Assistants de Preuve
Programmation fonctionnelle 1

Bruno Barras et Christine Paulin-Mohring

Université Paris-Sud 11, INRIA

9/12/08

Plan

Plan

Point fixe

$\texttt{fix}\ f\ : B := t$

- Typage : $f : B \vdash t : B$
- Décroissance structurelle : il existe n tel que
  - $B \equiv (x_1 : A_1) \ldots (x_n : A_n) B'$
  - A_n un type inductif
  - occurrences de f dans t de la forme $f\ u_1 \ldots u_n$ avec u_n structurellement plus petit que x
    essentiellement une variable d'un motif issu d'un filtrage sur x.
- Réduction $\texttt{fix}\ f\ : B := t \longrightarrow t[f := \texttt{fix}\ f\ : B := t]$ seulement dans des expressions $\texttt{fix}\ f\ : B := t\ y_1 \ldots y_n$ et y_n commence par un constructeur.

Suffisant pour coder les combinateurs de récursion structurelle.

Réduction de point fixe

Conserver la normalisation forte : réduction de termes avec variables quelle que soit la stratégie

```
Fixpoint R (n:nat) : C :=
    match n with O ⇒ x | (S m) ⇒ f m (R m) end.
```

- On veut :
  - $R\,0 \longrightarrow x$
  - $R\,(\mathtt{S}\,m) \longrightarrow f\,m\,(R\,m)$
- Perte de normalisation si :
  $R\,n \longrightarrow$ **match** n **with** $0 \Rightarrow x \mid (\mathtt{S}\,m) \Rightarrow f\,m\,(R\,m)$ **end**.

Définir des fonctions récursives générales

`fix` $f\ x : C := t$ avec des appels non structurels.

```
Fixpoint merge (l1 l2:list) : list :=
  match (l1,l2) with
    []  , _     ⇒ l2 | _   , []    ⇒ l1
  |a::m1,b::m2 ⇒ if a < b then a::merge m1 l2
                  else b::merge l1 m2 end.
```

- Etendre le schéma de définitions (normalisation forte).
- Codage
  - Double récursion
  - Argument de décroissance structurelle

Codage de la récursion

Double récursion

```
Fixpoint merge (l1 l2:list) : list :=
  match l1 with  []  ⇒ l2
  | a::m1 ⇒ (fix maux (m: list) : list :=
       match m with []   ⇒ l1
              |b::m2 ⇒ if a < b then a::merge m1 m
                       else b::maux m2 end) l2 end.
```

Reductions :

- merge [] *l2* $\longrightarrow$ *l2*
- merge (*a* :: *m1*) $\longrightarrow$ **fix** maux*m* :=
  match *m* **with** [] $\Rightarrow$ *a* :: *m1*
  |*b* :: *m2* $\Rightarrow$ **if** *a* < *b* **then** *a* :: merge *m1 m*
  else *b* :: maux *m2 end*)
- merge (*a* :: *m1*) [] $\longrightarrow$ *a* :: *m1*
- merge (*a* :: *m1*) (*b* :: *m2*) $\equiv$ **if** *a* < *b* **then** *a* :: merge *m1* (*b* :: *m2*)
  else*b* :: merge (*a* :: *m1*) *m2*

Codage de la récursion

Argument structurel entier (mesure la taille de *l1@l2*).

```
Fixpoint maux (l1 l2:list) (n:nat) : list :=
  match (l1,l2) with
    []  , _ ,n      ⇒ l2 | _   , [],n      ⇒ l1
  | _   , _ ,0      ⇒ [] (* cas absurde si |l1@l2| <= n *)
  |a::m1,b::m2,S n ⇒
    if a < b then a::maux m1 l2 n
    else b::maux l1 m2 n end.
Definition merge (l1 l2:list)
  := maux l1 l2 (length l1 + length l2).
Lemma merge_prop : ∀l1 l2 n, length l1 + length l2 <= n ⇒
  maux l1 l2 n = merge l1 l2.
```

Le nombre de réductions de `maux` *l1 l2 n* ne dépend que de *l1* et *l2*.

Minimalité

```
Fixpoint min (p:nat→bool) : nat :=
    if p 0 then 0 else S (min (fun x ⇒ p (S x)))
```

Termine s'il existe n tel que $p\,n = \texttt{true}$.

```
Fixpoint min2 (p:nat→bool) (n:nat) : nat :=
  if p 0 then 0 else
     match n with O ⇒ O (* cas absurde si p n = true *)
                 | S m ⇒ S (min2 (fun x ⇒ p (S x)) m)
     end.
Lemma min2min : ∀n k, k < min2 p n → p k = false.
Lemma min2p : ∀n, p n = true → p (min2 p n) = true.
```

L'entier passé en argument borne le nombre de réductions du point fixe.

Récursion générale

- Relation bien fondée R (pas de suite infinie décroissante)
- Vérifier que les appels récursifs dans $f\ t$ sont sur des termes u tels que $R\ u\ t$.
- Les points fixes structurels correspondent à la bonne fondaison de la relation sous-terme.

Accessibilité

```
Inductive Acc (x:A) : Prop :=
  Acc_intro : (∀y, R y x → Acc y) → Acc x.
Definition well_founded : Prop := ∀x, Acc x.
```

- `Acc` correspond à un arbre bien fondé (chaque branche est finie) avec un branchement arbitraire.
- Un nœud de type `Acc` x a des fils pour chaque y tel que $R\ y\ x$.
- Si $p : Acc\ x$ et $q : R\ y\ x$ alors
  `accR` $q :=$ **match** p **with** `Acc_intro` $h \Rightarrow h\ y\ q$ **end** est de type `Acc` y et structurellement plus petit que p.
- $f\ x := \ldots (f\ u_1) \ldots (f\ u_n) \ldots$ est codé comme
- $f\ x\ (p : \texttt{Acc}\ x) := \ldots (f\ u_1\ (\texttt{accR}\ ?_{R\,u_1\,x})) \ldots (f\ u_n\ (\texttt{accR}\ ?_{R\,u_n\,x})) \ldots$

Réductions

```
Variable B : Type.
Variable F : ∀x, (∀y, R y x → B) → B.
Fixpoint facc (x:A) (p:Acc x) {struct p} : B :=
   F x (fun y (q:R y x) ⇒ facc y (accR q)).
Variable rwf : well_founded.
Definition f (x:A) : B := facc x (rwf x).
```

- Réduction de $\texttt{facc}\, x\, (\texttt{Acc\_intro}\, h)$ vers $F\ x\ (\textbf{fun}\ y\ (q : R\ y\ x) \Rightarrow \texttt{facc}\, y\, (h\ y\ q))$
- $p : \texttt{Acc}\, x$ est de type **Prop** et disparait à l'extraction.
- Les preuves de $\texttt{Acc}\, x$ sont souvent opaques et ne se réduisent pas ...
- Utiliser $\forall p : \texttt{Acc}\, x, p = \texttt{Acc\_intro}\, (\textbf{fun} y\ (q : R\ y\ x) \Rightarrow \texttt{accR}\ q)$ prouvé par élimnation dépendante sur p.
- Preuve de $f\ x = F\ x\ (\textbf{fun}\ y\ q \Rightarrow f\ y)$, hypothèse d'extensionnalité de F : $\forall\ x\ f\ g, (\forall y\ (p : R\ y\ x), f\ y\ p = g\ y\ p) \rightarrow F\ x\ f = F\ x\ g$

Minimalité

- $x \prec y := p\,y = \texttt{false} \wedge x = \texttt{S}\,y$
  un chemin issu de x est fini s'il existe y tel que $x \leq y \wedge p\,y = \texttt{false}$
- Définition ad-hoc

```
Inductive bound (p:nat→bool) : Prop :=
   bound0 : p 0 = true → bound p
 | boundS : bound (fun x ⇒ p (S x)) → bound p.
```

- Predecesseur :

```
Definition boundP p (bp: bound p) (H:p 0 = false)
  : bound (fun x ⇒ p (S x))
  := match bp with
       bound0 H1 ⇒ match true_false (p 0) H1 H with end
     | boundS bp'⇒ bp' end.
```

- Point fixe

```
Fixpoint min (p:nat→bool) (bp:bound p) : nat :=
  match bool_dec (p 0) false with
    left H ⇒ S (min (fun x ⇒ p (S x)) (boundP p bp H))
  |  _ ⇒ 0 end.
```

Plan

Fonctions partielles

Fonction $f : A \to B$ définie uniquement sur un domaine D.

- Trouver une valeur par défaut dans B pour $x \notin D$
  Arbitraire si B est une variable : tête de liste
- Renvoyer une valeur dans le type `option` B.

```
Inductive option:Type :=
  Some : B → option | None : option.
```

  - Le programme teste si l'entrée est dans le domaine
  - Analogue à une exception
  - $\forall x, D\,x \Rightarrow g\,x = \texttt{Some}\,(f\,x)$.
- Argument de domaine : $\forall x, x \in D \to B$
  - Dépendance non calculatoire : $D : A \to Prop$.
  - Argument supplémentaire à chaque appel
  - Proof irrelevance : $f\,x\,d_1 = f\,x\,d_2$

Plan

Principes

- Un type ou une famille de types défini par ses constructeurs
- Toute valeur (terme clos normal) commence par un constructeur
  Construction par filtrage (**match** . . . **with** . . . **end**)
- Plus grand point fixe $\nu X.F\,X$: objets infinis
  - Co-iteration : $\forall X, (X \subseteq FX) \rightarrow X \subseteq \nu X.F\,X$
  - Co-recursion : $\forall X, (X \subseteq F\,(X + \nu X.FX)) \rightarrow X \subseteq \nu X.FX$
  - Co-fixpoint : $f := H(f) : \nu X.FX$
    Les appels récursifs à f sont gardés par des constructeurs de $\nu X.FX$.

Le cas des streams

```
Variable A : Type.
CoInductive Stream : Type := Cons : A → Stream → Stream.
Definition  hd (s:Stream) : A
  := match s with (Cons a _) ⇒ a end.
Definition  tl (s:Stream) : Stream
  := match s with (Cons a t) ⇒ t end.
CoFixpoint cte (a:A) := Cons a (cte a).
Lemma cte_hd : ∀a, hd (cte a) = a.
  trivial.
Lemma cte_tl : ∀a, tl (cte a) = cte a.
  trivial.
Lemma cte_eq : ∀a, cte a = Cons a (cte a).
  intros; transitivity (Cons (hd (cte a)) (tl (cte a)));
  trivial.
  case (cte a); auto.
```

Fonction non gardée

Filtre

```
Variable p:A→bool.
CoFixpoint filter (s:Stream) : Stream :=
   if p (hd s) then Cons (hd s) (filter (tl s))
   else filter (tl (p s))
```

Risque d'un objet clos de type `Stream` qui ne se réduit pas sur un constructeur.

Famille coinductive

Notion de preuve infinie :

```
CoFixpoint cte2 (a:A) := Cons a (Cons a (cte2 a)).
```

Comment prouver `cte` a = `cte2` a ?
Egalité extensionnelle des éléments de la stream.

```
CoInductive eqS (s t:Stream) : Prop :=
   eqS_intros : hd s = hd t → eqS (tl s) (tl t)
                → eqS s t.
```

Preuve

```
CoFixpoint cte_p1 a : eqS (cte a)  (cte2 a) :=
   eqS_intro (refl a) (cte_p2 a)
with cte_p2 a : eqS (cte a) (Cons a (cte2 a)) :=
   eqS_intro (refl a) (cte_p1 a).
```