

# 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 successfully 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 *)	(* Nested bullets *)	(* Brackets *)
tac1.	tac1.	tac1.
+ tac2.	+ tac2.	{ tac2. }
+ tac3.	* tac3	{ tac3. }
+ tac4.	* tac4.	tac4.
+ tac5.	+ tac5.	{ tac5. }
+ tac6.	++ tac6.	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
$P \rightarrow Q$	<code>intros H</code>
$\neg P$	<code>intros H</code> (Coq defines $\neg P$ as $P \rightarrow \text{False}$ )
$\forall x, P(x)$	<code>intros x</code>
$\exists x, P(x)$	<code>exists x, eexists</code>
$P \wedge Q$	<code>split</code> (also works for $P \leftrightarrow Q$ , which is defined as $(P \rightarrow Q) \wedge (Q \rightarrow P)$ )
$P \vee Q$	<code>left, right</code>
$Q$	<code>apply H, eapply H</code> (where $H : (...) \rightarrow Q$ is a lemma with conclusion $Q$ )
$\text{False}$	<code>apply H, eapply H</code> (where $H : (...) \rightarrow \neg P$ is a lemma with conclusion $\neg P$ )
Any goal	<code>exfalse</code> (turns any goal into $\text{False}$ )
Skip goal	<code>admit</code> (skips goal so that you can work on other subgoals)

When using `apply H` with a lemma  $H : P_1 \rightarrow P_2 \rightarrow Q$ , 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 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. 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
$H : \text{False}$	<code>destruct H</code>
$H : P \wedge Q$	<code>destruct H as [H1 H2]</code>
$H : P \vee Q$	<code>destruct H as [H1 H2]</code>
$H : \exists x, P(x)$	<code>destruct H as [x H]</code>
$H : \forall x, P(x)$	<code>specialize (H y)</code>
$H : P \rightarrow Q$	<code>specialize (H G)</code> (where $G : P$ is a lemma or hypothesis)
$H : P$	<code>apply G in H, eapply G in H</code> (where $G : P \rightarrow (...)$ is a lemma or hypothesis)
$H : P, x : A$	<code>clear H, clear x</code> (remove hypothesis $H$ or variable $x$ )

### 2.3 Forward reasoning

Tactic	Meaning
<code>assert P as H</code>	Create new hypothesis $H : P$ after proving subgoal $P$
<code>assert P as H by tac</code>	Create new hypothesis $H : P$ after proving subgoal $P$ using <code>tac</code>
<code>assert (G := H)</code>	Duplicate hypothesis
<code>cut P</code>	Split goal $Q$ into two subgoals $P \rightarrow Q$ and $P$

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

### 3 EQUALITY, REWRITING, AND COMPUTATION RULES

Tactic	Meaning
<code>reflexivity</code>	Solve goal of the form $x = x$ or $P \leftrightarrow P$
<code>symmetry</code>	Turn goal $x = y$ into $y = x$ (or $P \leftrightarrow Q$ )
<code>symmetry in H</code>	Turn hypothesis $H : x = y$ into $H : y = x$ (or $P \leftrightarrow Q$ )
<code>unfold f</code>	Replace constant $f$ with its definition (only in the goal)
<code>unfold f in H</code>	Replace constant $f$ with its definition (in hypothesis $H$ )
<code>unfold f in *</code>	Replace constant $f$ with its definition (everywhere)
<code>simpl</code>	Rewrite with computation rules (in the goal)
<code>simpl in H</code>	Rewrite with computation rules (in hypothesis $H$ )
<code>simpl in *</code>	Rewrite with computation rules (everywhere)
<code>rewrite H.</code>	Rewrite $H : x = y$ or $H : P \leftrightarrow Q$ (in the goal).
<code>rewrite H in G.</code>	Rewrite $H$ (in hypothesis $G$ ).
<code>rewrite H in *.</code>	Rewrite $H$ (everywhere).
<code>rewrite &lt;-H.</code>	Rewrite $H : x = y$ backwards.
<code>rewrite H,G.</code>	Rewrite using $H$ and then $G$ .
<code>rewrite !H.</code>	Repeatedly rewrite using $H$ .
<code>rewrite ?H.</code>	Try rewriting using $H$ .
<code>subst</code>	Substitute away all equations $H : x = A$ with a variable on one side.
<code>injection H as H</code>	Use injectivity of $C$ to turn $H : C x = C y$ into $H : x = y$ .
<code>discriminate H</code>	Solve goal with inconsistent assumption $H : C x = D y$ .
<code>simplify_eq</code>	Automated tactic that does <code>subst</code> , <code>injection</code> , and <code>discriminate</code> 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
$x : \text{Foo}$	<code>destruct x as [a b c d e f]</code>
$x : \text{Foo}$	<code>destruct x as [a b c d e f] eqn:E</code> (adds equation $E : x = (\dots)$ to context)
$x : \text{Foo}$	<code>induction x as [a b IH c d e IH1 IH2 f IH]</code>

### 4.2 Inductive relations Foo x y

Goal/Hypothesis	Tactic
$\text{Foo } x \ y$	<code>constructor, econstructor</code> (tries applying all constructors of Foo)
$H : \text{Foo } x \ y$	<code>inversion H</code> (use when $x, y$ are fixed terms)
$H : \text{Foo } x \ y$	<code>induction H</code> (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
$H : P$	<code>revert H</code> (opposite of <code>intros H</code> : turn goal $Q$ into $P \rightarrow Q$ )
$x : A$	<code>revert x</code> (opposite of <code>intros x</code> : 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  $(P \wedge \exists x : \text{option } A, Q_1 \vee Q_2) \rightarrow (\dots)$  then `intros [H [[x]] [G|G]]` splits the conjunction, unpacks the existential, case analyzes the  $x : \text{option } A$ , and case analyzes the disjunction (creating 4 subgoals). The `intros` tactic can also be chained to introduce multiple hypotheses: `intros x y.  $\equiv$  intros x. intros y.`

Data	Pattern
$\exists x, P$	[x H]
$P \wedge Q$	[H1 H2]
$P \vee Q$	[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 the equality $x \mapsto y$ )
$x = y$	<- (substitute the equality $y \mapsto x$ )
Any	? (introduce variable/hypothesis with name chosen by Coq)

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

Because  $\exists x, P, P \wedge Q, P \vee Q, \text{False}$  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
<code>tac1; tac2</code>	Do <code>tac2</code> on all subgoals created by <code>tac1</code> .
<code>tac1; [tac2 ..]</code>	Do <code>tac2</code> only on the first subgoal.
<code>tac1; [.. tac2]</code>	Do <code>tac2</code> only on the last subgoal.
<code>tac1; [tac2 .. tac3 tac4]</code>	Do tactics on corresponding subgoals.
<code>tac1; [tac2 tac3.. tac4]</code>	Do tactics on corresponding subgoals.
<code>tac1    tac2</code>	Try <code>tac1</code> and if it fails do <code>tac2</code> .
<code>try tac1</code>	Try <code>tac1</code> , and do nothing if it fails.
<code>repeat tac1</code>	Repeatedly do <code>tac1</code> until it fails.
<code>progress tac1</code>	Do <code>tac1</code> 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
<code>Search nat.</code>	Prints all lemmas and definitions about <code>nat</code> .
<code>Search (0 + _ = _).</code>	Prints all lemmas containing the pattern <code>0 + _ = _</code> .
<code>Search (_ + _ = _) 0.</code>	Prints all lemmas containing <code>_ + _ = _</code> and <code>0</code> .
<code>Search (list _ -&gt; list _).</code>	Prints all definitions and lemmas containing the pattern.
<code>Search Nat.add Nat.mul.</code>	Prints all lemmas relating addition and multiplication.
<code>Search "rev".</code>	Prints all definitions and lemmas containing substring "rev".
<code>Search "+" "*" "=".</code>	Prints all definitions and lemmas containing both <code>+</code> , <code>*</code> , <code>=</code> .
<hr/>	
<code>Check (1+1).</code>	Prints the type of <code>1+1</code>
<code>Compute (1+1).</code>	Prints the normal form of <code>1+1</code> .
<code>Print Nat.add.</code>	Prints the definition of <code>Nat.add</code>
<code>About Nat.add.</code>	Prints information about <code>Nat.add</code> .
<code>Locate "+".</code>	Prints information about notation <code>"+"</code> .

## 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>