Coq tutorial
Program verification using Coq

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

TYPES summer school – August 23th, 2005

Introduction

This lecture: how to use Coq to verify purely functional programs

Thursday's lecture: verification of imperative programs (using Coq and other provers)

Introduction

This lecture: how to use Coq to verify purely functional programs

Thursday's lecture: verification of imperative programs (using Coq and other provers)

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

Program extraction

Two sorts:

Prop	:	the sort of logic terms
Set	:	the sort of informative terms

Program extraction turns the informative contents of a Coq term into an ML program while removing the logical contents

Program extraction

Two sorts:

Prop	:	the sort of logic terms
Set	:	the sort of informative terms

Program extraction turns the informative contents of a Coq term into an ML program while removing the logical contents

Outline

1. Direct method (ML function defined in Coq)
2. Use of Coq dependent types
3. Modules and functors

Running example

Finite sets library implemented with balanced binary search trees

1. useful
2. complex
3. purely functional

The Ocaml library Set was verified using Coq
One (balancing) bug was found (fixed in current release)

Running example

Finite sets library implemented with balanced binary search trees

1. useful
2. complex
3. purely functional

The Ocaml library Set was verified using Coq
One (balancing) bug was found (fixed in current release)

Running example

Finite sets library implemented with balanced binary search trees

1. useful
2. complex
3. purely functional

The Ocaml library Set was verified using Coq
One (balancing) bug was found (fixed in current release)

Direct method

Most ML functions can be defined in Coq

$$f \quad : \quad \tau_1 \rightarrow \tau_2$$

A specification is a relation $S : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{Prop}$
f verifies S if

$$\forall x : \tau_1 . \, (S \; x \; (f \; x))$$

The proof is conducted following the definition of f

Direct method

Most ML functions can be defined in Coq

$$f \quad : \quad \tau_1 \rightarrow \tau_2$$

A specification is a relation $S : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{Prop}$
f verifies S if

$$\forall x : \tau_1.\ (S\ x\ (f\ x))$$

The proof is conducted following the definition of f

Direct method

Most ML functions can be defined in Coq

$$f \quad : \quad \tau_1 \rightarrow \tau_2$$

A specification is a relation $S : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{Prop}$
f verifies S if

$$\forall x : \tau_1.\ (S\ x\ (f\ x))$$

The proof is conducted following the definition of f

Binary search trees

The type of trees

```
Inductive tree : Set :=
  | Empty
  | Node : tree → Z → tree → tree.
```

The membership relation

```
Inductive In (x:Z) : tree → Prop :=
  | In_left : ∀l r y, In x l → In x (Node l y r)
  | In_right : ∀l r y, In x r → In x (Node l y r)
  | Is_root : ∀l r, In x (Node l x r).
```

Binary search trees

The type of trees

```
Inductive tree : Set :=
  | Empty
  | Node : tree → Z → tree → tree.
```

The membership relation

```
Inductive In (x:Z) : tree → Prop :=
  | In_left : ∀l r y, In x l → In x (Node l y r)
  | In_right : ∀l r y, In x r → In x (Node l y r)
  | Is_root : ∀l r, In x (Node l x r).
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function mem

ML

```
let rec mem x = function
  | Empty →
      false
  | Node (l, y, r) →
      let c = compare x y in
      if c < 0 then mem x l
      else if c = 0 then true
      else mem x r
```

The function mem

Coq

```
Fixpoint mem (x:Z) (s:tree) {struct s} : bool :=
  match s with
  | Empty ⇒
     false
  | Node l y r ⇒ match compare x y with
     | Lt ⇒ mem x l
     | Eq ⇒ true
     | Gt ⇒ mem x r
    end
  end.
```

assuming

```
Inductive order : Set := Lt | Eq | Gt.
Hypothesis compare : Z → Z → order.
```

The function mem

Coq

```
Fixpoint mem (x:Z) (s:tree) {struct s} : bool :=
  match s with
  | Empty ⇒
     false
  | Node l y r ⇒ match compare x y with
     | Lt ⇒ mem x l
     | Eq ⇒ true
     | Gt ⇒ mem x r
    end
  end.
```

assuming

```
Inductive order : Set := Lt | Eq | Gt.
Hypothesis compare : Z → Z → order.
```

Correctness of the function mem

to be a binary search tree

```
Inductive bst : tree → Prop :=
  | bst_empty :
     bst Empty
  | bst_node :
     ∀x l r,
     bst l → bst r →
     (∀y, In y l → y < x) →
     (∀y, In y r → x < y) → bst (Node l x r).
```

```
Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
```

specification S has the shape $P\ x \rightarrow Q\ x\ (f\ x)$

Correctness of the function mem

to be a binary search tree

```
Inductive bst : tree → Prop :=
  | bst_empty :
      bst Empty
  | bst_node :
      ∀x l r,
      bst l → bst r →
      (∀y, In y l → y < x) →
      (∀y, In y r → x < y) → bst (Node l x r).
```

```
Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
```

specification S has the shape $P\ x \rightarrow Q\ x\ (f\ x)$

Correctness of the function mem

to be a binary search tree

```
Inductive bst : tree → Prop :=
  | bst_empty :
     bst Empty
  | bst_node :
     ∀x l r,
     bst l → bst r →
     (∀y, In y l → y < x) →
     (∀y, In y r → x < y) → bst (Node l x r).

Theorem mem_correct :
 ∀x s, bst s → ((mem x s)=true ↔ In x s).
```

specification S has the shape $P\ x \rightarrow Q\ x\ (f\ x)$

Modularity

To prove mem correct requires a property for compare

```
Hypothesis compare_spec :
  ∀x y, match compare x y with
    | Lt ⇒ x < y
    | Eq ⇒ x = y
    | Gt ⇒ x > y
  end.

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
Proof.
  induction s; simpl.
  ...
  generalize (compare_spec x y); destruct (compare x y).
  ...
```

Modularity

To prove mem correct requires a property for compare

```
Hypothesis compare_spec :
  ∀x y, match compare x y with
    | Lt ⇒ x < y
    | Eq ⇒ x = y
    | Gt ⇒ x > y
  end.

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
Proof.
  induction s; simpl.
  ...
  generalize (compare_spec x y); destruct (compare x y).
  ...
```

Modularity

To prove mem correct requires a property for compare

```
Hypothesis compare_spec :
  ∀x y, match compare x y with
    | Lt ⇒ x < y
    | Eq ⇒ x = y
    | Gt ⇒ x > y
  end.

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
Proof.
  induction s; simpl.
  ...
  generalize (compare_spec x y); destruct (compare x y).
  ...
```

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \rightarrow \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \rightarrow \texttt{Z}$$

specification

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \text{bst}\ s \rightarrow$$
$$\texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \rightarrow \texttt{min\_elt}\ s\ h \leq x$$

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \rightarrow \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \rightarrow \texttt{Z}$$

specification

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \rightarrow$$
$$\texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \rightarrow \texttt{min\_elt}\ s\ h \leq x$$

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \rightarrow \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \rightarrow \texttt{Z}$$

specification

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \text{bst}\ s \rightarrow \\ \quad \texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \rightarrow \texttt{min\_elt}\ s\ h \leq x$$

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \rightarrow \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \rightarrow \texttt{Z}$$

specification

$$\begin{array}{l} \forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \rightarrow \\ \quad \texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \rightarrow \texttt{min\_elt}\ s\ h \leq x \end{array}$$

Even the definition of a partial function is not easy

ML

```
let rec min_elt = function
  | Empty → assert false
  | Node (Empty, x, _) → x
  | Node (l, _, _) → min_elt l
```

Coq

1. `assert false` ⇒ elimination on a proof of False
2. recursive call requires a proof that `l` is not empty

Even the definition of a partial function is not easy

ML

```
let rec min_elt = function
  | Empty → assert false
  | Node (Empty, x, _) → x
  | Node (l, _, _) → min_elt l
```

Coq

1. `assert false` ⇒ elimination on a proof of False
2. recursive call requires a proof that `l` is not empty

Even the definition of a partial function is not easy

ML

```
let rec min_elt = function
  | Empty → assert false
  | Node (Empty, x, _) → x
  | Node (l, _, _) → min_elt l
```

Coq

1. `assert false` ⇒ elimination on a proof of False
2. recursive call requires a proof that `l` is not empty

min_elt: a solution

```
Fixpoint min_elt (s:tree) (h:¬s=Empty) { struct s } : Z :=
  match s                          with
  | Empty ⇒


  | Node l x _ ⇒
               match l                             with
               | Empty ⇒                x
               | _ ⇒                  min_elt l

               end
  end   .
```

min_elt: a solution

```
Fixpoint min_elt (s:tree) (h:¬s=Empty) { struct s } : Z :=
  match s return ¬s=Empty → Z with
  | Empty ⇒
    (fun (h:¬Empty=Empty) ⇒
       False_rec _ (h (refl_equal Empty)))
  | Node l x _ ⇒
    (fun h ⇒ match l as a return a=l → Z with
             | Empty ⇒ (fun _ ⇒ x)
             | _ ⇒ (fun h ⇒ min_elt l
                              (Node_not_empty _ _ _ _ h))
             end (refl_equal l))
  end h.
```

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀ s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof (cont'd)

We can check the extracted code:

```
Extraction min_elt.

(** val min_elt : tree → z **)

let rec min_elt = function
  | Empty → assert false (* absurd case *)
  | Node (t0, z0, t1) →
      (match t0 with
         | Empty → z0
         | Node (s1_1, z1, s1_2) → min_elt t0)
```

The refine tactic

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  refine
    (fix min (s:tree) (h:¬s=Empty) { struct s } : Z :=
    match s return ¬s=Empty → Z with
    | Empty ⇒
        (fun h ⇒ _)
    | Node l x _ ⇒
        (fun h ⇒ match l as a return a=l → Z with
                  | Empty ⇒ (fun _ ⇒ x)
                  | _ ⇒ (fun h ⇒ min l _)
                  end _)
    end h).
...
```

The refine tactic

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  refine
    (fix min (s:tree) (h:¬s=Empty) { struct s } : Z :=
    match s return ¬s=Empty → Z with
    | Empty ⇒
        (fun h ⇒ _)
    | Node l x _ ⇒
        (fun h ⇒ match l as a return a=l → Z with
                  | Empty ⇒ (fun _ ⇒ x)
                  | _ ⇒ (fun h ⇒ min l _)
                  end _)
    end h).
...
```

A last solution

To make the function total

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty ⇒ 0
  | Node Empty z _ ⇒ z
  | Node l _ _ ⇒ min_elt l
end.
```

Correctness theorem almost unchanged:

```
Theorem min_elt_correct :
  ∀s, ¬s=Empty → bst s →
    In (min_elt s) s ∧
    ∀x, In x s → min_elt s <= x.
```

A last solution

To make the function total

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty ⇒ 0
  | Node Empty z _ ⇒ z
  | Node l _ _ ⇒ min_elt l
end.
```

Correctness theorem almost unchanged:

```
Theorem min_elt_correct :
  ∀s, ¬s=Empty → bst s →
    In (min_elt s) s ∧
    ∀x, In x s → min_elt s <= x.
```

A last solution

To make the function total

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty ⇒ 0
  | Node Empty z _ ⇒ z
  | Node l _ _ ⇒ min_elt l
end.
```

Correctness theorem almost unchanged:

```
Theorem min_elt_correct :
 ∀s, ¬s=Empty → bst s →
   In (min_elt s) s ∧
   ∀x, In x s → min_elt s <= x.
```

Functions that are not structurally recursive

One solution is to use a well-founded induction principle such as

```
well_founded_induction
   : ∀(A : Set) (R : A → A → Prop),
     well_founded R →
     ∀P : A → Set,
     (∀x : A, (∀y : A, R y x → P y) → P x) →
     ∀a : A, P a
```

Defining the function requires to build proof terms (of `R y x`) similar to partial functions ⇒ similar solutions

Functions that are not structurally recursive

One solution is to use a well-founded induction principle such as

```
well_founded_induction
   : ∀ (A : Set) (R : A → A → Prop),
     well_founded R →
     ∀ P : A → Set,
     (∀ x : A, (∀ y : A, R y x → P y) → P x) →
     ∀ a : A, P a
```

Defining the function requires to build proof terms (of `R y x`) similar to partial functions ⇒ similar solutions

Example: the subset function

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ →
      true
  | _, Empty →
      false
  | Node (l1, v1, r1), Node (l2, v2, r2) →
      let c = compare v1 v2 in
      if c = 0 then
        subset l1 l2 && subset r1 r2
      else if c < 0 then
        subset (Node (l1, v1, Empty)) l2 && subset r1 s2
      else
        subset (Node (Empty, v1, r1)) r2 && subset l1 s2
```

Example: the subset function

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ →
      true
  | _, Empty →
      false
  | Node (l1, v1, r1), Node (l2, v2, r2) →
      let c = compare v1 v2 in
      if c = 0 then
        subset l1 l2 && subset r1 r2
      else if c < 0 then
        subset (Node (l1, v1, Empty)) l2 && subset r1 s2
      else
        subset (Node (Empty, v1, r1)) r2 && subset l1 s2
```

Induction over two trees

```
Fixpoint cardinal_tree (s:tree) : nat := match s with
  | Empty ⇒
      O
  | Node l _ r ⇒
      (S (plus (cardinal_tree l) (cardinal_tree r)))
end.
```

```
Lemma cardinal_rec2 :
  ∀ (P:tree→tree→Set),
  (∀ (x x':tree),
     (∀ (y y':tree),
       (lt (plus (cardinal_tree y) (cardinal_tree y'))
           (plus (cardinal_tree x) (cardinal_tree x'))) →
     → (P x x')) →
  ∀ (x x':tree), (P x x').
```

Induction over two trees

```
Fixpoint cardinal_tree (s:tree) : nat := match s with
  | Empty ⇒
     0
  | Node l _ r ⇒
     (S (plus (cardinal_tree l) (cardinal_tree r)))
end.

Lemma cardinal_rec2 :
  ∀(P:tree→tree→Set),
  (∀(x x':tree),
     (∀(y y':tree),
       (lt (plus (cardinal_tree y) (cardinal_tree y'))
           (plus (cardinal_tree x) (cardinal_tree x'))) →
     → (P x x')) →
  ∀(x x':tree), (P x x').
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Extraction

```
Extraction well_founded_induction.
  let rec well_founded_induction x a =
    x a (fun y _ → well_founded_induction x y)
```

```
Extraction Inline cardinal_rec2 ...
Extraction subset.
```

gives the expected ML code

Extraction

```
Extraction well_founded_induction.
  let rec well_founded_induction x a =
    x a (fun y _ → well_founded_induction x y)
```

```
Extraction Inline cardinal_rec2 ...
Extraction subset.
```

gives the expected ML code

To sum up, defining an ML function in Coq and prove it correct seems the obvious way, but it can be rather complex when the function

- is partial, and/or
- is not structurally recursive

Use of dependent types

Instead of

1. defining a pure function, and
2. proving its correctness

let us do both at the same time

We can give Coq functions richer types that are specifications

Example

$$f : \{n : Z \mid n \geq 0\} \rightarrow \{p : Z \mid \texttt{prime}\ p\}$$

Use of dependent types

Instead of

1. defining a pure function, and
2. proving its correctness

let us do both at the same time

We can give Coq functions richer types that are specifications

Example

$$f : \{n : Z \mid n \geq 0\} \rightarrow \{p : Z \mid \texttt{prime}\ p\}$$

Use of dependent types

Instead of

1. defining a pure function, and
2. proving its correctness

let us do both at the same time

We can give Coq functions richer types that are specifications

Example

$$f : \{n : Z \mid n \geq 0\} \rightarrow \{p : Z \mid \texttt{prime}\ p\}$$

The type $\{x : A \mid P\}$

Notation for `sig` A (`fun` $x \Rightarrow P$) where

```
Inductive sig (A : Set) (P : A → Prop) : Set :=
  exist : ∀x:A, P x → sig P
```

In practice, we adopt the more general specification

$$f : \forall\ x : \tau_1,\ P\ x \rightarrow \{y : \tau_2 \mid Q\ x\ y\}$$

The type $\{x : A \mid P\}$

Notation for `sig` A (`fun` $x \Rightarrow P$) where

```
Inductive sig (A : Set) (P : A → Prop) : Set :=
  exist : ∀x:A, P x → sig P
```

In practice, we adopt the more general specification

$$f : \forall\ x : \tau_1,\ P\ x \rightarrow \{y : \tau_2 \mid Q\ x\ y\}$$

The type $\{x : A \mid P\}$

Notation for `sig` A (`fun` $x \Rightarrow P$) where

```
Inductive sig (A : Set) (P : A → Prop) : Set :=
  exist : ∀x:A, P x → sig P
```

In practice, we adopt the more general specification

$$f : \forall\, x : \tau_1,\ P\ x \rightarrow \{y : \tau_2 \mid Q\ x\ y\}$$

Example: the `min_elt` function

```
Definition min_elt :
  ∀s, ¬s=Empty → bst s →
    { m:Z | In m s ∧ ∀x, In x s → m <= x }.
```

We usually adopt a definition-by-proof
(which is now a definition-and-proof)

Still the same ML program

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

Example: the min_elt function

```
Definition min_elt :
  ∀s, ¬s=Empty → bst s →
    { m:Z | In m s ∧ ∀x, In x s → m <= x }.
```

We usually adopt a definition-by-proof
(which is now a definition-and-proof)

Still the same ML program

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

Example: the min_elt function

```
Definition min_elt :
  ∀s, ¬s=Empty → bst s →
    { m:Z | In m s ∧ ∀x, In x s → m <= x }.
```

We usually adopt a definition-by-proof
(which is now a definition-and-proof)

Still the same ML program

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

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

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

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

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

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

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

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

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Variant sumor A+{B}

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A → A + {B}
  | inright : B → A + {B}
```

Extracts to an option type

Example:

```
Definition min_elt :
  ∀s, bst s →
  { m:Z | In m s ∧ ∀x, In x s → m <= x } + { s=Empty }.
```

Variant sumor A+{B}

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A → A + {B}
  | inright : B → A + {B}
```

Extracts to an option type

Example:

```
Definition min_elt :
  ∀s, bst s →
  { m:Z | In m s ∧ ∀x, In x s → m <= x } + { s=Empty }.
```

Variant sumor A+{B}

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A → A + {B}
  | inright : B → A + {B}
```

Extracts to an option type

Example:

```
Definition min_elt :
  ∀s, bst s →
  { m:Z | In m s ∧ ∀x, In x s → m <= x } + { s=Empty }.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

To sum up, using dependent types

- we replace a definition and a proof by a single proof
- the ML function is still available using extraction

On the contrary, it is more difficult to prove several properties of the same function

To sum up, using dependent types

- we replace a definition and a proof by a single proof
- the ML function is still available using extraction

On the contrary, it is more difficult to prove several properties of the same function

Modules and functors

Coq has a module system similar to Objective Caml's one

Coq modules can contain definitions but also proofs, notations, hints for the `auto` tactic, etc.

As Ocaml, Coq has functors i.e. functions from modules to modules

Modules and functors

Coq has a module system similar to Objective Caml's one

Coq modules can contain definitions but also proofs, notations, hints for the `auto` tactic, etc.

As Ocaml, Coq has functors i.e. functions from modules to modules

Modules and functors

Coq has a module system similar to Objective Caml's one

Coq modules can contain definitions but also proofs, notations, hints for the `auto` tactic, etc.

As Ocaml, Coq has functors i.e. functions from modules to modules

ML modules

```
module type OrderedType = sig
  type t
  val compare: t → t → int
end

module Make(Ord: OrderedType) : sig
  type t
  val empty : t
  val mem : Ord.t → t → bool
  ...
end
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

The Coq functor for binary search trees

```
Module BST (X: OrderedType).
  Inductive tree : Set :=
  | Empty
  | Node : tree → X.t → tree → tree.

  Fixpoint mem (x:X.t) (s:tree) {struct s} : bool := ...

  Inductive In (x:X.t) : tree → Prop := ...
  Hint Constructors In.

  Inductive bst : tree → Prop :=
  | bst_empty : bst Empty
  | bst_node : ∀x l r, bst l → bst r →
      (∀y, In y l → X.lt y x) → ...
```

Conclusion

Coq is a tool of choice for the verification of purely functional programs, up to modules

ML or Haskell code can be obtained via program extraction

Conclusion

Coq is a tool of choice for the verification of purely functional programs, up to modules

ML or Haskell code can be obtained via program extraction