Introduction to Coq

Yves Bertot

August 2005

Running Coq

- the plain command : `coqtop`
  - use your favorite line-editor,
- the compilation command : `coqc`
- the interactive environment : `coqide`
- with the Emacs environment : open a file with suffix ".v"
- Also Pcoq developed at Sophia
- All commands terminate with a period at the end of a line.

The Check command

- Useful first step: load collections of known facts and functions.
  ```
  Require Import Arith.  Require Import ArithRing.
  Require Import Omega.
  ```
- First know how to construct well-formed terms.
  ```
  Check 3.
  ```
  3 : nat
  ```
  Check plus.
  ```
  plus : nat−>nat−>nat
  ```
  Check (nat->(nat->nat)).
  ```
  nat−>nat−>nat : Set
  ```
  Check (plus 3).
  ```
  plus 3 : nat−>nat

Basic constructs

- abstractions, applications.

  `Check (fun x => plus x x).`

  fun x:nat => x + x : nat->nat

- product types.

  `Check (fun (A:Set)(x:A)=>x).`

  fun (A:Set)(x:A) => x : forall A:Set, A->A

- Common notations.

  `Check (3=4).`

  3=4 : Prop

  `Check (fun (A:Set)(x:A)=>(3,x)).`

  *fun (A:Set)(x:A)=>(3,x) : forall A:Set, A ->nat*A*

Basic constructs (continued)

- Logical statements.
  ```
  Check (forall x y, x <= y -> y <= x -> x = y).
  ```
  forall x y:nat, x <= y -> y <= x -> x = y : Prop
- proofs.
  ```
  Check le_S.
  ```
  le_S : forall n m:nat, n <= m -> n <= S m
  ```
  Check (le_S 3 3).
  ```
  le_S 3 3 : 3 <= 3 -> 3 <= 4
  ```
  Check le_n.
  ```
  le_n : forall n:nat, n <= n
  ```
  Check (le_S 3 3 (le_n 3))
  ```
  le_S 3 3 (le_n 3) : 3 <= 4

Logical notations

- conjunction, disjunction, negation.
  `Check (forall A B, A /\ (B \/ ~ A)).`
  forall A B:Prop, A /\ (B \/ ~ A) : Prop
- Well-formed statements are not always true or provable.
- Existential quantification.
  `Check (exists x:nat, x = 3).`
  exists x:nat, x = 3 : Prop

Notations

- Know what function is hidden behind a notation:
  `Locate "_ + _".`
  Notation Scope
  "x + y" := sum x y : type_scope
  "x + y" := plus x y : nat_scope
  (default interpretation)

Computing

- Unlike Haskell, ML, or OCaml, values are not computed by default
  `Check (plus 3 4).`
  3+4:nat
- A command to require computation.
  `Eval compute in ((3+4)*5).`
  = 35 : nat
- A proposition is not a boolean value.
  `Eval compute in ((3+4)*5=61).`
  = 35=61:Prop
- Fast computation is not the main concern.

Definitions

- Define an object by providing a name and a value.
  `Definition ex1 := fun x => x + 3.`
  ex1 is defined
- Special notation for functions.
  `Definition ex2 (x:nat) := x + 3.`
  ex2 is defined
- See the value associated to definitions.
  `Print ex1.`
  ex1 = fun x : nat => x + 3 : nat -> nat
  Argument scope is [nat_scope]
  `Print ex2.`
  ex2 = fun x : nat => x + 3 : nat -> nat
  Argument scope is [nat_scope]

Sections

- Sections make it possible to have a local context.

```
Section sectA.
  Variable A:Set.
```

A is assumed

```
  Variables (x:A)(P:A->Prop)(R:A->A->Prop).
```

x is assumed

...

```
  Hypothesis Hyp1 :  forall x y, R x y -> P y.
```

...

```
  Check (Hyp1 x x).
```

Hyp x x : R x x −> P x

Sections (continued)

- Definitions can use local variables.

  `Definition ex3 (z:A) := Hyp1 z z.`

  `Print ex3.`

  ex3 = fun z:A => Hyp1 z z : forall z:A, R z z -> P z

- Defined values change at closing time.

  `End sectA.`

  ex3 is discharged.

  `Print ex3.`

  ex3 =
  fun (A:Set)(P:A->Prop)(R:A->A->Prop)
  (Hyp1:forall x y:A,Rxy->P y)(z :A)=>Hyp1 z z
  : forall(A:Set)(P:A->Prop)(R:A->A->Prop),
  (forall x y:A, R x y->P y)->forall z:A, R z z->P z

Parameters and Axioms

- Declaring variables and Hypotheses outside sections.
- Proofs will never be required for axioms.
- Make it possible to extend the logic.
- Make partial experiments easier.
- Beware of inconsistency!

Goal directed proof

- Finding inhabitants in types.
- Recursive technique:
  - observe a type in a given context.
  - find the shape of a term with holes with this type.
  - restart recursively with the new holes in new contexts.
- The commands to fill holes are called *tactics*.
  - arrow or forall types are function types and can be filled by an abstraction: the context increases (tactic `intro`).
  - For other types one may use existing functions or theorems (tactics `exact`, `apply`).
  - special tactics take care of classes of constructs (tactics `elim`, `split`, `exist`, `rewrite`, `omega`, `ring`).
- When no hole remains, the proof needs to be saved.

Example proof

```
Theorem example2 :  forall a b:Prop, a/\ b -> b /\ a.
```

1 subgoal

============================

forall a b : Prop, a /\ b −> b /\ a

```
Proof.
intros a b H.
```

1 subgoal

a : Prop

b : Prop

H : a /\ b

============================

b /\ a

Example proof (continued)

```
split.
```

2 subgoals

...

H : a /\ b

============================

b

subgoal 2 is:

a

Example proof (continued)

```
elim H.
...
  H : a /\ b
  ============================
   a -> b -> b
...
intros H1 H2.
...
  H1 : a
  H2 : b
  ============================
   b
```

Example proof (continued)

```
 exact b
```

1 subgoal ...

```
  ============================
```

a

```
intuition.
```

Proof completed.

```
Qed.
```

intros a b H.

...

intuition.

example2 is defined

Second example

```
Theorem square_lt :  forall n m, n < m -> n*n < m*m.
Proof.
intros n m H.
SearchPattern (_*_ < _*_).
```

mult_S_lt_compat_l:

*forall n m p : nat, m < p −> S n * m < S n * p*

mult_lt_compat_r:

*forall n m p : nat, n < m −> 0 < p −> n * p < m * p*

```
Check le_lt_trans.
```

le_lt_trans : forall n m p : nat, n <= m −> m < p −> n < p

Second example (continued)

```
apply le_lt_trans with (n * m).
...
```

H : n < m

============================

*n * n <= n * m*

```
...
SearchPattern (_ * _ <= _ * _).
```

*mult_le_compat_l: forall n m p : nat, n <= m -> p * n <= p * m*
*mult_le_compat_r: forall n m p : nat, n <= m -> n * p <= m * p*

```
...
Check lt_le_weak.
```

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

Second example (continued)

```
apply mult_le_compat_l; apply lt_le_weak; exact H.
...
```

H : n < m

==============================

*n * m < m * m*

```
apply mult_lt_compat_r.
```

2 subgoals

```
...
```

H : n < m

==============================

n < m

subgoal 2 is:

0 < m

```
assumption.
```

Second example (continued)

```
Show.
```

$H : n < m$

$==========================$

$0 < m$

```
omega.
```

Proof completed.

```
Qed.
```

Proofs : a synopsis

	$\Rightarrow$	$\forall$	$\wedge$	$\vee$	$\exists$
Hypothesis	`apply`	`apply`	`elim`	`elim`	`elim`
goal	`intros`	`intros`	`split`	`left` or `right`	`exists` v
	$\neg$	$=$			
Hypothesis	`elim`	`rewrite`			
goal	`intro`	`reflexivity`			

- Automatic tactics: `auto`, `auto with` *database*, `intuition`, `omega`, `ring`, `fourier`, `field`.
- Possibility to define your own tactics: `Ltac`.

Automatic tactics

- `intuition` Automatic proofs for 1st order intuitionnistic logic,
- `omega` Presburger arithmetic on types `nat` and `Z`,
- `ring` Polynomial equalities on types `Z` and `nat` (no subtraction for the latter)
- `fourier` Inequations between linear formulas in `R`,
- `field` Equations between fractional expressions in `R`.

Forward reasoning

- apply only supports backward reasoning (it does not implement ∀-elimination or -¿-elimination),
- Problem "I have `H: forall x, P x`" how can I add `P a` to the context"
  - `assert (H2 : P a)`, prove this by `apply H` and proceed,
  - alternatively `generalize (H a); intros H2`.
- Problem "I have `H: A -> B`" how can add `B` to the context and have an extra goal to prove `A`.
  - Use assert again,
  - alternatively use "lapply H;[intros H2 — idtac]".

Inductive types

- Inductive types extend the recursive (algebraic) data-types of Haskell, ML, ...
- An inductive type definition provides three kinds of elements:
  - A type (or a family of types),
  - Constructors,
  - A computation process (case-analysis and recursion),
  - A proof by induction principle.

```
Inductive bin :  Set :=
  L : bin
| N : bin -> bin -> bin.
```

Computation process

- Pattern-matching and structural recursion.

```
Fixpoint size (t1:bin):  nat :=
  match t1 with
    L => 1
  | N t1 t2 => 1 + size t1 + size t2
  end.
Fixpoint flatten_aux (t1 t2:bin) {struct t1} :  bin
:=
  match t1 with
    L => N L t2
  | N t'1 t'2 =>
    flatten_aux t'1 (flatten_aux t'2 t2)
  end.
```

Recursive definition (continued)

```
Fixpoint flatten (t:bin) :  bin :=
  match t with
    L => L
  | N t1 t2 => flatten_aux t1 (flatten t2)
  end.
```

Proof by induction principle

- Quantification over a predicate on the inductive type,
- Premises for all the cases represented by the constructors,
- Induction hypotheses for the subterms in the type.

Proof by induction principle

- Quantification over a predicate on the inductive type,
- Premises for all the cases represented by the constructors,
- Induction hypotheses for the subterms in the type.

```
Check bin_ind.
```

bin_ind : forall P:bin->Prop,
P L ->
(forall b:bin, P b -> forall b0:bin, P b0 -> P (N b b0)) ->
forall b : bin, P b

Proof by induction principle

- Quantification over a predicate on the inductive type,
- Premises for all the cases represented by the constructors,
- Induction hypotheses for the subterms in the type.

```
Check bin_ind.
```

bin_ind : forall P:bin->Prop,
P L ->
(forall b:bin, P b -> forall b0:bin, P b0 -> P (N b b0)) ->
forall b : bin, P b

- The tactic `elim` uses this theorem automatically.

Example proof by induction

```
Theorem forall_aux_size :
 forall t1 t,
    size(flatten_aux t1 t) = size t1+size t+1.
Proof.
 intros t1; elim t1.
...
```

$$\overline{}$$

forall t : bin, size (flatten_aux L t) = size L + size t + 1

subgoal 2 is:

...

size (flatten_aux (N b b0) t) = size (N b b0)+size t+1

Proof by induction (continued)

```
simpl.
...
```

============================

forall t : bin, S (S (size t)) = S (size t + 1)

```
...
intros; ring_nat.
```

Proof by induction (continued)

```
...
  ==============================
   forall t : bin, size (flatten_aux L t) = size L + size t + 1
```

```
simpl.
...
==============================
forall t : bin, S (S (size t)) = S (size t + 1)
...
intros; ring_nat.
```

- This goal is solved.

Proof by induction (continued)

```
  ============================
   forall b : bin,
   (forall t : bin, size (flatten_aux b t) = size b+size t+1) ->
   forall b0 : bin,
   (forall t : bin, size (flatten_aux b0 t) = size b0+size t+1) ->
   forall t : bin, size (flatten_aux (N b b0) t) =
                 size (N b b0)+size t+1
intros b Hrecb c Hrec t; simpl.
...
  ============================
   size(flatten_aux b (flatten_aux c t))=S(size b+size c+size t+1)
```

Proof by induction (continued)

```
...
  Hrec : forall t : bin, size(flatten_aux c t) = size c+size t+1
  t : bin
  ============================
   size(flatten_aux b (flatten_aux c t))=S(size b+size c+size t+1)
rewrite Hrecb.
...
  ============================
   size b+size(flatten_aux c t)+1 = S(size b+size c+size t+1)
rewrite Hrec; ring_nat.
Qed.
```

Inductive type and equality

- For inductive types of type `Set`, `Type`,
  - Constructors are distinguishable (strong elimination),
  - Constructors are injective.
  - Tactics: `discriminate` and `injection`.
- Not for inductive type of type `Prop`, bad interaction with impredicativity.

Discriminate example

```
Theorem discriminate_example :  forall t1 t2, L = N
t1 t2 -> 2 = 3.
...
intros t1 t2 H.
...
```

H : L = N t1 t2

============================

2 = 3

```
discriminate H.
```

Proof completed.

- With no argument, discriminate finds an hypothesis that fits.

Injection example

```
Theorem injection_example :
     forall t1 t2 t3, N t1 t2 = N t3 t3 -> t1 = t2.
...
intros t1 t2 t3 H.
```

H : N t1 t2 = N t3 t3
============================
t1 = t2

```
...
injection H.
...
```

============================
t2 = t3 −> t1 = t3 −> t1 = t2

```
intros H1 H2; rewrite H1; auto.
```

Proof completed.

Usual inductive data-types in Coq

- Most number types are inductive types,
  - Natural numbers *à la Peano*, the induction principle coincides with mathematical induction, `nat`,
  - Strictly positive integers as sequences of bits, `positive`,
  - Integers, as a three-branch disjoint sum, `Z`,
  - Strictly positive rational numbers can also be represented as an inductive type.
- Data structures: lists, binary search trees, finite sets.

Inductive propositions

- Dependent inductive types of sort `Prop`,
- The types of the constructors are logical statements,
- The induction principle is a simplified,
- Easy to understand as a minimal property for which the constructor hold.

Inductive proposition example

```
Inductive even :  nat -> Prop :=
  even0 :  even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

- `even` is a function that returns a type,
- When `x` varies, `even x` intuitively has one or zero element.

Simplified induction principle

```
Check even_ind.
```

even_ind : forall P : nat −> Prop,
P 0 −>
(forall x : nat, even x −> P x −> P (S (S x))) −>
forall n : nat, even n −> P n

- quantification over a predicate on the potential arguments of the inductive type,
- No universal quantification over elements of the type, only implication (*proof irrelevance*).

Example proof by induction on a proposition

```
Theorem even_mult :  forall x, even x -> exists y, x
= 2*y.
intros x H; elim H.
...
```

==============================

*exists y : nat, 0 = 2 * y*

subgoal 2 is:

forall x0 : nat,

*even x0 -> (exists y : nat, x0 = 2 * y) ->*

*exists y : nat, S (S x0) = 2 * y*

Proof by induction on a proposition (continued)

```
exists 0; ring_nat.
intros x0 Hevenx0 IHx.
...
```

*IHx : exists y : nat, x0 = 2 * y*
============================
*exists y : nat, S (S x0) = 2 * y*

```
destruct IHx as [y Heq]; rewrite Heq.
(*alternative to elim IHx; intros y Heq; rewrite Heq
*)
exists (S y); ring_nat.
Qed.
```

Inversion

- sometimes assumptions are false because no constructor proves them,
- sometimes the hypothesis of a constructor have to be tree because only this constructor could have been used.

Example inversion

```
 not_even_1 :  ~even 1.
```

intros even1. ...

even1 : even 1

==============================

False

```
inversion even1.
Qed.
```

Usual inductive propositions in Coq

- The order <= on natural numbers (type `le`).
- The logical connectives.
- The accessibility predicate with respect to a binary relation,

Logical connectives as inductive propositions

- Parallel with usual present of logic in sequent style,
- Right introduction rules are replaced by constructors,
- Left introduction is automatically given by the induction principle.

Inductive view of False

- No right introduction rule: no constructor.

```
Inductive False :  Prop := .
Check False_ind.
```

False_ind

: forall P : Prop, False −> P

Inductive view of and

- one constructor,
- two left introduction rules, but can be modeled as just one.

```
Print and.
```

Inductive and (A : Prop) (B : Prop) : Prop :=
    conj : A −> B −> A /\ B

```
Check and_ind.
```

and_ind
    : forall A B P : Prop, (A −> B −> P) −> A /\ B −> P

Inductive view of `or`

```
Print or.
```

Inductive or (A : Prop) (B : Prop) : Prop :=
or_introl : A −> A \/ B | or_intror : B −> A \/ B

```
Check or_ind.
```

or_ind : forall A B P : Prop, (A−>P)−>(B−>P)−>A \/ B−>P

Inductive view of exists

```
Print ex.
```

Inductive ex (A : Type) (P : A −> Prop) : Prop :=
ex_intro : forall x : A, P x −> ex P

```
Check ex_ind.
```

ex_ind : forall (A : Type) (P : A −> Prop) (P0 : Prop),
(forall x : A, P x −> P0) −> ex P −> P0

Inductive view of eq

```
Print eq.
```

Inductive eq (A : Type) (x : A) : A −> Prop :=
  refl_equal : x = x

```
Check eq_ind.
```

eq_ind : forall (A : Type) (x : A) (P : A −> Prop),
    P x −> forall y : A, x = y −> P y

Dependently typed pattern-matching

- Well-formed pattern-matching constructs where each branch has a different type,
- Still a constraint of being well-typed,
- Determine the type of the whole expression,
- Verify that each branch is well-typed,
- Dependence on the matched expression.

Syntax of dependently typed pattern-matching

- `match` e `as` x `return` T `with`
  p_1 `=>` v_1
  `|` p_2 `=>` v_2
  ...
  `end`
- The whole expression has type $T[e/x]$,
- Each value v_1 must have type $T[p_1/x]$.

Example of dependently typed programming

```
Print nat.
```

Inductive nat : Set := O : nat | S : nat −> nat

```
Fixpoint nat_ind (P:nat->Prop)(v0:P 0)
     (f:forall n, P n -> P (S n))
     (n:nat) {struct n} :  P n :=
   match n return P n with
     O => v0
   | S p => f p (nat_ind P v0 f p)
   end.
```

- Dependently-typed programming for logical purposes

Dependent pattern-matching with dependent inductive types

```
Fixpoint even_ind2 (P:nat->Prop)(v0:P 0)
     (f:forall n, P n -> P (S (S n)))
     (n:nat) (h:even n) {struct h} :  P n :=
     match h in even x return P x with
       even0 => v0
     | evenS a h' => f a (even_ind2 P v0 f a h')
     end.
```