent to `[x [y [z ...]]]`.

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

Intro 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 COMPOSING TACTICS

Tactic	Meaning
`tac1; tac2`	Do `tac2` on all subgoals created by `tac1`.
`tac1; [tac2|..]`	Do `tac2` only on the first subgoal.
`tac1; [..|tac2]`	Do `tac2` only on the last subgoal.
`tac1; [tac2|..|tac3|tac4]`	Do tactics on corresponding subgoals.
`tac1; [tac2|tac3..|tac4]`	Do tactics on corresponding subgoals.
`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.

In the examples above, the two dots are part of the Coq syntax.

8 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 "+".

9 EXAMPLES OF CUSTOM TACTICS

```
(* Simplifies equations by doing substitution and injection. *)
Tactic Notation "simplify_eq" := repeat match goal with
  | _ => congruence || (progress subst)
  | H : ?x = ?x |- _ => clear H
  | H : _ = _ |- _ => progress injection H as H
  | H1 : ?o = Some ?x, H2 : ?o = Some ?y |- _ =>
    assert (y = x) by congruence; clear H2
  end.

(* Inversion tactic that cleans up the original hypothesis and generated equalities. *)
Ltac inv H := inversion_clear H; simplify_eq.

Ltac simp := repeat match goal with
  | H : False |- _ => destruct H
  | H : ex _ |- _ => destruct H
  | H : _ /\ _ |- _ => destruct H
  | H : _ * _ |- _ => destruct H
  | H : ?P -> ?Q, H2 : ?P |- _ => specialize (H H2)
  | |- forall x,_ => intro
  | _ => progress (simpl in *; simplify_eq)
  | _ => solve [eauto]
  end.

Ltac cases := repeat match goal with
  | H : _ \/ _ |- _ => destruct H
  | |- _ /\ _ => split
  | |- context[if ?x then _ else _] => destruct x eqn:?
  | |- context[match ?x with _ => _ end] => destruct x eqn:?
  | H : context[if ?x then _ else _] |- _ => destruct x eqn:?
  | H : context[match ?x with _ => _ end] |- _ => destruct x eqn:?
  end.
```

http://julesjacobs.com/notes/coq-cheatsheet/tactics.v