

Assistants de Preuve

Programmation fonctionnelle 2

Christine Paulin-Mohring

Université Paris-Sud 11, INRIA

18/12/08

Plan

Preuve de programmes

Schéma structurel

Schéma général

Monades

Modules

Tactiques

Tactique automatique

Tactique par réflexion

Preuve de programmes

Correspondance entre :

- ▶ schéma de définition récursive
- ▶ schéma de preuve par récurrence

Introduire et utiliser le bon principe pour chaque définition de fonction.

Des schémas de preuve basiques sont introduits par la définition inductive / :

- ▶ Principe de minimalité si / est dans la sorte **Prop**.
- ▶ Principes d'induction si / est dans la sorte **Set** ou **Type**.
- ▶ Principe non mutuellement récursif, non imbriqué.

Définitions inductives complexes

```

Inductive term : Type :=
  Var: nat→term | App: nat→lterm→term
with lterm : Type := Emp | Add: term→lterm→lterm.

```

```

term_ind : ∀P : term → Prop,
  (∀n, P (Var n)) → (∀n (l:lterm), P (App n l))
→ ∀t:term, P t

```

Récursion mutuelle :

```

Scheme term_lterm := Induction for term Sort Prop
with lterm_term := Induction for lterm Sort Prop.
term_lterm : ∀ (P:term→Prop) (Q:lterm→Prop),
  (∀n, P (Var n)) → (∀n l, Q l → P (App n l))
→ Q Emp → (∀t, P t → ∀l, Q l → Q (Add t l))
→ ∀t, P t

```

Définitions imbriquées

```

Inductive term : Type :=
  Var : nat → term | App : nat → list term → term.
term_ind : ∀P : term → Set,
  (∀n, P (Var n)) → (∀n (l:list term), P (App n l))
  → ∀t:term, P t

```

Definition du principe complet

```

Fixpoint term_lterm (t:term) : P t :=
  match t return P t with
  Var n ⇒ Pv n
| App n l ⇒ Pa n l
  ((fix lterm_term (l:list term): Q l := match l return Q
    nil ⇒ Qn
    | t::lt ⇒ Qc t lt (term_lterm t) (lterm_term lt)
    end) l)
end.

```

Schémas ad-hoc

Exemple définition non récursive par cas

Definition $f (x:I) : A := \text{match } x \text{ with}$
 $p_1 \Rightarrow t_1 \quad \dots \mid p_n \Rightarrow t_n$
end.

Principes associés

- ▶ Analyse par cas :

$$\forall P : I \rightarrow \mathbf{Prop}, (\forall \vec{x}_1, P p_1) \rightarrow (\forall \vec{x}_2, P p_2) \rightarrow \dots \rightarrow (\forall \vec{x}_n, P p_n) \rightarrow \forall x : I, P x$$

- ▶ Analyse de la fonction :

$$\forall P : I \rightarrow A \rightarrow \mathbf{Prop}, (\forall \vec{x}_1, P p_1 t_1) \rightarrow (\forall \vec{x}_2, P p_2 t_2) \rightarrow \dots \rightarrow (\forall \vec{x}_n, P p_n t_n) \\ \rightarrow \forall x : I, P x (f x)$$

Construction de principes d'induction

Outils expérimentaux qui traitent une certaine classe de fonctions récursives structurelles.

- ▶ Principe engendré par :
Functional Scheme *name* := **Induction for function** **Sort** *s*.
- ▶ Principe utilisé avec :
elim *term* **using** *name*.
- ▶ Tactique : **functional induction** *function arg*
applique le principe fonctionnel associé à *function* sur l'argument *arg*.

Outil **Function**

```
Function insert (a:A) (l:list A) {struct l} : list A :=  
  match l with nil  $\Rightarrow$  nil  
    | cons b m  $\Rightarrow$  if leb a b then a::b::m  
                  else b::insert a m  
end.
```

► fonction sous forme de relation `R_insert` et algorithme `insert`.

► correspondance

`R_insert_complete`: $\forall (a:A) (l \text{ res}:list A),$
 `R_insert a l res` \rightarrow `res = insert a l`.

`R_insert_correct`: $\forall (a:A) (l \text{ res}:list A),$
 `res = insert a l` \rightarrow `R_insert a l res`.

► `insert_equation` preuve de l'équation récursive

► `insert_ind`, (`_rec`, `_rect`) principes d'induction associés à la fonction.

Récursion généralisée

Utilisation de mesures

`Require Export Recdef.`

```
Definition mes (l: list A*list A) : nat :=
  let (l1,l2):= l in length l1 + length l2.
```

```
Function merge (l:list A*list A) {measure mes} : list A :=
  match l with
  | (nil,l2)  => l2
  | (l1,nil)  => l1
  | (a::m1,b::m2) => if le a b then a::merge (m1,b::m2)
                     else b::merge (a::m1,m2)
  end.
{... tactics ...}
Defined.
```

Génération d'obligations de preuve.

Récursion généralisée

Utilisation d'ordres bien fondés.

```

Definition Rm (l m : list A * list A) :=
  let (l1, l2) := l in let (m1, m2) := m in
    length l1 < length m1 ∨
    (length l1 = length m1 ∧ length l2 < length m2).

```

```

Function merge (l : list A * list A) {wf Rm} : list A :=
  match l with
  | (nil, l2) ⇒ l2
  | (l1, nil) ⇒ l1
  | (a :: m1, b :: m2) ⇒ if le a b then a :: merge (m1, b :: m2)
                        else b :: merge (a :: m1, m2)
  end.

```

Construction par itération d'une fonctionnelle et preuve de terminaison.

Plan

Preuve de programmes

Schéma structurel

Schéma général

Monades

Modules

Tactiques

Tactique automatique

Tactique par réflexion

Monades

Principes

- ▶ Une manière systématique de représenter des constructions impératives dans un langage purement fonctionnel.
- ▶ Utilisé pour programmer en Haskell.
- ▶ Utilisé pour la preuve de programmes impératifs en Why.

Principe des monades

- ▶ Effets impératifs : références mémoire, exceptions, continuations, entrées-sorties, nombres aléatoires. . .
- ▶ Un programme impératif de type A est transformé systématiquement en un programme purement fonctionnel dans un type associé $M(A)$.
- ▶ $M(A)$ représente un type de **calculs**, l'évaluation du programme de type $M(A)$ va produire un effet et une valeur de type A .
- ▶ Deux opérateurs clé :
 - ▶ $\text{unit} : A \rightarrow M(A)$ explique comment une valeur pure (sans effets) est transformée en calcul
 - ▶ $\text{bind} : M(A) \rightarrow (A \rightarrow M(B)) \rightarrow M(B)$ explique comment les effets se propagent dans une construction **let $x = a$ in b** .
 - ▶ bind sert à interpréter la séquence d'instructions et l'application.
 - ▶ Propriétés algébriques : $\text{bind} (\text{unit } a) f = f a \dots$

Exemples de monades

Références

- ▶ Etat S (avec fonctions d'accès et de mise à jour)
- ▶ $M(A) := S \rightarrow A * S$
- ▶ `unit` $a := \mathbf{fun\ } s \Rightarrow (a, s)$
- ▶ `bind` $cf := \mathbf{fun\ } s \Rightarrow \mathbf{let\ } (a, s_1) := c \mathbf{ in\ } f a s_1$
- ▶ `!` $x := \mathbf{fun\ } s \Rightarrow \text{accès } s x$
- ▶ $(x := c) := \mathbf{fun\ } s \Rightarrow \mathbf{let\ } (a, s_1) := c \mathbf{ in\ } (a, \text{update } s_1 x a)$

Exemples de monades

Exception

- ▶ $M(A) := \text{option } A$
- ▶ $\text{unit } a := \text{Some } a$
- ▶ $\text{bind } c f := \text{match } c \text{ with } \text{Some } a \Rightarrow f a \mid \text{None} \Rightarrow \text{None}$
- ▶ $\text{fail} := \text{None}$
- ▶ $\text{try } c \text{ with fail} \Rightarrow d := \text{match } c \text{ with } \text{Some } a \Rightarrow \text{Some } a \mid \text{None} \Rightarrow d$

Exemples de monades

Générateurs aléatoires

Une stream infinie de tirages booléens indépendants vu comme une référence :

- ▶ $M(A) := \mathbb{B}^\infty \rightarrow A * \mathbb{B}^\infty$
- ▶ **flip** := **fun** $s \Rightarrow$ **let** $(b, s_1) := s$ **in** (b, s_1)

Continuations :

- ▶ $M(A) = (A \rightarrow C) \rightarrow C$
- ▶ **unit** $a :=$ **fun** $h \Rightarrow ha$
- ▶ **bind** $cf :=$ **fun** $h \Rightarrow c$ (**fun** $a \Rightarrow f a h$)

Plan

Preuve de programmes

Schéma structurel

Schéma général

Monades

Modules

Tactiques

Tactique automatique

Tactique par réflexion

Modules

- ▶ Extension du système de modules de Ocaml à la théorie des types.
 - ▶ Les modules ne sont pas des termes (\neq calculs de records avec définitions)
 - ▶ Les interfaces contiennent des définitions abstraites ou concrètes
 - ▶ Les modules peuvent être paramétrés (foncteurs)
- ▶ Meta-theorie faite par J. Courant (98)
- ▶ Implémentation par J. Chrzaszcz (02)
- ▶ Aspects non-logiques : notations, Hint databases, Extraction sont compatibles avec le système de modules.

Principes généraux

Interface

```
Module Type Modulename.  
Variable var : type.  
Definition def : type := term.  
Declare Module mod : Moduletype.  
End Modulename.
```

Implantation

```
Module Modulename.  
Variable var : type.  
Definition def : type := term.  
Module mod := modterm.  
End Modulename.
```

Utilisation des modules

Espace de nommage

- ▶ Accès aux éléments d'un module : *Modulename.varname*
- ▶ Possibilité d'ouvrir les modules : **Export** *Modulename*. permet d'accéder directement à *varname*.

Deux manières de contraindre le type d'un module :

- ▶ Vérifier la compatibilité :
Module *SetoidBool* <: *Setoid*.
- ▶ Masquer l'implantation :
Module *SetoidBool* : *Setoid*.

Exemple

Une interface de module :

```
Module Type Order.  
Variable A:Type.  
Variable leb:A→A→bool.  
End Setoid.
```

Une implantation de module :

```
Module OBool <: Order.  
Definition A:=bool.  
Definition leb (x y:A) := if x then y else true.  
End OBool.
```

Mélanger objets et preuves

```

Module Type Order.
Variable A:Type.
Variable leb:A→A→bool.
Hypothesis lebrefl :  $\forall x, \text{leb } x \ x = \text{true}$ .
Hypothesis lebtrans :  $\forall x \ y \ z,$ 
    leb  $x \ y = \text{true} \rightarrow \text{leb } y \ z = \text{true} \rightarrow \text{leb } x \ z = \text{true}$ .
End Order.

```

```

Module Type Setoid.
Variable A:Type.
Variable eqb:A→A→bool.
Hypothesis eqbrefl :  $\forall x, \text{eqb } x \ x = \text{true}$ .
Hypothesis eqbsym :  $\forall x \ y, \text{eqb } x \ y = \text{eqb } y \ x$ .
Hypothesis eqbtrans :  $\forall x \ y \ z,$ 
    eqb  $x \ y = \text{true} \rightarrow \text{eqb } y \ z = \text{true} \rightarrow \text{eqb } x \ z = \text{true}$ .
End Setoid.

```

Preuve des lemmes correspondants dans l'implantation.

Foncteurs

Definition d'un foncteur :

```

Module SetofOrder (O:Order) : Setoid.
Export O.
Definition A:=A.
Definition eqb (a b:A)
  := if leb a b then leb b a else false.
Lemma eqb_refl :  $\forall x$ , eqb x x = true.
unfold eqb; intros; rewrite (lebrefl x); simpl; auto.
Save.
Lemma eqbsym :  $\forall x y$ , eqb x y = eqb y x.
...
Save.
Lemma eqbtrans :  $\forall x y z$ ,
  eqb x y = true  $\rightarrow$  eqb y z = true  $\rightarrow$  eqb x z = true.
...
End SetofOrder.

```

Exemple :ordre produit

```

Module ProdOrder (O1 O2:Order) : Order.
Definition A := (O1.A*O2.A)%type.
Definition leb (x y:A) :=
  let (x1,x2):=x in let (y1,y2):=y in
  if O1.leb x1 y1 then O2.leb x2 y2 else false.
Lemma lebrefl :  $\forall x$ , leb x x = true.
...
Lemma lebtrans :  $\forall x y z$ ,
  leb x y = true  $\rightarrow$  leb y z = true  $\rightarrow$  leb x z = true.

```

Possibilité d'exporter certaines informations

```

Module ProdOrder (O1 O2:Order)
  : Order with Definition A:=(O1.A*O2.A)%type.

```


Plan

Preuve de programmes

Schéma structurel

Schéma général

Monades

Modules

Tactiques

Tactique automatique

Tactique par réflexion

Tactique automatique

- ▶ Toute tactique est une stratégie qui ultimement construit un terme de preuve vérifié par le noyau de Coq.
- ▶ Quelques tactiques automatiques de Coq :
 - ▶ `discriminate` : preuve d'inégalités de constructeurs
 - ▶ `auto` : recherche dans une base de lemmes
 - ▶ `intuition` : décomposition propositionnelle
 - ▶ `omega` : arithmétique de Pressburger
 - ▶ `ring` : simplification d'expressions dans les structures d'anneaux
- ▶ Les tactiques se composent :
 - ▶ séquence : `tac1 ; tac2`
 - ▶ répétition : `repeat tac`
 - ▶ essai : `try tac`
- ▶ L'efficacité est liée à la recherche de la solution **et** au temps de vérification du terme de preuve.

Tactique par réflexion

Principes généraux :

- ▶ Idée d'utiliser les capacités de calcul dans CIC pour faire des preuves **courtes**.
- ▶ La procédure de preuve est implantée et prouvée dans le système Coq puis réutilisée comme tactique.
- ▶ Expérimenté par S. Boutin pour la tactique `ring`.
- ▶ Mise en oeuvre de manière essentielle dans le théorème des 4 couleurs.
- ▶ A nécessité des optimisations de la fonction de réduction interne de Coq.

Tactique par réflexion

Mise en œuvre :

- ▶ Fonction $\text{dec} : A \rightarrow \text{bool}$ et une preuve
 $\text{dec_correct} : \forall a : A, \text{dec } a = \text{true} \rightarrow P a$
- ▶ Pour un terme clos a :

$$\text{dec_correct } a \text{ (refl true)} : P a$$

Cette preuve est bien typée quand $\text{dec } a \equiv \text{true}$

Tactique par réflexion

- ▶ Problèmes :
 - ▶ transformer le but en Pa
 - ▶ $a : A$ doit être clos (reification), mais il faut aussi que a reste petit
 - ▶ dec doit être prouvé (`dec_correct`) et se réduire efficacement
- ▶ Autres formes
 - ▶ Simplification : $nf : A \rightarrow A$ telle que $\forall a, P(nf\ a) \rightarrow Pa$
- ▶ Applications
 - ▶ Tactiques : ring, romega, (setoid) rewrite...
 - ▶ Interfaces entre Coq et d'autres systèmes en utilisant des traces (réécriture, prouveurs du premier ordre)

Exemple de réification

```
Inductive form : Set := T | F | Var : nat → form
  | Conj : form → form → form.
```

```
Definition env := list Prop.
```

```
Fixpoint find_env (e:env) (n:nat) {struct e} := ...
```

```
Fixpoint interp (e:env) (f:form) {struct f} :=
  match f with T ⇒ True | F ⇒ False
    | Conj A B ⇒ interp e A ∧ interp e B
    | Var n ⇒ find_env e n
  end.
```

```
Eval compute in
  (interp (True::False::(0=0)::nil)
    (Conj (Var 0) (Conj (Var 2) (Var 1))))).
= True ∧ 0 = 0 ∧ False : Prop
```

Fonction de simplification

Definition `simplform` : `form` \rightarrow `form` := ...

Lemma `simplcorrect` :

$\forall e f, \text{interp } e (\text{simplform } f) \rightarrow \text{interp } e f.$

Pour utiliser la simplification :

- ▶ Trouver *e* et *f* tels que *interp e f* est convertible avec le but (reification)
- ▶ Appliquer *simplcorrect*
- ▶ Calculer *simplform* et *interp*

Tactic language

Ltac conçu par D. Delahaye

- ▶ Une manière d'écrire des tactiques complexes sans utiliser de code ML.
- ▶ Un langage ad-hoc
 - ▶ Une classe de patterns adaptée au filtrage des buts (non-linéaire)

```
match goal with
  id:?A  $\wedge$  ?B |- ?A  $\Rightarrow$  case id; trivial
  | _  $\Rightarrow$  idtac
end.
```

```
match goal with |- context[?a+0]  $\Rightarrow$  rewrite ...
```

- ▶ notion spécifique de backtracking
 - ▶ les patterns sont essayés successivement tant que la partie droite échoue.
- ▶ constructions spécifiques : **fresh name**, **type of term** ...