Preuve Formelle Mecanisée

MOPS/ENSIIE
2010-2011

Systèmes formels Lambda Calcul Pur

O. Pons

Revision : 33
Date : $2011-09-3011:24:12+0200$(*ven*.30*sept*.2011)

Histoire

Modèle de calcul (Church, $\simeq$ 1930)

- Fondation des mathématiques, mécanisation de la logique, $\rightsquigarrow$ « échec »(paradoxe : Kleen-Rosser)
- Théorie de la fonctionnalité
- **language de programmation** de bas niveau

- Calculabilité (fonctions récursives, Turing-calculables)
- Programmation fonctionnelle
  - λ-calcul pur $\rightsquigarrow$ **`Lisp`**,**`Scheme`**,**`JavaScript`** etc.
  - λ-calcul typés $\rightsquigarrow$ famille **`ML`**, etc.

Lambda calcul (Syntaxe)

- Ensemble dénombrable de variables $V = \{x, y, z, \ldots\}$
- Termes du λ-calcul de la forme :

x		(variable)
$(\lambda x.M)$	M un terme	(abstraction)
$(M\ N)$	M et N des termes	(application)

Lambda calcul (Explications et exemples)

Explications ;

- $\lambda x.M$: fonction f telle que $f(x) = M$
- $x \mapsto M$ (notation *bourbakiste*)
- « fonction qui à x associe M »
  - x est la **variable liée** (*argument)*
  - *M le corps*

Exemples :

- $\lambda x.x$ « fonction identité »
- $\lambda x.c$ « fonction constante c »
- $(f\ x)$ « f appliquée à x »
- $\lambda f.\lambda x.(fx)$ « applique son 1^{er} argument au 2^{em} »
  notez l'omission des parenthèses superflues

Lambda calcul (Syntaxe)

- Alphabet : $A = V \cup \{\lambda, (,)\}$
- Définition Inductive :

  Le plus petit ensemble Λ tel que :

  - $x \in V$ alors $x \in \Lambda$
  - $M \in \Lambda$ et $N \in \Lambda$ alors $(M\ N) \in \Lambda$
  - $M \in \Lambda$ et $x \in X$ alors $(\lambda x.M) \in \Lambda$

Structure

Naturellement, raisonnement par induction sur les λ-termes :
Soit P une “propiété” de Λ,

- $P(x)$ pour tout $x \in X$
- $P(M)$ et $P(N) \Rightarrow P((M\ N))$
- $P(M) \Rightarrow P(\lambda x.M)$

Alors $P(M)$ pour tout $M \in \Lambda$

Lambda calcul (Syntaxe : Codage en caml)

```
(* definition des termes *)
type variable = string;;

type lambda=
  Var of variable
  |App of lambda*lambda
  |Lam of variable * lambda;;
```

Lambda calcul (Syntaxe : en Coq)

```
Coq <
Inductive lambda :Set :=
  var:nat->lambda
  |app:lambda->lambda->lambda
  |lam:nat ->lambda->lambda.Coq


lambda is defined
lambda_rect is defined
lambda_ind is defined
lambda_rec is defined
```

Lambda calcul (Syntaxe : en Coq)

```
Coq < Check lambda_ind.
lambda_ind
  : forall P : lambda -> Prop,
    (forall n : nat, P (var n)) ->
    (forall l : lambda, P l -> forall l0 : lambda,
                               P l0 -> P (app l l0)) ->
    (forall (n : nat) (l : lambda), P l ->
                                    P (lam n l)) ->
    forall l : lambda, P l
```

Variables libres, variables liées

- Occurrence de variable liée : dans la portée du lieur λ
  Ex. : $\lambda f.(f\ x)$, $(x \lambda x.(f\ x))$, $((\lambda x.x)\lambda f.(f x))$
- Variable Libre : non liée.
- Definitions inductives :
  - Ensemble des Variables libres d'un λ-terme t ($FV(t)$) (free variables) :
    - $FV(x) = \{x\}$
    - $FV(\lambda x.M) = FV(M) - \{x\}$
    - $FV((M\ N)) = FV(M) \cup FV(N)$
  - Ensemble des variables liées d'un λ-terme t ($BV(t)$) (bound variables)
    - $BV(x) = \{\}$
    - $BV(\lambda x.M) = BV(M) \cup x$
    - $BV(M\ N) = BV(M) \cup BV(N)$

Variables libres, variables liés (codage en caml)

```
let rec varLibres lambdaTerm = match lambdaTerm with
  Var (x) ->[x]
  |App (n,m)->union (varLibres n) (varLibres m)
  |Lam (x,m)->remove x  (varLibres m);;
```

```
let rec varLiees lambdaTerm = match lambdaTerm with
  Var (x) ->[]
  |App (n,m)->union (varLiees n) (varLiees m)
  |Lam (x,m)->add x  (varLiees m);;
```

Variables libres et variables liées

- Une même variable x dans un terme t :
  des occurences libres, d'autres liées

  Ex. : $(x\ \lambda x.x)$ ou $(y((\lambda x.(xy))(y\ x)))$
- dans $\lambda x.M$, x « muet »
  Ex. : $\lambda x.x$ et $\lambda y.y$ équivalentes pour identité

Substitution

Remplacement de variable par λ-terme dans λ-terme.

- $M[y := L]$ (ou $M[y \mapsto L]$ ou $M[L/y]$)
  λ-terme obtenu en substituant par L toutes les **occurrences libres** de y dans M
- Définie par induction :

$$x[y := L] = \begin{cases} L \text{ si } x = y \\ x \text{ sinon} \end{cases}$$

$$(\lambda x.M)[y := L] = \begin{cases} (\lambda x.M) \text{ si } x = y \\ (\lambda x.M[y := L]) \text{ sinon} \end{cases}$$

$$(M\ N)[y := L] = (M[y := L]\ N[y := L])$$

Substitution et capture de variables

- Exemples
  1. $(\lambda x.((x\ y))$ et $(\lambda t.(t\ y))$: même fonction

     $(\lambda x.(x\ y))[y := (z\ x)] \mapsto (\lambda x.(x\ (z\ x)))$ (capture)

     $(\lambda t.(t\ y))[y := (z\ x)] \mapsto (\lambda t.(t\ (z\ x)))$

  2. $(fun\ x -> \ x + y)[y := 4] \mapsto (fun\ x -> \ x + 4)$ (ajoute 4)

     $(fun\ x -> \ x + y)[y := x] \mapsto (fun\ x -> \ x + x)$ (double)

- Définition :
  La substitution $M[x := M]$ est dite **sure** si
  $BV(M) \cap FV(N) = \Phi$.
- Se limiter aux substitutions sures.
  Procéder à d'éventuels renommages ($\alpha - conversions$).

Variables, α-conversion

Relation α-réduction ($\rightarrow_\alpha$) :

$$\lambda x.M \rightarrow_\alpha \lambda y.M[x := y] \text{ pour tout } y \notin FV(E)$$

Relation α-conversion ($=_\alpha$) :

- côture de $\rightarrow_\alpha$, réflexive, symétrique, transitive, par contexte
- *i.e.* plus petite relation reflexive, symetrique transitive, stable par contexte qui contient $\rightarrow_\alpha$.
- α-conversion : congruence

Variables, α-conversion

Proposition.

M et N α-convertibles $\Leftrightarrow$

- $M = N = x \in X$
- $M = (F_1\ G_1)$ et $N = (F_2\ G_2)$ ou
  $F_1 =_\alpha F_2$ et $G_1 =_\alpha G_2$
- $M = \lambda x.F_1$ et $N = \lambda y.F_2$ ou
  $F_1[x \mapsto z] =_\alpha F_2[y \mapsto z]$ avec $z \notin FV(F_1) \cup FV(F_2)$

Dém. par recurrence

Corollaire.

$=_\alpha$: décidable

Désormais on travaille modulo $=_\alpha$-conversion

Renommage et substitution (codage en caml)

```
(* renommer var *)
let renomme  var listeVar =
  let rec renommeAux j=
    let varj=(var^(string_of_int j)) in
      if (member varj listeVar) then
      renommeAux (j+1)
      else varj
    in
   renommeAux 0;;
```

Renommage et substitution (codage en caml)

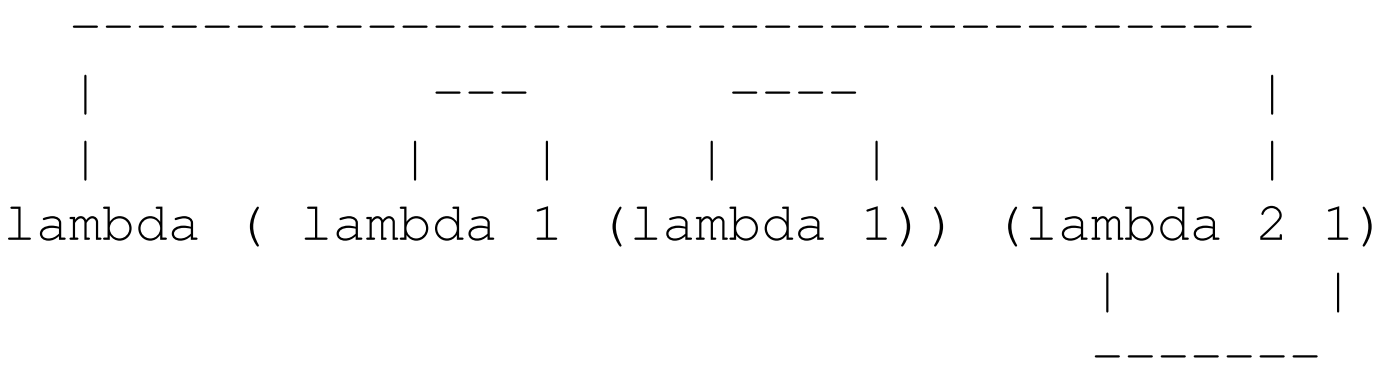
```
(* substituer terme a var dans exp *)

let rec substituer exp var terme = match exp with
  |Var(x) -> if (x=var) then terme else exp
  |App (n,m)->App((substituer n var terme),
                  (substituer m var terme))
  |Lam (x,m)->
     (* pas d'occurence libre on en fait rien *)
     if (not (member var (varLibres exp)))
     then exp
     else
       (* si capture on renome *)
       if (member x (varLibres terme)) then
         let newV=renomme x (varLibres terme) in
         let newCorps=substituer m x (Var(newV)) in
           Lam(newV,(substituer newCorps var terme))
       else
       (* sinon *)
          Lam(x,(substituer m var terme));;
```

Notation alternative : indices de De Bruijn

- Du nom de leur inventeur : Nicolaas Govert de Bruijn
- But : Éliminer les noms de variable de la représentation des termes
  - Termes écrits invariants par rapport à l'α*conversion*
  - Vérifier l' α-équivalence $\longrightarrow$ vérifier l'égalité syntaxique
- Comment :
  - Indice de De Bruijn : un entier naturel représentant une occurrence d'une variable dans un λ-terme
  - dénote le nombre de lieurs (λ) entre cette occurrence et le lieur qui lui correspond.

Notation alternative : indices de De Bruijn

- Exemples :
  - $\lambda x.\lambda y.x$ s'écrit $\lambda\lambda 2$
    Le lieur pour l'occurrence de x est le second λ.
  - $\lambda x.\lambda y.\lambda z.xz(yz)$ s'écrit $\lambda\lambda\lambda 31(21)$.
  - $\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.zx)$ s'écrit $\lambda(\lambda 1(\lambda 1))(\lambda 21)$.
    On peut donner une représentation graphique :



Réduction

- Idée : Exprimer l'évaluation de l'application d'un λ-terme(abstraction, représentant "une fonction") à un λ-terme
- Définition : **Redex** application d'une abstraction à un terme :

$$((\lambda x.M)N)$$

- β-réduction d'un redex : $((\lambda x.M)N) \rightarrow_\beta M[x := N])$ si la substitution est sure

Réduction

Étendus aux sous-termes (passage au contexte) :

$(\lambda x.M) \rightarrow_\beta (\lambda x.M')$ si $M \rightarrow_\beta M'$

$(M\ N) \rightarrow_\beta (M'\ N)$ si $M \rightarrow_\beta M'$

$(M\ N) \rightarrow_\beta (M\ N')$ si $N \rightarrow_\beta N'$

- β-réduction(plusieurs étape de calcul) :
  - $\rightarrow_\beta^*$:clôture transitive et reflexive de $\rightarrow_\beta$
  - $\rightarrow_\beta^* = \cup_{n \in N} \rightarrow_\beta^n$
- Exemple :
  $((\lambda x.(\lambda y.(xy))b)c) \rightarrow_\beta^* (bc)$ car
  $((\lambda x.(\lambda y.(xy))b)c) \rightarrow_\beta ((\lambda y.(by))c) \rightarrow_\beta (bc)$

Forme Normale

- Définition (Forme normale)
  - Terme M en **forme normale** : si il ne contient aucun rédex.
  - *i.e.* ne peut plus être β-réduit
  - il n'existe aucun N tel que $M \rightarrow_\beta N$
  - *i.e.* calcul terminé
  - Un λ-terme$N$, forme normale d'un λ-terme$M$ si
    1. $M \rightarrow_\beta^\star N$
    2. N en forme normale
- Exemple
  $(b\ c)$ forme normale de $((\lambda x.(\lambda y.(xy))b)c)$ car :
  $((\lambda x.(\lambda y.(xy))b)c) \rightarrow_\beta^* (b\ c)$ et $(b\ c)$ irréductible

Forme normale

- Définition (Terme normalisable)
  Un terme E est normalisable si il existe un terme E' en forme normale telle que $E \rightarrow_\beta^* E'$.
- Tous terme normalisable ?
  - Exemples :

$$\begin{array}{rcll} \Delta & = & \lambda x.(x\ x) & \text{forme normale} \\ \Omega & = & (\Delta\ \Delta) & = \quad (\lambda x.(x\ x)\ \Delta) \\ & & & \rightarrow_\beta \quad (x\ x)[x := \Delta] \\ & & & = \quad (\Delta\ \Delta) \quad = \quad \Omega \end{array}$$

  - Un λ-terme n'a pas toujours de forme normale

Stratégie de réduction

- Il existe plusieurs facons de réduire un terme.
- Exemple

$$\begin{array}{ccccc} ((\lambda x.((\lambda y.(x\ y)u))z)) & \rightarrow_\beta & ((\lambda y.(z\ y))u) & \rightarrow_\beta & (z\ u) \\ \downarrow_\beta & & & & \\ ((\lambda x.(x\ u))z) & & & & \\ \downarrow_\beta & & & & \\ (z\ u) & & & & \end{array}$$

- Les reductions même t-elle toujours au même résultat ?

- **Théorème.** Church-Rosser
  Si $E \rightarrow_\beta^* E_1$ et $E \rightarrow_\beta^* E_2$, alors il existe E' tel que $E_1 \rightarrow_\beta^* E'$ et $E_2 \rightarrow_\beta^* E'$.
- Ce théorème assure l'unicité de la forme normale
  i.e. Même résultat quel que soit l'ordre des calculs

Stratégie de réduction

- exemple

$$\lambda y.y \leftarrow_\beta (\lambda x.\lambda y.y\ \Omega) \rightarrow_\beta \cdots$$

- Suivant stratégie, forme normale atteinte. . . ou pas !
- existe t'il une stratégie gagnante ?
  - Réduction normale d'un λ-terme :
    appliquer la β-réduction au rédex le plus a gauche.
  - Théorème (Curry)
    Si E est normalisable vers une forme normale N alors, la réduction normale de E mène aussi à N.

    i.e. : La réduction normale d'un terme normalisable termine toujours (mais n'est pas toujours la plus rapide !)

Réduction (codage en caml)

```
let estRedex terme = match terme with
  |App(Lam(_,_),_) ->true
  |_ -> false;;

exception NOTREDEX;;
let  betaReducRedex redex = match redex with
      |App(Lam(x,m),n) ->substituer m x n
      |_ ->raise NOTREDEX;;
```

Réduction (codage en caml)

```
exception IRREDUCTIBLE;;

let rec reduc1Normale terme = match terme with
   Var(x)->raise IRREDUCTIBLE
  |Lam(x,m)->Lam(x,(reduc1Normale m))
  |App(n,m)->
     if (estRedex terme)
      then  (betaReducRedex terme)
      else
         try
            App((reduc1Normale n),m)
         with IRREDUCTIBLE ->App(n,(reduc1Normale m));;
```

Réduction (codage en caml)

```
let rec reduc1Valeur terme =
  match terme with
      Var(x)->raise IRREDUCTIBLE
    |Lam(x,m)->Lam(x,(reduc1Valeur m))
    |App(n,m)->
       try
          App(n,(reduc1Valeur m))
          with IRREDUCTIBLE ->
              try
              App((reduc1Valeur n),m)
              with IRREDUCTIBLE->
                 try
                 (betaReducRedex terme)
                 with NOTREDEX ->raise IRREDUCTIBLE;;
```

Representation des données en λ-calcul

- Booléen : 2 combinateurs non β-convertibles, relier par la négations

$$\begin{array}{rcl} \mathit{VRAI} & = & (\lambda x.(\lambda y.x)) \\ \mathit{FAUX} & = & (\lambda x.(\lambda y.y)) \\ \mathit{NON} & = & (\lambda b.((b\ \mathit{FAUX})\ \mathit{VRAI})) \end{array}$$

$(\mathit{NON}\ \mathit{VRAI}) = ((\lambda b.((b\ \mathit{FAUX})\ \mathit{VRAI}))(\lambda x.(\lambda y.x))) \rightarrow_\beta$
$(((\lambda x.(\lambda y.x))\ \mathit{FAUX})\ \mathit{VRAI}) \rightarrow_\beta ((\lambda y.\mathit{FAUX})\ \mathit{VRAI}) \rightarrow_\beta$
FAUX

- combinateur ternaire SI

$$\mathit{SI} \quad = \quad (\lambda b.(\lambda e_1.(\lambda e_2.((b\ e1)\ e2))))$$

On vérifie :

$$\begin{array}{rcl} (((\mathit{SI}\ \mathit{VRAI})\ x)\ y) & = \beta & x \\ (((\mathit{SI}\ \mathit{FAUX})\ x)\ y) & = \beta & y \end{array}$$

Les entiers de Church

Codage des entiers en $\lambda - calcul$,

- Des itérateurs de fonction :

$$\begin{array}{lcl} \overline{0} & = & \lambda f.\lambda x.x \\ \overline{1} & = & \lambda f.\lambda x.(f\ x) \\ \overline{2} & = & \lambda f.\lambda x.(f(f\ x)) \\ & \ldots & \\ \overline{n} & = & \lambda f.\lambda x.\underbrace{(f(f\ldots x)\ldots)}_{\text{n fois}} \end{array}$$

- fonction sur les entiers

$$\begin{array}{rcl} SUCC & = & (\lambda n.(\lambda f.(\lambda x.(f\ (n\ f)\ x)))) \\ ADD & = & (\lambda m.(\lambda n.(\lambda f.(\lambda x.((m\ f)((n\ f)x)))))) \end{array}$$

Fonctions λ-représentables et Thèse de Church

- Fonctions λ-représentables
  Soit f une fonction de $N^k \to N$. f est λ-représentable par un terme F si $\forall n_1, ..., n_k \in N, ((F\overline{n_1}) \dots \overline{n_k}) \to_\beta^* \overline{f(n_1, \dots, n_k)}$

- Thèse de Church : Les fonctions calculables sont les fonctions λ-représentables.

Representation des données en λ-calcul

- Les listes

$$\begin{array}{rcc} \mathit{CONS} & = & (\lambda x.(\lambda y.(\lambda f.((f\ x)y)))) \\ \mathit{CAR} & = & (\lambda l.(l\ \mathit{VRAI}) \\ \mathit{CDR} & = & (\lambda l.(l\ \mathit{FAUX})) \\ \mathit{VIDE} & = & (\lambda l.\mathit{VRAI}) \\ \mathit{VIDE?} & = & (\lambda l.(l\ (\lambda a.(\lambda d.\mathit{FAUX})))) \end{array}$$

Qui vérifient :

$$\begin{array}{lcl} (\mathit{CAR}\ ((\mathit{CONS}\ x)\ y)) & =\beta & x \\ (\mathit{CDR}\ ((\mathit{CONS}\ x)\ y)) & =\beta & y \\ (\mathit{VIDE?}\ \mathit{VIDE}) & =\beta & \mathit{VRAI} \\ (\mathit{VIDE?}\ ((\mathit{CONS}\ x)\ y)) & =\beta & \mathit{FAUX} \end{array}$$

Récursion et points fixes

- Fonction renvoyant le dernier élément d'une liste non vide

```
let rec dernier =
  function l ->match l with
      x::[] ->x
    | x::tl->dernier tl;;
```

- abstraction de l'appel récursif à dernier

```
let phi = function f ->
   function l ->match l with
      x::[] ->x
    | x::tl->f tl;;
```

  - On a l'égalité : `dernier=phi(dernier)`
  - *dernier* est un point fixe de *Phi*

- Soit M un λ-terme existe t'il un λ-calcul N tq. $(M\ N) =_\beta N$?
- Oui (*Fix* :combinateur de point fixe)
  $(Fix\ M) =_\beta (M\ (Fix\ M))$

Récursion et points fixes

- Combinateur Y (dû à Curry)

$$Y \quad = \quad (\lambda f.((\lambda x.(f(x\ x)))(\lambda x.(f\ (x\ x)))))$$

- on vérifie : $(Y\ M) \rightarrow_\beta \cdots \rightarrow_\beta (M\ (Y\ M))$
- Définition de la fonction dernier :
  $(Y\ \lambda f.(\lambda l.(((Sl(VIDE?\ (CDR\ l)))(CAR\ L))(f\ (CDR\ l)))))$