

# Basic Proving with Coq

Pierre Letouzey

Fréjus, May 2015

# Coq Expressions

- ▶ In Coq, we manipulate *expressions* (a.k.a. *terms*).
- ▶ Syntactically, one big unique class of expressions.
- ▶ Some examples:

0

33

(1 + 2) \* pred 3

nat

True

False

False -> ((False /\ True) <-> True)

forall n:nat, n<>0 -> exists p, n = S p

Prop

...

# Syntactic Notations

Some of the previous expressions incorporate *notations*:

- ▶ 0 is 0
- ▶ 1 is S 0
- ▶ 33 is S (S (S ... 0))
- ▶ 1+2 is plus 1 2
- ▶  $A \leftrightarrow B$  is iff A B  
which is *defined* as  $(A \rightarrow B) / \backslash (B \rightarrow A)$
- ▶  $n \neq 0$  is  $\sim(n=0)$
- ▶ = is eq
- ▶  $\sim A$  is not A,  
which is *defined* as  $A \rightarrow \text{False}$
- ▶ even  $\rightarrow$  is a special case of forall

# Priorities

Be careful with notation precedence:  
what you write might not be what Coq reads...

- ▶  $A \rightarrow B \rightarrow C$  is  $A \rightarrow (B \rightarrow C)$
- ▶  $A /\backslash B /\backslash C$  is  $A /\backslash (B /\backslash C)$
- ▶  $A \rightarrow B \leftrightarrow C$  is now  $A \rightarrow (B \leftrightarrow C)$  in Coq 8.5  
but used to be  $(A \rightarrow B) \leftrightarrow C$  in earlier versions
- ▶ ...

# Priorities

Be careful with notation precedence:  
what you write might not be what Coq reads...

- ▶  $A \rightarrow B \rightarrow C$  is  $A \rightarrow (B \rightarrow C)$
- ▶  $A /\backslash B /\backslash C$  is  $A /\backslash (B /\backslash C)$
- ▶  $A \rightarrow B \leftrightarrow C$  is now  $A \rightarrow (B \leftrightarrow C)$  in Coq 8.5  
but used to be  $(A \rightarrow B) \leftrightarrow C$  in earlier versions
- ▶ ...

If in doubt:

- ▶ try putting parenthesis !
- ▶ in CoqIDE, experiment with View / Display notation.
- ▶ in Emacs: (Set/Unset) Printing Notations.

# Types

- ▶ Coq provides a rich type system
- ▶ All correct expressions are well-typed

0 : nat

33 : nat

1 + 2 \* pred 3 : nat

nat : Set

Set : Type

Type : Type

# Types

- ▶ Coq provides a rich type system
- ▶ All correct expressions are well-typed

0 : nat

33 : nat

1 + 2 \* pred 3 : nat

nat : Set

Set : Type

Type : Type

True : Prop

False : Prop

False -> ((False /\ True) <-> True) : Prop

forall n:nat, n<>0 -> exists p, n = S p : Prop

Prop : Type

# Classification of terms

Thanks to typing, we can now classify Coq terms:

- ▶ Set, Prop and Type are **sorts** or **universes** i.e. types of types.
- ▶ A **type** is anything typed by a sort.  
Examples : `nat`, `(False->True)`, ...
- ▶ In particular, a **data-type** has type `Set`.
- ▶ Similarly, a **proposition** is anything of type `Prop`.
- ▶ A **proof** of a proposition `A` is simply a term of type `A`
- ▶ A type may be inhabited (`0:nat`, `I:True`) or empty (no *closed* proof of `False`)



# Classification of terms

Two more notions:

- ▶ A **predicate** is a function returning propositions.  
Its type will hence look like  $X \rightarrow \text{Prop}$  for some domain  $X$ .
- ▶ A **relation** is a predicate with two arguments:  
 $R : X \rightarrow Y \rightarrow \text{Prop}$ , or frequently  $R : X \rightarrow X \rightarrow \text{Prop}$ .  
Examples: equality, order, ...

# Main Commands: Extending the Environment

- ▶ **Definition**: give a name to some expression.
- ▶ **Fixpoint**: same, for recursive functions.
- ▶ **Lemma / Theorem**:  
interactive build of a proof term of a given type.
- ▶ **Axiom / Parameter**:  
assume we have a term of a given type (be careful!).
- ▶ **Inductive**: add a new inductive type to the system.

# Main Commands: Queries

- ▶ `Print iff.` (or `Print "iff".`)
- ▶ `Locate "+".`
- ▶ `Check True->False.`
- ▶ `Compute 1+2*3.`
- ▶ `Search "+".` This was `SearchAbout` before Coq 8.5.

# A Basic Proof

Lemma obvious : forall A:Prop, A\ / A -> A.

Proof.

intro A. intro H.

destruct H.

- assumption.

- assumption.

Qed.

# A Basic Proof

```
Lemma obvious : forall A:Prop, A\ / A -> A.
```

```
Proof.
```

```
  intro A. intro H.
```

```
  destruct H.
```

```
  - assumption.
```

```
  - assumption.
```

```
Qed.
```

All the actions made between Proof and Qed are called *tactics*.

# A Proof Context

Before the destruct:

```
1 subgoal
```

```
A : Prop
```

```
H : A \ / A
```

```
----- (1/1)
```

```
A
```

# A Proof Context

After the destruct:

2 subgoals

A : Prop

H : A

----- (1/2)

A

----- (2/2)

A

# A Proof Context

After the destruct:

2 subgoals

$A : \text{Prop}$

$H : A$

----- (1/2)

$A$

----- (2/2)

$A$

Note the distinction between  $A : \text{Prop}$  and  $H : A$  !



# Proof Structure

You may give a tree-like structure to your proof via *bullets* - + \*.

Proof.

```
tactic_with_two_subgoals.
```

```
- tactic_for_subgoal_1.
```

```
  + foo.
```

```
  + bar.
```

```
- etc_for_subgoal_2.
```

Qed.

# Proof Structure

You could also delimit a zone via `{ }`.

Proof.

```
...
assert (some_intermediate_statement).
{
  its_proof.
}
```

...  
Qed.

# Basic Tactics: Misc

- ▶ `assumption`
- ▶ `assert` : detour (or “cut”) via an intermediate statement.
- ▶ `revert` : the opposite of `intro`.
- ▶ `unfold` : expand a definition name to its body.
- ▶ `simpl` : do some computations.

## Basic Tactics: Connectors

	introduction (in goal)	elimination (in hypothesis H)
core connectors : $\forall, \rightarrow$	<code>intro(s)</code>	<code>apply H</code>
defined connectors : $\perp, \wedge, \vee, \exists$	<code>constructor</code>	<code>destruct H</code>

# Basic Tactics: Connectors

	introduction (in goal)	elimination (in hypothesis H)
core connectors : $\forall, \rightarrow$	<code>intro(s)</code>	<code>apply H</code>
defined connectors : $\perp, \wedge, \vee, \exists$	<code>constructor</code>	<code>destruct H</code>

In fact, instead of `constructor`, some dedicated introduction tactics:

$\perp$	-
$\wedge$	<code>split</code>
$\vee$	<code>left, right</code>
$\exists$	<code>exists ...</code>

## Basic Tactics: Connectors

	introduction (in goal)	elimination (in hypothesis H)
core connectors : $\forall, \rightarrow$	<code>intro(s)</code>	<code>apply H</code>
defined connectors : $\perp, \wedge, \vee, \exists$	<code>constructor</code>	<code>destruct H</code>

In fact, instead of `constructor`, some dedicated introduction tactics:

$\perp$	-
$\wedge$	<code>split</code>
$\vee$	<code>left, right</code>
$\exists$	<code>exists ...</code>

NB: for  $\leftrightarrow$ , see it as a conjunction of implications.

# Basic Tactics: Negation

Remember that  $\sim A$  is  $A \rightarrow \text{False}$ .

- ▶ To explicit this: `unfold not in *`.  
Not mandatory, mostly for helping the user.
- ▶ To introduce a negation: `intro`.  
NB: `intros` does nothing on  $\sim$ , unless given enough names.
- ▶ To eliminate a negation, we play with the final `False`,  
hence `destruct`.

Two more related tactics:

- ▶ `exfalse` : replace the goal by `False`.
- ▶ `contradict H` : when both `H` and the goal are negated.

# Basic Tactics: Equality

- ▶ `reflexivity`.
- ▶ `symmetry`.
- ▶ `transitivity` ....
- ▶ `rewrite` ....
- ▶ `f_equal` : prove  $f\ x = f\ y$  via  $x = y$ .



# Basic Tactics: Automation

General-purpose automatic tactics:

- ▶ `trivial` : direct consequences of hypothesis.
- ▶ `auto` : goal-directed prolog-like proof search.
- ▶  `eauto` : same with possible unresolved holes ?.
- ▶ `intuition` : propositional solver.
- ▶ `firstorder` : first-order proof search.

# Basic Tactics: Automation

Many more domain-specific automatic tactics:

- ▶ `congruence` : saturation of equalities.
- ▶ `omega` : Presburger arithmetic ( $=$ ,  $<$ ,  $+$ ,  $-$ , but no  $*$ ).
- ▶ `ring` : algebraic rules of rings ( $=$ ,  $+$ ,  $-$ ,  $*$ ).
- ▶ `field` : algebraic rules of fields ( $=$ ,  $+$ ,  $-$ ,  $*$ ,  $/$ ).
- ▶ ...

# Tactic combinators

- ▶ sequence : `tac1 ; tac2`
- ▶ conditional : `try tac, tac1 || tac2`
- ▶ iteration : `do n tac, repeat tac`

In fact, a whole programming language of tactics (Ltac).

# Coq Logic is Intuitionistic

With the core logic of Coq, no proofs of:

- ▶ Excluded middle :  $A \vee \sim A$
- ▶ Double negation :  $\sim\sim A \rightarrow A$
- ▶ Pierce's Law
- ▶ ...

If you want to use classical logic, add an axiom.

For instance :

```
Require Import Classical.  
Print classic.
```

# Coq Logic is no Boolean Logic

Do not confuse `Prop` and `bool` !

- ▶ `bool` is a datatype containing two values `true` and `false`.
- ▶ No other closed reduced expressions of type `bool`!
- ▶ `Prop` is the universe of all propositions, including `True` and `False` and many others.
- ▶ These propositions may be infinitary and/or non-decidable.
- ▶ Conversely `bool` is meant for programming (for instance equality tests).

# Interactions between bool and Prop

- ▶ `true` and `false` are no types, they aren't statements per se :  
Lemma oups : true
- ▶ To place boolean values in a statement, use equalities:  
Lemma ok : forall b:bool, orb b (negb b) = true.
- ▶ Conversely, no generic way to turn a proposition into a corresponding boolean.
- ▶ When feasible, this could be a proof method.  
For instance, relate a order relation with a comparison test, and do proof by computation.

Showtime!