Assistants de Preuve
Programmation fonctionnelle 2

Bruno Barras et Christine Paulin-Mohring

INRIA, Université Paris-Sud 11

16/12/08

Plan

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 := match x with
   p1 ⇒ t1  ... |  pn ⇒ tn
end.
```

Principes associés

- Analyse par cas :
  $\forall P : I \to \mathbf{Prop}, (\forall \vec{x_1}\, P\, p_1) \to (\forall \vec{x_2}, P\, p_2) \to \cdots \to (\forall \vec{x_n}, P\, p_n) \to \forall x : I, P\, x$
- Analyse de la fonction :
  $\forall P : I \to A \to \mathbf{Prop}, (\forall \vec{x_1}\, P\, p_1\, t_1) \to (\forall \vec{x_2}, P\, p_2\, t_2) \to \cdots \to (\forall \vec{x_n}, P\, p_n\, t_n)$
  $\to \forall x : I, P\, x\, (f\, x)$

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 ⇒ nil
             | cons b m ⇒ 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: ∀(a:A) (l res:list A),
   R_insert a l res → res = insert a l.
R_insert_correct: ∀(a:A) (l res:list A),
   res = insert a l → 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.

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 :
  - `unit` : $A \to M(A)$ explique comment une valeur pure (sans effets) est transformée en calcul
  - `bind` : $M(A) \to (A \to M(B)) \to M(B)$ 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 : `bind` (`unit` a) $f = f\ a$ ...

Exemples de monades

Références

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

Exemples de monades

Exception

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

Exemples de monades

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

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

Continuations :

- $M(A) = (A \to C) \to C$
- $\mathtt{unit}\, a := \mathbf{fun}\, h \Rightarrow h\, a$
- $\mathtt{bind}\, c\, f := \mathbf{fun}\, h \Rightarrow c\, (\mathbf{fun}\, a \Rightarrow f\ a\ h)$

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. Chrząszcz (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 : **Import** *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 : ∀x, leb x x = true.
Hypothesis lebtrans : ∀x y z,
     leb x y = true → leb y z = true → leb x z = true.
End Order.

Module Type Setoid.
Variable A:Type.
Variable eqb:A→A→bool.
Hypothesis eqbrefl : ∀x, eqb x x = true.
Hypothesis eqbsym : ∀x y, eqb x y = eqb y x.
Hypothesis eqbtrans : ∀x y z,
     eqb x y = true → eqb y z = true → eqb x z = 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 : ∀x, eqb x x = true.
unfold eqb; intros; rewrite (lebrefl x); simpl; auto.
Save.
Lemma eqbsym : ∀x y, eqb x y = eqb y x.
...
Save.
Lemma eqbtrans : ∀x y z,
     eqb x y = true → eqb y z = true → 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 : ∀x, leb x x = true.
...
Lemma lebtrans : ∀x y z,
     leb x y = true → leb y z = true → leb x z = true.
```

Possibilité d'exporter certaines informations

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

Tactique automatique

- Toute tactique est une stratégie qui ultimement construit un terme de preuve revérifié 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 proprositionnelle
  - `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 revérification 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 (originalement 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 $\texttt{dec} : A \to \mathit{bool}$ et une preuve $\texttt{dec\_correct} : \forall a : A, \texttt{dec}\ a = \texttt{true} \to P\ a$
- Pour un terme clos a :

$$\texttt{dec\_correct}\ a\ (\texttt{refl}\ \mathit{true}) : P\ a$$

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

Tactique par réflexion

- Problèmes :
  - transformer le but en $P\,a$
  - $a : A$ 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 : $\texttt{nf} : A \to A$ telle que $\forall a, P\,(\texttt{nf}\,a) \to P\,a$
- 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 → form := ...
Lemma simplcorrect :
    ∀e f, interp e (simplform f) → 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 ∧ ?B |- ?A ⇒ case id; trivial
    | _ ⇒ idtac
end.
match goal with |- context[?a+0] ⇒ rewrite ...
```

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