

Master Parisien de Recherche en Informatique
cours 2-7-2 : assistants de preuve

Preuve de programmes fonctionnels

Jean-Christophe Filliâtre

2005–2006

Ce cours : comment utiliser Coq pour vérifier des programmes **purement fonctionnels**

Obtenir un programme purement fonctionnel (ML) **prouvé correct**

Il y a au moins deux façons de procéder :

- ① définir sa fonction ML en Coq et en prouver ensuite la correction
- ② donner à la fonction Coq un type plus riche (= sa spécification) et obtenir ensuite le programme ML par **extraction**

Deux sortes :

Prop : la sorte des termes **logiques**

Set : la sorte des termes **informatifs**

L'extraction de programmes transforme le contenu informatif d'un terme Coq en un programme ML tout en supprimant le contenu logique

- ① Méthode directe (fonction ML définie en Coq)
- ② Utilisation de types dépendants
- ③ Modules et foncteurs

Bibliothèque d'ensembles finis représentés par des arbres binaires de recherche

- ① utile
- ② complexe
- ③ purement fonctionnel

La bibliothèque Ocaml **Set** a été vérifiée avec Coq
Un bug (d'équilibrage) a été trouvé (corrigé dans la version 3.07)

La plupart des fonctions ML peuvent être définies en Coq

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

Une spécification est une relation $S : \tau_1 \rightarrow \tau_2 \rightarrow \text{Prop}$
 f vérifie S si

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

La preuve suit la définition de f

Le type des arbres

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

La relation d'appartenance

```
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).
```


La fonction `is_empty`

ML

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

Coq

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

Correction

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
```

Proof.

```
destruct s; simpl; intuition.
```

...

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
```

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.
```

en supposant

```
Inductive order : Set := Lt | Eq | Gt.
```

```
Hypothesis compare : Z → Z → order.
```

être un **arbre binaire de recherche**

```
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).
```

S a la forme $P x \rightarrow Q x (f x)$

Prouver la correction de **mem** nécessite une propriété de **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).
```

```
...
```

Si la fonction f est partielle, elle a un type Coq

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

Exemple : `min_elt` retournant le plus petit élément d'un arbre

$$\text{min_elt} : \forall s : \text{tree}. \neg s = \text{Empty} \rightarrow \mathbb{Z}$$

spécification

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

La seule **définition** d'une fonction partielle peut être difficile

ML

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

Coq

- 1 `assert false` \Rightarrow élimination d'une preuve de **False**
- 2 l'appel récursif nécessite une preuve que `l` n'est pas vide

min_elt : une 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
  end .
```


min_elt : une 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.
```

Définition par preuve

Idée : utiliser l'éditeur de preuve pour construire la définition

Definition min_elt : $\forall s, \neg s = \text{Empty} \rightarrow Z$.

Proof.

```
induction s; intro h.  
elim h; auto.  
destruct s1.  
exact z.  
apply IHs1; discriminate.
```

Defined.

Mais avons-nous défini la bonne fonction ?

Définition par preuve (suite)

On peut vérifier le code extrait :

```
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)
```

La tactique refine

Definition min_elt : $\forall s, \neg s = \text{Empty} \rightarrow Z$.

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).
```

...

Une dernière solution

Rendre la fonction **totale**

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

L'énoncé de correction est presque inchangé :

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

Une solution est d'utiliser un principe de **récurrence bien fondée** tel que

`well_founded_induction`

`: $\forall (A : \text{Set}) (R : A \rightarrow A \rightarrow \text{Prop}),$`

`well_founded R \rightarrow`

`$\forall P : A \rightarrow \text{Set},$`

`$(\forall x : A, (\forall y : A, R\ y\ x \rightarrow P\ y) \rightarrow P\ x) \rightarrow$`

`$\forall a : A, P\ a$`

nécessite alors de construire des termes de preuve (de $R\ y\ x$)
même problème que les fonctions partielles \Rightarrow même solutions

Exemple : la fonction subset

```
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
```

Récurrence sur deux arbres

```
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
        → (P x x')) →
    ∀ (x x':tree), (P x x')).
```


Définir la fonction subset

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 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.
```

donne le code ML escompté

En résumé, définir une fonction ML en Coq et la prouver ensuite correcte semble la manière naturelle, mais peut s'avérer **complexe** lorsque la fonction

- est **partielle**, et/ou
- n'est **pas structurellement réursive**

Utilisation de types dépendants

Au lieu de

- ① définir une fonction pure, puis
- ② prouver sa correction

faisons les deux en même temps

On peut donner aux fonctions Coq des types plus riches qui **sont des spécifications**

Exemple

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

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

Notation pour $\text{sig } A \text{ (fun } x \Rightarrow P)$ où

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

En pratique, on adopte la spécification plus générale

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

Exemple : la fonction min_elt

Definition min_elt :

$$\forall s, \neg s = \text{Empty} \rightarrow \text{bst } s \rightarrow \\ \{ m : \mathbb{Z} \mid \text{In } m \ s \wedge \forall x, \text{In } x \ s \rightarrow m \leq x \}.$$

On adopte généralement une **définition par preuve**
(qui est maintenant une **définition-preuve**)

Toujours le même programme ML

Coq < **Extraction** sig.

```
type 'a sig = 'a
```

```
(* singleton inductive, whose constructor was exist *)
```

Spécification d'une fonction booléenne : $\{A\} + \{B\}$

Notation pour `sumbool A B` où

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

c'est une **disjonction informative**

Exemple :

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

L'extraction est un booléen

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

Variante $A+\{B\}$

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

S'extrait en un **type option**

Exemple :

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


Hypothesis compare : $\forall x y, \{x < y\} + \{x = y\} + \{x > y\}$.

Definition mem : $\forall x s, \text{bst } s \rightarrow \{ \text{In } x s \} + \{ \neg(\text{In } x s) \}$.

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.

En résumé, en utilisant des **types dépendants**

- on remplace une définition et une preuve par **une seule preuve**
- la fonction ML est toujours disponible par **extraction**

Note : Il est maintenant plus difficile de prouver **plusieurs propriétés** de la même fonction

Coq a un **système de modules** similaire à celui d'Objective Caml

Les modules de Coq peuvent contenir des définitions mais aussi des preuves, notations, indications pour la tactique auto, etc.

Comme Ocaml, Coq a des **foncteurs** i.e. des fonctions des modules vers les 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
```

```
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.
```

Le foncteur Coq des arbres binaires de recherche

```
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) → ...
```

Coq est **l'outil de choix** pour la vérification de programmes purement fonctionnels, jusqu'au modules

Du code ML ou Haskell, certifié, peut être obtenu par extraction