Représentation des langages fonctionnels

Joëlle Despeyroux
INRIA Sophia-Antipolis

Module "Mécanisation des Preuves"
Dea d'informatique, Nice, janvier-mars 2000

Représentation des variables

Exemple: le λ-calcul simple non typé.
Expressions:

$$exp\ e \ ::= x \mid \lambda x.e \mid (e_1\ e_2) \qquad x \in \mathcal{V}ar$$

Syntaxe abstraite:

sortes exp, var

sous sortes var $\subset$ exp

constructeurs

$$\begin{array}{lcccl} var & : & & \rightarrow & \text{var} \\ app & : & \text{exp} \times \text{exp} & \rightarrow & \text{exp} \\ lam & : & \text{var} \times \text{exp} & \rightarrow & \text{exp} \end{array}$$

Réduction:

$$(\lambda x.e\ e') \hookrightarrow_\beta e[e'/x]$$

Rappels:

Problème de **capture de variable**:
$\lambda x.\lambda x.x \equiv \lambda x'.\lambda x.x$ ou $\lambda x.\lambda x'.x$?

Solution: **α-conversion:** $\lambda x.x \equiv \lambda y.y$ et $\lambda x.\lambda x.x \equiv \lambda x.\lambda y.y$.

Autres réductions:

$$\begin{array}{rcl} \lambda x : A.M \hookrightarrow_\alpha \lambda y : A.M[y/x] & - & \text{renommage} \\ \lambda x : A.(M\ x) \hookrightarrow_\eta M \text{ si } x \notin M & - & \text{Ex : } \lambda x : i.\ (S\ x) = S \end{array}$$

I. Codes de Bruijn

Les variables liées sont remplacées par des entiers indiquant la profondeur de leur lieur:

$$\lambda x.\lambda y.\lambda z.(x\ y\ z) \text{ est représenté par } \lambda\lambda\lambda(2\ 1\ 0)$$

Problème: il faut décaler tous les indices d'une expression que l'on plonge dans une autre, typiquement lors d'une substitution.

- Les spécifications deviennent vite obscures.
- Les preuves sont artificiellement longues: 75% des développements concernent les codes de Bruijn.

Morale:
Technique à utiliser (seulement!) pour spécifier complètement un système implémenté en utilisant les de Bruijn. Ex: Coq en Coq.

Le λ-calcul (*)

```
Inductive l : Set :=
    Ref : nat -> l
  | Abs : l -> l
  | App : l -> l -> l.

Lemma test : (n,m:nat){le n m}+{gt n m}.
Lemma compare : (n,m:nat){lt n m}+{n=m}+{gt n m}.

Definition relocate := [i,k,n:nat]
   Cases (test k i) of
     (* k<=i *) (left _) => (plus n i)
   | (* k>i  *) _         => i
   end.

Fixpoint  lift_rec [L:l] : nat -> nat -> l
:= [k,n:nat]
   Cases L of
     (Ref i)   => (Ref (relocate i k n))
   | (Abs M)   => (Abs (lift_rec M (S k) n))
   | (App M N) => (App (lift_rec M k n) (lift_rec N k n))
   end.

Definition lift := [n:nat][M:l](lift_rec M O n).
```

(*) Développement du à Gérard Huet, Lemme du cube, nov. 94.

Substitution

```
Definition insert_Ref := [N:l][i,k:nat]
   Cases (compare k i) of
     (* k<i *)  (inleft  (left _)) => (Ref (pred i))
  |  (* k=i *)  (inleft  _)         => (lift k N)
  |  (* k>i *)   _                  => (Ref i)
  end.

Fixpoint subst_rec [L:l] : l -> nat -> l :=
[N:l][k:nat]
  Cases L of
    (Ref i)     => (insert_Ref N i k)
  | (Abs M)     => (Abs (subst_rec M N (S k)))
  | (App M M') => (App (subst_rec M N k)
                      (subst_rec M' N k))
  end.

Definition subst := [N,M:l](subst_rec M N O).
```

Reduction

```
Inductive red1 : l -> l -> Prop :=
   beta:      (M,N:l)(red1 (App (Abs M) N) (subst N M))
 | abs_red:   (M,N:l)(red1 M N) -> (red1 (Abs M) (Abs N))
 | app_red_l: (M1,N1:l)(red1 M1 N1) ->
              (M2:l)(red1 (App M1 M2) (App N1 M2))
 | app_red_r: (M2,N2:l)(red1 M2 N2) ->
              (M1:l)(red1 (App M1 M2) (App M1 N2)).
```

Cloture transitive de red1:

```
Inductive red : l -> l -> Prop :=
   one_step_red: (M,N:l)(red1 M N) -> (red M N)
 | refl_red: (M:l)(red M M)
 | trans_red: (M,N,P:l)(red M N)->(red N P) -> (red M P).
```

Lemmes:

```
Lemma red_abs :
   (M,M':l)(red M M') -> (red (Abs M) (Abs M')).


Lemma red_app :
   (M,M',N,N':l)(red M M') -> (red N N') ->
   (red (App M N) (App M' N')).


Lemma red_beta :
  (M,M',N,N':l)(red M M') -> (red N N') ->
  (red (App (Abs M) N) (subst N' M')).
```

Reduction (suite)

Reduction parallèle:

```
Inductive par_red1 : l -> l -> Prop :=
   par_beta :
      (M,M':l)(par_red1 M M') -> (N,N':l)(par_red1 N N') ->
      (par_red1 (App (Abs M) N) (subst N' M'))
 | ref_par_red : (n:nat)(par_red1 (Ref n) (Ref n))
 | abs_par_red :
      (M,M':l)(par_red1 M M') ->
      (par_red1 (Abs M) (Abs M'))
 | app_par_red :
      (M,M':l)(par_red1 M M') -> (N,N':l)(par_red1 N N') ->
      (par_red1 (App M N) (App M' N')).


Lemma refl_par_red1 : (M:l)(par_red1 M M).

Inductive par_red : l -> l -> Prop :=
   one_step_par_red :
      (M,N:l)(par_red1 M N) -> (par_red M N)
 | trans_par_red :
      (M,N,P:l)(par_red M N) -> (par_red N P) ->
      (par_red M P).
```

II. Variables et paramètres

Les variables liées sont représentées par des *variables*, les variables libres par des *paramètres*.

$$exp\ e \ ::= x \mid p \mid \lambda x : e_1.\ e_2 \mid (e_1\ e_2) \quad - x \in \mathcal{V}ar,\ p \in \mathcal{P}ar$$

Lorsque l'on descend sous un lambda, on remplace toutes les occurrences de la *variable* liée par un *paramètre* frais.

Technique due à J. Mc Kinna et R. Pollack, utilisée pour spécifier Lego en Lego (Lego est un système proche de Coq).

```
Parameter PP, VV : Set.
 Axiom PP_eq_dec : (p,q:PP){(p=q)}+{(~p=q)}.
 Axiom VV_eq_dec : (x,y:VV){(x=y)}+{(~x=y)}.

Inductive L : Set :=
    var : VV -> L
  | par : PP -> L
  | lam : VV -> L -> L -> L
  | app : L -> L -> L.
```

Le λ-calcul

$$exp\ e \ ::= x \mid p \mid \lambda x.\ e \mid (e\ e') \ \text{ - } x \in \mathcal{V}ar,\ p \in \mathcal{P}ar$$

Substitutions textuelles, et prédicats auxilaires (non donnees ici):

$$e[a/p],\ e[a/v],\ p \notin e$$

Termes clos:

par: $\text{Closed}(p)$

lam: $\dfrac{\text{Closed}(e[p/x])}{\text{Closed}(\lambda x.e)}$

app: $\dfrac{\text{Closed}(e_1) \quad \text{Closed}(e_2)}{\text{Closed}(e\ e') : t}$

Alternative donnant un principe d'élimination plus fort:

lam: $\dfrac{(\forall p : \mathcal{P}ar.\ \text{Closed}(e[p/x]))}{\text{Closed}(\lambda x.e)}$

Réduction

$$\mathrm{red}_\beta : \ (\lambda x.m \ \ n) \hookrightarrow m[n/x] \quad if \ (Closed \ n)$$

$$\mathrm{red}_\lambda : \ \frac{e[p/x] \hookrightarrow e'[p/y]}{\lambda x.e \hookrightarrow \lambda y.e'} \quad p \notin e, \ p \notin e'$$

$$\mathrm{red}_{app1} : \ \frac{e_1 \hookrightarrow e_1'}{(e_1 \ e_2) \hookrightarrow (e_1' \ e_2)}$$

$$\mathrm{red}_{app2} : \ \frac{e_2 \hookrightarrow e_2'}{(e_1 \ e_2) \hookrightarrow (e_1 \ e_2')}$$

Alternative:

$$\mathrm{red}_\lambda : \ \frac{(\forall p : \mathcal{P}ar. \ e[p/x] \hookrightarrow e'[p/y])}{\lambda x.e \hookrightarrow \lambda y.e'}$$

- Les spécifications sont lisibles.
- Les preuves sont relativement concises.
- Les occurences de $Closed$ et $\notin$ polluent un peu les développements.

III. Variables définies à α-conversion près

Technique due à A. Gordon.
On définit d'abord les termes de Bruijn. Pour le λ-calcul:

```
Inductive term: Set :=
     con: cons -> term
   | var: var -> term
   | bvar: nat -> term
   | abs: term -> term
   | app: term -> term -> term .
```

Puis des fonctions sur les termes de Bruijn:

```
Fixpoint abst[u:term]: nat->var->term := [i:nat] [x:var]
   Cases u of
        (con c) => (con c)
      | (var y) => Cases (dec_var x y) of
                        (left _) => (bvar i)
                      | (right _) => (var y)
                   end
      | (bvar j) => (bvar j)
      | (abs t)   => (abs (abst t (S i) x))
      | (app u1 u2) => (app (abst u1 i x) (abst u2 i x))
   end.
```

Le λ-calcul

```
Fixpoint inst[u:term]: nat->term->term := [i:nat] [v:term]
   Cases u of
        (con c)  => (con c)
      | (var y)  => (var y)
      | (bvar j) => Cases (dec_nat i j) of
                          (left _)  => v
                        | (right _) => (bvar j)
                      end
      | (abs t)  => (abs (inst t (S i) v))
      | (app u1 u2) => (app (inst u1 i v) (inst u2 i v))
   end.

Definition lambda: var -> term -> term :=
   [x:var] [u:term](abs (abst u O x)).

Inductive Term: term ->Prop :=
     Con: (c:cons)(Term (con c))
   | Var: (x:var)(Term (var x))
   | Lam: (u:term) (x:var) (Term u) -> (Term (lambda x u))
   | App: (u1, u2:term) (Term u1) -> (Term u2) ->
          (Term (app u1 u2)) .

Definition subst: term -> term -> var -> term :=
   [t, u:term] [x:var](inst (abst t O x) O u).
```

Inductions sur la longueur des termes

```
Lemma alpha_lambda:
  (m:term) (x, y:var) ~(In var (dfv m) y) ->
  (lambda x m) = (lambda y (subst m (var y) x)).

Lemma length_ind: (P:term -> Prop)
  ((v:term) ((u:term) (smaller u v) ->(P u)) ->(P v)) ->
  (u:term)(P u).

Lemma length_ind1: (P:term -> Prop)
  ((v:term) ((u:term) (Term u) -> (smaller u v) -> (P u)) ->
    (Term v) ->(P v)) ->
  (u:term) (Term u) ->(P u).

Lemma length_ind2:
  (P:term -> Prop)
  ((c:cons)(P (con c))) ->
  ((x:var)(P (var x))) ->
  ((u:term) (Term u) ->
    ((v:term) (Term v) -> (length u)=(length v) ->(P v)) ->
    (x:var)(P (lambda x u))) ->
  ((u1, u2:term) (Term u1) -> (P u1) -> (Term u2) ->
    (P u2) -> (P (app u1 u2))) ->
  (d:term) (Term d) ->(P d).
```

Induction

```
(*  u:Term -> [x/y]u:Term  *)
Lemma Term_subst_cons:
  (u:term) (Term u) -> (x:var) (v:term) (Term v) ->
  (Term (subst u v x)).

Lemma induction:
  (P:term -> Prop)
  ((c:cons)(P (con c))) ->
  ((x:var)(P (var x))) ->
  ((exT (Ensemble var) [X:(Ensemble var)]
     (Finite var X) /\
     ((u:term)(x:var) ~(In var X x) -> (Term u) -> (P u) ->
      (P (lambda x u))))) ->
  ((u,v:term) (Term u) -> (P u) -> (Term v) -> (P v) ->
    (P (app u v))) ->
  (d:term) (Term d) ->(P d).
```

Dans la suite du développement, on ne voit plus de code de Bruijn, sauf si l'on introduit de nouvelles fonctions sur les termes, auquel cas il faut prouver que ces fonctions sont définies à α-conversion près.

IV. Syntaxe abstraite d'ordre supérieur

Church(40): 'on peut représenter toute construction liante d'un langage en utilisant la λ-abstraction'.

$$exp\ e\ \ ::= x \mid (lam\ \lambda x.e) \mid (app\ e_1\ e_2)$$

où $\lambda x.e$ est une lambda-expression du "Logical Framework" choisi (typiquement LF) pour formaliser le language.

Fonction de représentation de *exp* dans LF:

$$\ulcorner \_ \urcorner : exp \rightarrow \mathrm{LF}$$

Les variables de *exp* sont représentées par les vars de LF. Les vars liées dans *exp* le seront donc dans LF.

$$lam : (exp \rightarrow exp) \rightarrow exp$$
$$app : exp \rightarrow exp$$

$$\ulcorner x \urcorner = x$$
$$\ulcorner \lambda x.e \urcorner = (lam\ \lambda x : exp. \ulcorner e \urcorner)$$
$$\ulcorner (e\ e') \urcorner = (app\ \ulcorner e \urcorner\ \ulcorner e' \urcorner)$$

L'application objet (dans *exp*) utilise l'application méta (de LF).

β-réduction: $(app\ (lam\ M)\ N) \hookrightarrow (M\ N)$

Adéquation. Comme d'habitude, il faut prouver l'adéquation de la représentation de la syntaxe et des sémantiques.