Coq Tactics Cheat Sheet

Martin Irungu

2025-04-26

Contents

Overview

Tactics specify how to transform the *proof state* of an incomplete proof to eventually generate a complete proof. Here's a general guide on how to use tactics in Coq:

If you wish to . . .	try the tactic
prove by contradiction `p /\ ~p`	`absurd p`
simplify expressions	`simpl`
prove via the intermediate goal `p`	`cut p` or `enough(H:p)`
prove by induction on `t`	`induction t`
skip a goal so you can work on others	`admit`
turn current goal into `False`	`exfalso`

A general guide on goal transformation:

When the goal is . . .	try the tactic
very simple	`auto` or `tauto`
`p /\ q`	`split`
`p \/ q`	`left` or `right`
`p -> q`	`intros H`
`~p`	`intros H` (recall $\neg p$ is defined as `p -> False`)
`p <-> q`	`split` (recall this is `(p -> q) /\ (q -> p)`)
an assumption	`assumption`
`forall x, p`	`intros x`
`forall x, p -> q`	`intros x H`
`exists x, p`	`exists t`

Here's a summary on how to manipulate hypotheses.

To use hypothesis H :	try the tactic
`p = q`	`rewrite H` or `rewrite <- H`
`p /\ q`	`destruct H as [H1 H2]`
`p \/ q`	`destruct H as [H1 | H2]`
`p`	`apply G in H` (assuming `G: p -> (...)`)
`p -> q`	`apply H` (or `specialize (H G)` assuming `G:p`)
`~p`	`apply H` or `elim H`
`p <-> q`	`apply H`
`False`	`contradiction` or `destruct H`
`forall x, p`	`specialize (H y)`
`exists x, p`	`destruct H as [x G]`

Intro patterns

An introduction pattern is either:

- The wildcard: `_`
- A variable: e.g., `x`, `H`, `p`
- A disjunction of lists of patterns: `[ p11...p1m_1 | ...| pn1...pnm_n ]`
- A conjunction of patterns: `( p1,...,pn )`

The behavior of the `intros` tactic is defined inductively over the structure of the given pattern:

- introduction on the wildcard do the introduction and then immediately clears the corresponding hypothesis, effectively skipping naming.
- introduction on a variable behaves like a plain `intro`, naming the hypothesis or binder locally.
- introduction over a *list of patterns* `p1...pn` is equivalent to the sequence of introductions over the patterns namely: `intros p1;...;intros pn`, the goal should start with at least `n` products (Π-types like implications or universal quantifiers).

- introduction over a *disjunction of list of patterns* `[p11...p1m_1 | ...| pn1...pnm_n]`. It introduces a new variable `X`, (its type should be an inductive definition with `n` constructors (the number of disjunctions), then it performs a case analysis over `X` (which generates `n` subgoals), it clears `X` and performs on each generated subgoals the corresponding `intros pi1...pim_i` tactic;
- introduction over a *conjunction of patterns* `(p1,...,pn)`, it introduces a new variable `X`, its type should be an inductive definition with 1 constructor with (at least) `n` arguments, then it performs a case analysis over `X` (which generates 1 subgoal with at least `n` products), it clears `X` and performs an introduction over the list of patterns `p1 ... pn`.

Basic Usage and Examples

You can unpack multiple levels at once using nested intro patterns: if the goal is `P /\ exists x : option A, Q1 \/ Q2) ->(...)` then `intros [H [[x|] [G|G]]]` splits the conjunction, unpacks the existential, case analyzes the `x : 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	Description
$\exists x,\ P$	`[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)

```
Goal forall A B C:Prop, A \/ B /\ C -> (A -> C) -> C.
intros A B C [HA|[_ HC]] HAC.
```

```
1 goal

  ============================
  forall A B C : Prop, A \/ B /\ C -> (A -> C) -> C

2 goals

  A, B, C : Prop
  HA : A
  HAC : A -> C
  ============================
  C

goal 2 is:
 C
```

Summary of intro patterns

Tactic	Description	Transform
`Name`	Introduce a term (hypothesis or variable) as `Name`	$\frac{}{H : P} \rightsquigarrow \frac{Name : H}{P}$
`[x|n H1 H2]`	Introduce a term and destruct it. If the type has multiple constructors they are separated by |. The arguments in a single constructor are separated by spaces.	$\frac{}{(\exists k, H) \to P} \rightsquigarrow \frac{k : \mathbb{N} \quad H1 : H}{P}$
`*`	Introduce one or more quantified variables until there are no more quantified variables.	$\frac{}{\forall H1\, H2, H3 \to P} \rightsquigarrow \frac{n1 : H1, n2 : H2}{H3 \to P}$
`H1%H`	Introduce a term and apply `H` in it.	$\frac{H : X \to Y}{X \to P} \rightsquigarrow \frac{H : X \to Y \quad H1 : Y}{P}$
`->`	Rewrite the equation. Can also be used as `<-`.	$\frac{}{(X = Y) \to Y} \rightsquigarrow \frac{}{X}$
`(H1 & H2 & H3)`	Introduce a nested term with multiple arguments and split it.	$\frac{}{X \wedge Y \wedge Z \to P} \rightsquigarrow \frac{H1 : X \quad H2 : Y \quad H3 : Z}{P}$
`[= H]`	Introduce a term and apply injectivity and discriminate on it.	$\frac{}{Sx = Sy \to P} \rightsquigarrow \frac{H : x = y}{P}$

Note that `(H1 & H2 & H3 & ...)` is equivalent to `[H1 [H2 [H3 ...]]]`.

Basic proof management

`intros`: Introduces hypotheses, variables, or decomposes implications/universal quantifiers.

The `intros` tactic finds assumptions built into your goal (usually in the form of a `forall` quantifier) and moves them to the goal's context (or hypothesis space). This is similar to the first step of many informal, paper proofs, when the prover states “let there be some number $n \ldots$" More specifically, **intros** *specializes* a goal by looking for type inhabitation and proposition assumptions and moving them into the assumption space. For example, if you write

```
forall (n : nat), n + 0 = n
```

the `forall` is acting as an assumption that there is a value of type `nat` that we can call `n`. Calling `intros` here will provide you an assumption `n` that there is a value of type `nat`.
If used without arguments, the names of the assumptions are chosen by default to be the variables. then the hypotheses H, H_0, H_1 etc. It is good practice to name these yourself.

$$\frac{}{\forall x\,y, P \to Q \to R} \quad \rightsquigarrow \quad \begin{array}{c} x : X \\ y : Y \\ H1 : P \\ H2 : Q \\ \hline R \end{array}$$

Some examples. Pay attention to the state before and after:

```
Theorem plus_0_n: forall(n:nat),
    0 + n = n.
Proof.
  intros n.
```

```
1 goal

  ============================
  forall n : nat, 0 + n = n
```

```
1 goal

  n : nat
  ============================
  0 + n = n
```

```
Theorem plus_id_exercise : forall n m o : nat,
  n = m -> m = o -> n + m = m + o.
Proof. intros n m o H H'. Show Proof.
```

```
1 goal

  ============================
  forall n m o : nat, n = m -> m = o -> n + m = m + o
```

```
1 goal

  n, m, o : nat
  H : n = m
  H' : m = o
  ============================
  n + m = m + o
```

```
(fun (n m o : nat) (H : n = m) (H' : m = o) => ?Goal)
```

The variant `introv` allows to automatically introduce the variables of a theorem and explicitly name the hypotheses involved.

Proof term

$(\lambda\, x\, y\, H_1\, H_2 \Rightarrow \blacksquare)$ where $\blacksquare$ is the gap in the proof term which still needs to be closed.

`apply`: Uses implications to transform goals and hypotheses.

If we have a hypothesis that says that $H : X \implies Y$, to prove Y all we really have to do is prove X. The tactic `apply` tries to match the current goal against the conclusion of the type of term. We can apply that hypothesis to a goal of y to transform it into x. It can be thought of as applying the function $H : X \to Y$ by Curry-Howard.

$$\frac{H : X \to Y}{Y} \quad \rightsquigarrow \quad \frac{H : X \to Y}{X}$$

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 `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)`.

```
Lemma modus_ponens:
  forall x y : Prop, (x -> y) -> x -> y.
Proof.
  intros.
  apply H.
```

```
1 goal

  ============================
  forall x y : Prop, (x -> y) -> x -> y
```

```
1 goal

  x, y : Prop
```

```
  H : x -> y
  H0 : x
  ============================
  y

1 goal

  x, y : Prop
  H : x -> y
  H0 : x
  ============================
  x
```

`apply H with (x :=a)`

This variant applies H where x is instantiated with a. Example:

```
Example trans_eq_example' : forall (a b c d e f : nat),
    [a;b] = [c;d] ->
    [c;d] = [e;f] ->
    [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  apply trans_eq with (m:=[c;d]).
  apply eq1. apply eq2.   Qed.
```

`eapply`

The tactic `eapply` behaves as `apply` but does not fail when no instantiation are deducible for some variables in the premises. Rather, it turns these variables into so-called *existential variables* or e-vars, which are variables still to instantiate. You can use `eapply H` to use an e-var `?x`, which means that the instantiation will be determined later in the proof. If there are multiple forall-quantifiers you can do `apply (H _ _ 4 _)`, to let Coq determine the ones where you put `_`.

`exact`: Solves a goal by supplying the exact proof term.

Suppose your current goal is to prove some proposition `P`. If you already have a term `t` in scope (e.g., from hypotheses or definitions) such that `t : P`, then `exact t.` tells Coq: "Here is the proof you're asking for." Coq checks whether `t` indeed has the required type (i.e., it matches the goal exactly). If it does, the goal is discharged.

```
Goal forall A B : Prop, A -> (A -> B) -> B.
Proof.
  intros A B HA HAB.
  exact (HAB HA). (* Applying a function of type A -> B to a term of type A *)
Qed.
```

```
1 goal

  ============================
  forall A B : Prop, A -> (A -> B) -> B

1 goal

  A, B : Prop
  HA : A
  HAB : A -> B
  ============================
  B

No more goals.
```

`assumption`: Solves the goal using an existing hypothesis.

The tactic `assumption` attempts to close the current proof goal by finding a hypothesis in the context **exactly** matching the goal's conclusion. It searches your local context for a hypothesis `H : G` where `G` is convertible (i.e., judgmentally equal) to the current goal. If found, it applies `H` and the proof is complete. The `eassumption` variant behaves like `assumption` but can handle goals with existential variables.

Negation and contradiction

`absurd`: Proves a goal by showing a specific proposition `P` is both true and false

The tactic `absurd P` applies `False` elimination, i.e. it deduces the current goal from `False`, then generates as subgoals `~P` and `P`. It is very useful in proofs by cases, where some cases are impossible. In most cases, `P` or `~P` is one of the hypotheses of the local context.

```
 Goal Q.
Proof.
  absurd P. (* Subgoals: [P] and [~P] *)
  - (* Prove P here *)
  - (* Prove ~P here *)
Qed.
```

`contradiction`: Solves any goal if the context contains `False` or contradictory hypotheses.

The `contradiction` tactic attempts to find in the current context (after all `intros`) one which is equivalent to `False`. It searches for hypotheses of the form `P` and `~P` (or `False`). If found, it immediately solves the goal. This tactic is a macro for the tactics sequence `intros; elimtype False; assumption`.

Examples

```
Theorem law_of_contradiction : forall (P Q : Prop),
  P /\ ~P -> Q.
Proof.
  intros P Q P_and_not_P.
  destruct P_and_not_P as [P_holds not_P].
  contradiction.
```

```
1 goal

  ============================
  forall P Q : Prop, P /\ ~ P -> Q

1 goal

  P, Q : Prop
  P_and_not_P : P /\ ~ P
  ============================
  Q

1 goal

  P, Q : Prop
  P_holds : P
  not_P : ~ P
  ============================
  Q

No more goals.
```

`exfalso` Replaces the current goal with `False`.

The tactic `exfalso` (from the Latin *"ex falso quodlibet"*: from falsehood, anything, or the *principle of explosion*) changes the goal to `False`, allowing you to derive a contradiction from the context afterward.

`contradict`: Manipulates negated hypotheses and goals,

The `contradict` tactic in Coq is designed for manipulating negated hypotheses and goals, effectively transforming the proof state by "flipping" negations between hypotheses and the current goal. When invoked as `contradict H`, where `H` names a hypothesis, it rewrites the proof state according to four core transformation rules:

- From $H : \neg A \vdash B$ it produces $\vdash A$.
- From $H : \neg A \vdash \neg B$ it produces the new context $H : B \vdash A$.
- From $H : A \vdash B$ it produces the goal $\vdash \neg A$.
- From$H : A \vdash \neg B$ it produces the new context $H : B \vdash \neg A$.

Notice the hypothesis gets removed from the context if the goal is negated.

`false` and `tryfalse`

This can derive the current goal from `False`. It is a shorthand for `exfalso`, but `false` proves the goal if it contains an absurd assumption, such as `False` or `0 = S n`, or if it contains contradictory assumptions, such as `x = true` and `x = false`. The tactic `false` can take an argument; `false H` replace the goals with `False` and then applies `H`. The tactic `tryfalse` is a shorthand for `try solve [false]`: it tries to find a contradiction in the goal. It is generally called after a case analysis.

Hypothesis and context management

`clear`: Removes hypotheses

The tactic `clear H` removes the named hypothesis from the current proof context. Fails if you try to remove a hypothesis that other hypotheses or the goal depend upon (unless you use `clear dependent`).

```
Lemma many_hypotheses :
  forall A B C : Prop, A -> B -> C -> A /\ C.
Proof.
  intros A B C HA HB HC.
  clear HB.          (* B wasn't needed for this goal *)
  split; [ exact HA | exact HC ].
Qed.
```

```
1 goal

  ============================
  forall A B C : Prop, A -> B -> C -> A /\ C

1 goal

  A, B, C : Prop
  HA : A
  HB : B
  HC : C
  ============================
  A /\ C

1 goal

  A, B, C : Prop
  HA : A
  HC : C
  ============================
  A /\ C

No more goals.
```

`rename`: Renames hypotheses and variables

The tactic `rename` changes the name of an introduced variable or assumption.

```
Goal forall n:nat, n=n. intros. rename n into x.
```

```
1 goal

  ============================
  forall n : nat, n = n

1 goal

  n : nat
  ============================
  n = n

1 goal

  x : nat
  ============================
  x = x
```

remember: Names an expression to avoid losing it across destruct or induction.

The `remember` tactic in Coq abstracts a complex term by replacing all its occurrences with a fresh variable and simultaneously introducing an equality hypothesis relating that variable to the original term. Given a term `t` of type `T`, `remember t` simply introduces a new variable `x` of type `T`, an hypothesis stating the equality `x = t` and replaces the instances of the term `t` by the variable `x`. The usual variants work:

```
Goal forall x y: nat, x+y=y -> y=0.
intros x y H.  remember (x + y) as sum eqn: Hsum.
```

```
1 goal

  ============================
  forall x y : nat, x + y = y -> y = 0

1 goal

  x, y : nat
  H : x + y = y
  ============================
  y = 0

1 goal

  x, y, sum : nat
  Hsum : sum = x + y
  H : sum = y
  ============================
  y = 0
```

This is mainly useful because, for technical reasons, when you perform induction over an hypothesis of an inductive type, it first generalizes all of its arguments. If your hypothesis is too weak, you may find yourself without enough information to complete the proof, and this . (see the IndProp chapter in (Pierce *and others* 2025) for more)

revert: Reverts a hypothesis back into the goal.

This tactic moves a hypotheses from the context back into the goal, turning a hypothesis `H : P` into a premise `P -> G` of the current goal. It is the logical inverse of `intros`; while `intros H` moves a universal quantifier or implication from the goal into the context as a hypothesis, `revert H` pushes it back out.

Hypothesis	Tactic
`H:P`	`revert H` (opposite of `intros H`; turns the goal Q into $P \to Q$)
`x:T`	`revert x` (opposite of `intros x`; turns the 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`).

Examples

$$\frac{\mathsf{H : P}}{\mathsf{Q}} \rightsquigarrow \frac{}{\mathsf{P \to Q}} \quad \text{and} \quad \frac{\mathsf{x : T}}{\mathsf{Q}} \rightsquigarrow \frac{}{\mathsf{\forall x.Q}}$$

The specialized form `revert dependent x` will automatically detect and move `x` along with all hypotheses that depend on `x`.

generalize: Adds universal quantification to the goal.

The tactic applies to any goal. It generalizes the conclusion w.r.t. one subterm of it. If the goal is `G` and `t` is a subterm of type `T` in the goal, then `generalize t` replaces the goal by `forall (x:T), G'` where `G'` is obtained from `G` by replacing all occurrences of `t` by `x`. The name of the variable is chosen accordingly to `T`. For example:

```
Lemma positivity:
  forall (x y : nat), 0 <= x + y + y.
Proof.
  intros. generalize (x + y + y).
```

```
1 goal

  ============================
  forall x y : nat, 0 <= x + y + y

1 goal

  x, y : nat
  ============================
  0 <= x + y + y
```

```
1 goal

  x, y : nat
  ============================
  forall n : nat, 0 <= n
```

$$\frac{}{\mathsf{P}} \quad \rightsquigarrow \quad \frac{}{\forall \mathsf{x}.\mathsf{P}'} \text{ (where } \mathsf{P}' \text{ is } \mathsf{P} \text{ with all occurences of the term replaced by } \mathsf{x}.)$$

Note that unlike `revert`, `generalize` works on arbitrary terms, even those not named by identifiers, allowing you to abstract complex subexpressions directly.
The `generalize dependent` variant goes further by also moving any hypotheses that depend on the given term back into the goal, ensuring that dependencies are correctly captured when reordering or reusing variables.

`specialize`: Instantiates a universally quantified hypothesis with specific arguments

`specialize(H:e)` instantiates an assumption H by passing it an argument e If H is a quantified hypothesis in the current context i.e., `H : (x:T), P` then `specialize H with (x :=e)` will change H so that it looks like `[x:=e]P`, that is, P with x replaced by e.

$$\frac{\mathsf{H} : \forall \mathsf{x}, \mathsf{P}}{\mathsf{Q}} \quad \rightsquigarrow \quad \frac{\mathsf{H} : \mathsf{P}^{\mathsf{x}}_{\mathsf{e}}}{\mathsf{Q}}$$

- We can also use the `as` variant.

Examples

```
Goal (forall n m : nat, n + m = m + n) -> True.
Proof.
  intros H.
  (* Before: H : forall n m : nat, n + m = m + n *)
  specialize (H 1 2).
  (* After:  H : 1 + 2 = 2 + 1 *)
  exact I.
Qed.
```

```
1 goal

  ============================
  (forall n m : nat, n + m = m + n) -> True

1 goal

  H : forall n m : nat, n + m = m + n
  ============================
  True
```

```
1 goal

  H : 1 + 2 = 2 + 1
  ============================
  True

No more goals.
```

- **Proof term:** $(\lambda(H : P^x_e \Rightarrow \blacksquare)(He)$

`pose proof`: Creates new hypotheses out of existing hypotheses

The `pose proof` tactic allows you to introduce a new hypothesis into the context by applying an existing lemma, hypothesis, or constructed term, without modifying the original source of that fact. Unlike `assert`, which generates subgoals for proving the asserted fact, or `specialize`, which transforms an existing hypothesis in place, `pose proof` purely adds a fresh copy.

Examples

- `pose proof (eq_refl a)` creates a new hypothesis `a = a`.
- `pose proof (H x)`. Given $H : \forall m, P$, create a new hypothesis stating that `P` holds for `x`. For example, `pose proof (Nat.add_comm 3 5) as H` adds the hypothesis `H: 3 + 5 = 5 + 3` to the context.
- `pose proof (H1 H2)`: Given `H1 : P -> Q` and `H2 : P` , create a new hypothesis `Q`.

Intermediate steps

Use `assert` for helper lemmas, `enough` when you want to “reverse-engineer” the goal, and `cut` for classical reasoning or splitting proofs into phases.

`assert`: Add a hypothesis to the context by proving it first.

The tactic `assert (H : Q)` (or `assert Q as H`) adds a new hypothesis of name `H` asserting `Q` to the current goal and opens a new subgoal `Q`. The subgoal `Q` comes first in the list of subgoals remaining to prove.
Example:

```
Theorem mult_0_plus' : n m : nat,
  (n + 0 + 0) × m = n × m.
Proof.
  intros n m.
  assert (H: n + 0 + 0 = n).
    { rewrite add_comm. simpl. rewrite add_comm. reflexivity. }
  rewrite → H.
  reflexivity. Qed.
```

$$\frac{}{\mathsf{P}} \quad \rightsquigarrow \quad \frac{}{\mathsf{Q}} \text{ and } \frac{\mathsf{H} : \mathsf{Q}}{\mathsf{P}}$$

enough: Prove that a hypothesis implies the goal, then proves the assumption.

This behaves just like `assert` but puts the goal for the stated fact after the current goal rather than before. The tactic `enough (H: Q)` allows you to prove `P` under the assumption `H` first and then `H` remains to be shown.

$$\boxed{\frac{}{\mathsf{P}} \quad \rightsquigarrow \quad \frac{\mathsf{H} : \mathsf{Q}}{\mathsf{P}} \quad \text{and} \quad \frac{}{\mathsf{Q}}}$$

cut: Prove an intermediate hypothesis first, then show it implies the goal..

Sometimes to prove a goal you need an extra hypothesis. You can add the hypothesis using `cut`. This allows you to first prove your goal using the new hypothesis, and then prove that the new hypothesis is also true. `cut P` transforms the current goal `P` into the two following subgoals: `Q -> P` and `Q`. The subgoal `Q -> P` comes first in the list of remaining subgoals to prove.

$$\boxed{\frac{}{\mathsf{P}} \quad \rightsquigarrow \quad \frac{}{\mathsf{Q} \to \mathsf{P}} \quad \text{and} \quad \frac{}{\mathsf{Q}}}$$

In this example we will prove that if `x = y` and `y = z` then `f x = f z`, for any function `f`. We first add the intermediate proposition that `x = z`. Then we have to prove that `x = z` implies `f x = f z`, and that `x` is actually equal to `z`.

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

Lemma xyz:
  forall (f: bool->bool) x y z,
    x  = y -> y = z -> f x = f z.
Proof.
  intros.
  cut (x = z).
```

```
bool is defined
bool_rect is defined
bool_ind is defined
bool_rec is defined
bool_sind is defined

1 goal

  ============================
  forall (f : bool -> bool) (x y z : bool), x = y -> y = z -> f x = f z

1 goal

  f : bool -> bool
  x, y, z : bool
  H : x = y
  H0 : y = z
  ============================
  f x = f z

2 goals

  f : bool -> bool
  x, y, z : bool
  H : x = y
  H0 : y = z
  ============================
  x = z -> f x = f z

goal 2 is:
 x = z
```

Proof term

```
let H1 :  = ?Goal0 in ?Goal H1
```

Equality and rewriting

rewrite: Replaces a term with an equivalent term if the equivalence of the terms has already been proven.

The `rewrite` tactic replaces occurrences of term `a` with term `b` if `H: a = b` is an assumption. The `rewrite` tactic takes an equivalence proof as input, like `a = b`, and replaces all occurances of `a` with `b`. Replacement of `b` with `a` can be achieved with the variant `rewrite <-` (rewrite backwards). Multiple rewrites can be chained with one tactic via a list of comma-separated equivalence proofs. Each of the equivalence proofs in the chain may be rewritten forwards or backwards.

$$\boxed{\begin{array}{c}\dfrac{\mathsf{H} : \mathsf{a} = \mathsf{b}}{\mathsf{P}} \\ \text{(where } \mathsf{a} \text{ occurs in } \mathsf{P}\text{)}\end{array} \quad \rightsquigarrow \quad \begin{array}{c}\dfrac{\mathsf{H} : \mathsf{a} = \mathsf{b}}{\mathsf{P}'} \\ \text{(where } \mathsf{a} \text{ is replaced with } \mathsf{b} \text{ in } \mathsf{P}.\text{)}\end{array}}$$

Usage

- `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`.

Rewriting also works with quantified equalities. If you have `H : forall n m, n + m = m + n` then you can still do `rewrite H` and 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)`.

```
Axiom H : forall n m, n + m = m + n.
Theorem add_assoc : forall n m p : nat,
  n + (p + m) = (n + m) + p.
Proof. intros. rewrite (H p m).
```

```
H is declared

1 goal

  ============================
  forall n m p : nat, n + (p + m) = n + m + p

1 goal

  n, m, p : nat
  ============================
  n + (p + m) = n + m + p

1 goal

  n, m, p : nat
  ============================
  n + (m + p) = n + m + p
```

Proof term

`eq_indr _ ?Goal H`

`reflexivity`: Prove `x = x` by reflexivity.

This tactic dischares goals that assert an equality between two terms known to be definitionally identical. It operates by simplifying both sides of the equation using Coq's $(\beta, \delta, \iota, \zeta)$-conversion rules and, if they coincide, applies the constructor `eq_refl` of the `eq` inductive type to close the goal.

`subst`: Transforms an identifier into an equivalent term.

The tactic `subst x` replaces `x` with an equivalent value defined by an equation involving `x` in the assumptions or a definition of `x`. After that the assumption is removed. When called without any arguments it substitutes everything it can.

$$\boxed{\frac{\mathsf{x} := \mathsf{t}}{\mathsf{p}\,\mathsf{x}} \quad \rightsquigarrow \quad \frac{}{\mathsf{p}\,\mathsf{t}}}$$

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

Lemma equality_commutes:
  forall (a: bool) (b: bool), a = b -> b = a.
Proof.
  intros.
  subst a.
```

```
bool is defined
bool_rect is defined
bool_ind is defined
bool_rec is defined
bool_sind is defined

1 goal

  ============================
  forall a b : bool, a = b -> b = a

1 goal

  a, b : bool
  H : a = b
  ============================
  b = a

1 goal

  b : bool
  ============================
  b = b
```

Proof term

(unchanged)

`symmetry`: Flips `x = y` to `y = x`.

This tactic applies symmetry to equalities.

$$\boxed{\frac{}{\mathsf{x} = \mathsf{y}} \quad \rightsquigarrow \quad \frac{}{\mathsf{y} = \mathsf{x}}}$$

Usage

- `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`)

Proof term

The goal `?Goal` gets replaced with `eq_sym ?Goal`

```
Theorem silly3 : forall (n m : nat),
  n = m ->
  m = n.
Proof.
  intros n m H. symmetry. Show Proof.
```

```
1 goal

  ============================
  forall n m : nat, n = m -> m = n

1 goal

  n, m : nat
  H : n = m
  ============================
  m = n

1 goal

  n, m : nat
  H : n = m
  ============================
  n = m

(fun (n m : nat) (H : n = m) => eq_sym ?Goal)
```

`f_equal`: Proves equality of structured terms (e.g., `f a = g b`).

For a goal with the form `f a1 ... an = g b1 ... bn`, creates subgoals `f = g` and `ai = bi` for the `n` arguments. Subgoals that can be proven by `reflexivity` or `congruence` are solved automatically.

$$\frac{}{\mathsf{f\,a1\cdots an = g\,b1\cdots bn}} \quad \rightsquigarrow \quad \frac{}{\mathsf{f = g}}\ \frac{}{\mathsf{a1 = b1}}\cdots\frac{}{\mathsf{an = bn}}$$

An example

```
Goal forall (n m : nat),  n = m -> S n = S m.
intros n m H. f_equal. assumption.
```

```
1 goal

  ============================
  forall n m : nat, n = m -> S n = S m

1 goal

  n, m : nat
  H : n = m
  ============================
  S n = S m

1 goal

  n, m : nat
  H : n = m
  ============================
  n = m

No more goals.
```

Functions are functional, thus if we want to show `f x = f y` it's always sufficient to show `x = y`. This is also true for constructors by injectivity.

```
Require Import List.

Goal forall (a a' : nat) s, a :: s = a' :: s.
intros.
f_equal.
```

```
1 goal

  ============================
  forall (a a' : nat) (s : list nat), a :: s = a' :: s

1 goal

  a, a' : nat
  s : list nat
  ============================
  a :: s = a' :: s

1 goal

  a, a' : nat
  s : list nat
  ============================
  a = a'
```

Caveat: Using `f_equal` on non-injective functions can produce absurd subgoals.

```
Require Import Arith.
Goal 1 mod 2 = 3 mod 4.
f_equal.
```

```
1 goal

  ============================
  1 mod 2 = 3 mod 4

2 goals

  ============================
  1 = 3

goal 2 is:
 2 = 4
```

`discriminate`: Proves goals by contradiction of distinct constructor terms.

The *principle of disjointness* says that two terms beginning with different constructors (like `0` and `S`, or `true` and `false`) can never be equal. This means that, any time we find ourselves in a context where we've assumed that two such terms are equal, we are justified in concluding anything we want, since the assumption is nonsensical.
The `discriminate` tactic embodies this principle: it proves any goal from an absurd hypothesis stating that two structurally different terms of an inductive set are equal. In other words, if you have a hypothesis `H : C x = D y`, `discriminate H` solves the goal immediately.

Examples

```
Goal forall n: nat, S n = 0 -> 2 + 2 = 5. intros. discriminate.
```

```
1 goal

  ============================
  forall n : nat, S n = 0 -> 2 + 2 = 5

1 goal

  n : nat
  H : S n = 0
  ============================
  2 + 2 = 5

No more goals.
```

```
Require Import List.
Goal forall (X : Type) (x y z : X) (l j : list X),
    x :: y :: l = nil ->
    x = z.
intros X x y z l j H. discriminate H.
```

```
1 goal

  ============================
  forall (X : Type) (x y z : X) (l : list X),
  list X -> x :: y :: l = nil -> x = z

1 goal

  X : Type
  x, y, z : X
  l, j : list X
  H : x :: y :: l = nil
  ============================
  x = z

No more goals.
```

`injection`: Decompose `H : C x = C y` into `x = y`.

The injection tactic is based on the fact that constructors of inductive sets are injections i.e., that whenever two objects were built using the same introduction rule, then this rule should have been applied to the same element.
This tactic is applied to a term t of type $C\, x_1 \ldots x_n = C\, y_1 \ldots y_n$], where C is some constructor of an inductive type. The tactic `injection` is applied as deep as possible to derive the equality of all pairs of subterms of t_i and t'_i placed in the same position. All these equalities are put as antecedents of the current goal.

$$\boxed{\frac{\mathsf{H : C\,x1\,xn = C\,y1\,yn}}{\mathsf{P}} \quad \rightsquigarrow \quad \frac{}{\mathsf{x1 = y1 \to xn = yn \to P}}}$$

Given `H : C x1 x2 ... = C y1 y2 ..`, the tactic `injection H as H1 H2 ...` generates one new hypothesis per constructor argument, e.g., `H1 : x1=y1, H2 : x2=y2 ...`. The hypothesis itself is removed from context: sometimes we write `injection H as H` if there is only one argument, and we want to peel off the constructor on the hypothesis.

```
Theorem S_injective' : forall(n m : nat),
  S n = S m ->
  n = m.
Proof.
  intros n m H.
```

```
  injection H as H. apply H.
Qed.
```

```
1 goal

  ============================
  forall n m : nat, S n = S m -> n = m

1 goal

  n, m : nat
  H : S n = S m
  ============================
  n = m

1 goal

  n, m : nat
  H : n = m
  ============================
  n = m

No more goals.
```

Please make sure the tactic is running on a constructor!!!!

`replace`: Replaces a term with a equivalent term and generates a subgoal to prove that the equality holds.

The tactic `replace A with B` replaces all occurrences of `A` with `B` in the goal. A new subgoal of the form `A = B` is generated and solved if it occurs in the assumptions.

```
Theorem one_x_one : forall (x : nat),
  1 + x + 1 = 2 + x.
Proof.
  intro; simpl.
  replace (x + 1) with (S x).
```

```
1 goal

  ============================
  forall x : nat, 1 + x + 1 = 2 + x

1 goal

  x : nat
  ============================
  S (x + 1) = S (S x)

2 goals

  x : nat
  ============================
  S (S x) = S (S x)

goal 2 is:
 S x = x + 1
```

$$\boxed{\frac{}{\mathsf{P}} \quad \rightsquigarrow \quad \frac{\overline{\mathsf{P}[\mathsf{A} \leftarrow \mathsf{B}]}}{\text{(where } \mathsf{A} \text{ is replaced with } \mathsf{B} \text{ in } \mathsf{P})} \quad \text{and} \quad \frac{}{\mathsf{A} = \mathsf{B}}}$$

Common variants: in, by

Case analysis and induction

`destruct`: Performs case analysis without recursion

The `destruct` tactic in Coq is one of the fundamental ways to perform case-analysis on an inductive data type. At a high level, when you `destruct t`, you:

1. Inspect the term `t` in your goal or context.
2. Generate one subgoal per constructor of =t='s inductive type.
3. Replace occurrences of `t` in the goal (and context) with the constructor for that branch, introducing any arguments of that constructor as new hypotheses.

Usage

```
destruct t as [ c1_args | c2_args |...].
```

- `t` must be of some inductive type (e.g. `bool`, `nat`, `list T`, or a user-defined type).
- Coq creates one subgoal per constructor of `t`.
- In each subgoal, every occurrence of `t` is replaced by the constructor, and its arguments become fresh hypotheses with the names you give in the brackets.

Examples

```
Goal forall b : bool, b = true \/ b = false.
Proof.
  intros b.
  destruct b as [ | ].  (* Two constructors: true and false *)
  - left. reflexivity.
  - right. reflexivity.
```

Generating equations (eqn:E)

Sometimes you need to keep track of which case you're in by recalling the original identifier. The `eqn:` annotation generates a new hypothesis in each new subgoal that is an equality between the term being case-analyzed and the associated constructor.

```
Theorem idempotence:  forall (f:bool -> bool) (b:bool), f (f (f b)) = f b.
Proof.
  intros f b. destruct (f true) eqn:Hft;
    destruct (f false) eqn:Hff;
    destruct b; congruence. (* also do 2 (try rewrite Hft; try rewrite Hff); auto. *)
Qed.
```

```
1 goal

  ============================
  forall (f : bool -> bool) (b : bool), f (f (f b)) = f b

1 goal

  f : bool -> bool
  b : bool
  ============================
  f (f (f b)) = f b

No more goals.
```

edestruct variant

Similar to `destruct`, but can also deal with existential variables: if it does not know how to instantiate variables, it does not fail, but instead introduces existential variables which need to be instantiated later.

`induction`: Performs case analysis with recursion

The `induction` tactic applies the automatically generated induction lemma for an inductive type to the current goal and introduces assumptions. Like `destruct`, it splits on an inductive value, but additionally provides you with an induction hypothesis in each non-base case.

Usage

```
induction t as [ C0_args | C1_args IH1 | ... | Cn_args IHn ].
```

- `t` must be of some inductive type (e.g. `nat`, `list T`, or a user-defined family).
- Coq generates one subgoal for each constructor of `t`.
- In non-base cases (where the constructor is recursive), it introduces one induction hypothesis per recursive occurrence of `t` in that constructor's argument types.

You name:

- Constructor arguments (`Ck_args`),
- Induction hypotheses (e.g. `IH1, IH2, ...`).

Examples

```
Goal forall n, n + 0 = n.
Proof.
  intros n.
  induction n as [      (* base: n = 0, no IH *)
  | n' IHn' ].          (* inductive: n = S n', with IHn': n' + 0 = n' *)
  - simpl. reflexivity.
  - simpl. rewrite IHn'. reflexivity.
Qed.
```

```
1 goal

  ============================
  forall n : nat, n + 0 = n

1 goal

  n : nat
  ============================
  n + 0 = n

2 goals

  ============================
  0 + 0 = 0

goal 2 is:
 S n' + 0 = S n'

1 goal

  ============================
  0 + 0 = 0

1 goal

  ============================
  0 = 0

This subproof is complete, but there are some unfocused goals.
Focus next goal with bullet -.
```

```
1 goal

goal 1 is:
 S n' + 0 = S n'

1 goal

  n' : nat
  IHn' : n' + 0 = n'
  ============================
  S n' + 0 = S n'

1 goal

  n' : nat
  IHn' : n' + 0 = n'
  ============================
  S (n' + 0) = S n'

1 goal

  n' : nat
  IHn' : n' + 0 = n'
  ============================
  S n' = S n'

No more goals.
```

- **Base case:** `n = 0`. No `IH` since 0 has no recursive arguments.
- **Inductive case:** `n = S n'`. We get `IHn' : n' + 0 = n'`.

You can also give explicit names to the generated equations like in `destruct`:

```
induction n as [| n' IHn'] eqn:En.
```

`eqn:En` adds `En : n = 0` in the first branch and `En : n = S n'` in the second.

`inversion`: Deduces equalities that must be true given an equality between two constructors.

Sometimes you have a hypothesis that can't be true unless other things are also true. We can use inversion to discover other necessary conditions for a hypothesis to be true. The inversion tactic is used like this. Suppose `H` is a hypothesis of the form `C a1 a2 ... an = D b1 b2 ... bm` for some constructors c and d and arguments `a1 ... an` and `b1 ... bm`. Then `inversion H` instructs Coq to "invert" this equality to extract the information it contains about these terms:

- If `C` and `D` are the same constructor, then we know, by the injectivity of this constructor, that `a1 = b1, a2 = b2`, etc.; `inversion H` adds these facts to the context, and tries to use them to rewrite the goal.
- If `C` and `D` are different constructors, then the hypothesis `H` is contradictory. In this case, `inversion H` marks the current goal as completed and pops it off the goal stack.

```
Inductive ev : nat -> Prop :=
  | ev_0 : ev 0
  | ev_SS : forall n, ev n -> ev (S (S n)).

Theorem evSS_ev' : forall n, ev (S (S n)) -> ev n.
Proof.
  intros n E. inversion E as [| n' E' Hnn'].
  (* now in the ev_SS case: n = S (S n') and Hnn': n = n' *)
  apply E'.
Qed.
```

```
ev is defined
ev_ind is defined
ev_sind is defined

1 goal

  ============================
  forall n : nat, ev (S (S n)) -> ev n

1 goal

  n : nat
  E : ev (S (S n))
  ============================
  ev n

1 goal

  n : nat
  E : ev (S (S n))
  n' : nat
  E' : ev n
  Hnn' : n' = n
  ============================
  ev n

No more goals.
```

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

`constructor`: Applies a constructor of an inductive type (e.g., prove `A \/ B` via left).

This tactic applies to a goal such that the head of its conclusion is an inductive constant (say `T`). Let `Ci` be the i^{th} constructor of `T`, then `constructor i` is equivalent to `intros; apply Ci`. If no number is specified the constructors in the premises are tried in order and the first one whose result type matches the goal is selected and unfolded. The advantage over a simple `apply` is isn't necessary to explicitly name the constructor.

```
Inductive even : nat -> Prop:=
 | even_O: even O
 | even_S: forall n, even n -> even (S(S n)).

Lemma four_is_even:
  even (S (S (S (S O)))).
Proof.
  constructor.  (* picks even_S: generates subgoal even 2 *)
  constructor.  (* picks even_S again: generates subgoal even 0 *)
  constructor.  (* picks even_0: no subgoals left *)
Qed.
```

```
even is defined
even_ind is defined
even_sind is defined

1 goal

  ============================
  even 4

1 goal

  ============================
  even 2

1 goal

  ============================
  even 0

No more goals.
```

$$\frac{\begin{array}{c} T \text{ has some constructor} \\ C_i : \forall x_i\, A_1 \to A_2 \cdots \to A_k \to T\, x_1 \cdots x_n \end{array}}{T\, u_1 u_2 \cdots u_n} \quad \rightsquigarrow \quad \frac{}{A_1 \to A_2 \cdots \to A_k}$$

The tactics `split`, `exists`, `left` and `right` are all versions of this. When your goal is to show that you can build up a term that has some type and you have a constructor to do just that, use `constructor`!

`split`: Replaces a goal consisting of a conjunction `P /\ Q` with two subgoals `P` and `Q`.

Equivalent to `constructor 1`. Applies if `I` has only one constructor, typically in the case of conjunction `P /\ Q`.

`left` / `right`: Replaces a goal consisting of a disjunction `P \/ Q` with just `P` or `Q`.

Apply if `I` has two constructors, for instance in the case of disjunction `P \/ Q`. Then, they are respectively equivalent to `constructor 1` and `constructor 2`.

```
Goal forall (P Q : Prop), P -> P \/ Q.
intros P Q HP. left. Show Proof.
```

```
1 goal

  ============================
  forall P Q : Prop, P -> P \/ Q

1 goal

  P, Q : Prop
  HP : P
  ============================
  P \/ Q

1 goal

  P, Q : Prop
  HP : P
  ============================
  P

(fun (P Q : Prop) (HP : P) => or_introl ?Goal)
```

Proof term

`or_introl ?Goal` or `or_intror ?Goal`

`exists`: Puts in a witness into a proof with an existential quantifier.

Applies if `I` has only one constructor, for instance in the case of existential quantification $\exists x \mid P(x)$. Then, `exists t` is equivalent to `intros; constructor 1 with t`.

$$\frac{}{\exists x. p\, x} \quad \rightsquigarrow \quad \frac{}{p\, a}$$

The `eexists` variant 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.

```
Goal forall x:nat, exists y:nat, x*x=y.
```

```
1 goal

  ============================
  forall x : nat, exists y : nat, x * x = y
```

Simplification and computation

simpl: Simplifies expressions using definitions.

The `simpl` tactic reduces complex terms to simpler forms. It reduces matches and fixpoints when applied to a constructor. It's not always necessary because other tactics (e.g. `discriminate`) can do the simplification themselves. It is meant to be "human readable": `simpl` does not perform full δ-expansion unless it simplifies immediately after.

$$\boxed{\frac{}{\mathsf{p(Sx+y)}} \quad \rightsquigarrow \quad \frac{}{\mathsf{p(S(x+y))}}}$$

Usage

- `simpl`: rewrite with computation rules in the goal.
- `simpl in H`: rewrite with computation rules in the hypothesis H.
- `simpl in *`: rewrite with computation rules in all hypotheses and the goal. Nice to try when you don't know what to do.

```
Theorem plus_1_l_1__1olll: forall(n:nat),
    1 + n = S n.
Proof.
  intros n. simpl.
```

```
1 goal

  ============================
  forall n : nat, 1 + n = S n

1 goal

  n : nat
  ============================
  1 + n = S n

1 goal

  n : nat
  ============================
  S n = S n
```

Proof term

(unchanged).

unfold: Unfolds the definitions of terms.

Applies δ-reduction to the constants i.e. replaces constants with their definitions (right-hand sides). The selected hypotheses and/or goals are then reduced to $\beta\iota\zeta$-normal form

$$\boxed{\frac{}{\mathsf{P}} \text{ (where P contains the constant f)} \quad \rightsquigarrow \quad \frac{}{\mathsf{P'}} \text{ (where P}' \text{ is P with f substituted.)}}$$

Usage

- `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)

Proof term

(unchanged)

hnf

cbv

Automation

auto / eauto: Solve goals using a hint database (lemmas/constructors).

The `auto` tactic performs a bounded backward search using only the primitive tactics `reflexivity`, `assumption`, and `apply`, trying hints (lemmas or tactics) whose conclusion matches the goal head. The depth of proof search is limited to 5 by default, writing `auto n` uses `n` instead of 5.
The `eauto` tactic extends auto by also using the `eapply` primitive, which allows it to defer instantiation of existential variables, thereby solving goals involving existential quantifiers more flexibly.

```
Lemma exists_example : forall (P Q : nat -> Prop) x,
    P x -> (exists y, Q y) -> P x.
Proof. eauto. Qed.
```

```
1 goal

  ============================
  forall (P Q : nat -> Prop) (x : nat), P x -> (exists y : nat, Q y) -> P x

No more goals.
```

`trivial`: Solve goals using lemmas that exactly fit the goal.

This is a lightweight shortcut for the special case of `auto` where only cost-0 hints are used (i.e., depth 0). It succeeds only if the goal already matches exactly one of:

- A hypothesis in the context (`assumption`).
- An equality like `x = x` (`reflexivity`).
- A contradiction (`discriminate`, `congruence`).
- Any other lemma/constructor that has been declared with cost 0.

Thus, `trivial` is essentially **non-recursive** proof-search: it makes **one** pass over the hint database and solves the goal only if an exact match is found immediately.
Because it never recurses, `trivial` is extremely fast and predictable, but only useful when you know that a direct lemma exactly matches your goal (for instance, after an `apply` that leaves trivial subgoals).

`tauto`: Solves goals consisting of tautologies that hold in constructive logic.

Solves all goals that can be solved by purely propositional reasoning. It can solve all tautological intuitionistic propositions. `tauto` will not instantiate universal quantifiers.

`intuition`: Splits along the search tree of the decision procedures from `tauto` and apply `auto`.

The tactic `intuition` automates the propositional-logic decision procedure (`tauto`) to break down complex propositional goals into simpler subgoals. Roughly:

1. It **analyzes** the goal's Boolean structure (conjunctions, disjunctions, implications).
2. It **generates** an equivalent set of simpler subgoals by case-splitting on these connectives (using `destruct`, `split`, etc.).
3. It then **applies** a user-supplied tactic (by default, `auto`) to each resulting subgoal.

For example, on a goal like `(P  Q) → (R  S)`, `intuition auto` might split into subgoals `P → Q → R` and `P → Q → S`, then use `auto` to close them if possible. If the secondary tactic fails on **any** subgoal, `intuition` itself fails. Internally, it uses the same search-tree structure as `tauto`, but exposes the intermediate subgoals to further automation.

`firstorder`: More powerful automation for first-order logic.

`firstorder` is an experimental extension of `tauto` (and thus `intuition`) to a **restricted first-order fragment** of Coq's logic. It supports:

- Universal and existential quantifiers $(\forall, \exists)$
- Propositional connectives $(\wedge, \vee, \rightarrow, \neg)$
- Equality reasoning only insofar as the above connectives allow.

Under the hood, `firstorder` performs a **tableau-style proof search**: it systematically instantiates quantifiers (within user-provided bounds) and applies propositional steps; if a branch closes (e.g., leads to a contradiction), it is pruned. Because it does **not** attempt arbitrary higher-order matching or induction, `firstorder` is generally fast and predictable but limited to goals that lie squarely within first-order logic (no dependent-type reasoning or complex higher-order unification).

`congruence`: Solves equational reasoning problems.

Solves all goals that can be solved using purely equational reasoning, i.e reflexivity, transitivity, symmetry and rewriting. It uses the Nelson and Oppen closure algorithm. It subsumes the power of `injectivity` and `discriminate`.

`lia` / `nia`: Solves linear/nonlinear arithmetic computational problems.

The tactic `lia` uses linear positivstellensatz refutations, cutting plane proofs (rounding rational constants) and case analysis for possible values. Has the power of `omega` (Presburger Arithmetic) and normalization of ring and semiring structures. Lia module has to be loaded before (`Require Import Lia.`)
`nia` is a variant of `lia` that can not only deal with linear arithmetic, but also with non-linear arithmetic (i.e. multiplication). Essentially heuristically transforms the goal to eliminate non-linearities and then calls `lia`. This is not a complete decision procedure and may fail on many goals or take prohibitely long. Lia again has to be loaded before (`Require Import Lia.`)

`autorewrite`:

Advanced tactics

`setoid_rewrite`: Rewrites under equivalence relations.

`functional induction`: Performs induction on function definitions.

`stdpp` tactics

Activated using:

```
From stdpp Require Export tactics.
```

`done`

Solves trivial goals by reflexivity, discrimination, splitting, and with `trivial`. Faster than Coq's built-in `easy`.

`simplify_eq`: Does `subst`, `injection`, and `discriminate` automatically.

Repeatedly substitutes, discriminates, and injects equalities, and tries to contradict impossible inequalities. The variant `simplify_eq/=` additionally performs simplification.

`by tac`

Calls `tac` and executes done afterwards. Faster than Coq's built-in `now`.

`split_and`

Destructs a conjunction in the goal (and only conjunctions, in contrast to Coq's built-in `split`, which also splits other inductives). The variant `split_and` splits multiple conjunctions, but at least one. The variant `split_and?` splits zero or more conjunctions.

naive_solver

A `firstorder`-like tactic. `firstorder` can "loop" on quite small goals already, `naive_solver` fixes that by implementing a breadth-first search with limited depth. It implements some ad-hoc rules for logical connectives that in practice work quite well, and usually works better than `firstorder` for our purposes.

Tacticals

Tactical	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
`by tac`	Shorthand for `tac; done`

`;` (semicolon): Applies the tactic on the right to all subgoals produced by the tactic on the left.

The infix `;` tactical is the sequencing tactical. It applies the right tactic to all of the goals generated by the left tactic.
It is binary, so it takes two tactics `A` and `B` as input. `A` is executed. If `A` does not fail and does not solve the goal, then `B` is executed for every goal that results from applying `A`. If `A` solves the goal, then `B` is never called and the entire tactic succeeds. This is useful when `A` generates lots of very simple subgoals (like preconditions of a theorem application) that can all be handled with another automation tactic.
The `;` tactical is left-associative. Consider the tactic `A; B; C`. If `A` generates goals `A1` and `A2`, then `B` will be applied to each. Let's say that this results in a state with goals `A1', A2'`, and `B'`. `C` will now be applied to each of these. This may not always be desired, and so parentheses can be used to force right-associativity. Consider the tactic `A; (B; C)`. If `A` again generates goals `A1` and `A2`, then `B; C` will be applied to each. The difference may not be crystal-clear in an abstract example such as this one, so check out the script below. Keep in mind that the difference is in the resulting state tree from calling these tactics:

```
A; B; C
|-- A1              /* Call B */
|   +-- A1'         /* Call C */
|       +-- A1''
+-- A2              /* Call B */
    +-- A2'         /* Call C */
        +-- A2''

A;(B;C)             /* Call A */
|-- A1              /* Call B;C */
|   +-- A1''
+-- A2              /* Call B;C */
    +-- A2''
```

This tactical also has a more general form than the simple `tac1;tac2` we've seen above. If `T, T1, ..., Tn` are tactics, then `T; [T1 | T2 | ... | Tn]` is a tactic that first performs `T` and then performs `T1` on the first subgoal generated by `T`, performs `T2` on the second subgoal, etc. So `T;T'` is just special notation for the case when all of the `Ti`'s are the same tactic; i.e., `T;T'` is shorthand for: `T; [T' | T' | ... | T']`

`try`: Attempts to apply the given tactic but does not fail even if the given tactic fails.

`||` (or): Tries to apply the tactic on the left; if that fails, tries to apply the tactic on the right.

`all :` Applies the given tactic to all remaining subgoals.

`repeat`: Applies the given tactic repeatedly until it fails.

`!`: Applies the tactic if only one goal is in focus. If not, this tactic fails

`par`: Applies the tactic to all goals in focus in parallel.

The tactic provided must solve all goals or do nothing, otherwise this tactic fails.

`n-m:`: Applies the tactic to goals with indices between n and m, inclusive

The tactics `1:, 2:`, etc. solves a specifically numbered subgoal with a tactic or bracketed logic. Useful when the goal splits into two or more cases where a particular case is very easy and I don't want to spend a layer of bullets on this split. For example, if my goal splits into two cases and the second case can be proven with reflexivity but the first case is very complex, I might do `split. 2: reflexivity.` and then continue with the proof of case 1. Comma-separated numbers and ranges denoted with hyphens both work. Cases are 1-indexed.

`progress`: Applies the tactic and fail if it does nothing.

Guiding automation and custom tactics

Hints

Hint databases are the collections of lemmas and rewrite rules that automation tactics consult when solving goals or performing rewrites. To declare a hint database, use `Create HintDb my_db`. To populate hint databases:

- `Hint Resolve q1 q2 ... : dbs.` adds lemmas `q1, q2, ...` to databases `dbs` with default cost and an inferred pattern from each lemma's conclusion.

- `Hint Constructors ind1 ind2 ... : dbs.` adds all constructors of the given inductive types to the databases, enabling `auto` to apply them when the goal's head symbol matches

- Hint Rewrite lemmas : dbs. adds rewrite rules from the given lemmas to the database, which autorewrite with dbs will apply exhaustively or up to a given limit. If you want to autorewrite using a theorem with premises that might not always be true, and you only want to autorewrite when the premises can be automatically proven, you can say that the premises must be solved with auto to use the hint Hint Rewrite lemma using (solve [auto]) : dbs.
- Hint Extern n pattern => tactic : dbs. adds a tactic-based hint of priority n, firing when the current goal matches pattern; useful for hooking custom tactics into auto or eauto Once hints are in place, direct Coq's automation to use them like auto with my_db

Custom tactics

The simplest way to define new tactics:

```
Ltac my_tactic := tactic1; tactic2; tactic3.
```

More interesting tactics will pattern match on the goal or on hypotheses. Use the syntax match goal with ... end. The pattern has (entails) |- to separate hypotheses from the goal. You can name hypotheses. To name variables or expressions that might appear in the hypotheses or goal but might vary and that you want to use in the tactic, use a question mark in front of a name in the pattern. Don't use that question mark in the body of the tactic. Example:

```
Ltac subst_Some_eq :=
  match goal with
  | H : Some ?a = Some ?b |- _ =>
        inversion H; subst b
  end.
```

If you want to pattern-match on any expression anywhere, use context[ ].

```
Ltac destruct_if :=
  match goal with
  | |- context[if ?a then _ else _] => destruct a
  end.
```

If you only want to do something if some variable ?x is a variable, you can use is_var x before the tactic. match backtracks and tries matching its pattern to more hypotheses; it also tries the next branch if the branch pattern doesn't match or if the tactics fail.

Searching for lemmas and definitions

Use the queries. Search "foo". Check foo. About foo. Compute foo. Print foo. To look up notation (e.g. what does <> mean?) use Locate (e.g. Locate "<>"). To look up a simple abstract theorem about natural numbers or equality without knowing what it's called, SearchPattern is useful. For example, if looking to prove x <> y given y <> x., SearchPattern (_ <> _ -> _ <> _). finds the theorem not_eq_sym. Similarly SearchRewrite on some expression pattern searches for theorems you could use to rewrite that expression.

Variants of tactics

This is a small list of common variants of tactics (e.g. apply has a variant apply _ in _) together with the behavior one can in general expect. There are some exceptions in behavior like induction which we explained earlier. The in variant is used in hypotheses for a form of forward reasoning.

Variant	Usual description	Examples
as <intropattern>	Use an intropattern to specify the names given to new assumptions introduced or to directly destruct it.	destruct H as [H1 H2] apply H in H' as [H1 H2]
by <tactical>	Directly dispatch a new goal that is generated by a given tactical, which should completely solve the goal.	assert (x = y) by (intros H; now apply H2) rewrite H by eauto
in <assumption>	If a tactic should not be applied to the goal, specify to which assumption it should be applied.	rewrite H in H1 apply H1 in H2
at <occurrencelist>	For rewriting-based tactics: give the occurrence (s) at which the rewrite shall be performed.	rewrite H at 1 3 change y at 2 with ((fun x => x) y)

Proof General

To see the shortcuts,

Key	Action
C-c C-n	
C-c C-u	
C-c C-RET	

Some company-coq tips:

Bibliography

Apart from (Pierce *and others* 2025), the Coq manual and the book (Bertot and Castéran 2004) I used many other online cheatsheets and sources, like this one and Jules Jacobs' notes and this and this and this and this and Coq Tactics in Plain English and Smolka's overview and Castegren's KTH course and Robert Krebbers' notes and Al-hassy's notes and the Coq Survival Kit and the Coq discourse group and stack exchange sites.

Bertot, Y. and Castéran, P. (2004). Interactive theorem proving and program development: Coq'art: The calculus of inductive constructions. Springer.

Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V. and Yorgey, B. (2025). Logical foundations. Electronic textbook.