Coq Version 8.3pl2 for the Clueless
(177 Hints)

Pierre Castéran Hugo Herbelin Florent Kirchner Benjamin Monate
Julien Narboux

October 28, 2011

Abstract

This note intends to provide an easy way to get acquainted with the Coq theorem prover. It tries to formulate appropriate answers to some of the questions any newcomers will face, and to give pointers to other references when possible.

Contents

1 Introduction

This FAQ is the sum of the questions that came to mind as we developed proofs in Coq. Since we are singularly short-minded, we wrote the answers we found on bits of papers to have them at hand whenever the situation occurs again. This is pretty much the result of that: a collection of tips one can refer to when proofs become intricate. Yes, this means we won't take the blame for the shortcomings of this FAQ. But if you want to contribute and send in your own question and answers, feel free to write to us...

2 Presentation

1 What is Coq?

The Coq tool is a formal proof management system: a proof done with Coq is mechanically checked by the machine. In particular, Coq allows:

- the definition of mathematical objects and programming objects,
- to state mathematical theorems and software specifications,
- to interactively develop formal proofs of these theorems,
- to check these proofs by a small certification "kernel".

Coq is based on a logical framework called "Calculus of Inductive Constructions" extended by a modular development system for theories.

2 Did you really need to name it like that?

Some French computer scientists have a tradition of naming their software as animal species: Caml, Elan, Foc or Phox are examples of this tacit convention. In French, "coq" means rooster, and it sounds like the initials of the Calculus of Constructions CoC on which it is based.

3 Is Coq a theorem prover?

Coq comes with decision and semi-decision procedures (propositional calculus, Presburger's arithmetic, ring and field simplification, resolution, ...) but the main style for proving theorems is interactively by using LCF-style tactics.

4 What are the other theorem provers?

Many other theorem provers are available for use nowadays. Isabelle, HOL, HOL Light, Lego, Nuprl, PVS are examples of provers that are fairly similar to Coq by the way they interact with the user. Other relatives of Coq are ACL2, Agda/Alfa, Twelf, Kiv, Mizar, NqThm, Ωmega...

5 What do I have to trust when I see a proof checked by Coq?

You have to trust:

The theory behind Coq The theory of Coq version 8.0 is generally admitted to be consistent wrt Zermelo-Fraenkel set theory + inaccessible cardinals. Proofs of consistency of subsystems of the theory of Coq can be found in the literature.

The Coq kernel implementation You have to trust that the implementation of the Coq kernel mirrors the theory behind Coq. The kernel is intentionally small to limit the risk of conceptual or accidental implementation bugs.

The Objective Caml compiler The Coq kernel is written using the Objective Caml language but it uses only the most standard features (no object, no label ...), so that it is highly unprobable that an Objective Caml bug breaks the consistency of Coq without breaking all other kinds of features of Coq or of other software compiled with Objective Caml.

Your hardware In theory, if your hardware does not work properly, it can accidentally be the case that False becomes provable. But it is more likely the case that the whole Coq system will be unusable. You can check your proof using different computers if you feel the need to.

Your axioms Your axioms must be consistent with the theory behind Coq.

6 Where can I find information about the theory behind Coq?

The Calculus of Inductive Constructions The corresponding chapter and the chapter on modules in the Coq Reference Manual.

Type theory A book [11] or some lecture notes [8].

Inductive types Christine Paulin-Mohring's habilitation thesis [19].

Co-Inductive types Eduardo Giménez' thesis [9].

Miscellaneous A bibliography about Coq

7 How can I use Coq to prove programs?

You can either extract a program from a proof by using the extraction mechanism or use dedicated tools, such as Why, Krakatoa, Caduceus, to prove annotated programs written in other languages.

8 How old is Coq?

The first implementation is from 1985 (it was named CoC which is the acronym of the name of the logic it implemented: the Calculus of Constructions). The first official release of Coq (version 4.10) was distributed in 1989.

9 What are the Coq-related tools?

There are graphical user interfaces:

Coqide A GTK based GUI for Coq.

Pcoq A GUI for Coq with proof by pointing and pretty printing.

coqwc A tool similar to `wc` to count lines in Coq files.

Proof General A emacs mode for Coq and many other proof assistants.

ProofWeb The ProofWeb online web interface for Coq (and other proof assistants), with a focus on teaching.

ProverEditor is an experimental Eclipse plugin with support for Coq.

There are documentation and browsing tools:

Helm/Mowgli A rendering, searching and publishing tool.

coq-tex A tool to insert Coq examples within .tex files.

coqdoc A documentation tool for Coq.

coqgraph A tool to generate a dependency graph from Coq sources.

There are front-ends for specific languages:

Why A back-end generator of verification conditions.

Krakatoa A Java code certification tool that uses both Coq and Why to verify the soundness of implementations with regards to the specifications.

Caduceus A C code certification tool that uses both Coq and Why.

Zenon A first-order theorem prover.

Focal The Focal project aims at building an environment to develop certified computer algebra libraries.

Concoqtion is a dependently-typed extension of Objective Caml (and of MetaOCaml) with specifications expressed and proved in Coq.

Ynot is an extension of Coq providing a "Hoare Type Theory" for specifying higher-order, imperative and concurrent programs.

Ott is a tool to translate the descriptions of the syntax and semantics of programming languages to the syntax of Coq, or of other provers.

10 What are the high-level tactics of Coq

- Decision of quantifier-free Presburger's Arithmetic
- Simplification of expressions on rings and fields
- Decision of closed systems of equations
- Semi-decision of first-order logic
- Prolog-style proof search, possibly involving equalities

11 What are the main libraries available for Coq

- Basic Peano's arithmetic, binary integer numbers, rational numbers,
- Real analysis,
- Libraries for lists, boolean, maps, floating-point numbers,
- Libraries for relations, sets and constructive algebra,
- Geometry

12 What are the mathematical applications for Coq?

Coq is used for formalizing mathematical theories, for teaching, and for proving properties of algorithms or programs libraries.

The largest mathematical formalization has been done at the University of Nijmegen (see the Constructive Coq Repository at Nijmegen).

A symbolic step has also been obtained by formalizing in full a proof of the Four Color Theorem.

13 What are the industrial applications for Coq?

Coq is used e.g. to prove properties of the JavaCard system (especially by Schlumberger and Trusted Logic). It has also been used to formalize the semantics of the Lucid-Synchrone data-flow synchronous calculus used by Esterel-Technologies.

3 Documentation

14 Where can I find documentation about Coq?

All the documentation about Coq, from the reference manual [17] to friendly tutorials [15] and documentation of the standard library, is available online. All these documents are viewable either in browsable HTML, or as downloadable postscripts.

15 Where can I find this FAQ on the web?

This FAQ is available online at `http://coq.inria.fr/faq`.

16 How can I submit suggestions / improvements / additions for this FAQ?

This FAQ is unfinished (in the sense that there are some obvious sections that are missing). Please send contributions to Coq-Club.

17 Is there any mailing list about Coq?

The main Coq mailing list is `coq-club@inria.fr`, which broadcasts questions and suggestions about the implementation, the logical formalism or proof developments. See `https://sympa-roc.inria.fr/wws/info/coq-club` for subscription. For bugs reports see question 22.

18 Where can I find an archive of the list?

The archives of the Coq mailing list are available at `https://sympa-roc.inria.fr/wws/arc/coq-club`.

19 How can I be kept informed of new releases of Coq?

New versions of Coq are announced on the coq-club mailing list.

20 Is there any book about Coq?

The first book on Coq, Yves Bertot and Pierre Castéran's Coq'Art has been published by Springer-Verlag in 2004:

> "This book provides a pragmatic introduction to the development of proofs and certified programs using Coq. With its large collection of examples and exercises it is an invaluable tool for researchers, students, and engineers interested in formal methods and the development of zero-default software."

21 Where can I find some Coq examples?

There are examples in the manual [17] and in the Coq'Art [2] exercises `http://www.labri.fr/Perso/~casteran/CoqArt/index.html`. You can also find large developments using Coq in the Coq user contributions: `http://coq.inria.fr/contribs`.

22 How can I report a bug?

You can use the web interface accessible at `http://coq.inria.fr/bugs`.

4 Installation

23 What is the license of Coq?

Coq is distributed under the GNU Lesser General License (LGPL).

24 Where can I find the sources of Coq?

The sources of Coq can be found online in the tar.gz'ed packages (`http://coq.inria.fr`, link "download"). Development sources can be accessed at `https://gforge.inria.fr/scm/?group_id=269`

25 On which platform is Coq available?

Compiled binaries are available for Linux, MacOS X, and Windows. The sources can be easily compiled on all platforms supporting Objective Caml.

5 The logic of Coq

5.1 General

26 What is the logic of Coq?

Coq is based on an axiom-free type theory called the Calculus of Inductive Constructions (see Coquand [6], Luo [16] and Coquand–Paulin-Mohring [7]). It includes higher-order functions and predicates, inductive and co-inductive datatypes and predicates, and a stratified hierarchy of sets.

27 Is Coq's logic intuitionistic or classical?

Coq's logic is modular. The core logic is intuitionistic (i.e. excluded-middle $A \vee \neg A$ is not granted by default). It can be extended to classical logic on demand by requiring an optional module stating $A \vee \neg A$.

28 Can I define non-terminating programs in Coq?

All programs in Coq are terminating. Especially, loops must come with an evidence of their termination.

Non-terminating programs can be simulated by passing around a bound on how long the program is allowed to run before dying.

29 How is equational reasoning working in Coq?

Coq comes with an internal notion of computation called *conversion* (e.g. $(x + 1) + y$ is internally equivalent to $(x + y) + 1$; similarly applying argument a to a function mapping x to some expression t converts to the expression t where x is replaced by a). This notion of conversion (which is decidable because Coq programs are terminating) covers a certain part of equational reasoning but is limited to sequential evaluation of expressions of (not necessarily closed) programs. Besides conversion, equations have to be treated by hand or using specialised tactics.

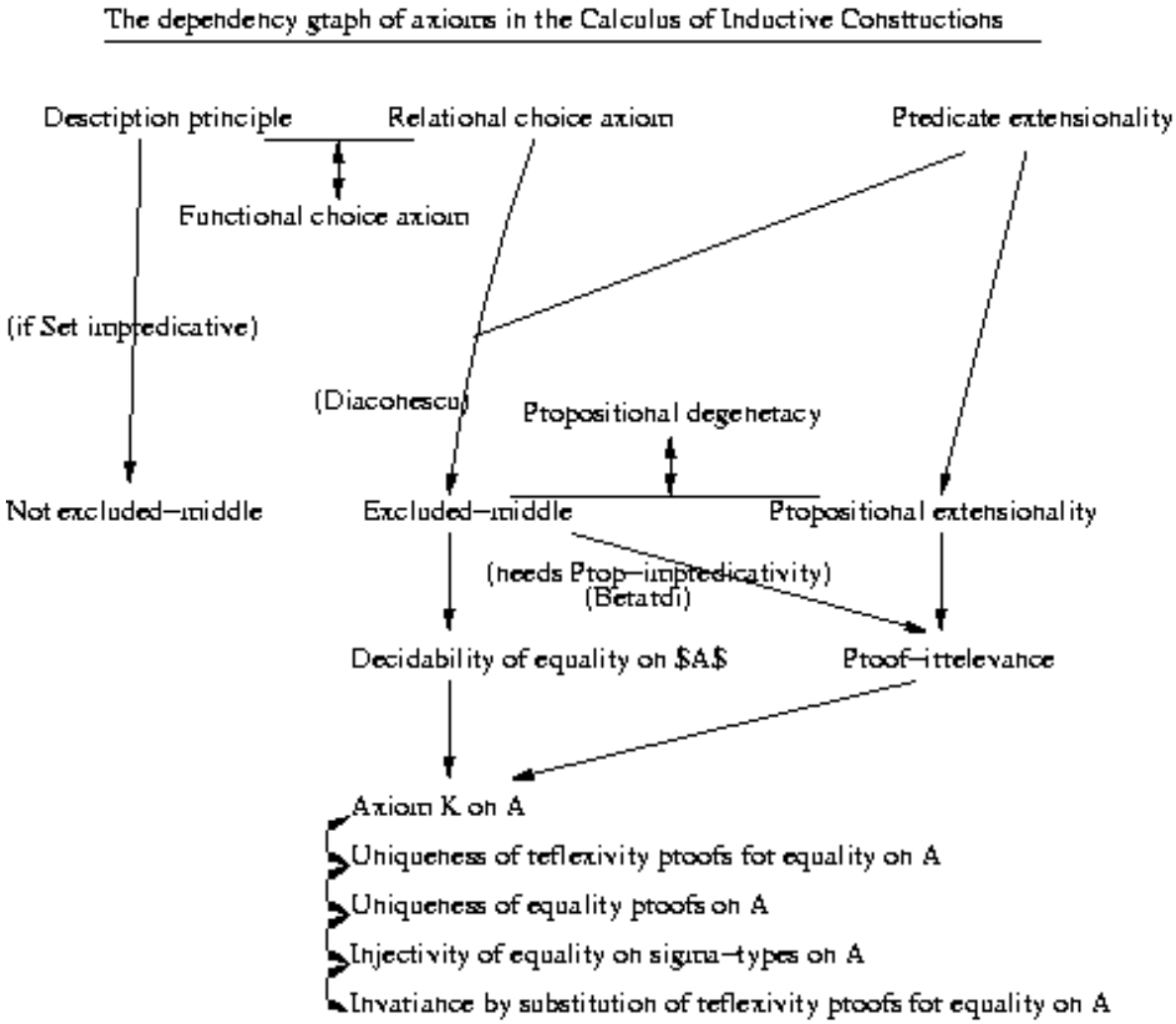
5.2 Axioms

30 What axioms can be safely added to Coq?

There are a few typical useful axioms that are independent from the Calculus of Inductive Constructions and that are considered consistent with the theory of Coq. Most of these axioms are stated in the directory `Logic` of the standard library of Coq. The most interesting ones are

- Excluded-middle: $\forall A : Prop, A \vee \neg A$
- Proof-irrelevance: $\forall A : Prop \forall p_1 p_2 : A, p_1 = p_2$
- Unicity of equality proofs (or equivalently Streicher's axiom K): $\forall A \forall xy : A \forall p_1 p_2 : x = y, p_1 = p_2$
- Hilbert's ϵ operator: if $A \neq \emptyset$, then there is ϵ_P such that $\exists x P(x) \rightarrow P(\epsilon_P)$
- Church's ι operator: if $A \neq \emptyset$, then there is ι_P such that $\exists! x P(x) \rightarrow P(\iota_P)$
- The axiom of unique choice: $\forall x \exists! y R(x, y) \rightarrow \exists f \forall x R(x, f(x))$
- The functional axiom of choice: $\forall x \exists y R(x, y) \rightarrow \exists f \forall x R(x, f(x))$
- Extensionality of predicates: $\forall PQ : A \rightarrow Prop, (\forall x, P(x) \leftrightarrow Q(x)) \rightarrow P = Q$
- Extensionality of functions: $\forall fg : A \rightarrow B, (\forall x, f(x) = g(x)) \rightarrow f = g$

Here is a summary of the relative strength of these axioms, most proofs can be found in directory `Logic` of the standard library. The justification of their validity relies on the interpretability in set theory.



31 What standard axioms are inconsistent with Coq?

The axiom of unique choice together with classical logic (e.g. excluded-middle) are inconsistent in the variant of the Calculus of Inductive Constructions where `Set` is impredicative.

As a consequence, the functional form of the axiom of choice and excluded-middle, or any form of the axiom of choice together with predicate extensionality are inconsistent in the `Set`-impredicative version of the Calculus of Inductive Constructions.

The main purpose of the `Set`-predicative restriction of the Calculus of Inductive Constructions is precisely to accommodate these axioms which are quite standard in mathematical usage.

The `Set`-predicative system is commonly considered consistent by interpreting it in a standard set-theoretic boolean model, even with classical logic, axiom of choice and predicate extensionality added.

32 What is Streicher's axiom K

Streicher's axiom K [13] is an axiom that asserts dependent elimination of reflexive equality proofs.

```
Coq < Axiom Streicher_K :
Coq <   forall (A:Type) (x:A) (P: x=x -> Prop),
Coq <     P (refl_equal x) -> forall p: x=x, P p.
```

In the general case, axiom K is an independent statement of the Calculus of Inductive Constructions. However, it is true on decidable domains (see file `Eqdep_dec.v`). It is also trivially a consequence of proof-irrelevance (see 33) hence of classical logic.

Axiom K is equivalent to *Uniqueness of Identity Proofs* [13]

```
Coq < Axiom UIP : forall (A:Set) (x y:A) (p1 p2: x=y), p1 = p2.
```

Axiom K is also equivalent to *Uniqueness of Reflexive Identity Proofs* [13]

```
Coq < Axiom UIP_refl : forall (A:Set) (x:A) (p: x=x), p = refl_equal x.
```

Axiom K is also equivalent to

```
Coq < Axiom
Coq <   eq_rec_eq :
Coq <     forall (A:Set) (x:A) (P: A->Set) (p:P x) (h: x=x),
Coq <       p = eq_rect x P p x h.
```

It is also equivalent to the injectivity of dependent equality (dependent equality is itself equivalent to equality of dependent pairs).

```
Coq < Inductive eq_dep (U:Set) (P:U -> Set) (p:U) (x:P p) :
Coq < forall q:U, P q -> Prop :=
Coq <     eq_dep_intro : eq_dep U P p x p x.
```

```
Coq < Axiom
Coq <   eq_dep_eq :
Coq <     forall (U:Set) (u:U) (P:U -> Set) (p1 p2:P u),
Coq <       eq_dep U P u p1 u p2 -> p1 = p2.
```

33 What is proof-irrelevance

A specificity of the Calculus of Inductive Constructions is to permit statements about proofs. This leads to the question of comparing two proofs of the same proposition. Identifying all proofs of the same proposition is called *proof-irrelevance*:

$$\forall A : \texttt{Prop}, \forall pq : A, p = q$$

Proof-irrelevance (in `Prop`) can be assumed without contradiction in Coq. It expresses that only provability matters, whatever the exact form of the proof is. This is in harmony with the common purely logical interpretation of `Prop`. Contrastingly, proof-irrelevance is inconsistent in `Set` since there are types in `Set`, such as the type of booleans, that provably have at least two distinct elements.

Proof-irrelevance (in `Prop`) is a consequence of classical logic (see proofs in file `Classical.v` and `Berardi.v`). Proof-irrelevance is also a consequence of propositional extensionality (i.e. `(A <-> B) -> A=B`, see the proof in file `ClassicalFacts.v`).

Proof-irrelevance directly implies Streicher's axiom K.

34 What about functional extensionality?

Extensionality of functions is admittedly consistent with the Set-predicative Calculus of Inductive Constructions.

Let `A`, `B` be types. To deal with extensionality on `A->B` without relying on a general extensionality axiom, a possible approach is to define one's own extensional equality on `A->B`.

```
Coq < Definition ext_eq (f g: A->B) := forall x:A, f x = g x.
```

and to reason on `A->B` as a setoid (see the Chapter on Setoids in the Reference Manual).

35 Is `Prop` impredicative?

Yes, the sort `Prop` of propositions is *impredicative*. Otherwise said, a statement of the form $\forall A : Prop, P(A)$ can be instantiated by itself: if $\forall A : \texttt{Prop}, P(A)$ is provable, then $P(\forall A : \texttt{Prop}, P(A))$ is.

36 Is `Set` impredicative?

No, the sort `Set` lying at the bottom of the hierarchy of computational types is *predicative* in the basic Coq system. This means that a family of types in `Set`, e.g. $\forall A : \texttt{Set}, A \to A$, is not a type in `Set` and it cannot be applied on itself.

However, the sort `Set` was impredicative in the original versions of Coq. For backward compatibility, or for experiments by knowledgeable users, the logic of Coq can be set impredicative for `Set` by calling Coq with the option `-impredicative-set`.

`Set` has been made predicative from version 8.0 of Coq. The main reason is to interact smoothly with a classical mathematical world where both excluded-middle and the axiom of description are valid (see file `ClassicalDescription.v` for a proof that excluded-middle and description implies the double negation of excluded-middle in `Set` and file `Hurkens_Set.v` from the user contribution `Rocq/PARADOXES` for a proof that impredicativity of `Set` implies the simple negation of excluded-middle in `Set`).

37 Is `Type` impredicative?

No, `Type` is stratified. This is hidden for the user, but Coq internally maintains a set of constraints ensuring stratification.

If `Type` were impredicative then it would be possible to encode Girard's systems $U-$ and U in Coq and it is known from Girard, Coquand, Hurkens and Miquel that systems $U-$ and U are inconsistent [Girard 1972, Coquand 1991, Hurkens 1993, Miquel 2001]. This encoding can be found in file `Logic/Hurkens.v` of Coq standard library.

For instance, when the user see $\forall$ `X:Type, X->X : Type`, each occurrence of `Type` is implicitly bound to a different level, say α and β and the actual statement is `forall X:Type(`α`), X->X : Type(`β`)` with the constraint $\alpha < \beta$.

When a statement violates a constraint, the message `Universe inconsistency` appears. Example: `fun (x:Type) (y:`$\forall$ `X:Type, X -> X) => y x x`.

38 I have two proofs of the same proposition. Can I prove they are equal?

In the base Coq system, the answer is generally no. However, if classical logic is set, the answer is yes for propositions in `Prop`. The answer is also yes if proof irrelevance holds (see question 33).

There are also "simple enough" propositions for which you can prove the equality without requiring any extra axioms. This is typically the case for propositions defined deterministically as a first-order inductive predicate on decidable sets. See for instance in question 138 an axiom-free proof of the unicity of the proofs of the proposition `le m n` (less or equal on `nat`).

39 I have two proofs of an equality statement. Can I prove they are equal?

Yes, if equality is decidable on the domain considered (which is the case for `nat`, `bool`, etc): see Coq file `Eqdep_dec.v`). No otherwise, unless assuming Streicher's axiom K (see [13]) or a more general assumption such as proof-irrelevance (see 33) or classical logic.

All of these statements can be found in file `Eqdep.v`.

40 Can I prove that the second components of equal dependent pairs are equal?

The answer is the same as for proofs of equality statements. It is provable if equality on the domain of the first component is decidable (look at `inj_right_pair` from file `Eqdep_dec.v`), but not provable in the general case. However, it is consistent (with the Calculus of Constructions) to assume it is true. The file `Eqdep.v` actually provides an axiom (equivalent to Streicher's axiom K) which entails the result (look at `inj_pair2` in `Eqdep.v`).

5.3 Impredicativity

41 Why `injection` does not work on impredicative `Set`?

E.g. in this case (this occurs only in the `Set`-impredicative variant of Coq):

```
Coq < Inductive I : Type :=
Coq <   intro : forall k:Set, k -> I.

Coq < Lemma eq_jdef :
Coq <    forall x y:nat, intro _ x = intro _ y -> x = y.

Coq < Proof.

Coq <    intros x y H; injection H.
```

Injectivity of constructors is restricted to predicative types. If injectivity on large inductive types were not restricted, we would be allowed to derive an inconsistency (e.g. following the lines of Burali-Forti paradox). The question remains open whether injectivity is consistent on some large inductive types not expressive enough to encode known paradoxes (such as type I above).

42 What is a "large inductive definition"?

An inductive definition in `Prop` or `Set` is called large if its constructors embed sets or propositions. As an example, here is a large inductive type:

```
Coq < Inductive sigST (P:Set -> Set) : Type :=
Coq <   existST : forall X:Set, P X -> sigST P.
```

In the `Set` impredicative variant of Coq, large inductive definitions in `Set` have restricted elimination schemes to prevent inconsistencies. Especially, projecting the set or the proposition content of a large inductive definition is forbidden. If it were allowed, it would be possible to encode e.g. Burali-Forti paradox [10, 5].

43 Is Coq's logic conservative over Coquand's Calculus of Constructions?

Yes for the non Set-impredicative version of the Calculus of Inductive Constructions. Indeed, the impredicative sort of the Calculus of Constructions can only be interpreted as the sort `Prop` since `Set` is predicative. But `Prop` can be

6 Talkin' with the Rooster

6.1 My goal is ..., how can I prove it?

44 My goal is a conjunction, how can I prove it?

Use some theorem or assumption or use the `split` tactic.

```
Coq < Goal forall A B:Prop, A->B-> A/\B.
1 subgoal

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

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
   A /\ B

Coq < split.
2 subgoals

  A : Prop
```

```
  B : Prop
  H : A
  H0 : B
  ============================
   A
subgoal 2 is:
 B
```

```
Coq < assumption.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
   B
```

```
Coq < assumption.
Proof completed.
```

```
Coq < Qed.
intros.
split.
 assumption.

 assumption.

Unnamed_thm is defined
```

45 My goal contains a conjunction as an hypothesis, how can I use it?

If you want to decompose your hypothesis into other hypothesis you can use the `decompose` tactic:

```
Coq < Goal forall A B:Prop, A/\B-> B.
1 subgoal

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

```
Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  H : A /\ B
  ============================
   B
```

```
Coq < decompose [and] H.
1 subgoal

  A : Prop
  B : Prop
  H : A /\ B
  H0 : A
  H1 : B
  ============================
   B
```

```
Coq < assumption.
Proof completed.
```

```
Coq < Qed.
```

```
intros.
decompose [and] H.
assumption.
Unnamed_thm0 is defined
```

46 My goal is a disjunction, how can I prove it?

You can prove the left part or the right part of the disjunction using `left` or `right` tactics. If you want to do a classical reasoning step, use the `classic` axiom to prove the right part with the assumption that the left part of the disjunction is false.

```
Coq < Goal forall A B:Prop, A-> A\/B.
1 subgoal

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

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  H : A
  ============================
   A \/ B

Coq < left.
1 subgoal

  A : Prop
  B : Prop
  H : A
  ============================
   A

Coq < assumption.
Proof completed.

Coq < Qed.
intros.
left.
assumption.
Unnamed_thm1 is defined
```

An example using classical reasoning:

```
Coq < Require Import Classical.

Coq <
Coq < Ltac classical_right :=
Coq < match goal with
Coq < | _:_ |-?X1 \/ _ => (elim (classic X1);intro;[left;trivial|right])
Coq < end.
classical_right is defined

Coq <
Coq < Ltac classical_left :=
Coq < match goal with
Coq < | _:_ |- _ \/?X1 => (elim (classic X1);intro;[right;trivial|left])
Coq < end.
classical_left is defined

Coq <
Coq <
```

```
Coq < Goal forall A B:Prop, (~A -> B) -> A\/B.
1 subgoal

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

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  H : ~ A -> B
  ============================
   A \/ B

Coq < classical_right.
1 subgoal

  A : Prop
  B : Prop
  H : ~ A -> B
  H0 : ~ A
  ============================
   B

Coq < auto.
Proof completed.

Coq < Qed.
intros.
classical_right.
auto.
Unnamed_thm2 is defined
```

47 My goal is an universally quantified statement, how can I prove it?

Use some theorem or assumption or introduce the quantified variable in the context using the `intro` tactic. If there are several variables you can use the `intros` tactic. A good habit is to provide names for these variables: Coq will do it anyway, but such automatic naming decreases legibility and robustness.

48 My goal contains an universally quantified statement, how can I use it?

If the universally quantified assumption matches the goal you can use the `apply` tactic. If it is an equation you can use the `rewrite` tactic. Otherwise you can use the `specialize` tactic to instantiate the quantified variables with terms. The variant `assert(Ht := H t)` makes a copy of assumption `H` before instantiating it.

49 My goal is an existential, how can I prove it?

Use some theorem or assumption or exhibit the witness using the `exists` tactic.

```
Coq < Goal exists x:nat, forall y, x+y=y.
1 subgoal

  ============================
   exists x : nat, forall y : nat, x + y = y

Coq < exists 0.
1 subgoal

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

Coq < intros.
```

```
1 subgoal

  y : nat
  ============================
   0 + y = y

Coq < auto.
Proof completed.

Coq < Qed.
exists 0.
intros.
auto.
Unnamed_thm3 is defined
```

50 My goal is solvable by some lemma, how can I prove it?

Just use the `apply` tactic.

```
Coq < Lemma mylemma : forall x, x+0 = x.
1 subgoal

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

Coq < auto.
Proof completed.

Coq < Qed.
auto.
mylemma is defined

Coq <
Coq < Goal 3+0 = 3.
1 subgoal

  ============================
   3 + 0 = 3

Coq < apply mylemma.
Proof completed.

Coq < Qed.
apply mylemma.
Unnamed_thm is defined
```

51 My goal contains False as an hypothesis, how can I prove it?

You can use the `contradiction` or `intuition` tactics.

52 My goal is an equality of two convertible terms, how can I prove it?

Just use the `reflexivity` tactic.

```
Coq < Goal forall x, 0+x = x.
1 subgoal

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

Coq < intros.
1 subgoal

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

```
  0 + x = x
```

```
Coq < reflexivity.
Proof completed.
```

```
Coq < Qed.
intros.
reflexivity.
Unnamed_thm0 is defined
```

53 My goal is a `let x := a in ...`, how can I prove it?

Just use the `intro` tactic.

54 My goal is a `let (a, ..., b) := c in`, how can I prove it?

Just use the `destruct` c as (a,...,b) tactic.

55 My goal contains some existential hypotheses, how can I use it?

You can use the tactic `elim` with you hypotheses as an argument.

56 My goal contains some existential hypotheses, how can I use it and decompose my knowledge about this new thing into different hypotheses?

```
Ltac DecompEx H P := elim H;intro P;intro TO;decompose [and] TO;clear TO;clear H.
```

57 My goal is an equality, how can I swap the left and right hand terms?

Just use the `symmetry` tactic.

```
Coq < Goal forall x y : nat, x=y -> y=x.
1 subgoal

  ============================
   forall x y : nat, x = y -> y = x
```

```
Coq < intros.
1 subgoal

  x : nat
  y : nat
  H : x = y
  ============================
   y = x
```

```
Coq < symmetry.
1 subgoal

  x : nat
  y : nat
  H : x = y
  ============================
   x = y
```

```
Coq < assumption.
Proof completed.
```

```
Coq < Qed.
intros.
symmetry .
assumption.
Unnamed_thm1 is defined
```

58 My hypothesis is an equality, how can I swap the left and right hand terms?

Just use the `symmetryin` tactic.

```
Coq < Goal forall x y : nat, x=y -> y=x.
1 subgoal

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

Coq < intros.
1 subgoal

  x : nat
  y : nat
  H : x = y
  ============================
   y = x

Coq < symmetry in H.
1 subgoal

  x : nat
  y : nat
  H : y = x
  ============================
   y = x

Coq < assumption.
Proof completed.

Coq < Qed.
intros.
symmetry  in H.
assumption.
Unnamed_thm2 is defined
```

59 My goal is an equality, how can I prove it by transitivity?

Just use the `transitivity` tactic.

```
Coq < Goal forall x y z : nat, x=y -> y=z -> x=z.
1 subgoal

  ============================
   forall x y z : nat, x = y -> y = z -> x = z

Coq < intros.
1 subgoal

  x : nat
  y : nat
  z : nat
  H : x = y
  H0 : y = z
  ============================
   x = z

Coq < transitivity y.
2 subgoals

  x : nat
  y : nat
  z : nat
  H : x = y
```

```
  H0 : y = z
  ============================
   x = y
subgoal 2 is:
 y = z
```

```
Coq < assumption.
1 subgoal

  x : nat
  y : nat
  z : nat
  H : x = y
  H0 : y = z
  ============================
   y = z
```

```
Coq < assumption.
Proof completed.
```

```
Coq < Qed.
intros.
transitivity y.
 assumption.

 assumption.

Unnamed_thm3 is defined
```

60 My goal would be solvable using `apply;assumption` if it would not create meta-variables, how can I prove it?

You can use `eapply yourtheorem;eauto` but it won't work in all cases ! (for example if more than one hypothesis match one of the subgoals generated by `eapply`) so you should rather use `try solve [eapply yourtheorem;eauto]`, otherwise some metavariables may be incorrectly instantiated.

```
Coq < Lemma trans : forall x y z : nat, x=y -> y=z -> x=z.
1 subgoal

  ============================
   forall x y z : nat, x = y -> y = z -> x = z
```

```
Coq < intros.
1 subgoal

  x : nat
  y : nat
  z : nat
  H : x = y
  H0 : y = z
  ============================
   x = z
```

```
Coq < transitivity y;assumption.
Proof completed.
```

```
Coq < Qed.
intros.
transitivity y; assumption.
trans is defined
```

```
Coq <
Coq < Goal forall x y z : nat, x=y -> y=z -> x=z.
1 subgoal
```

```
  ============================
   forall x y z : nat, x = y -> y = z -> x = z

Coq < intros.
1 subgoal

  x : nat
  y : nat
  z : nat
  H : x = y
  H0 : y = z
  ============================
   x = z

Coq < eapply trans;eauto.
Proof completed.

Coq < Qed.
intros.
eapply trans; eauto   .
Unnamed_thm4 is defined

Coq <
Coq < Goal forall x y z t : nat, x=y -> x=t -> y=z -> x=z.
1 subgoal

  ============================
   forall x y z t : nat, x = y -> x = t -> y = z -> x = z

Coq < intros.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   x = z

Coq < eapply trans;eauto.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   t = z

Coq < Undo.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
```

```
  H0 : x = t
  H1 : y = z
  ============================
   x = z

Coq < eapply trans.
2 subgoals

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   x = ?146
subgoal 2 is:
 ?146 = z

Coq < apply H.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   y = z

Coq < auto.
Proof completed.

Coq < Qed.
intros.
eapply trans.
 apply H.

 auto.

Unnamed_thm5 is defined

Coq <
Coq < Goal forall x y z t : nat, x=y -> x=t -> y=z -> x=z.
1 subgoal

  ============================
   forall x y z t : nat, x = y -> x = t -> y = z -> x = z

Coq < intros.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   x = z
```

```
Coq < eapply trans;eauto.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   t = z

Coq < Undo.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   x = z

Coq < try solve [eapply trans;eauto].
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   x = z

Coq < eapply trans.
2 subgoals

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
  H1 : y = z
  ============================
   x = ?161
subgoal 2 is:
 ?161 = z

Coq < apply H.
1 subgoal

  x : nat
  y : nat
  z : nat
  t : nat
  H : x = y
  H0 : x = t
```

```
  H1 : y = z
  ============================
   y = z

Coq < auto.
Proof completed.

Coq < Qed.
intros.
try (solve [ eapply trans; eauto    ]).
eapply trans.
 apply H.

 auto.

Unnamed_thm6 is defined

Coq <
```

61 My goal is solvable by some lemma within a set of lemmas and I don't want to remember which one, how can I prove it?

You can use a what is called a hints' base.

```
Coq < Require Import ZArith.
Coq < [Loading ML file z_syntax_plugin.cmxs ... done]
[Loading ML file quote_plugin.cmxs ... done]
[Loading ML file newring_plugin.cmxs ... done]
[Loading ML file omega_plugin.cmxs ... done]

Coq < Require Ring.

Coq < Open Local Scope Z_scope.
Warning: Obsolete syntax: use "Local" as a prefix.

Coq < Lemma toto1 : 1+1 = 2.
1 subgoal

  ============================
   1 + 1 = 2

Coq < ring.
Proof completed.

Coq < Qed.
ring.
toto1 is defined

Coq < Lemma toto2 : 2+2 = 4.
1 subgoal

  ============================
   2 + 2 = 4

Coq < ring.
Proof completed.

Coq < Qed.
ring.
toto2 is defined

Coq < Lemma toto3 : 2+1 = 3.
1 subgoal

  ============================
   2 + 1 = 3
```

```
Coq < ring.
Proof completed.

Coq < Qed.
ring.
toto3 is defined

Coq <
Coq < Hint Resolve toto1 toto2 toto3 : mybase.

Coq <
Coq < Goal 2+(1+1)=4.
1 subgoal

  ============================
   2 + (1 + 1) = 4

Coq < auto with mybase.
Proof completed.

Coq < Qed.
auto with mybase.
Unnamed_thm7 is defined
```

62 My goal is one of the hypotheses, how can I prove it?

Use the `assumption` tactic.

```
Coq < Goal 1=1 -> 1=1.
1 subgoal

  ============================
   1 = 1 -> 1 = 1

Coq < intro.
1 subgoal

  H : 1 = 1
  ============================
   1 = 1

Coq < assumption.
Proof completed.

Coq < Qed.
intro.
assumption.
Unnamed_thm8 is defined
```

63 My goal appears twice in the hypotheses and I want to choose which one is used, how can I do it?

Use the `exact` tactic.

```
Coq < Goal 1=1 -> 1=1 -> 1=1.
1 subgoal

  ============================
   1 = 1 -> 1 = 1 -> 1 = 1

Coq < intros.
1 subgoal

  H : 1 = 1
  H0 : 1 = 1
```

```
  ============================
   1 = 1

Coq < exact H0.
Proof completed.

Coq < Qed.
intros.
exact H0.
Unnamed_thm9 is defined
```

64 What can be the difference between applying one hypothesis or another in the context of the last question?

From a proof point of view it is equivalent but if you want to extract a program from your proof, the two hypotheses can lead to different programs.

65 My goal is a propositional tautology, how can I prove it?

Just use the `tauto` tactic.

```
Coq < Goal forall A B:Prop, A-> (A\/B) /\ A.
1 subgoal

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

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  H : A
  ============================
   (A \/ B) /\ A

Coq < tauto.
Proof completed.

Coq < Qed.
intros.
tauto.
Unnamed_thm10 is defined
```

66 My goal is a first order formula, how can I prove it?

Just use the semi-decision tactic: `firstorder`.

67 My goal is solvable by a sequence of rewrites, how can I prove it?

Just use the `congruence` tactic.

```
Coq < Goal forall a b c d e, a=d -> b=e -> c+b=d -> c+e=a.
1 subgoal

  ============================
   forall a b c d e : Z, a = d -> b = e -> c + b = d -> c + e = a

Coq < intros.
1 subgoal

  a : Z
  b : Z
  c : Z
```

```
  d : Z
  e : Z
  H : a = d
  H0 : b = e
  H1 : c + b = d
  ============================
   c + e = a

Coq < congruence.
Proof completed.

Coq < Qed.
intros.
congruence.
Unnamed_thm11 is defined
```

68 My goal is a disequality solvable by a sequence of rewrites, how can I prove it?

Just use the congruence tactic.

```
Coq < Goal forall a b c d, a<>d -> b=a -> d=c+b -> b<>c+b.
1 subgoal

  ============================
   forall a b c d : Z, a <> d -> b = a -> d = c + b -> b <> c + b

Coq < intros.
1 subgoal

  a : Z
  b : Z
  c : Z
  d : Z
  H : a <> d
  H0 : b = a
  H1 : d = c + b
  ============================
   b <> c + b

Coq < congruence.
Proof completed.

Coq < Qed.
intros.
congruence.
Unnamed_thm12 is defined
```

69 My goal is an equality on some ring (e.g. natural numbers), how can I prove it?

Just use the ring tactic.

```
Coq < Require Import ZArith.

Coq < Require Ring.

Coq < Open Local Scope Z_scope.
Warning: Obsolete syntax: use "Local" as a prefix.

Coq < Goal forall a b : Z, (a+b)*(a+b) = a*a + 2*a*b + b*b.
1 subgoal

  ============================
   forall a b : Z, (a + b) * (a + b) = a * a + 2 * a * b + b * b

Coq < intros.
```

```
1 subgoal

  a : Z
  b : Z
  ============================
   (a + b) * (a + b) = a * a + 2 * a * b + b * b

Coq < ring.
Proof completed.

Coq < Qed.
intros.
ring.
Unnamed_thm13 is defined
```

70 My goal is an equality on some field (e.g. real numbers), how can I prove it?

Just use the `field` tactic.

```
Coq < Require Import Reals.
[Loading ML file r_syntax_plugin.cmxs ... done]
[Loading ML file ring_plugin.cmxs ... done]
[Loading ML file field_plugin.cmxs ... done]
[Loading ML file fourier_plugin.cmxs ... done]

Coq < Require Ring.

Coq < Open Local Scope R_scope.
Warning: Obsolete syntax: use "Local" as a prefix.

Coq < Goal forall a b : R, b*a<>0 -> (a/b) * (b/a) = 1.
1 subgoal

  ============================
   forall a b : R, b * a <> 0 -> a / b * (b / a) = 1

Coq < intros.
1 subgoal

  a : R
  b : R
  H : b * a <> 0
  ============================
   a / b * (b / a) = 1

Coq < field.
1 subgoal

  a : R
  b : R
  H : b * a <> 0
  ============================
   a <> 0 /\ b <> 0

Coq < cut (b*a <>0 -> a<>0).
2 subgoals

  a : R
  b : R
  H : b * a <> 0
  ============================
   (b * a <> 0 -> a <> 0) -> a <> 0 /\ b <> 0
subgoal 2 is:
 b * a <> 0 -> a <> 0
```

```
Coq < cut (b*a <>0 -> b<>0).
3 subgoals

  a : R
  b : R
  H : b * a <> 0
  ============================
   (b * a <> 0 -> b <> 0) -> (b * a <> 0 -> a <> 0) -> a <> 0 /\ b <> 0
subgoal 2 is:
 b * a <> 0 -> b <> 0
subgoal 3 is:
 b * a <> 0 -> a <> 0

Coq < auto.
2 subgoals

  a : R
  b : R
  H : b * a <> 0
  ============================
   b * a <> 0 -> b <> 0
subgoal 2 is:
 b * a <> 0 -> a <> 0

Coq < auto with real.
1 subgoal

  a : R
  b : R
  H : b * a <> 0
  ============================
   b * a <> 0 -> a <> 0

Coq < auto with real.
Proof completed.

Coq < Qed.
intros.
field.
cut (b * a <> 0 -> a <> 0).
 cut (b * a <> 0 -> b <> 0).
  auto.

  auto with real.

 auto with real.

Unnamed_thm14 is defined
```

71 My goal is an inequality on integers in Presburger's arithmetic (an expression build from +,-,constants and variables), how can I prove it?

```
Coq < Require Import ZArith.

Coq < Require Omega.

Coq < Open Local Scope Z_scope.
Warning: Obsolete syntax: use "Local" as a prefix.

Coq < Goal forall a : Z, a>0 -> a+a > a.
1 subgoal

  ============================
   forall a : Z, a > 0 -> a + a > a
```

```
Coq < intros.
1 subgoal

  a : Z
  H : a > 0
  ============================
   a + a > a

Coq < omega.
Proof completed.

Coq < Qed.
intros.
omega.
Unnamed_thm15 is defined
```

72 My goal is an equation solvable using equational hypothesis on some ring (e.g. natural numbers), how can I prove it?

You need the `gb` tactic (see Loïc Pottier's homepage).

6.2 Tactics usage

73 I want to state a fact that I will use later as an hypothesis, how can I do it?

If you want to use forward reasoning (first proving the fact and then using it) you just need to use the `assert` tactic. If you want to use backward reasoning (proving your goal using an assumption and then proving the assumption) use the `cut` tactic.

```
Coq < Goal forall A B C D : Prop, (A -> B) -> (B->C) -> A -> C.
1 subgoal

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

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  ============================
   C

Coq < assert (A->C).
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  ============================
   A -> C
subgoal 2 is:
 C
```

```
Coq < intro;apply H0;apply H;assumption.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  H2 : A -> C
  ============================
   C

Coq < apply H2.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  H2 : A -> C
  ============================
   A

Coq < assumption.
Proof completed.

Coq < Qed.
intros.
assert (A -> C).
 intro; apply H0; apply H; assumption.

 apply H2.
 assumption.

Unnamed_thm16 is defined

Coq <
Coq < Goal forall A B C D : Prop, (A -> B) -> (B->C) -> A -> C.
1 subgoal

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

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  ============================
   C

Coq < cut (A->C).
2 subgoals
```

```
  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  ============================
   (A -> C) -> C
subgoal 2 is:
 A -> C
```

```
Coq < intro.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  H2 : A -> C
  ============================
   C
subgoal 2 is:
 A -> C
```

```
Coq < apply H2;assumption.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H : A -> B
  H0 : B -> C
  H1 : A
  ============================
   A -> C
```

```
Coq < intro;apply H0;apply H;assumption.
Proof completed.
```

```
Coq < Qed.
intros.
cut (A -> C).
 intro.
 apply H2; assumption.

 intro; apply H0; apply H; assumption.

Unnamed_thm17 is defined
```

74 I want to state a fact that I will use later as an hypothesis and prove it later, how can I do it?

You can use `cut` followed by `intro` or you can use the following Ltac command:

```
Ltac assert_later t := cut t;[intro|idtac].
```

75 What is the difference between `Qed` and `Defined`?

These two commands perform type checking, but when `Defined` is used the new definition is set as transparent, otherwise it is defined as opaque (see 170).

76 How can I know what a tactic does?

You can use the `info` command.

77 Why `auto` does not work? How can I fix it?

You can increase the depth of the proof search or add some lemmas in the base of hints. Perhaps you may need to use `eauto`.

78 What is `eauto`?

This is the same tactic as `auto`, but it relies on `eapply` instead of `apply`.

79 How can I speed up `auto`?

You can use `info auto` to replace `auto` by the tactics it generates. You can split your hint bases into smaller ones.

80 What is the equivalent of `tauto` for classical logic?

Currently there are no equivalent tactic for classical logic. You can use Gödel's "not not" translation.

81 I want to replace some term with another in the goal, how can I do it?

If one of your hypothesis (say `H`) states that the terms are equal you can use the `rewrite` tactic. Otherwise you can use the `replace with` tactic.

82 I want to replace some term with another in an hypothesis, how can I do it?

You can use the `rewrite in` tactic.

83 I want to replace some symbol with its definition, how can I do it?

You can use the `unfold` tactic.

84 How can I reduce some term?

You can use the `simpl` tactic.

85 How can I declare a shortcut for some term?

You can use the `set` or `pose` tactics.

86 How can I perform case analysis?

You can use the `case` or `destruct` tactics.

87 How can I prevent the case tactic from losing information ?

You may want to use the (now standard) `case_eq` tactic. See the Coq'Art page 159.

88 Why should I name my intros?

When you use the `intro` tactic you don't have to give a name to your hypothesis. If you do so the name will be generated by Coq but your scripts may be less robust. If you add some hypothesis to your theorem (or change their order), you will have to change your proof to adapt to the new names.

89 How can I automatize the naming?

You can use the `Show Intro.` or `Show Intros.` commands to generate the names and use your editor to generate a fully named `intro` tactic. This can be automatized within `xemacs`.

```
Coq < Goal forall A B C : Prop, A -> B -> C -> A/\B/\C.
1 subgoal

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

Coq < Show Intros.
A B C H H0 H1

Coq < (*
Coq < A B C H H0
Coq < H1
Coq < *)
Coq < intros A B C H H0 H1.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A
  H0 : B
  H1 : C
  ============================
   A /\ B /\ C

Coq < repeat split;assumption.
Proof completed.

Coq < Qed.
intros A B C H H0 H1.
repeat split; assumption.
Unnamed_thm18 is defined
```

90 I want to automatize the use of some tactic, how can I do it?

You need to use the `proof with T` command and add ... at the end of your sentences.

For instance:

```
Coq < Goal forall A B C : Prop, A -> B/\C -> A/\B/\C.
1 subgoal

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

Coq < Proof with assumption.

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A
```

```
  H0 : B /\ C
  ============================
   A /\ B /\ C

Coq < split...
Proof completed.

Coq < Qed.
intros.
split...
Unnamed_thm19 is defined
```

91 I want to execute the proof with tactic only if it solves the goal, how can I do it?

You need to use the `try` and `solve` tactics. For instance:

```
Coq < Require Import ZArith.

Coq < Require Ring.

Coq < Open Local Scope Z_scope.
Warning: Obsolete syntax: use "Local" as a prefix.

Coq < Goal forall a b c : Z, a+b=b+a.
1 subgoal

  ============================
   forall a b : Z, Z -> a + b = b + a

Coq < Proof with try solve [ring].

Coq < intros...
Proof completed.

Coq < Qed.
intros...
Unnamed_thm20 is defined
```

92 How can I do the opposite of the `intro` tactic?

You can use the `generalize` tactic.

```
Coq < Goal forall A B : Prop, A->B-> A/\B.
1 subgoal

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

Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
   A /\ B

Coq < generalize H.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
```

```
   A -> A /\ B

Coq < intro.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  H1 : A
  ============================
   A /\ B

Coq < auto.
Proof completed.

Coq < Qed.
intros.
generalize H.
intro.
auto.
Unnamed_thm21 is defined
```

93 One of the hypothesis is an equality between a variable and some term, I want to get rid of this variable, how can I do it?

You can use the `subst` tactic. This will rewrite the equality everywhere and clear the assumption.

94 What can I do if I get "`generated subgoal term has metavariables in it `"?

You should use the `eapply` tactic, this will generate some goals containing metavariables.

95 How can I instantiate some metavariable?

Just use the `instantiate` tactic.

96 What is the use of the `pattern` tactic?

The `pattern` tactic transforms the current goal, performing beta-expansion on all the applications featuring this tactic's argument. For instance, if the current goal includes a subterm `phi(t)`, then `pattern t` transforms the subterm into `(fun x:A => phi(x)) t`. This can be useful when `apply` fails on matching, to abstract the appropriate terms.

97 What is the difference between assert, cut and generalize?

PS: Notice for people that are interested in proof rendering that `assert`and `pose` (and `cut`) are not rendered the same as `generalize` (see the HELM experimental rendering tool at `http://helm.cs.unibo.it`, link HELM, link COQ Online). Indeed `generalize` builds a beta-expanded term while `assert`, `pose` and `cut` uses a let-in.

```
(* Goal is T *)
generalize (H1 H2).
(* Goal is A->T *)
... a proof of A->T ...
```

is rendered into something like

```
(h) ... the proof of A->T ...
    we proved A->T
(h0) by (H1 H2) we proved A
by (h h0) we proved T
```

while

```
(* Goal is T *)
assert q := (H1 H2).
(* Goal is A *)
... a proof of A ...
(* Goal is A |- T *)
... a proof of T ...
```

is rendered into something like

```
(q) ... the proof of A ...
    we proved A
... the proof of T ...
we proved T
```

Otherwise said, `generalize` is not rendered in a forward-reasoning way, while `assert` is.

98 What can I do if Coqcan not infer some implicit argument ?

You can state explicitely what this implicit argument is. See 99.

99 How can I explicit some implicit argument ?

Just use `A:=term` where `A` is the argument.

For instance if you want to use the existence of "nil" on nat*nat lists:

```
exists (nil (A:=(nat*nat))).
```

6.3 Proof management

100 How can I change the order of the subgoals?

You can use the `Focus` command to concentrate on some goal. When the goal is proved you will see the remaining goals.

101 How can I change the order of the hypothesis?

You can use the `Move ... after` command.

102 How can I change the name of an hypothesis?

You can use the `Rename ... into` command.

103 How can I delete some hypothesis?

You can use the `Clear` command.

104 How can use a proof which is not finished?

You can use the `Admitted` command to state your current proof as an axiom. You can use the `admit` tactic to omit a portion of a proof.

105 How can I state a conjecture?

You can use the `Admitted` command to state your current proof as an axiom.

106 What is the difference between a lemma, a fact and a theorem?

From Coq point of view there are no difference. But some tools can have a different behavior when you use a lemma rather than a theorem. For instance `coqdoc` will not generate documentation for the lemmas within your development.

107 How can I organize my proofs?

You can organize your proofs using the section mechanism of Coq. Have a look at the manual for further information.

7 Inductive and Co-inductive types

7.1 General

108 How can I prove that two constructors are different?

You can use the `discriminate` tactic.

```
Coq < Inductive toto : Set := | C1 : toto | C2 : toto.
toto is defined
toto_rect is defined
toto_ind is defined
toto_rec is defined

Coq < Goal C1 <> C2.
1 subgoal

  ============================
   C1 <> C2

Coq < discriminate.
Proof completed.

Coq < Qed.
discriminate.
Unnamed_thm22 is defined
```

109 During an inductive proof, how to get rid of impossible cases of an inductive definition?

Use the `inversion` tactic.

110 How can I prove that 2 terms in an inductive set are equal? Or different?

Have a look at `decide equality` and `discriminate` in the Reference Manual.

111 Why is the proof of `0+n=n` on natural numbers trivial but the proof of `n+0=n` is not?

Since `+` (`plus`) on natural numbers is defined by analysis on its first argument

```
Coq < Print plus.
plus =
fix plus (n m : nat) : nat :=
  match n with
  | O%nat => m
  | S p => S (plus p m)
  end
     : nat -> nat -> nat
Argument scopes are [nat_scope nat_scope]
```

The expression `0+n` evaluates to `n`. As Coq reasons modulo evaluation of expressions, `0+n` and `n` are considered equal and the theorem `0+n=n` is an instance of the reflexivity of equality. On the other side, `n+0` does not evaluate to `n` and a proof by induction on `n` is necessary to trigger the evaluation of `+`.

112 Why is dependent elimination in Prop not available by default?

This is just because most of the time it is not needed. To derive a dependent elimination principle in `Prop`, use the command `Scheme` and apply the elimination scheme using the `using` option of `elim`, `destruct` or `induction`.

113 Argh! I cannot write expressions like " `if n <= p then p else n` ", as in any programming language

The short answer : You should use `le_lt_dec n p` instead.

That's right, you can't. If you type for instance the following "definition":

```
Coq < Definition max (n p : nat) := if n <= p then p else n.
Toplevel input, characters 33-39:
> Definition max (n p : nat) := if n <= p then p else n.
>                                 ^^^^^^
Error: The term "n <= p" has type "Prop" which is not a (co-)inductive type.
```

As Coq says, the term " `n <= p` " is a proposition, i.e. a statement that belongs to the mathematical world. There are many ways to prove such a proposition, either by some computation, or using some already proven theoremas. For instance, proving $3 - 2 \leq 2^{45503}$ is very easy, using some theorems on arithmetical operations. If you compute both numbers before comparing them, you risk to use a lot of time and space.

On the contrary, a function for computing the greatest of two natural numbers is an algorithm which, called on two natural numbers n and p, determines wether $n \leq p$ or $p < n$. Such a function is a *decision procedure* for the inequality of `nat`. The possibility of writing such a procedure comes directly from de decidability of the order $\leq$ on natural numbers.

When you write a piece of code like " `if n <= p then ... else ...` " in a programming language like *ML* or *Java*, a call to such a decision procedure is generated. The decision procedure is in general a primitive function, written in a low-level language, in the correctness of which you have to trust.

The standard Library of the system *Coq* contains a (constructive) proof of decidability of the order $\leq$ on `nat` : the function `le_lt_dec` of the module `Compare_dec` of library `Arith`.

The following code shows how to define correctly `min` and `max`, and prove some properties of these functions.

```
Coq < Require Import Compare_dec.

Coq <
Coq < Definition max (n p : nat) := if le_lt_dec n p then p else n.
max is defined

Coq <
Coq < Definition min (n p : nat) := if le_lt_dec n p then n else p.
min is defined

Coq <
Coq < Eval compute in (min 4 7).
     = 4
     : nat

Coq <
Coq < Theorem min_plus_max : forall n p, min n p + max n p  = n + p.
1 subgoal

  ============================
   forall n p : nat, min n p + max n p = n + p

Coq < Proof.

Coq <  intros n p;
Coq <  unfold min, max;
Coq <  case (le_lt_dec n p);
```

```
Coq <  simpl; auto with arith.
Proof completed.

Coq < Qed.
intros n p; unfold min, max; case (le_lt_dec n p); simpl; auto with arith.
min_plus_max is defined

Coq <
Coq < Theorem max_equiv : forall n p, max n p = p <-> n <= p.
1 subgoal

  ============================
   forall n p : nat, max n p = p <-> n <= p

Coq < Proof.

Coq <  unfold max; intros n p; case (le_lt_dec n p);simpl; auto.
2 subgoals

  n : nat
  p : nat
  ============================
   n <= p -> (p = p <-> n <= p)
subgoal 2 is:
 p < n -> (n = p <-> n <= p)

Coq <  intuition auto with arith.
1 subgoal

  n : nat
  p : nat
  ============================
   p < n -> (n = p <-> n <= p)

Coq <  split.
2 subgoals

  n : nat
  p : nat
  l : p < n
  ============================
   n = p -> n <= p
subgoal 2 is:
 n <= p -> n = p

Coq <  intro e; rewrite e; auto with arith.
1 subgoal

  n : nat
  p : nat
  l : p < n
  ============================
   n <= p -> n = p

Coq <  intro H; absurd (p < p); eauto with arith.
Proof completed.

Coq < Qed.
unfold max; intros n p; case (le_lt_dec n p); simpl; auto.
 intuition auto with arith.

 split.
  intro e; rewrite e; auto with arith.

  intro H; absurd (p < p); eauto    with arith.
```

```
max_equiv is defined
```

114 I wrote my own decision procedure for $\leq$, which is much faster than yours, but proving such theorems as max_equiv seems to be quite difficult

Your code is probably the following one:

```
Coq < Fixpoint my_le_lt_dec (n p :nat) {struct n}: bool  :=
Coq <  match n, p with 0, _ => true
Coq <                | S n', S p' => my_le_lt_dec n' p'
Coq <                | _   , _    => false
Coq <  end.
my_le_lt_dec is recursively defined (decreasing on 1st argument)

Coq <
Coq < Definition my_max (n p:nat) := if my_le_lt_dec n p then p else n.
my_max is defined

Coq <
Coq < Definition my_min (n p:nat) := if my_le_lt_dec n p then n else p.
my_min is defined
```

For instance, the computation of `my_max 567 321` is almost immediate, whereas one can't wait for the result of `max 56 32`, using *Coq's* `le_lt_dec`.

This is normal. Your definition is a simple recursive function which returns a boolean value. Coq's `le_lt_dec` is a *certified function*, i.e. a complex object, able not only to tell wether $n \leq p$ or $p < n$, but also of building a complete proof of the correct inequality. What make `le_lt_dec` inefficient for computing `min` and `max` is the building of a huge proof term.

Nevertheless, `le_lt_dec` is very useful. Its type is a strong specification, using the `sumbool` type (look at the reference manual or chapter 9 of [1]). Eliminations of the form " `case (le_lt_dec n p)` " provide proofs of either $n \leq p$ or $p < n$, allowing to prove easily theorems as in question 113. Unfortunately, this not the case of your `my_le_lt_dec`, which returns a quite non-informative boolean value.

```
Coq < Check le_lt_dec.
le_lt_dec
     : forall n m : nat, {n <= m} + {m < n}
```

You should keep in mind that `le_lt_dec` is useful to build certified programs which need to compare natural numbers, and is not designed to compare quickly two numbers.

Nevertheless, the *extraction* of `le_lt_dec` towards *Ocaml* or *Haskell*, is a reasonable program for comparing two natural numbers in Peano form in linear time.

It is also possible to keep your boolean function as a decision procedure, but you have to establish yourself the relationship between `my_le_lt_dec` and the propositions $n \leq p$ and $p < n$:

```
Coq < Theorem my_le_lt_dec_true :
Coq <     forall n p, my_le_lt_dec n p = true <-> n <= p.

Coq <
Coq < Theorem  my_le_lt_dec_false :
Coq <    forall n p, my_le_lt_dec n p = false <-> p < n.
```

7.2 Recursion

115 Why can't I define a non terminating program?

Because otherwise the decidability of the type-checking algorithm (which involves evaluation of programs) is not ensured. On another side, if non terminating proofs were allowed, we could get a proof of `False`:

```
Coq < (* This is fortunately not allowed! *)
Coq < Fixpoint InfiniteProof (n:nat) : False := InfiniteProof n.

Coq < Theorem Paradox : False.

Coq < Proof (InfiniteProof 0).
```

116 Why only structurally well-founded loops are allowed?

The structural order on inductive types is a simple and powerful notion of termination. The consistency of the Calculus of Inductive Constructions relies on it and another consistency proof would have to be made for stronger termination arguments (such as the termination of the evaluation of CIC programs themselves!).

In spite of this, all non-pathological termination orders can be mapped to a structural order. Tools to do this are provided in the file `Wf.v` of the standard library of Coq.

117 How to define loops based on non structurally smaller recursive calls?

The procedure is as follows (we consider the definition of `mergesort` as an example).

- Define the termination order, say `R` on the type `A` of the arguments of the loop.

```
Coq < Definition R (a b:list nat) := length a < length b.
```

- Prove that this order is well-founded (in fact that all elements in `A` are accessible along `R`).

```
Coq < Lemma Rwf : well_founded R.
```

- Define the step function (which needs proofs that recursive calls are on smaller arguments).

```
Coq < Definition split (l : list nat)
Coq <    : {l1: list nat | R l1 l} * {l2 : list nat | R l2 l}
Coq <    := (* ... *) .
Coq < Definition concat (l1 l2 : list nat) : list nat := (* ... *) .
Coq < Definition merge_step (l : list nat) (f: forall l':list nat, R l' l -> list nat) :=
Coq <   let (lH1,lH2) := (split l) in
Coq <   let (l1,H1) := lH1 in
Coq <   let (l2,H2) := lH2 in
Coq <   concat (f l1 H1) (f l2 H2).
```

- Define the recursive function by fixpoint on the step function.

```
Coq < Definition merge := Fix Rwf (fun _ => list nat) merge_step.
```

118 What is behind the accessibility and well-foundedness proofs?

Well-foundedness of some relation `R` on some type `A` is defined as the accessibility of all elements of `A` along `R`.

```
Coq < Print well_founded.
well_founded =
fun (A : Type) (R : A -> A -> Prop) => forall a : A, Acc R a
     : forall A : Type, (A -> A -> Prop) -> Prop
Argument A is implicit
Argument scopes are [type_scope _]
Coq < Print Acc.
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
    Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
For Acc: Argument A is implicit
For Acc_intro: Arguments A, R are implicit
For Acc: Argument scopes are [type_scope _ _]
For Acc_intro: Argument scopes are [type_scope _ _ _]
```

The structure of the accessibility predicate is a well-founded tree branching at each node `x` in `A` along all the nodes `x'` less than `x` along `R`. Any sequence of elements of `A` decreasing along the order `R` are branches in the accessibility tree. Hence any decreasing along `R` is mapped into a structural decreasing in the accessibility tree of `R`. This is emphasised in the definition of `fix` which recurs not on its argument `x:A` but on the accessibility of this argument along `R`.

See file `Wf.v`.

119 How to perform simultaneous double induction?

In general a (simultaneous) double induction is simply solved by an induction on the first hypothesis followed by an inversion over the second hypothesis. Here is an example

```
Coq < Inductive even : nat -> Prop :=
Coq <   | even_0 : even 0
Coq <   | even_S : forall n:nat, even n -> even (S (S n)).
even is defined
even_ind is defined

Coq <
Coq < Inductive odd : nat -> Prop :=
Coq <   | odd_S0 : odd 1
Coq <   | odd_S : forall n:nat, odd n -> odd (S (S n)).
odd is defined
odd_ind is defined

Coq <
Coq < Lemma not_even_and_odd : forall n:nat, even n -> odd n -> False.
1 subgoal

  ============================
   forall n : nat, even n -> odd n -> False

Coq < induction 1.
2 subgoals

  ============================
   odd 0 -> False
subgoal 2 is:
 odd (S (S n)) -> False

Coq <   inversion 1.
1 subgoal

  n : nat
  H : even n
  IHeven : odd n -> False
  ============================
   odd (S (S n)) -> False

Coq <   inversion 1. apply IHeven; trivial.
1 subgoal

  n : nat
  H : even n
  IHeven : odd n -> False
  H0 : odd (S (S n))
  n0 : nat
  H2 : odd n
  H1 : n0 = n
  ============================
   False
Proof completed.
```

In case the type of the second induction hypothesis is not dependent, `inversion` can just be replaced by `destruct`.

120 How to define a function by simultaneous double recursion?

The same trick applies, you can even use the pattern-matching compilation algorithm to do the work for you. Here is an example:

```
Coq < Fixpoint minus (n m:nat) {struct n} : nat :=
Coq <   match n, m with
Coq <   | O, _ => O
Coq <   | S k, O => S k
Coq <   | S k, S l => minus k l
Coq <   end.
minus is recursively defined (decreasing on 1st argument)

Coq < Print  minus.
minus =
fix minus (n m : nat) : nat :=
  match n with
  | O => O
  | S k => match m with
           | O => S k
           | S l => minus k l
           end
  end
     : nat -> nat -> nat
Argument scopes are [nat_scope nat_scope]
```

In case of dependencies in the type of the induction objects t_1 and t_2, an extra argument stating $t_1 = t_2$ must be given to the fixpoint definition

121 How to perform nested and double induction?

To reason by nested (i.e. lexicographic) induction, just reason by induction on the successive components.

Double induction (or induction on pairs) is a restriction of the lexicographic induction. Here is an example of double induction.

```
Coq < Lemma nat_double_ind :
Coq < forall P : nat -> nat -> Prop, P 0 0 ->
Coq <   (forall m n, P m n -> P m (S n)) ->
Coq <   (forall m n, P m n -> P (S m) n) ->
Coq <      forall m n, P m n.
1 subgoal

  ============================
   forall P : nat -> nat -> Prop,
   P 0 0 ->
   (forall m n : nat, P m n -> P m (S n)) ->
   (forall m n : nat, P m n -> P (S m) n) -> forall m n : nat, P m n

Coq < intros P H00 HmS HSn; induction m.
2 subgoals

  P : nat -> nat -> Prop
  H00 : P 0 0
  HmS : forall m n : nat, P m n -> P m (S n)
  HSn : forall m n : nat, P m n -> P (S m) n
  ============================
   forall n : nat, P 0 n
subgoal 2 is:
 forall n : nat, P (S m) n

Coq < (* case 0 *)
Coq < induction n; [assumption | apply HmS; apply IHn].
1 subgoal

  P : nat -> nat -> Prop
  H00 : P 0 0
  HmS : forall m n : nat, P m n -> P m (S n)
```

```
  HSn : forall m n : nat, P m n -> P (S m) n
  m : nat
  IHm : forall n : nat, P m n
  ============================
   forall n : nat, P (S m) n

Coq < (* case Sm *)
Coq < intro n; apply HSn; apply IHm.
Proof completed.
```

122 How to define a function by nested recursion?

The same trick applies. Here is the example of Ackermann function.

```
Coq < Fixpoint ack (n:nat) : nat -> nat :=
Coq <   match n with
Coq <   | O => S
Coq <   | S n' =>
Coq <      (fix ack' (m:nat) : nat :=
Coq <         match m with
Coq <         | O => ack n' 1
Coq <         | S m' => ack n' (ack' m')
Coq <         end)
Coq <   end.
ack is recursively defined (decreasing on 1st argument)
```

7.3 Co-inductive types

123 I have a cofixpoint $t := F(t)$ and I want to prove $t = F(t)$. How to do it?

Just case-expand $F(\mathtt{t})$ then complete by a trivial case analysis. Here is what it gives on e.g. the type of streams on naturals

```
Coq < CoInductive Stream (A:Set) : Set :=
Coq <   Cons : A -> Stream A -> Stream A.
Stream is defined

Coq < CoFixpoint nats (n:nat) : Stream nat := Cons n (nats (S n)).
nats is corecursively defined

Coq < Lemma Stream_unfold :
Coq <    forall n:nat, nats n = Cons n (nats (S n)).
1 subgoal

  ============================
   forall n : nat, nats n = Cons n (nats (S n))

Coq < Proof.

Coq <   intro;
Coq <   change (nats n = match nats n with
Coq <                    | Cons x s => Cons x s
Coq <                    end).
1 subgoal

  n : nat
  ============================
   nats n = match nats n with
            | Cons x s => Cons x s
            end

Coq <   case (nats n); reflexivity.
Proof completed.

Coq < Qed.
```

```
intro; change (nats n = match nats n with
                        | Cons x s => Cons x s
                        end).
case (nats n); reflexivity.
Stream_unfold is defined
```

8 Syntax and notations

124 I do not want to type "forall" because it is too long, what can I do?

You can define your own notation for forall:

```
Notation "fa x : t, P" := (forall x:t, P) (at level 200, x ident).
```

or if your are using CoqIde you can define a pretty symbol for for all and an input method (see 152).

125 How can I define a notation for square?

You can use for instance:

```
Notation "x ^2" := (Rmult x x) (at level 20).
```

Note that you can not use: `Notation "x ² " := (Rmult x x) (at level 20).` because "²" is an iso-latin character. If you really want this kind of notation you should use UTF-8.

126 Why "no associativity" and "left associativity" at the same level does not work?

Because we relie on camlp4 for syntactical analysis and camlp4 does not really implement no associativity. By default, non associative operators are defined as right associative.

127 How can I know the associativity associated with a level?

You can do "Print Grammar constr", and decode the output from camlp4, good luck !

9 Modules

10 Ltac

128 What is Ltac?

Ltac is the tactic language for Coq. It provides the user with a high-level "toolbox" for tactic creation.

129 Is there any printing command in Ltac?

You can use the `idtac` tactic with a string argument. This string will be printed out. The same applies to the `fail` tactic

130 What is the syntax for let in Ltac?

If x_i are identifiers and e_i and *expr* are tactic expressions, then let reads:

`let` x_1`:=`e_1 `with` x_2`:=`e_2`...with` x_n`:=`e_n `in` *expr*.

Beware that if *expr* is complex (i.e. features at least a sequence) parenthesis should be added around it. For example:

```
Coq < Ltac twoIntro := let x:=intro in (x;x).
twoIntro is defined
```

131 What is the syntax for pattern matching in Ltac?

Pattern matching on a term *expr* (non-linear first order unification) with patterns p_i and tactic expressions e_i reads:

`match` *expr* `with` p_1 `=>` e_1 |p_2 `=>` e_2 `...` |p_n `=>` e_n | _ `=>` e_{n+1} `end.`

Underscore matches all terms.

132 What is the semantics for "match goal"?

The semantics of `match goal` depends on whether it returns tactics or not. The `match goal` expression matches the current goal against a series of patterns: $hyp_1 \ldots hyp_n$ |- *ccl*. It uses a first-order unification algorithm and in case of success, if the right-hand-side is an expression, it tries to type it while if the right-hand-side is a tactic, it tries to apply it. If the typing or the tactic application fails, the `match goal` tries all the possible combinations of hyp_i before dropping the branch and moving to the next one. Underscore matches all terms.

133 Why can't I use a "match goal" returning a tactic in a non tail-recursive position?

This is precisely because the semantics of `match goal` is to apply the tactic on the right as soon as a pattern unifies what is meaningful only in tail-recursive uses.

The semantics in non tail-recursive call could have been the one used for terms (i.e. fail if the tactic expression is not typable, but don't try to apply it). For uniformity of semantics though, this has been rejected.

134 How can I generate a new name?

You can use the following syntax: `let id:=fresh in ...`
For example:

```
Coq < Ltac introIdGen := let id:=fresh in intro id.
introIdGen is defined
```

11 Tactics written in Ocaml

135 Can you show me an example of a tactic written in OCaml?

You have some examples of tactics written in Ocaml in the "plugins" directory of Coq sources.

12 Case studies

136 How to prove that 2 sets are different?

You need to find a property true on one set and false on the other one. As an example we show how to prove that `bool` and `nat` are discriminable. As discrimination property we take the property to have no more than 2 elements.

```
Coq < Theorem nat_bool_discr : bool <> nat.

Coq < Proof.

Coq <   pose (discr :=
Coq <     fun X:Set =>
Coq <       ~ (forall a b:X, ~ (forall x:X, x <> a -> x <> b -> False))).

Coq <   intro Heq; assert (H: discr bool).

Coq <   intro H; apply (H true false); destruct x; auto.

Coq <   rewrite Heq in H; apply H; clear H.
```

```
Coq <   destruct a; destruct b as [|n]; intro H0; eauto.
Coq <   destruct n; [ apply (H0 2); discriminate | eauto ].
Coq < Qed.
```

137 Is there an axiom-free proof of Streicher's axiom K for the equality on nat?

Yes, because equality is decidable on nat. Here is the proof.

```
Coq < Require Import Eqdep_dec.
Coq < Require Import Peano_dec.
Coq < Theorem K_nat :
Coq <   forall (x:nat) (P:x = x -> Prop), P (refl_equal x) -> forall p:x = x, P p.
Coq < Proof.
Coq < intros; apply K_dec_set with (p := p).
Coq < apply eq_nat_dec.
Coq < assumption.
Coq < Qed.
```

Similarly, we have

```
Coq < Theorem eq_rect_eq_nat :
Coq <   forall (p:nat) (Q:nat->Type) (x:Q p) (h:p=p), x = eq_rect p Q x p h.
Coq < Proof.
Coq < intros; apply K_nat with (p := h); reflexivity.
Coq < Qed.
```

138 How to prove that two proofs of n<=m on nat are equal?

This is provable without requiring any axiom because axiom K directly holds on nat. Here is a proof using question 137.

```
Coq < Require Import Arith.
Coq < Scheme le_ind' := Induction for le Sort Prop.
Coq < Theorem le_uniqueness_proof : forall (n m : nat) (p q : n <= m), p = q.
Coq < Proof.
Coq < induction p using le_ind'; intro q.
Coq <  replace (le_n n) with
Coq <   (eq_rect _ (fun n0 => n <= n0) (le_n n) _ (refl_equal n)).
Coq <  2:reflexivity.
Coq <   generalize (refl_equal n).
Coq <     pattern n at 2 4 6 10, q; case q; [intro | intros m l e].
Coq <      rewrite <- eq_rect_eq_nat; trivial.
Coq <      contradiction (le_Sn_n m); rewrite <- e; assumption.
Coq <  replace (le_S n m p) with
Coq <   (eq_rect _ (fun n0 => n <= n0) (le_S n m p) _ (refl_equal (S m))).
Coq <  2:reflexivity.
Coq <   generalize (refl_equal (S m)).
Coq <     pattern (S m) at 1 3 4 6, q; case q; [intro Heq | intros m0 l HeqS].
```

```
Coq <      contradiction (le_Sn_n m); rewrite Heq; assumption.
Coq <      injection HeqS; intro Heq; generalize l HeqS.
Coq <       rewrite <- Heq; intros; rewrite <- eq_rect_eq_nat.
Coq <       rewrite (IHp l0); reflexivity.
Coq < Qed.
```

139 How to exploit equalities on sets

To extract information from an equality on sets, you need to find a predicate of sets satisfied by the elements of the sets. As an example, let's consider the following theorem.

```
Coq < Theorem interval_discr :
Coq <   forall m n:nat,
Coq <     {x : nat | x <= m} = {x : nat | x <= n} -> m = n.
```

We have a proof requiring the axiom of proof-irrelevance. We conjecture that proof-irrelevance can be circumvented by introducing a primitive definition of discrimination of the proofs of `{x : nat | x <= m}`.

The proof can be found in file `interval_discr.v` in this directory.

140 I have a problem of dependent elimination on proofs, how to solve it?

```
Coq < Inductive Def1 : Set := c1 : Def1.
Coq < Inductive DefProp : Def1 -> Prop :=
Coq <   c2 : forall d:Def1, DefProp d.
Coq < Inductive Comb : Set :=
Coq <   c3 : forall d:Def1, DefProp d -> Comb.
Coq < Lemma eq_comb :
Coq <   forall (d1 d1':Def1) (d2:DefProp d1) (d2':DefProp d1'),
Coq <     d1 = d1' -> c3 d1 d2 = c3 d1' d2'.
```

You need to derive the dependent elimination scheme for DefProp by hand using Scheme.

```
Coq < Scheme DefProp_elim := Induction for DefProp Sort Prop.
Coq < Lemma eq_comb :
Coq <   forall d1 d1':Def1,
Coq <     d1 = d1' ->
Coq <     forall (d2:DefProp d1) (d2':DefProp d1'), c3 d1 d2 = c3 d1' d2'.
Coq < intros.
Coq < destruct H.
Coq < destruct d2 using DefProp_elim.
Coq < destruct d2' using DefProp_elim.
Coq < reflexivity.
Coq < Qed.
```

141 And what if I want to prove the following?

```
Coq < Inductive natProp : nat -> Prop :=
Coq <   | p0 : natProp 0
Coq <   | pS : forall n:nat, natProp n -> natProp (S n).
Coq < Inductive package : Set :=
Coq <   pack : forall n:nat, natProp n -> package.
Coq < Lemma eq_pack :
Coq <  forall n n':nat,
Coq <    n = n' ->
Coq <    forall (np:natProp n) (np':natProp n'), pack n np = pack n' np'.
```

```
Coq < Scheme natProp_elim := Induction for natProp Sort Prop.

Coq < Definition pack_S : package -> package.

Coq < destruct 1.

Coq < apply (pack (S n)).

Coq < apply pS; assumption.

Coq < Defined.

Coq < Lemma eq_pack :
Coq <   forall n n':nat,
Coq <     n = n' ->
Coq <     forall (np:natProp n) (np':natProp n'), pack n np = pack n' np'.

Coq < intros n n' Heq np np'.

Coq < generalize dependent n'.

Coq < induction np using natProp_elim.

Coq < induction np' using natProp_elim; intros; auto.

Coq <  discriminate Heq.

Coq < induction np' using natProp_elim; intros; auto.

Coq <  discriminate Heq.

Coq < change (pack_S (pack n np) = pack_S (pack n0 np')).

Coq < apply (f_equal (A:=package)).

Coq < apply IHnp.

Coq < auto.

Coq < Qed.
```

13 Publishing tools

142 How can I generate some latex from my development?

You can use `coqdoc`.

143 How can I generate some HTML from my development?

You can use `coqdoc`.

144 How can I generate some dependency graph from my development?

You can use the tool `coqgraph` developped by Philippe Audebaud in 2002. This tool transforms dependancies generated by `coqdep` into 'dot' files which can be visualized using the Graphviz software (http://www.graphviz.org/).

145 How can I cite some Coq in my latex document?

You can use `coq_tex`.

146 How can I cite the Coq reference manual?

You can use this bibtex entry:

```
@Manual{Coq:manual,
  title =        {The Coq proof assistant reference manual},
  author =       {\mbox{The Coq development team}},
  note =         {Version 8.3},
  year =         {2009},
  url =          "http://coq.inria.fr"
}
```

147 Where can I publish my developments in Coq?

You can submit your developments as a user contribution to the Coq development team. This ensures its liveness along the evolution and possible changes of Coq.

You can also submit your developments to the HELM/MoWGLI repository at the University of Bologna (see `http://mowgli.cs.unibo.it`). For developments submitted in this database, it is possible to visualize the developments in natural language and execute various retrieving requests.

148 How can I read my proof in natural language?

You can submit your proof to the HELM/MoWGLI repository and use the rendering tool provided by the server (see `http://mowgli.cs.unibo.it`).

14 CoqIde

149 What is CoqIde?

CoqIde is a gtk based GUI for Coq.

150 How to enable Emacs keybindings?

Depending on your configuration, use either one of these two methods

- Insert `gtk-key-theme-name = "Emacs"` in your `.coqide-gtk2rc` file. It may be in the current dir or in `$HOME` dir. This is done by default.
- If in Gnome, run the gnome configuration editor (`gconf-editor`) and set key `gtk-key-theme` to `Emacs` in the category `desktop/gnome/interface`.

151 How to enable antialiased fonts?

Set the `GDK_USE_XFT` variable to `1`. This is by default with `Gtk >= 2.2`. If some of your fonts are not available, set `GDK_USE_XFT` to `0`.

152 How to use those Forall and Exists pretty symbols?

Thanks to the notation features in Coq, you just need to insert these lines in your Coq buffer:

```
Notation "∀ x :  t, P" := (forall x:t, P) (at level 200, x ident).
Notation "∃ x :  t, P" := (exists x:t, P) (at level 200, x ident).
```

Copy/Paste of these lines from this file will not work outside of CoqIde. You need to load a file containing these lines or to enter the ∀ using an input method (see 153). To try it just use `Require Import utf8` from inside CoqIde. To enable these notations automatically start coqide with

```
coqide -l utf8
```

In the ide subdir of Coq library, you will find a sample utf8.v with some pretty simple notations.

153 How to define an input method for non ASCII symbols?

- First solution: type <CONTROL><SHIFT>2200 to enter a forall in the script widow. 2200 is the hexadecimal code for forall in unicode charts and is encoded as in UTF-8. 2203 is for exists. See http://www.unicode.org for more codes.
- Second solution: rebind <AltGr>a to forall and <AltGr>e to exists. Under X11, you need to use something like

  ```
  xmodmap -e "keycode  24 = a A F13 F13"
  xmodmap -e "keycode  26 = e E F14 F14"
  ```

  and then to add

  ```
  bind "F13" {"insert-at-cursor" ("")}
  bind "F14" {"insert-at-cursor" ("")}
  ```

  to your "binding "text"" section in .coqiderc-gtk2rc. The strange ("") argument is the UTF-8 encoding for 0x2200. You can compute these encodings using the lablgtk2 toplevel with

  ```
  Glib.Utf8.from_unichar 0x2200;;
  ```

  Further symbols can be bound on higher Fxx keys or on even on other keys you do not need .

154 How to build a custom CoqIde with user ml code?

Use coqmktop -ide -byte m1.cmo...mi.cmo or coqmktop -ide -opt m1.cmx...mi.cmx

155 How to customize the shortcuts for menus?

Two solutions are offered:

- Edit $HOME/.coqide.keys by hand or
- Add "gtk-can-change-accels = 1" in your .coqide-gtk2rc file. Then from CoqIde, you may select a menu entry and press the desired shortcut.

156 What encoding should I use? What is this \x{iiii} in my file?

The encoding option is related to the way files are saved. Keep it as UTF-8 until it becomes important for you to exchange files with non UTF-8 aware applications. If you choose something else than UTF-8, then missing characters will be encoded by \x{....} or \x{........} where each dot is an hex. digit. The number between braces is the hexadecimal UNICODE index for the missing character.

157 How to get rid of annoying unwanted automatic templates?

Some users may experiment problems with unwanted automatic templates while using Coqide. This is due to a change in the modifiers keys available through GTK. The straightest way to get rid of the problem is to edit by hand your .coqiderc (either /home/<user>/.coqiderc under Linux, or C:\Documents and Settings\<user>\.coqiderc under Windows) and replace any occurence of MOD4 by MOD1.

15 Extraction

158 What is program extraction?

Program extraction consist in generating a program from a constructive proof.

159 Which language can I extract to?

You can extract your programs to Objective Caml and Haskell.

160 How can I extract an incomplete proof?

You can provide programs for your axioms.

16 Glossary

161 Can you explain me what an evaluable constant is?

An evaluable constant is a constant which is unfoldable.

162 What is a goal?

The goal is the statement to be proved.

163 What is a meta variable?

A meta variable in Coq represents a "hole", i.e. a part of a proof that is still unknown.

164 What is Gallina?

Gallina is the specification language of Coq. Complete documentation of this language can be found in the Reference Manual.

165 What is The Vernacular?

It is the language of commands of Gallina i.e. definitions, lemmas, ...

166 What is a dependent type?

A dependant type is a type which depends on some term. For instance "vector of size n" is a dependant type representing all the vectors of size n. Its type depends on n

167 What is a proof by reflection?

This is a proof generated by some computation which is done using the internal reduction of Coq (not using the tactic language of Coq (Ltac) nor the implementation language for Coq). An example of tactic using the reflection mechanism is the `ring` tactic. The reflection method consist in reflecting a subset of Coq language (for example the arithmetical expressions) into an object of the Coq language itself (in this case an inductive type denoting arithmetical expressions). For more information see [14, 12, 3] and the last chapter of the Coq'Art.

168 What is intuitionistic logic?

This is any logic which does not assume that "A or not A".

169 What is proof-irrelevance?

See question 33

170 What is the difference between opaque and transparent?

Opaque definitions can not be unfolded but transparent ones can.

17 Troubleshooting

171 What can I do when `Qed.` is slow?

Sometime you can use the `abstract` tactic, which makes as if you had stated some local lemma, this speeds up the typing process.

172 Why `Reset Initial.` does not work when using `coqc`?

The initial state corresponds to the state of `coqtop` when the interactive session began. It does not make sense in files to compile.

173 What can I do if I get "No more subgoals but non-instantiated existential variables"?

This means that `eauto` or `eapply` didn't instantiate an existential variable which eventually got erased by some computation. You may backtrack to the faulty occurrence of `eauto` or `eapply` and give the missing argument an explicit value. Alternatively, you can use the commands `Show Existentials.` and `Existential.` to display and instantiate the remainig existential variables.

```
Coq < Lemma example_show_existentials :  forall a b c:nat, a=b -> b=c -> a=c.
1 subgoal

  ============================
   forall a b c : nat, a = b -> b = c -> a = c

Coq < Proof.

Coq < intros.
1 subgoal

  a : nat
  b : nat
  c : nat
  H : a = b
  H0 : b = c
  ============================
   a = c

Coq < eapply trans_equal.
2 subgoals

  a : nat
  b : nat
  c : nat
  H : a = b
  H0 : b = c
  ============================
   a = ?567
subgoal 2 is:
 ?567 = c

Coq < Show Existentials.
Existential 1 =
?567 : [a : nat  b : nat  c : nat  H : a = b  H0 : b = c |- nat]

Coq < eassumption.
1 subgoal

  a : nat
  b : nat
  c : nat
  H : a = b
  H0 : b = c
```

```
  ============================
   b = c

Coq < assumption.
Proof completed.

Coq < Qed.
intros.
eapply eq_trans.
 eassumption.

 assumption.

example_show_existentials is defined
```

174 What can I do if I get "Cannot solve a second-order unification problem"?

You can help Coq using the `pattern` tactic.

175 Why does Coq tell me that {x:A|(P x)} is not convertible with (sig A P)?

This is because {x:A|P x} is a notation for `sig (fun x:A => P x)`. Since Coq does not reason up to η-conversion, this is different from `sig P`.

176 I copy-paste a term and Coq says it is not convertible to the original term. Sometimes it even says the copied term is not well-typed.

This is probably due to invisible implicit information (implicit arguments, coercions and Cases annotations) in the printed term, which is not re-synthesised from the copied-pasted term in the same way as it is in the original term.

Consider for instance `(@eq Type True True)`. This term is printed as `True=True` and re-parsed as `(@eq Prop True True)`. The two terms are not convertible (hence they fool tactics like `pattern`).

There is currently no satisfactory answer to the problem. However, the command `Set Printing All` is useful for diagnosing the problem.

Due to coercions, one may even face type-checking errors. In some rare cases, the criterion to hide coercions is a bit too loose, which may result in a typing error message if the parser is not able to find again the missing coercion.

18 Conclusion and Farewell.

177 What if my question isn't answered here?

Don't panic `:-)`. You can try the Coq manual [17] for a technical description of the prover. The Coq'Art [2] is the first book written on Coq and provides a comprehensive review of the theorem prover as well as a number of example and exercises. Finally, the tutorial [15] provides a smooth introduction to theorem proving in Coq.

References

[1] Yves bertot and Pierre Castéran. *Coq'Art*. Springer-Verlag, 2004. To appear.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS series. Springer Verlag, 2004.

[3] Samuel Boutin. Using reflection to build efficient and certified decision pro cedures. In M. Abadi and T. Ito, editors, *Proceedings of TACS'97*, volume 1281 of *LNCS*. Springer-Verlag, 1997.

[4] David Carlisle, Scott Pakin, and Alexander Holt. *The Great, Big List of LaTeX Symbols*, February 2001.

[5] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.

[6] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[7] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[8] Gilles Dowek. Théorie des types. Lecture notes, 2002.

[9] Eduardo Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. thèse d'université, Ecole Normale Supérieure de Lyon, December 1996.

[10] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the 2nd Scandinavian Logic Symposium*. North-Holland, 1970.

[11] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambrige Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[12] John Harrison. Meta theory and reflection in theorem proving:a survey and cri tique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center, 1995.

[13] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Proceedings of the meeting Twenty-five years of constructive type theory*. Oxford University Press, 1998.

[14] Doug Howe. Computation meta theory in nuprl. In E. Lusk and R. Overbeek, editors, *The Proceedings of the Ninth International Conference of Autom ated Deduction*, volume 310, pages 238–257. Springer-Verlag, 1988.

[15] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistant A Tutorial*, 2004.

[16] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[17] The Coq development team. *The Coq proof assistant reference manual*, 2010. Version 8.3.

[18] Tobias Oetiker. *The Not So Short Introduction to LaTeX2e*, January 1999.

[19] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.