A Coq tutorial for confirmed Proof system users

Yves Bertot

August 2008

Presentation

- Get it at `http://coq.inria.fr`
- pre-compiled binaries for Linux, Windows, Mac OS,
- commands: `coqtop` or `coqide` (user interface),
- Also user interface based on Proof General,
- Historical overview and developers: refer to the introduction of the reference manual.

Libraries and Uses

- Numbers (nat, Z, rationals, real) , Strings, Lists, Finite Sets and maps,
- User contributions
  - Constructive mathematics (R. U., Nijmegen),
  - Electronic banking protocols (Trusted Logic, Gemalto),
  - Programming languages semantics and tools (Compcert, Möbius, Princeton, U. Penn, U. C. Berkeley),
  - Large prime number certification, elliptic curves,
  - Geometry: elements, algorithms,
- A book with many examples and exercises: the Coq'Art (Springer, 2004), `http://www.labri.fr/Perso/~casteran/CoqArt`

A programming language

- Typed lambda calculus with inductive and co-inductive data-types,
- Pattern-matching,
- Dependent types,
- No side-effect, no exception: pure functional programming,
- Recursion safeguard: structural recursion,
- Special notations for numbers and lists.

A few inductive types

- `Inductive nat : Set := O | S (n:nat).`
- `Inductive bool : Set : true | false.`
- Obtained when typing `Require Import ZArith`:
  `Inductive positive : Set :=`
  `xI (p:positive) | xO (p:positive) | xH.`
- `Inductive Z : Set :=`
  `ZO | Zpos (p:positive) | Zneg (p:positive).`
- Obtained when typing `Require Import List`:
  `Inductive list (A:Type) : Type :=`
  `nil | cons (a:A)(l:list A).`

Recursive definitions and pattern-matching

- The `Fixpoint` command,

```
Fixpoint app (A:Type) (l1 l2:list A) : list A :=
match l1 with
   nil => l2
| cons a l1' => cons a (app l1' l2)
end.
```

- reminiscent of Ocaml's pattern-matching (using => to separate sides of rules),
- Recursive calls only on variables out of pattern-matching,
  - for one argument that can be guessed by Coq,
- Structural recursion,
- More forms of recursion, to be studied later.

Example recursive function

- The following function computes whether the input is even
- Patterns need not be simple,
- They need to be linear (or will be read as such),

```
Fixpoint e_b (x:nat) : bool :=
 match x with
  S (S x) => e_b x
| O => true
| _ => false
end.
```

Dependent types

- A distinguishing feature.
- Functions may return results in different types,
- The result type is chosen from the input (with a function, too),

```
Definition T (b:bool) : Type := if b then nat else bool.

Definition f (b:bool) : T b :=
  if b return T b then 0 else true.
```

- New notation for types: `f :  forall b:bool, T b`

Dependency in inductive types

- Several extensions:
  - Add dependency only in constructors: dependent records,
  - Define families of types,
  - Mix the two aspects.

Dependent records

```
Inductive bt : Type := Cbt b (v:T b).
```

- The following returns the second component of a `bt` pair, or its even value when this second component is a number.

```
Definition g(c:bt) : bool :=
  let (b, v) := c in
  (if b return T b -> bool
   then fun v:nat => e_b v
   else fun v:bool => v) v.
```

Inductive families

- An inductive definition may not construct one type but a family of types,
- Examples : `list : Type -> Type`, `vector : Type -> nat -> Type`

```
Inductive list (A:Type) : Type :=
  nil | cons (a:A) (l:list A).

Inductive vector (A:Type) : nat -> Type :=
  Vnil : vector A 0
| Vcons : forall n, A -> Vector A n -> Vector A (S n).
```

- Beware: even simple functions on type `vector` are a challenge to write.
- Better representation of vectors described later.

Explicit polymorphism and implicit parameters

- In usual functional programming languages, polymorphism is implicit,
- type variables are universally quantified by default,
- Here polymorphism is explicit:
  `cons :  forall A:Type, A -> list A -> list A`
- The first argument of `cons` is declared *implicit*.
- Should not be written by the user, but guessed at type-verification time,
- The same for `nil`, but type information guessed from the context,
- Implicit argument mechanism is overriden by writing `@cons`, `@nil`,
- Notations: `a::tl` is `cons a tl`, also `@cons _ a tl`.

Logic and proofs

- Programming and constructing proofs are the same activity in Coq,
- The programming language is used directly to represent logical statements,
- Some types are reserved for logical reasoning,
- Because of explicit typing, terms contain redundant information,
- A tactic language is provided to avoid constructing terms by hand.

The Curry-Howard isomorphism

- Read arrows as implications,
- Read dependent types as universal quantifications,
- Read types as logical formula,
- Read "t has type T" as "t is a proof of T",
- Read some inductive types families as logical connectives,
- Functions are total, type `A -> B` can be read as "if you have a proof of A, you can construct a proof of B",
- Reserve a collection of types (a *sort*) for logical propositions `Prop`.

Logical connectives

```
Inductive and (A B:Prop) : Prop :=
  conj : A -> B -> and A B.

Definition proj1 (A B:Prop) (c: and A B) : A :=
  match c with conj p1 _ => p1 end.
```

- Notation : `A /\ B` for `and A B`,
- The same for `\/` (disjunction), `False`, `~` (negation),

Inductive representation of order

```
Inductive le (n:nat) : nat -> Prop :=
  le_n : le n n
| le_S : forall m, le n m -> le n (S m).

Fixpoint le_ind (n:nat)(P:nat->Prop)
 (Hn : P n)(HS : forall m, le n m -> P m -> P (S m))
 (p : nat)(np : le n p) : P p :=
 match np in le _ x return P x with
   le_n => Hn
 | le_S m nm => HS m nm (le_ind n P Hn HS m nm)
 end.
```

Inductive representation of order

```
Inductive le (n:nat) : nat -> Prop :=
  le_n : le n n
| le_S : forall m, le n m -> le n (S m).

Fixpoint le_ind (n:nat)(P:nat->Prop)
 (Hn : P n)(HS : forall m, le n m -> P m -> P (S m))
 (m:nat)(h:le n m) : P n :=
 match h in le _ x return P x with
   le_n => Hn : P n
 | le_S m nm => HS m nm (le_ind n Hn Hs m) : P (S m)
 end.
```

Inductive representation of equality

```
Inductive eq (A:Type)(x:A) : A -> Prop :=
  refl_equal : eq A x x.

Notation "x = y" := eq _ x y.

Definition eq_ind :
  forall (A:Type)(P:A->Prop)(x:A), P x ->
  forall y, x = y -> P y :=
fun A P x px y q =>
  match q in @eq _ _ y return P y with
    refl_equal => px  : P x
  end  : P y.
```

Slides from here to section on co-recursion were not presented at the conference.

Classical and constructive logic

- Interpretation of arrows and universal quantification does not give provability for all formulas provable with truth tables,
- Example: Peirce's law `((A -> B) -> A) -> A`,
- Inductive connectives in their current form do not extend the logic,
- This logic is *constructive*,
- Advantage: constructive proofs contain algorithms,
- No logical inconsistency in using classical logic (by admitting *excluded middle*, `∀ P, P \/ ¬ P`, as in other systems),

Classical logic

- Separation of `Prop` and `Type` allows for this,
- The barrier is "weak elimination": no case analysis on `Prop` inductive types to obtain `Type` values,
- `exists x, P x` means *there is an* `x` *satisfying* `P`
  `{x | P x}` means *a pair of an* `x` *and a certificate that it satisfies* `P`,
- In a constructive setting, the latter is existential quantification,
- Even in presence of excluded middle (for `Prop` types), values of the form `{x | P x}` can always be computed,
- Some other classical axioms may remove this property (axiom of definite description, axiom of choice).

Proofs: the Coq toplevel

- Basic categories of commands:
  - Definitions: `Definition`, `Fixpoint`, `Inductive`,
  - Queries: `Search`, `Check`, `Locate`,
  - Goal handling: `Theorem`, `Goal`, `Lemma`, `Qed`
  - Tactics (possibly preceded by a goal number), `elim`, `intro`, `apply`,
- Advanced features:
  - Notations and scopes,
  - General recursion,
  - Module system,
  - "Program" presentation of terms,
  - Canonical structures and type classes.

An example of proof

```
Lemma ex1 : forall a b:Prop, a /\ b -> b /\ a.
1 subgoal

  ============================
   forall a b : Prop, a /\ b -> b /\ a

ex1 < intros a b c.
1 subgoal

  a : Prop
  b : Prop
  c : a /\ b
  ============================
   b /\ a
```

An example of proof (continued)

```
  ...
  c : a /\ b
  ============================
   b /\ a
case c.
  ...
  ============================
   a -> b -> b /\ a
intros ha hb.
  ...
  ha : a
  hb : b
  ============================
   b /\ a
```

An example of proof (continued)

```
  ...
  ============================
   b /\ a
split.
2 subgoals
  ...
  hb : b
  ============================
  b

subgoal 2 is:
 a
```

An example of proof (continued)

```
exact hb.
  ...
  ha : a
  ...
  ============================
   a
assumption.
Proof completed.
Qed.
intros a b c.
case c.
...
ex1 is defined
```

About tactics

- The tactic `apply` performs backward chaining with a theorem's goal,
- the tactic `elim` looks systematically for a theorem shaped like an induction principle,
- The tactic `intro` can destructure inductive types,
- The tactics `change`, `simpl` replace the goal with a convertible one,
- The tactic `rewrite` uses equalities (hides a case analysis,
- Automatic tactics are provided for decidable fragments: `intuition`, `firstorder`, `ring`, `field`, `omega`.

Programs as proofs

- Use tactics to develop algorithms,
- `apply` calls a function,
- `case` describes case analysis (with dependencies),
- `elim` describes a recursive computation,
- More complex tactics should be avoided.

Mixing algorithmic and logical content

- Inductive types can contain both data and proofs,
- Function can take as argument both data and proofs,
- Allow for partial functions,
- More expressive types,
- Examples follow.

Constructive disjunction

```
Inductive sumbool (A B:Prop) : Set :=
  left (h:A) | right (h:B).

Notation { A } + { B } := sumbool A B.
```

- Functions returning a `sumbool` type are like boolean functions,
- `sumbool` types can be used in proofs like disjunctions,
- Pattern matching on sumbool values increases the context.

Learning from experience

- Comparing pattern-matching constructs:

```
match vb with true => e1 | false => e2 end

match vsb with left h => e'1 | right h' => e'2 end
```

- `e1` and `e2` live in the same context,
- `e'1` and `e'2` are distinguished by the knowledge `h` and `h'`,
- Extra knowledge used to
  - add knowledge to results,
  - justify calls to partial functions,
  - or discard unreachable cases.

certified values

- Sigma types: a generalization of constructive disjunction,
- Combine an index and a element of a family at this index,
- Usable like an existential statement,
- Like the earlier `bt`, but with a proof as second component.

```
Inductive sig (A:Type)(P:A->Prop) : Type :=
 exist (x:A)(H:P x).

Notation "{ x : A | P x } " := sig A (fun x => P x).
```

Better representation of vectors

- make sure that the length information can be forgotten easily,

```
Definition vector (A:Type)(n:nat) :=
   {l:list A | length l = n}.
```

Example: insertion sort

```
Variables (A : Type)(le : A -> A -> Prop).
Infix "<=" := le.
Variable le_dec : forall x y, {x <= y}+{y <= x}.

Inductive sorted : list A -> Prop :=
  s0 : sorted nil
| s1 : forall x, sorted (x::nil)
| s2 : forall x y l, x <= y -> sorted (y::l) ->
   sorted (x::y::nil).

Hint Resolve s0 s1 s2.
```

The sort function

```
Check insert.
 : A -> forall l:list A, sorted l -> {l' | sorted l'}.

Fixpoint sort (l:list A) : {l' | sorted l'} :=
  match l with
    nil => exist _ nil s0
  | a::tl => let (l', p) := sort tl in insert a l' p
  end.
```

The insert function

```
Definition insert : A -> list A -> {l' | sorted l'}.
intros x l sl; assert
 (S : {l' | sorted l' /\
    forall b, sorted (b::l) -> b <= x -> sorted (b::l')}).
induction l.
  sl : sorted nil
  ======================
   {l' | sorted l' /\ ...}
exists (x::nil); auto.
```

insert (continued)

```
  sl : sorted (a :: l)
  IHl : sorted l -> {l' : list A | sorted l' /\ ... }
  ==============================
  {l' : list A | sorted l' /\ ... }

case (le_dec x a); intros cmp.

exists (x::a::l).
  cmp : x <= a
  ==============================
   sorted (x :: a :: l) /\
   (forall b : A, sorted (b :: a :: l) -> b <= x ->
      sorted (b :: x :: a :: l))
auto.
```

insert (continued)

```
  sl : sorted (a :: l)
  IHl : sorted l -> {l' | sorted l' /\ forall b, ...}
  cmp : a <= x
  ===========================
   {l' | sorted l' /\ ...}
assert (sl1 : sorted l) by (inversion sl; auto).
destruct (IHl sl) as [l' [_ sl']].
  sl' : forall b, sorted (b :: l) -> b <= x ->
     sorted (b :: l').
```

insert (continued)

```
exists (a::l').
split; try (intros b s'; inversion s'); firstorder.

(* unloading the recursion. *)
S : {l' : list A |
      sorted l' /\ (forall b, sorted (b::l) -> ...)
  ============================
   {l' : list A | sorted l'}
destruct S as [l' [sl' _]]; exists l'; exact sl'.
Proof completed.
Defined.
```

insert and sort: testing

```
Require Import Arith Omega.

Definition le_dec : forall x y : nat, {x <= y}+{y <= x}.
...
Defined.

Eval vm_compute in
  let (l, _) := sort _ _ le_dec (1::7::3::2::nil).
     = 1 :: 2 :: 3 :: 7 :: nil
     : list nat
```

Algorithmic content

```
Extraction insert.
(** val insert : ('a1 -> 'a1 -> sumbool) ->
    'a1 -> 'a1 list -> 'a1 list **)

let rec insert le_dec x = function
  | Nil -> Cons (x, Nil)
  | Cons (a, l0) ->
      (match le_dec x a with
         | Left -> Cons (x, (Cons (a, l0)))
         | Right -> Cons (a, (insert le_dec x l0)))
```

General recursion

- The foundation : *well-founded induction*,
- Directly describable as structural recursion over accessibility, viewed as an inductive proposition,
- Allow recursive calls only on predecessors for a well-founded relation,
- Discipline enforced by typing,
- Promotes types as strong specifications.

```
Fix : forall (A : Type) (R : A -> A -> Prop),
   well_founded R ->
   forall P : A -> Type,
   (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
   forall x : A, P x
```

The Function command

- Add support for various forms of terminating recursion,
- Uniform syntax for structural, well-founded, or measure-based termination criteria,
- Induction principle (somehow: induction on the computation tree),
- Avoids dependent types in definitions (write ML-like code),
- Less complete than the basic well-founded induction.

Example with Function

```
Function sum (x:Z) {measure Zabs_nat} : Z :=
  if Z_le_dec x 0 then 0 else x + sum (x-1).
1 subgoal

  ============================
   forall (x : Z) (anonymous : ~ x <= 0),
   Z_le_dec x 0 = right (x <= 0) anonymous ->
   (Zabs_nat (x - 1) < Zabs_nat x)%nat
intros x xneg _; apply Zabs_nat_lt; omega.
Defined.
```

Function example

```
Lemma sum_p : forall x, 0 <= x -> 2*sum x = x*(x+1).
2 subgoals

  ...
  _x : x <= 0
  ============================
   0 <= x -> 2 * 0 = x * (x + 1)
intros; assert (x = 0) by omega; subst x; auto.
...
  _x : ~ x <= 0
  IHz : 0 <= x - 1 ->
        2 * sum (x - 1) = (x - 1) * (x - 1 + 1)
  ============================
   0 <= x -> 2 * (x + sum (x - 1)) = x * (x + 1)
```

Function example

```
...
  _x : ~ x <= 0
  IHz : 0 <= x - 1 ->
        2 * sum (x - 1) = (x - 1) * (x - 1 + 1)
  ============================
   0 <= x -> 2 * (x + sum (x - 1)) = x * (x + 1)
intros;
 replace (2*(x+sum(x-1))) with (2*x + 2*sum(x-1)) by ring;
 rewrite IHz;[ring | omega].
Proof completed.
Qed.
```

Next four slides were presented at the conference.

Co-induction

- A different form of recursion,
- Data is not necessarily finite,
- Recursion is allowed only if data is being produced,
- Computation is lazy.

```
CoInductive Stream (A:Type) : Type :=
  Scons (a:A)(s:Stream A).

Implicit Arguments Scons [A].
Infix "::" := Scons (at level 60, right associativity).

CoFixpoint zeros : Stream nat := 0::zeros.

CoFixpoint nums (n:nat) : Stream nat := n::nums (n+1).
```

Lazy computation

```
Fixpoint explore (A:Type)(s:Stream A)(n:nat): A :=
  match s, n with
    a::_, 0 => a
  | _::t, S p => explore _ t p
  end.
Implicit Arguments explore [A].

Definition nats := nums 0.
Time Eval vm_compute in explore nats 10000.
   = 10000 : nat
Finished transaction in 10. secs (...)
Time Eval vm_compute in explore nats 10000.
   = 10000 : nat
Finished transaction in 0. secs (...)
```

Erastothene's Sieve in 50 lines

```
(* Definitions of Stream, nums, take divides : 22 lines *)

Fixpoint bfilter (p:nat->bool)(n:nat)(s:Stream nat)
  {struct n} : nat*Stream nat :=
 match n with
   0 => let (a, tl) := s in (a, tl)
 | S k =>
  let (a, tl) := s in
  if p a then (a,tl) else bfilter p k tl
 end.

CoFixpoint filter (p:nat->bool)(k:nat)(s:Stream nat)
    : Stream nat :=
  let (a,tl) := bfilter p k s in a::filter p a tl.
```

Eratosthene's sieve, continued

```
CoFixpoint sieve (s:Stream nat) : Stream nat :=
   let (a,tl) := s in
   a::sieve (filter (not_divides a) a tl).

Definition primes := sieve (nums 2).

Eval vm_compute in take 20 primes.
  = 2 :: 3 :: 5 :: 7 :: 11 :: 13 :: 17 :: 19 :: 23
     :: 29 :: 31 :: 37 :: 41 :: 43 :: 47 :: 53 :: 59
     :: 61 :: 67 :: 71 :: nil
```

Slides beyond this one were not presented at the conference.

Co-Inductive predicates

- Predicates with "infinite proofs",
- Same well-formedness criterion as co-recursive data,
  - Proofs actually not more infinite than proofs by induction,

Example of co-inductive predicates

```
CoInductive prime_spec : Stream nat -> Prop :=
 cp1 : forall a tl, prime a -> prime_spec tl ->
      prime_spec (a::tl).

CoInductive all_prime_spec (p:nat) : Stream nat -> Prop :=
  cp2 : forall a tl, p < a -> prime a ->
         (forall x, p < x < a -> ~prime a) ->
         all_prime_spec a tl ->
         all_prime_spec p (a::tl).

CoInductive bisimilar (A:Type) :
  Stream A -> Stream A -> Prop :=
  cb : forall a tl1 tl2, bisimilar tl1 tl2 ->
      bisimilar (a::tl1) (a::tl2).
```

Reflexion

- Define a function that computes inside the theorem prover,
- Establish a theorem the results of the function,
- Use the theorem to prove results,
- Approach used inside Coq for ring equalities,
- Our example : associativity.

Re-organizing binary trees

```
Require Import Arith.
Set Implicit Arguments.

Section fl.

Variables (A : Type) (op : A -> A -> A).
Hypothesis assoc : forall x y z, op x (op y z) = op (op x y

Inductive bin : Type := L (v:A) | N (x y : bin).

Function fl1 (x y : bin) struct x : bin :=
  match x with
    L v => N (L v) y
  | N t1 t2 => fl1 t1 (fl1 t2 y)
  end.
```

Re-organizing binary trees

```
Function fl (x : bin) struct x : bin :=
  match x with L v => L v | N t1 t2 => fl1 t1 (fl t2) end.

Function it (t:bin) struct t : A :=
 match t with
   L v => v | N t1 t2 => op (it t1) (it t2)
 end.
```

Re-organizing binary trees (proofs)

```
Lemma fl1_s : forall t1 t2,
   it (fl1 t1 t2) = op (it t1) (it t2).
intros t1 t2; functional induction (fl1 t1 t2).
  ================
  it (N (L v) y) = op (it (L v)) (it y)
auto.
  IHb : it (fl1 t2 y) = op (it t2) (it y)
  IHb0 : it (fl1 t1 (fl1 t2 y)) =
     op (it t1) (it (fl1 t2 y))
  ================
  it (fl1 t1 (fl1 t2 y)) = op (it (N t1 t2)) (it y)
simpl; rewrite IHb0, IHb.
auto.
Qed.
```

Re-organizing binary trees (proofs)

```
Lemma fl_s :  forall t, it (fl t) = it t.
intros t; functional induction (fl t); auto.
rewrite fl1_s, IHb; simpl; auto.
Qed.

Lemma fl2 : forall t1 t2, it (fl t1) = it (fl t2) ->
   it t1 = it t2.
intros t1 t2; repeat rewrite fl_s; auto.
Qed.

End fl.
```

Transforming problem into data

```
Ltac mkt f v :=
  match v with
  | (f ?X1 ?X2) =>
    let r1 := mkt f X1 with r2 := mkt f X2 in
    constr:(N r1 r2)
  | ?X => constr:(L X)
  end.

Ltac abstract_plus := intros;
  match goal with
  |- ?X1 = ?X2 =>
    let r1 := mkt plus X1 with r2 := mkt plus X2 in
    change (it plus r1 = it plus r2)
  end.
```

Example on a goal

```
Lemma ex1 : forall x y, 1 + x + 3 + y = (1 + x) + (3 + y).
abstract_plus.
  ============================
   it plus (N (N (N (L 1) (L x)) (L 3)) (L y)) =
   it plus (N (N (L 1) (L x)) (N (L 3) (L y)))
apply fl2 with (1 := plus_assoc).
  ============================
   it plus (fl (N (N (N (L 1) (L x)) (L 3)) (L y))) =
   it plus (fl (N (N (L 1) (L x)) (N (L 3) (L y))))
simpl fl.
  ============================
   it plus (N (L 1) (N (L x) (N (L 3) (L y)))) =
   it plus (N (L 1) (N (L x) (N (L 3) (L y))))
reflexivity.
Qed.
```

Topics not covered

- Subtyping: simulated with the help of coercions,
- Polymorphism: simulated with implicit arguments,
- Modularity,
- Defined equality: the `Setoid` approach,
- Type classes and canonical structures,
- small-scale reflection.