

Introduction au lambda-calcul

Michel Lévy

26 septembre 2001

Table des matières

1	Réduction	5
1.1	Syntaxe	5
1.1.1	Variable	5
1.1.2	Application	6
1.1.3	Abstraction	6
1.1.4	Syntaxe concrète	6
1.2	Changement de variables liées	6
1.2.1	Sous-terme, occurrence libre et liée d'une variable	7
1.2.2	Variables libres	7
1.2.3	Substitution simple	8
1.2.4	α -équivalence	8
1.2.5	Substitution avec renommage	10
1.3	Réduction	11
1.4	Stratégies de réduction	14
2	Programmation en λ-calcul	17
2.1	Codage des booléens	17
2.2	Codage des couples	18
2.3	Codages des entiers naturels	18
2.4	Codage des fonctions	18
2.5	Codage des définitions récursives	19
2.5.1	Combinateur point-fixe	19
2.5.2	Exemples de définitions récursives	20
2.5.3	Conclusion	21
2.6	Entiers de Church	21
2.6.1	Des entiers de Church vers ceux de Barendregt	21
2.6.2	Des entiers de Barendregt vers ceux de Church	21
3	Typage	23
3.1	Système simple	23
3.1.1	Les règles de typage	23
3.1.2	Calcul du type principal d'un terme	25
3.2	Système T	28
3.2.1	Extension du langage	29

3.2.2	Programmation dans le système T	30
3.3	Système F	32
3.3.1	Définition du typage	32
3.3.2	Programmation dans le système F	33

Chapitre 1

Réduction

Introduction

Le lambda-calcul est un langage fonctionnel permettant de calculer toute fonction intuitivement calculable sur les entiers. L'intérêt du lambda-calcul est de concentrer dans un formalisme très réduit l'essentiel des problèmes posés par la définition du sens des langages de programmation.

1.1 Syntaxe

Les lambda-termes (le mot lambda sera omis dans la suite) sont construits en utilisant des variables, des abstractions et des applications. L'ensemble des termes est défini par la grammaire "abstraite" :

terme ::= variable | (terme terme) | (λ variable . terme).

L'ensemble des termes peut être représenté par le type Ocaml :

```
type terme =  
  | Var of string  
  | App of terme * terme  
  | Lambda of string * terme ;;
```

Notez que dans cette représentation, il faut distinguer une variable, qui est une chaîne, et un terme réduit à une variable, qui est obtenu en appliquant le constructeur Var à la variable.

1.1.1 Variable

Une variable est une suite de lettres, de chiffres et de blancs soulignés commençant par une lettre.

1.1.2 Application

Soient u et v deux termes, $(u v)$ est une application : le terme u est une “fonction” appliquée à l’argument v .

L’application est une opération non notée, et l’application de gauche est prioritaire. Avec ces conventions $x y z$ est une abréviation pour le terme $((x y)z)$.

Dans la suite, sauf exception, on réserve les lettres x, y, z pour les variables, les autres lettres désignant des termes.

1.1.3 Abstraction

Soit x une variable et u un terme, $(\lambda x . u)$ est une abstraction, c’est-à-dire une “fonction” qui à la valeur x associe le terme u ¹. Le mot fonction est mis entre guillemets, car il ne s’agit pas d’une fonction au sens de la théorie des ensembles, mais d’une fonction définie par des règles d’évaluation.

Entre 2 abstractions, celle de droite est prioritaire et l’abstraction est moins prioritaire que l’application. De plus on peut regrouper plusieurs abstractions avec un seul point suivant l’exemple ci-dessous :

Exemple 1.1.1 *Écriture abrégée des termes*

$\lambda x y . x y$ abrège $(\lambda x . (\lambda y . (x y)))$

1.1.4 Syntaxe concrète

En utilisant les conventions ci-dessus, on peut écrire les termes avec la syntaxe suivante :

terme : $:=$ variable | terme terme | λ {variable}⁺. terme | (terme).

Les parenthèses permettent d’échapper aux priorités implicites.

Exemple 1.1.2 *Passage d’un terme concret à sa forme complètement parenthésée*

Soit $S = \lambda x y z . x z(y z)$.

Avec toutes les parenthèses, $S = (\lambda x . (\lambda y . (\lambda z . ((x z)(y z))))))$.

1.2 Changement de variables liées

Dans le terme $(\lambda x . \underline{x})$, l’occurrence soulignée de x est liée. En tant que fonction, ce terme représente la même fonction que $(\lambda y . \underline{y})$. Cette notion de liaison et d’identité de 2 expressions au changement près des variables liées est courante en mathématiques.

Exemple 1.2.1 *Variables liées en mathématiques*

1. $f : x \rightarrow 4x + 3$

-
- $g : y \rightarrow 4y + 3$

f et g désignent la même fonction

¹En Ocaml, la même notion est notée ($fun x \rightarrow u$)

$$2. \{x \in \mathbb{N} \mid x > a\} = \{y \in \mathbb{N} \mid y > a\}$$

Cet exemple montre que les changements de variables liées doivent être effectués avec prudence, il est interdit de changer x en a , en effet :

$$\{x \in \mathbb{N} \mid x > a\} \neq \{a \in \mathbb{N} \mid a > a\}$$

1.2.1 Sous-terme, occurrence libre et liée d'une variable

Un terme est sous-terme de lui-même. Soient u et v deux termes, les sous-termes du terme $(u v)$ comprennent lui-même, les sous-termes de u et les sous-termes de v . Les sous-termes du terme $(\lambda x . u)$ comprennent lui-même et les sous-termes de u .

Une occurrence d'une variable dans un terme est une occurrence de la variable qui est un sous-terme du terme, donc une occurrence qui ne suit pas immédiatement λ . Par exemple dans le terme $(\lambda x . (\underline{x} y))$, la variable x n'a qu'une occurrence, qui est soulignée.

Soit le terme $(\lambda x . u)$: la portée du lier λx est le terme u et toutes les occurrences de x dans le terme $(\lambda x . u)$ sont liées. Une occurrence d'une variable est libre si cette occurrence n'est pas liée.

Exemple 1.2.2 Dans le terme $(\underline{y} (\lambda y . (\underline{y} \underline{x})))$ on a souligné les occurrences des variables. La première occurrence de y est libre et la deuxième est liée. L'unique occurrence de x est libre.

1.2.2 Variables libres

Une variable est une variable libre d'un terme, si ce terme comporte au moins une occurrence libre de la variable. Un terme fermé est un terme sans variable libre.

Exemple 1.2.3 Les variables x et y sont les variables libres du terme $(y \lambda y . y x)$. Le terme $\lambda x y . x$ est fermé.

Soit vl la fonction qui associe à un terme l'ensemble de ses variables libres. Cette fonction est définie ci-dessous par récurrence structurelle :

- $vl(x) = \{x\}$ où x est une variable
- $vl((u v)) = vl(u) \cup vl(v)$ où u, v sont des termes
- $vl((\lambda x . u)) = vl(u) - \{x\}$ où u, v sont des termes et x une variable

Cette définition peut être traduite en OCaml, un terme étant représenté par une valeur de type `terme`. Dans cette traduction, les ensembles de variables sont représentés par des listes de variables.

```
(* (enleve x l) enlève toutes les occurrences x de la liste l *)
let enleve x l =
  let rec aux = fonction
    [] -> []
    | a::m -> if x = a then aux m else a::(aux m)
  in aux l;
```

```
(* (vl m) donne une liste des variables libres du terme m *)
let rec vl = fonction
  Var v -> [v]
  | App (u,v) -> (vl u) @ (vl v)
  | Lambda (x,u) -> enleve x (vl u);;
```

1.2.3 Substitution simple

$u < x := v >$ est le terme obtenu en remplaçant dans le terme u toute occurrence *libre* de la variable x par le terme v .

La substitution simple est réalisée en Ocaml par le programme ci-dessous.

```
(* (substsimp u x v) =
remplace dans u les occurrences non liées du terme (Var x) par v *)
let substsimp u x v =
  let rec aux = fonction
    (Var y) as w -> if x = y then v else w
  | App (n,p) -> App (aux n,aux p)
  | (Lambda (y, t)) as w ->
    if x = y then w else Lambda (y, aux t)
  in aux u ;;
```

Exemple 1.2.4 *substitution simple*

$(\lambda z . x z) < x := y > = (\lambda z . y z)$

$(\lambda x . x) < x := y > = (\lambda x . x)$

$(\lambda y . x) < x := y > = (\lambda y . y)$: la variable y est capturée au cours de la substitution $< x := y >$, cette capture se manifeste par la création d'une liaison nouvelle. On définit ci-dessous une substitution qui évite les captures de variables.

1.2.4 α -équivalence

Considérons le dessin des liaisons de deux termes

$$\lambda \begin{array}{c} \boxed{} \\ xy . x(xy) \end{array} =_{\alpha} \lambda \begin{array}{c} \boxed{} \\ uv . u(uv) \end{array}$$

Lorsque ces dessins sont identiques, ces termes sont des variantes l'un de l'autre. On dit aussi que ces deux termes sont α -équivalents. Avec les conventions mathématiques usuelles, ils représentent la même expression. On note par $u =_{\alpha} v$ le fait que u et v sont des termes α -équivalents et on définit précisément cette relation par récurrence sur la longueur des termes.

Définition 1.2.1 $=_{\alpha}$

1. Soit x une variable et u un terme. $x =_{\alpha} u$ si et seulement si $x = u$

2. Soient u, v, w trois termes.

$(u \ v) =_{\alpha} w$ si et seulement si $w = (u' \ v')$ et $u =_{\alpha} u'$ et $v =_{\alpha} v'$

3. Soit x une variable et u, v deux termes.

Si $(\lambda x . u) =_{\alpha} v$ alors il y a une variable x' et un terme u' tel que $v = (\lambda x' . u')$ et pour toute variable z absente des termes u et u' , on a $u < x := z > =_{\alpha} u' < x' := z >$

4. S'il existe une variable z absente des termes u et u' telle que

$u < x := z > =_{\alpha} u' < x' := z >$ alors $(\lambda x . u) =_{\alpha} (\lambda x' . u')$

La définition ci-dessus nous un moyen de calculer la relation $=_{\alpha}$. Supposons que l'on souhaite savoir si : $(\lambda x . u) =_{\alpha} v$.

1. si v n'est pas une λ -abstraction, alors d'après (3) $(\lambda x . u) \neq_{\alpha} v$.

2. si $v = (\lambda x' . u')$ alors on choisit une variable z quelconque absente de u et de u' .

(a) Supposons que $u < x := z > =_{\alpha} u' < x' := z >$. Alors, d'après (4), $(\lambda x . u) =_{\alpha} (\lambda x' . u')$.

(b) Supposons que $u < x := z > \neq_{\alpha} u' < x' := z >$. Montrons qu'alors $(\lambda x . u) \neq_{\alpha} (\lambda x' . u')$.

En effet supposons le contraire, d'après (3) et le choix de z , on aurait $u < x := z > =_{\alpha} u' < x' := z >$ ce qui contredit l'hypothèse.

La définition et son commentaire ci-dessus justifie sans peine la correction de la fonction **alpha** qui vérifie l' α -équivalence de deux termes.

Cette fonction utilise la fonction auxiliaire `nv`, qui sert à produire des nouvelles variables, et la fonction `vars`, qui permet de calculer la liste des variables d'un terme.

(* (nv lc) construit une chaîne s où s est la lettre v suivie de l'écriture décimale du plus petit entier i tel que s n'est pas élément de la liste `lc` *)

```
let rec nv lc =
  let rec aux i = let s = "v"^(string_of_int i) in
    if (mem s lc) then aux (i+1) else s
  in aux 0;;
```

(* (vars m) donne une liste des variables du terme m *)

```
let rec vars = fonction
  Var x -> [x]
  | App (n,p) -> (vars n)@(vars p)
  | Lambda (y,t) -> y::(vars t);;
```

(* (alpha u v) teste l' α -équivalence de u et v *)

```
let rec alpha u v = match u,v with
  Var x, Var y -> (x = y)
  | App (m,n), App (m',n') -> (alpha m m') && (alpha n n')
  | Lambda (x,m), Lambda (y,n) -> let z = nv ((vars m)@(vars n)) in
    alpha (substsimp m x (Var z)) (substsimp n y (Var z))
```

```
|      _ -> false;;
```

On donne une seule propriété de la relation $=_\alpha$, celle qui permet de changer le nom des variables liées d'un terme pour obtenir un autre terme qui lui soit α -équivalent.

Propriété 1.2.1 *Soit y une variable qui ne figure pas dans le terme u . On a :*

$$(\lambda x . u) =_\alpha (\lambda y . u < x := y >)$$

preuve : Vu le choix de y , la substitution simple vérifie pour tout terme v

$$u < x := v > = u < x := y > \times y := v >.$$

Puisque $=_\alpha$ est une équivalence, on peut trouver une variable z absente de u et de $u < x := y >$ telle que

$$u < x := z > =_\alpha u < x := y > \times y := z >.$$

$$\text{Donc par définition de } =_\alpha, (\lambda x . u) =_\alpha (\lambda y . u < x := y >).$$

C.Q.F.D

En répétant l'application de cette propriété à toutes les abstractions d'un terme, on peut construire un terme α -équivalent à ce terme et ne comportant aucune liaison portant sur un ensemble fini de variables.

1.2.5 Substitution avec renommage

La substitution avec renommage est une substitution qui évite les captures de variables dans les termes que l'on substitue, à l'aide de changements de variables liées.

Définition 1.2.2 *Substitution avec renommage.*

$u[x := v]$ est le résultat de la substitution avec renommage de la variable x par le terme v dans le terme u .

Le terme $u[x := v]$ est défini par récurrence sur u .

1. u est une variable.

$$\text{Si } x = u \text{ alors } u[x := v] = v \text{ sinon } u[x := v] = u.$$

2. u est une application, autrement dit $u = (n \ p)$

$$u[x := v] = (n[x := v] \ p[x := v])$$

3. u est une abstraction, autrement dit $u = (\lambda y . t)$.

On distingue plusieurs cas de façon à minimiser le nombre de changement de variable à faire pour empêcher la capture des variables du terme v .

$$(a) \text{ Si } x \notin vl(u) \text{ alors } u[x := v] = u.$$

$$(b) \text{ Si } x \in vl(u) \text{ et } y \notin vl(v) \text{ alors } u[x := v] = (\lambda y . t[x := v]).$$

$$(c) \text{ Si } x \in vl(u) \text{ et } y \in vl(v) \text{ alors } u[x := v] = (\lambda s . t < y := s > [x := v]),$$

où la variable s n'est ni une variable de u , ni une variable de v . Dans les exemples et dans le programme Ocaml suivant, on choisit pour s , la lettre v suivie de l'écriture décimale du plus petit entier i tel que s n'est ni une variable de u , ni une variable de v .

Exemple 1.2.5 *On a*

$$\begin{aligned} (\lambda x . x y)[y := x] &= (\lambda v0 . v0 x) \\ (\lambda v0 . v0 x)[x := v0] &= (\lambda v1 . v1 v0) \end{aligned}$$

On donne un programme Ocaml qui réalise cette substitution en suivant la définition ci-dessus.

```
(* subst u x v = u[x:=v], u et v sont du type terme, x du type string *)
let subst u x v =
  let vlv = vl v in
  let rec aux = function
    (Var y) as w -> if x = y then v else w
  | App (n, p) -> App (aux n, aux p)
  | (Lambda (y, t)) as w ->
    if not (mem x (vl w))
    then w
    else if not (mem y vlv)
    then Lambda (y, aux t)
    else (* y capture une variable libre de v *)
      let z = nv ((vars t)@ vlv)
      in Lambda (z, aux (substsimp t y (Var z)))
  in aux u;;
```

Il y a d'autres façons de définir la substitution avec renommage, toutes ont en commun les deux propriétés ci-dessous. La première propriété montre la relation entre la substitution simple et la substitution avec renommage, la deuxième propriété met en évidence que la substitution avec renommage est définie à un changement près des variables liées.

Propriété 1.2.2 *Soit x une variable et u, v deux termes.*

Soit u' un terme α -équivalent à u et dont les variables liées ne sont pas des variables libres de v . On a : $u[x := v] =_{\alpha} u' < x := v >$

La propriété ci-dessus suggère une autre *réalisation* du calcul de $u[x := v]$: changer u en un terme α -équivalent u' , dont les variables liées ne sont pas des variables libres de v , puis calculer $u' < x := v >$.

Propriété 1.2.3 *Supposons $u =_{\alpha} u', v =_{\alpha} v'$. On a : $u[x := v] =_{\alpha} u'[x := v']$.*

1.3 Réduction

Définition 1.3.1 *Soit R une relation sur les termes. La relation R est α -compatible si et seulement si $=_{\alpha} R \subseteq R =_{\alpha}$.*

Remarque 1.3.1 *Montrons une conséquence de cette définition, qui est d'ailleurs équivalente à la définition.*

Soit R une relation α -compatible. Supposons que $u R v$ et $u =_{\alpha} u'$. Puisque la relation $=_{\alpha}$ est symétrique, on a : $u' =_{\alpha} R v$, donc $u' R =_{\alpha} v$. Par suite il existe v' tel que $u' R v'$ et $v' =_{\alpha} v$.

Définition 1.3.2 *Réduction en 1 pas.*

- Un redex (reducible expression) est un terme de la forme $((\lambda x . u) v)$
- Le contracté de ce redex est $u[x := v]$.
- On note par $u \rightarrow v$, le fait que le terme v est obtenu en contractant un redex de u ².

Cette relation est appelée la réduction en 1 pas, elle est aussi notée \rightarrow_β pour la distinguer d'autres réductions.

Exemple 1.3.1 *le redex contracté est souligné.*

- $(\lambda x . x) y \rightarrow y$
- $\lambda y . ((\lambda z . y z) x) \rightarrow \lambda y . (y x)$
- $(z ((\lambda x y . y x) y)) \rightarrow (z (\lambda v 0 . v 0 y))$

Propriété 1.3.1 *La réduction en 1 pas est α -compatible.*

Propriété 1.3.2 *Si une relation sur les termes est α -compatible, il en est de même de sa fermeture réflexive et transitive.*

preuve : Soit R une relation α -compatible. Pour prouver que R^* est α -compatible, il suffit de prouver par récurrence sur n , que la relation R^n est α -compatible.

- pour $n = 0$, c'est évident.
- Montrons que R^{n+1} est α -compatible.
Par hypothèse de récurrence on a : $=_\alpha R^n \subseteq R^n =_\alpha$.
Par monotonie de la composition de relation, on a : $=_\alpha R^{n+1} \subseteq R^n =_\alpha R$.
Puisque R est α -compatible, on a : $=_\alpha R \subseteq R =_\alpha$.
Par monotonie de la composition de relation, on en déduit :
 $R^n =_\alpha R \subseteq R^{n+1} =_\alpha$.
Par suite $=_\alpha R^{n+1} \subseteq R^{n+1} =_\alpha$.

C.Q.F.D

D'après les deux propriétés 1.3.1 et 1.3.2, la relation \rightarrow^* est α -compatible.

Définition 1.3.3 *Réduction, chemin de réduction.*

Le fait que $M \rightarrow^* N$ se lit : M se réduit en N .

Un chemin du graphe de la relation \rightarrow est appelé un chemin de réduction.

Définition 1.3.4 *Terme normal, normalisable, fortement normalisable.*

- Un terme est dit normal ou sous forme normale, s'il ne contient aucun redex.
- Un terme est normalisable, s'il peut se réduire en un terme normal.
- Un terme est fortement normalisable s'il n'existe aucun chemin infini de réduction ayant ce terme comme origine.

Le terme $(\lambda x . x)$ est normal.

Le terme $(\lambda x . x x)(\lambda x . x)$ est fortement normalisable, en effet il se contracte l'unique terme $(\lambda x . x)(\lambda x . x)$, qui lui-même se contracte en l'unique terme normal $(\lambda x . x)$.

Soit $\Omega = (\lambda x . x x)(\lambda x . x x)$. Ce terme n'est pas normalisable, en effet en un pas, ce terme se réduit *uniquement* en lui-même, donc il est l'origine d'un chemin

²plus précisément en contractant une occurrence d'un redex de u

infini de réduction et il n'est l'origine d'aucun chemin de réduction aboutissant à un terme normal.

Le terme $((\lambda x . y) \Omega)$ est normalisable, car $((\lambda x . y) \Omega) \rightarrow y$ mais il n'est pas fortement normalisable, car il est l'origine d'un chemin infini de réduction.

Définition 1.3.5 β -équivalence

On donne deux définitions de cette relation entre termes, qui est notée $=_\beta$.

1. $=_\beta$ est la plus petite relation d'équivalence contenant \rightarrow et $=_\alpha$.
2. $u =_\beta^n v$ si et seulement si il existe une suite $w_{i(i=0\dots n)}$ telle que
 - (a) $u = w_0, v = w_n$
 - (b) pour i de 1 à $n - 1$, $w_i =_\alpha w_{i+1}$ ou $w_i \rightarrow w_{i+1}$ ou $w_{i+1} \rightarrow w_i$
3. $u =_\beta v$ si et seulement si il existe un naturel n tel que $u =_\beta^n v$

Définition 1.3.6 Relation Church-Rosser.

Une relation R , sur les termes, est Church-Rosser lorsqu'elle vérifie : $u R v$ et $u R w$ implique il existe un terme t tels que $v R t$ et $w R t$.

Théorème 1.3.1 Théorème de Church-Rosser.

La relation $\rightarrow^* =_\alpha$ est CR.

La preuve de ce théorème peut être lue dans [1]. Ce théorème a des conséquences importantes, corollaires 1.3.1, 1.3.2, 1.3.3 que nous prouvons ci-dessous.

Corollaire 1.3.1 Si $u =_\beta v$ alors il existe w tel que $u \rightarrow^* =_\alpha w$ et $v \rightarrow^* =_\alpha w$.

preuve : Soit $P(n)$ la propriété suivante : si $u =_\beta^n v$ alors il existe w tel que $u \rightarrow^* =_\alpha w$ et $v \rightarrow^* =_\alpha w$.

Pour établir le corollaire, il suffit de montrer que $P(n)$ est vérifiée pour tout entier naturel n .

Pour $n = 0$, c'est évident. Admettons $P(n)$ et prouvons $P(n + 1)$.

Supposons $u =_\beta^{n+1} v$ et montrons qu'il existe w tel que $u \rightarrow^* =_\alpha w$ et $v \rightarrow^* =_\alpha w$.

Par définition de $=_\beta^{n+1}$, il existe w tel que $u =_\beta^n w$ et l'on a un des trois cas suivants $w =_\alpha v$, $v \rightarrow w$ ou $w \rightarrow v$.

D'après l'hypothèse de récurrence, il y a un terme r tel que $u \rightarrow^* =_\alpha r$ et $w \rightarrow^* =_\alpha r$.

Examinons les trois cas possibles suivant les relations entre v et w .

1. Supposons que $w =_\alpha v$.

Puisque que $=_\alpha$ est symétrique, on a : $v =_\alpha \rightarrow^* =_\alpha r$.

Puisque que \rightarrow^* est α -compatible, on en déduit : $v \rightarrow^* =_\alpha =_\alpha r$.

Puisque que $=_\alpha$ est transitive, il en résulte que : $v \rightarrow^* =_\alpha r$.

2. Supposons que $v \rightarrow w$.

On a immédiatement $v \rightarrow^* =_\alpha r$.

3. Supposons que $w \rightarrow v$.

Puisque $w \rightarrow^* =_\alpha r$, d'après le théorème de Church-Rosser, il existe un terme s tel que $v \rightarrow^* =_\alpha s$ et $r \rightarrow^* =_\alpha s$.

Puisque $u \rightarrow^* =_\alpha r$, il existe un terme t tel que $u \rightarrow^* t$ et $t =_\alpha r$.

Puisque $r \rightarrow^* =_\alpha s$ et que \rightarrow^* est α -compatible, on a : $t \rightarrow^* =_\alpha =_\alpha s$.

Puisque $u \rightarrow^* t$, on a : $u \rightarrow^* \rightarrow^* =_\alpha =_\alpha s$.

Puisque \rightarrow^* et $=_\alpha$ sont transitives, on déduit que : $u \rightarrow^* =_\alpha s$.

Ainsi dans les trois cas ci-dessus, il existe w tel que $u \rightarrow^* =_\alpha w$ et $v \rightarrow^* =_\alpha w$.

Par suite la propriété $P(n+1)$ est prouvée.

C.Q.F.D

Le corollaire ci-dessous indique que si un terme est β -égal à un terme normal, il se réduit en ce terme normal à un changement près de variables liées.

Corollaire 1.3.2 *Si $u =_\beta v$ et si v est normal alors $u \rightarrow^* =_\alpha v$.*

preuve : Supposons que $u =_\beta v$ et que v est normal.

D'après le corollaire 1.3.1, il existe w tel que $u \rightarrow^* =_\alpha w$ et $v \rightarrow^* =_\alpha w$.

Puisque v est un terme normal, $v =_\alpha w$, donc par symétrie et transitivité de $=_\alpha$, on a : $u \rightarrow^* =_\alpha v$

C.Q.F.D

Le terme v est une *forme normale de u* , si u se réduit en v et si v est normal. Le corollaire ci-dessous montre que cette forme normale est unique à un changement près de variables liées. Aussi après la preuve de ce corollaire, on pourra légitimement parler de *la* forme normale d'un terme.

Corollaire 1.3.3 *Si $u =_\beta v$, $u =_\beta w$ et si v et w sont normaux alors $v =_\alpha w$.*

preuve : Supposons que $u =_\beta v$, $u =_\beta w$ et que v et w sont normaux.

D'après le corollaire précédent, $u \rightarrow^* =_\alpha v$ et $u \rightarrow^* =_\alpha w$.

D'après le théorème de Church-Rosser, il existe un terme s tel que

$v \rightarrow^* =_\alpha s$ et $w \rightarrow^* =_\alpha s$.

Puisque v et w sont normaux, $v =_\alpha s$ et $w =_\alpha s$.

Puisque $=_\alpha$ est une équivalence, on a : $v =_\alpha w$.

C.Q.F.D

1.4 Stratégies de réduction

Un terme peut comporter plusieurs redex. Une stratégie de réduction consiste à préciser le (ou les) redex qu'il faut réduire dans un terme. On donne deux exemples classiques de stratégie.

1. La réduction gauche consiste à contracter le redex le plus à gauche.

Soient u et v deux termes. On note $u \rightarrow_g v$ le fait que v est obtenu en contractant le redex le plus à gauche de u .

2. La réduction droite consiste à contracter le redex le plus à droite.

Une stratégie est normalisante si elle permet de calculer (par application répétée de la stratégie) la forme normale de tout terme normalisable.

Théorème 1.4.1 *La réduction gauche est une stratégie normalisante. Autrement dit, si le terme u est β -équivalent au terme normal v alors $u \rightarrow_g^* =_\alpha v$.*

On ne donne pas la preuve de ce théorème, mais quelques indications qui le justifient. D'après le corollaire 1.3.2, si le terme u est β -équivalent au terme normal v alors $u \rightarrow^* =_\alpha v$. Il suffit donc de prouver que tout chemin de réduction conduisant à un terme normal peut être transformé en un chemin de réduction gauche.

Voici un exemple qui justifie le choix de la réduction gauche comme stratégie normalisante :

- $((\lambda x . y) \Omega) \rightarrow ((\lambda x . y) \Omega)$, par réduction droite, donc de $((\lambda x . y) \Omega)$ part un chemin de réduction droite infini.
- $((\lambda x . y) \Omega) \rightarrow_g y$, en ignorant l'argument Ω dont l'évaluation ne se termine pas, on obtient immédiatement un terme normal.

En calculant la forme normale d'un terme, la réduction gauche ne fait, comme cet exemple le suggère, que les contractions nécessaires pour obtenir cette forme : si le chemin de réduction gauche est infini, le terme n'est pas normalisable.

Corollaire 1.4.1 *Un terme est normalisable si et seulement si son chemin de réduction gauche est fini.*

preuve : L'implication de droite à gauche est évidente. Réciproquement, supposons qu'un terme u soit normalisable. Il existe un terme v tel que $u =_\beta v$ et v est normal. D'après le théorème 1.4.1, $u \rightarrow_g^* =_\alpha v$, donc le chemin de réduction gauche de u est fini.

C.Q.F.D

Remarque 1.4.1 *Il n'y a aucun algorithme permettant de décider si un terme est normalisable.*

On termine ce chapitre en écrivant en Ocaml un algorithme pour calculer la forme normale d'un terme. Mais *attention*, lorsque cet algorithme est appliqué à un terme non normalisable, il ne se termine pas.

```
(* (normal u) = true si et seulement si le terme u est normal *)
let rec normal = fonction
  Var x -> true
  | App (Lambda (x,t), u) -> false
  | App (u, v) -> (normal u) && (normal v)
  | Lambda (x,t) -> normal t ;;

(* Si u contient un redex, (betag1 u) effectue un pas
de réduction gauche sur u *)
let rec betag1 = fonction
  (App (Lambda (x,t),v)) -> (subst t x v)
  | (App (p, q)) -> if normal p then (App (p, betag1 q))
    else (App (betag1 p,q))
  | (Lambda (x,t)) -> (Lambda (x, betag1 t)) ;;
```

```
(* (betag u) calcule si elle existe la forme normale
de u en itérant l'application de la réduction gauche en un pas *)
let rec betag u =
  if normal u then u else betag (betag1 u);;
```

Chapitre 2

Programmation en λ -calcul

Introduction

Pour transformer le λ -calcul en langage de programmation, il faut répondre aux questions suivantes :

- Quels termes sont des valeurs ?
- Comment calculer ces valeurs ?
- Comment représenter les booléens, les entiers, les couples, ...

Dans les livres [1] et [2], on trouve les choix suivants pour les valeurs et leur mode de calcul :

1. Les valeurs sont les termes normaux
 - (a) On calcule par réduction gauche
 - (b) On calcule par réduction droite
2. Les valeurs sont les termes normaux de tête et on calcule par réduction de tête.

Expliquons ces termes. Le redex de tête d'un terme est le redex qui n'est pas à droite d'une application. Un terme est en forme normale de tête s'il n'a pas de redex de tête et on calcule en contractant uniquement les redex de tête.

Dans la suite de chapitre, les valeurs sont des termes *normaux* et on évalue les termes par réduction *gauche*. Ci-dessous, on propose un codage des booléens, des couples et des entiers.

2.1 Codage des booléens

Le booléen vrai est représenté par le terme $T = \lambda x y . x$.

Le booléen faux est représenté par le terme $F = \lambda x y . y$.

L'expression conditionnelle est représentée par le terme $if = \lambda b x y . b x y$.

Propriété 2.1.1 *Soient u et v deux termes.*

$$\begin{aligned} \text{if } T \text{ } u \text{ } v &=_{\beta} u \\ \text{if } F \text{ } u \text{ } v &=_{\beta} v \end{aligned}$$

2.2 Codage des couples

Soient u et v deux termes, on note par $\langle u, v \rangle$ un terme $(\lambda p . p \ u \ v)$, où p est une variable qui n'est libre ni dans u , ni dans v .

Pour accéder aux éléments d'un couple $\langle u, v \rangle$ on utilise les termes

$$\begin{aligned} \text{pi1} &= \lambda p . p \ T \\ \text{pi2} &= \lambda p . p \ F \end{aligned}$$

Propriété 2.2.1 *Soient u et v deux termes.*

$$\begin{aligned} \text{pi1 } \langle u, v \rangle &=_{\beta} u \\ \text{pi2 } \langle u, v \rangle &=_{\beta} v \end{aligned}$$

preuve :

$$\begin{aligned} \text{pi1 } \langle u, v \rangle &=_{\beta} \langle u, v \rangle \ T && \text{par réduction gauche} \\ &= (\lambda p . p \ u \ v) \ T && \text{par définition de } \langle u, v \rangle \\ &=_{\beta} T \ u \ v && \text{par réduction gauche, car } p \text{ n'est libre ni dans } u, \text{ ni dans } v. \\ &=_{\beta} u && \text{par réduction gauche} \end{aligned}$$

C.Q.F.D

2.3 Codages des entiers naturels

On choisit une représentation qui facilite le calcul du prédécesseur d'un entier et le test d'égalité à 0. Soit n un entier naturel, on note b_n la représentation de n préconisée par Barendregt :

$$\begin{aligned} - b_0 &= \langle T, T \rangle = \lambda p . p \ T \ T \\ - b_{n+1} &= \langle F, b_n \rangle = \lambda p . p \ F \ b_n \end{aligned}$$

On note que les entiers de Barendregt sont des termes normaux et on donne quelques propriétés évidentes de cette représentation :

$$\begin{aligned} - \text{pi1 } b_0 &=_{\beta} T \\ - \text{pi1 } b_{n+1} &=_{\beta} F \\ - \text{pi2 } b_{n+1} &=_{\beta} b_n \end{aligned}$$

La fonction successeur est calculée par le terme $sb = \lambda x \ p . p \ F \ x$, car il est évident que $sb \ b_n =_{\beta} b_{n+1}$.

2.4 Codage des fonctions

On précise ce qu'on entend par calcul d'une fonction au moyen d'un terme. On considère uniquement le cas des fonctions partielles de \mathbb{N} dans \mathbb{N} , une fonction partielle de \mathbb{N} dans \mathbb{N} étant une fonction d'un sous-ensemble de \mathbb{N} dans \mathbb{N} .

Définition 2.4.1 *Le terme F calcule la fonction f signifie que pour tout entier naturel n élément du domaine de f , on a $(F \ b_n) \rightarrow_{\beta}^* =_{\alpha} b_{f(n)}$.*

On donne une propriété qui justifie notre choix de privilégier l'évaluation par réduction gauche.

Propriété 2.4.1 *Le terme F calcule la fonction f si et seulement si, pour tout entier naturel n élément du domaine de f , on a $(F b_n) =_{\beta} b_{f(n)}$.*

preuve :

- Si $(F b_n) \rightarrow_g^* =_{\alpha} b_{f(n)}$ alors par définition de $=_{\beta}$, on a $(F b_n) =_{\beta} b_{f(n)}$.
- Réciproquement supposons que $(F b_n) =_{\beta} b_{f(n)}$. On a choisi de représenter les valeurs par des termes normaux, en particulier $b_{f(n)}$ est un terme normal. Donc d'après le théorème 1.4.1, on a : $(F b_n) \rightarrow_g^* =_{\alpha} b_{f(n)}$.

C.Q.F.D

Remarque 2.4.1 *Pour des raisons qui sont exposées dans les ouvrages [1] et [2], dans le codage des fonctions partielles de \mathbb{N} dans \mathbb{N} , on peut être amené à préciser le comportement du terme F calculant la fonction partielle f , lorsqu'on évalue $(F b_n)$ pour un entier naturel n non élément du domaine de f .*

Un choix fréquent est la convention suivante : lorsque n est un entier naturel non élément du domaine de f , le chemin de réduction gauche de $(F b_n)$ est infini. D'après le théorème 1.4.1, c'est équivalent à dire que $(F b_n)$ n'est pas normalisable.

Lorsque le terme F respecte cette convention, on dit dans les livres cités que F représente f . Sur un exemple, on peut observer la nuance entre calculer une fonction et la représenter. le terme $\text{pi}2$ (défini et utilisé dans 2.2 et 2.3) calcule la fonction prédécesseur mais ne la représente pas : en effet sur les entiers naturels, 0 n'a pas de prédécesseur, mais $(\text{pi}2 b_0) =_{\beta} T$, donc $(\text{pi}2 b_0)$ est normalisable.

2.5 Codage des définitions récursives

2.5.1 Combinateur point-fixe

On appelle combinateur tout terme sans variable libre. Le terme C sans variable libre est un combinateur de point fixe s'il vérifie pour tout terme u : $u(Cu) =_{\beta} Cu$.

Théorème 2.5.1 *Soit $Y = \lambda f . ((\lambda x . f(x x))(\lambda x . f(x x)))$.*

Y est un combinateur de point-fixe.

preuve : Soit u un terme quelconque.

$$Y u =_{\beta} (\lambda y . u(y y))(\lambda y . u(y y))$$

$$=_{\beta} u(\lambda y . u(y y))(\lambda y . u(y y))$$

$$=_{\beta} u(Y u)$$

C.Q.F.D

Si x est libre dans u , on change x en y
une variable non libre dans u

puis on réduit à gauche

par réduction gauche

par la β -équivalence de la première ligne

2.5.2 Exemples de définitions récursives

addition

L'addition sur les entiers naturels est définie par :

$$add(x, y) = \text{si } x = 0 \text{ alors } y \text{ sinon } 1 + add(x - 1, y)$$

C'est-à-dire : $add = \lambda x y. \text{si } x = 0 \text{ alors } y \text{ sinon } 1 + add(x - 1, y)$.

Posons : $fadd = \lambda f x y. \text{si } x = 0 \text{ alors } y \text{ sinon } 1 + f(x - 1, y)$.

Il est clair que $add = fadd \text{ add}$, autrement dit add est un point-fixe de la fonctionnelle $fadd$.

On traduit cette idée dans le λ -calcul en écrivant $faddb$ un terme représentant la fonctionnelle $fadd$ pour les entiers de Barendregt :

$$faddb = \lambda f x y . \text{if } (pi1 x) y (sb (f (pi2 x) y))$$

$$addb = Y faddb$$

Prouvons la correction du programme en montrant par récurrence sur x que pour tous les entiers naturels x et y , on a : $addb b_x b_y =_{\beta} b_{x+y}$.

1. $addb b_0 b_y =_{\beta} faddb addb b_0 b_y$
 car d'après le théorème du point-fixe (cf 2.5.1), $addb =_{\beta} faddb addb$
 $=_{\beta} \text{if } (pi1 b_0) b_y (sb (addb (pi2 b_0) b_y))$
 par trois réductions gauches.
 $=_{\beta} b_y$
 car $(pi1 b_0) =_{\beta} T$ et $(\text{if } T b_y (sb (addb (pi2 b_0) b_y))) =_{\beta} b_y$.
2. $addb b_{x+1} b_y =_{\beta} faddb addb b_{x+1} b_y$
 car $addb =_{\beta} faddb addb$.
 $=_{\beta} \text{if } (pi1 b_{x+1}) b_y (sb (addb (pi2 b_{x+1}) b_y))$
 par trois réductions gauches.
 $=_{\beta} (sb (addb (pi2 b_{x+1}) b_y))$
 car $(pi1 b_{x+1}) =_{\beta} F$
 et $(\text{if } F b_y (sb (addb (pi2 b_{x+1}) b_y))) =_{\beta} (sb (addb (pi2 b_{x+1}) b_y))$
 $=_{\beta} (sb (addb b_x b_y))$
 car $pi2 b_{x+1} =_{\beta} b_x$
 $=_{\beta} sb b_{x+y}$
 car par hypothèse de récurrence $(addb b_x b_y) =_{\beta} b_{x+y}$.
 $=_{\beta} b_{x+y+1}$
 car sb calcule le successeur.

soustraction

La soustraction sur les entiers naturels est définie par :

$$sub(x, y) = \text{si } y = 0 \text{ alors } x \text{ sinon } sub(x, y - 1) - 1$$

Dans la suite, on considère que $sub(x, y)$ n'est pas définie quand $x < y$.

De cette définition, on déduit le programme qui calcule la soustraction :

$$- fsubb = \lambda f x y . \text{if } (pi1 y) x (pi2 (f x (pi2 y)))$$

$$- subb = Y fsubb$$

Comme pour l'addition, il est facile de prouver que $subb$ calcule correctement la soustraction, c'est à dire que si $x \geq y$ alors $subb b_x b_y =_{\beta} b_{x-y}$.

Ce programme ne représente pas la soustraction, car pour $x < y$, $sub(x, y)$ n'est pas défini mais le terme $subb\ b_x\ b_y$ est normalisable.

2.5.3 Conclusion

Dans le λ -calcul, on sait représenter les données usuelles (booléens, couples, entiers), l'expression conditionnelle, les opérations successeur, prédécesseur, le test d'égalité à 0 et les définitions récursives. Donc le λ -calcul peut être considéré comme un langage de programmation :

- Il est peu pratique à cause des codages compliqués pour représenter les entiers, booléens, ...
- Il permet de programmer toutes les fonctions intuitivement calculables. Cette affirmation est appelée la thèse de Church.

2.6 Entiers de Church

Le codage le plus fréquent des entiers, mais pas le plus simple, est celui de Church. On définit par récurrence la notation suivante $(u^n\ v)$ pour tous les termes u, v et tout entier naturel n :

- $(u^0\ v) = v$
- $(u^{n+1}\ v) = u\ (u^n\ v)$

L'entier naturel n est représenté par l'entier de Church $c_n = \lambda f\ x . (f^n\ x)$. Par exemple $c_0 = \lambda f\ x . x$, $c_1 = \lambda f\ x . f\ x$, $c_2 = \lambda f\ x . f\ (f\ x)$.

Avec ce codage des entiers, la fonction successeur est représentée par :

$$sc = \lambda n\ f\ x . f\ (n\ f\ x), \text{ c'est-à-dire que } sc\ c_n =_{\beta} c_{n+1}.$$

Il n'est pas facile de programmer directement la fonction prédécesseur avec les entiers de Church, mais on peut écrire dans le λ -calcul des programmes de conversion des entiers de Church à ceux de Barendregt et inversement.

2.6.1 Des entiers de Church vers ceux de Barendregt

Soit $c2b = \lambda n . n\ sb\ b_0$.

Pour tout entier naturel n , on a : $c2b\ c_n =_{\beta} b_n$.

En effet $c2b\ c_n =_{\beta} c_n\ sb\ b_0 =_{\beta} sb^n\ b_0 =_{\beta} b_n$

2.6.2 Des entiers de Barendregt vers ceux de Church

Le programme $b2c$, qui convertit un entier de Barendregt en un entier de Church, a la définition informelle suivante :

$$b2c(n) = \text{si } (pi1\ n) \text{ alors } c_0 \text{ sinon } sc\ (b2c\ (pi2\ n)).$$

Par suite ce programme s'écrit :

- $fb2c = \lambda f\ n . \text{if } (pi1\ n)\ c_0\ (sc\ (f\ (pi2\ n)))$
- $b2c = Y\ fb2c$

Et pour tout entier naturel n , on a : $b2c b_n =_{\beta} c_n$.

Avec ces deux conversions, il est facile d'écrire par exemple l'addition pour les entiers de Church à partir de l'addition pour les entiers de Barendregt :

$$addc = \lambda x y . b2c (addb (c2b x) (c2b y))$$

Chapitre 3

Typage

Introduction

Dans les langages de programmation, le typage d'un programme permet de vérifier avant l'exécution du programme que cette exécution évitera certaines erreurs. Les systèmes de types que nous examinons dans la suite ont des objectifs analogues : un *lambda*-terme typable est fortement normalisable (1.3.4). Les systèmes de type peuvent être plus riches que ceux des langages de programmation, les types peuvent constituer des spécifications complètes des programmes et la vérification des types devient une *preuve de programme*.

3.1 Système simple

Dans ce paragraphe, on définit un système de typage dit simple. Les règles de typage des applications et des abstractions sont les mêmes que celles du langage de programmation Ocaml, les algorithmes permettant de calculer le type d'un terme constituent ainsi une introduction aux algorithmes permettant de typer une expression du langage Ocaml.

3.1.1 Les règles de typage

Définition 3.1.1 *Les types.*

Les types sont construits à partir de variables de types et de l'opération \rightarrow . Dans la suite de ce chapitre, on note les variables de types avec des minuscules du début de l'alphabet, pouvant être suivies de chiffres et on réserve la fin de l'alphabet pour les variables des termes. Le type $a \rightarrow b$ est intuitivement celui des fonctions de domaine a et de portée b . La syntaxe concrète des types est définie par la règle suivante :

$type := variable_de_type \mid type \rightarrow type \mid (type)$.

Dans cette syntaxe, l'opération \rightarrow est prioritaire à droite. Par exemple le type $a \rightarrow b \rightarrow c$ est un type dont la forme parenthésée est $(a \rightarrow (b \rightarrow c))$.

Définition 3.1.2 *La relation de typage.*

- Une déclaration de variable est un couple $x : A$ où x est une variable du λ -calcul et A un type.
- Un contexte est une liste de déclarations
- Soit Γ un contexte, t un λ -terme et A un type. La relation t est de type A dans le contexte Γ est notée $\Gamma \vdash t : A$. Cette relation est définie par les règles suivantes :

1. *axiome*

$\Gamma \vdash x : A$ est un axiome si la déclaration la plus à droite de x dans le contexte Γ est $x : A$

2. *application*

$$\frac{\Gamma \vdash u : A \rightarrow B \quad \Gamma \vdash v : A}{\Gamma \vdash (u v) : B} \text{ APP}$$

3. *abstraction*

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x . t) : A \rightarrow B} \text{ ABS}$$

Le terme t est typable s'il existe un contexte Γ et un type A tel que $\Gamma \vdash t : A$.

Le contexte dans lequel un terme est typé sert à déclarer les variables libres du terme. D'ailleurs, par récurrence sur les termes, on peut prouver les résultats intuitifs suivants :

- pour qu'un terme ait un type dans un contexte donné, il faut que ses variables libres soient déclarées dans le contexte.
- il est inutile de déclarer des variables qui ne sont pas libres dans le terme dont on veut calculer le type

Supposons que le terme t ayant x pour unique variable libre soit typable. Par définition, il existe des types A et B tels que $x : A \vdash t : B$. Donc par la règle de typage de l'abstraction, on déduit que $\vdash (\lambda x . t) : A \rightarrow B$, autrement dit la fermeture du terme t est typable. La réciproque est aussi vérifiée, puisque la règle de l'abstraction est l'unique règle applicable au terme $(\lambda x . t)$: si ce terme est typable, t est aussi typable.

En généralisant à plusieurs variables, on en déduit qu'un terme est typable si et seulement si sa fermeture est typable, ce qui nous permet dans la suite de nous limiter à typer des types sans variable libre.

Exemple 3.1.1 *On déduit que $\vdash \lambda xy . x : a \rightarrow b \rightarrow a$ par la preuve suivante.*

- (1) $x : a, y : b \vdash x : a$ *axiome*
- (2) $x : a \vdash \lambda y . x : b \rightarrow a$ *app 1*
- (3) $\vdash \lambda xy . x : a \rightarrow b \rightarrow a$ *app 2*

Une preuve analogue permet de déduire $\vdash \lambda xy . x : A \rightarrow B \rightarrow A$ pour tous types A et B . Mais $a \rightarrow b \rightarrow a$ est un type principal du terme $(\lambda xy . x)$. De ce type principal on peut déduire les autres types en substituant des types quelconques aux variables a et b . Puisque qu'entre deux types principaux d'un terme, il y a une substitution sur les variables de types, qui permet de passer d'un type

principal à l'autre, deux types principaux sont égaux au nom près des variables liées. Aussi dans la suite on parlera du type principal d'un terme.

3.1.2 Calcul du type principal d'un terme

Pour typer un terme sans variable libre, on procède en 3 étapes.

1. On associe une variable de type à chaque sous-terme et à chaque variable liée. Cette association est notée de la façon suivante :

(a) $(u_a v_b)_c$

(b) $(\lambda x_a . t_b)_c$

Par exemple le terme $(\lambda xy.x(xy))$ est noté ainsi $(\lambda x_a(\lambda y_b.(x_c(x_d y_e)_f)_g)_h)_i$

2. On pose les équations entre les variables de type correspondant aux règles de typage

- (a) si une occurrence de la variable x de type a est associée à la déclaration de x de type b , on pose l'équation $a = b$.

Donc sur l'exemple, on pose les équations : $a = c, a = d, b = e$ ¹.

- (b) à l'application $(u_a v_b)_c$, on associe l'équation $a = b \rightarrow c$.

Donc sur l'exemple, on pose les équations : $d = e \rightarrow f, c = f \rightarrow g$.

- (c) à l'abstraction $(\lambda x_a . t_b)_c$, on associe l'équation $c = a \rightarrow b$.

Donc sur l'exemple, on pose les équations : $h = b \rightarrow g, i = a \rightarrow h$.

3. On calcule la solution principale du système d'équations obtenues, c'est-à-dire une solution dont on peut déduire les autres par substitution ². La valeur que cette solution donne à la variable de type associée au terme est le type principal du terme.

Dans la description de l'algorithme de calcul, les lettres A, B, C, D désignent des types et a une variable de type.

L'ensemble des équations est séparé en deux, les équations non résolues et les équations résolues.

- (a) Une équation non résolue de la forme $A = A$ est supprimée.
- (b) Une équation non résolue de la forme $A \rightarrow B = C \rightarrow D$ est *décomposée* en 2 équations non résolues $A = C, B = D$.
- (c) Si une équation non résolue s'écrit $a = A$ (resp. $A = a$) où a est une variable de type distincte du type A et si a figure dans A , alors le système n'a pas de solution.
- (d) Si une équation non résolue s'écrit $a = A$ (resp. $A = a$) où a est une variable de type distincte du type A et si a ne figure pas dans A , alors on *élimine* a des autres équations (résolues ou non) en remplaçant a par A , et on place $a = A$ (resp. $A = a$) parmi les équations résolues.

¹Quand on travaille sans machine, il est plus simple de remplacer c, d directement par a et e par b , ce qui permet d'éviter d'introduire les variables c, d, e et les équations correspondantes

²Un algorithme utilisé pour trouver la solution principale d'un système d'équations est souvent appelé algorithme d'unification, car il est utilisé en logique et en Prolog pour trouver une substitution unifiant des expressions. On a choisi de présenter un algorithme simple à comprendre mais peu efficace et le lecteur pourra trouver dans [4] des variantes plus efficaces.

Reprenons l'exemple précédent. Pour typer $(\lambda xy . x(xy))$ avec i comme variable de type associé au terme, on a posé les équations suivantes :
 $a = c, a = d, b = e, d = e \rightarrow f, c = f \rightarrow g, h = b \rightarrow g, i = a \rightarrow h$.
 Appliquons l'algorithme à ces équations

opérations effectuées	équations non résolues	équations résolues
éliminons a grâce à $a = c$	$c = d, b = e, d = e \rightarrow f$ $c = f \rightarrow g, h = b \rightarrow g$ $i = c \rightarrow h$	$a = c$
éliminons c grâce à $c = d$	$b = e, d = e \rightarrow f$ $d = f \rightarrow g$ $h = b \rightarrow g, i = d \rightarrow h$	$a = d, c = d$
éliminons b grâce à $b = e$	$d = e \rightarrow f$ $d = f \rightarrow g$ $h = e \rightarrow g, i = d \rightarrow h$	$a = d, c = d, b = e$
éliminons d grâce à $d = e \rightarrow f$	$e \rightarrow f = f \rightarrow g$ $h = e \rightarrow g, i = (e \rightarrow f) \rightarrow h$	$a = e \rightarrow f, c = e \rightarrow f$ $b = e$
décomposons	$e = f, f = g$ $h = e \rightarrow g, i = (e \rightarrow f) \rightarrow h$	$a = e \rightarrow f, c = e \rightarrow f$ $b = e$
éliminons e grâce à $e = f$	$f = g$ $h = f \rightarrow g, i = (f \rightarrow f) \rightarrow h$	$a = f \rightarrow f, c = f \rightarrow f$ $b = f$
éliminons f grâce à $f = g$	$h = g \rightarrow g, i = (g \rightarrow g) \rightarrow h$	$a = g \rightarrow g, c = g \rightarrow g$ $b = g$
éliminons h grâce à $h = g \rightarrow g$	$i = (g \rightarrow g) \rightarrow (g \rightarrow g)$	$a = g \rightarrow g, c = g \rightarrow g$ $b = g, h = g \rightarrow g$
éliminons i grâce à $i = (g \rightarrow g) \rightarrow (g \rightarrow g)$		$a = g \rightarrow g, c = g \rightarrow g$ $b = g, h = g \rightarrow g$ $i = (g \rightarrow g) \rightarrow (g \rightarrow g)$

Le type $(g \rightarrow g) \rightarrow (g \rightarrow g)$, qui est la valeur de i , est donc le type principal du terme $(\lambda xy . x(xy))$.

En absence de machine, on aurait pu éviter les trois premières étapes en identifiant dès le départ le type d'une occurrence de variable du terme avec le type de sa déclaration. Sur l'exemple cela consiste à partir du terme anoté par les variables de types $(\lambda x_a(\lambda y_b . (x_a(x_a y_b)_c)_d)_e)_f$.

L'algorithme de typage est décrit ci-dessous en Ocaml. Observer que la fonction `type_terme_ferme` associe une variable de type `a` au λ -terme `t`, pose les équations sur les types, les résoud et recherche le type qui est associé à `a` dans les équations résolues.

```
(* termes *)
type terme =
  | Var of string
  | App of terme * terme
  | Lambda of string * terme ;;

(* types *)
```

```

type type_terme =
  | Var_type of int
  | App_type of type_terme * type_terme;;

(* resolution des equations *)
exception Non_typable;;

(* present x t = true ssi (Var_type x) apparait dans le type t *)
let present x t =
  let rec aux = function
    Var_type y -> x = y
  | App_type (a,b) -> (aux a) or (aux b)
  in aux t;;

(* (remplacer u x t) remplace dans le type u , (Var_type x) par le type t *)
let remplacer u x t =
  let rec aux = function
    (Var_type y) as w -> if x = y then t else w
  | App_type (a, b) -> App_type (aux a, aux b)
  in aux u;;

(* (remplacer_equation (u,v) x t) remplace dans les types u et v,
(Var_type x) par le type t *)
let remplacer_equation (u,v) x t = remplacer u x t, remplacer v x t;;

(* (remplacer_systeme s x t) remplace dans la liste s d'équations entre type
(Var_type x) par le type t *)
let remplacer_systeme s x t = List.map (function e -> remplacer_equation e x t) s;;

(* effectuer une étape de résolution des équations entre types *)
let pas (((t1,t2)as eq1)::nr, r) =
  if t1 = t2 then (* supprimer l'équation *) nr, r
  else match eq1 with
    Var_type x, t -> if present x t
      then raise Non_typable else
        (* éliminer x *)
        remplacer_systeme nr x t,((Var_type x, t)::(remplacer_systeme r x t))
  | t, Var_type x -> if present x t
      then raise Non_typable else
        (* éliminer x *)
        remplacer_systeme nr x t ,((Var_type x, t)::(remplacer_systeme r x t))
  | App_type (a,b), App_type (a',b') -> ((a,a')::((b,b')::nr)), r ;;

(* resolution d'un système :
on effectue une suite de pas jusqu'à vider le système non résolu *)
let rec resolution_aux (nr,r) =

```

```

    if nr = [] then r else resolution_aux (pas (nr,r));;
let resolution nr = resolution_aux (nr, []);;

(* FIN de la resolution des equations*)

(* typage d'un terme *)

(* recherche associative si x n'est pas trouvé on lui associe x *)
let cassoc x la = try List.assoc x la with Not_found -> x;;

let i = ref 0;;

let nouvelle_variable_type () = (i:=!i+1;Var_type !i);;

(* (equations a env t) pose les équations pour typer t dans le contexte env,
a est la variable de type associée au terme t *)
let rec equations a env = function
  Var x -> [a, (List.assoc x env)]
| App (u, v) ->
  let b = nouvelle_variable_type () and c = nouvelle_variable_type () in
  (b, App_type (c,a))::
  ((equations b env u) @ (equations c env v))
| Lambda (x, t) ->
  let b = nouvelle_variable_type () and c = nouvelle_variable_type () in
  (a, App_type (b,c))::(equations c ((x,b)::env) t);;

(* calcul du type d'un terme sans variable libre *)
let type_terme_ferme t =
  let a = nouvelle_variable_type () in
  cassoc a (resolution (equations a [] t));;

```

3.2 Système T

Le système simple que nous avons étudié a très peu de pouvoir expressif. Il est impossible avec les λ -termes typés de représenter les fonctions arithmétiques usuelles, notamment la fonction puissance $\lambda ab.a^b$. Nous ne prouverons pas cette impossibilité mais nous donnerons des raisons intuitives qui confortent cette affirmation.

- le terme $\lambda ab.ba$ représente la fonction puissance pour les entiers de Church, ce terme est typable mais les applications de ce terme à deux entiers de Church ne sont pas typables.
- les termes comme $(x x)$ ne sont pas typables, ce qui interdit de typer les combinateurs de point-fixe.

Aussi pour obtenir un langage de programmation plus expressif, on étend le langage des termes et les règles de typage pour les nouveaux termes. C'est ainsi que le langage Ocaml peut être considéré comme une extension du système précédent. Dans ce paragraphe, nous examinons une des premières propositions d'extension du λ -calcul, le système T de Gödel.

3.2.1 Extension du langage

Aux λ -termes, on ajoute les 3 constantes O, S, r . On introduit *nat* comme une constante de type pour typer les entiers naturels. Soit Γ un contexte quelconque, et a une variable de type, les types des constantes sont définies par les axiomes suivants :

1. $\Gamma \vdash O : nat$
0 est un entier naturel
2. $\Gamma \vdash S : nat \rightarrow nat$
 s est une fonction des entiers naturels dans les entiers naturels
3. $\Gamma \vdash r : nat \rightarrow a \rightarrow (nat \rightarrow a \rightarrow a) \rightarrow a$
 r est une fonctionnelle à 3 arguments

Le codage des entiers est immédiat. Notons \underline{n} le code de l'entier n . Le codage est défini par :

1. $\underline{0} = O$
2. $\underline{n+1} = (S \underline{n})$

Par exemple $\underline{2} = (S(S O))$.

Définition 3.2.1 *Réduction en 1 pas.*

Soient k, u, v des termes, aux redex de la forme $((\lambda x . u) v)$ on ajoute deux nouveaux redex avec leur contracté :

1. $(r O u v)$ est contracté en u
2. $(r(S k)u v)$ est contracté en $(v k(r k u v))$

On note par $u \rightarrow_T v$, le fait que le terme v est obtenu en contractant un redex de u . L'indice T peut être omis quand il n'y a pas d'ambiguïté.

On note par $=_T$ la plus petite relation d'équivalence contenant \rightarrow_T et $=_\alpha$.

Nous donnons sans preuve les résultats principaux concernant le système T.

Théorème 3.2.1

1. La relation $\rightarrow_T^* =_\alpha$ est Church-Rosser
2. Les termes typables sont fortement normalisables

La preuve du premier résultat (CR) peut être obtenu en entendant au système T, les résultats analogues prouvés pour le λ -calcul dans [1]. La preuve du deuxième résultat (normalisation forte) figure dans [3]. Une conséquence immédiate de ce théorème est que la relation $=_T$ est décidable : en effet pour savoir si $u =_T v$, il suffit de normaliser les deux termes u et v et de tester si les deux formes normales sont égales au changement près des variables liées.

Ci-dessous nous représentons en Ocaml le type nat et la fonction r.

```

type nat =
  0
  | S of nat
let rec r n u v = match n with
  0 -> u
  | S k -> v k (r k u v)

```

3.2.2 Programmation dans le système T

Dans le système T, on peut programmer toutes les fonctions calculables (au sens de Church), dont la terminaison est prouvable dans l'arithmétique de Péano³. Nous ne prouvons pas cette propriété (voir [3]) mais nous montrons sur des exemples comment programmer dans ce système.

addition

Posons $add = \lambda xy . r x y (\lambda u v . S v)$.

On vérifie aisément (si possible avec une machine) que le type de add est $nat \rightarrow nat \rightarrow nat$. On prouve que add représente bien l'addition. Puisque le système T est CR, il suffit que tous entiers naturels m, n vérifient la propriété $add \underline{m} \underline{n} =_T m + n$.

On prouve cette propriété par récurrence sur m :

- prouvons $add \underline{0} \underline{n} =_T \underline{n}$

$$\begin{aligned} add \underline{0} \underline{n} &= r \underline{0} \underline{n} (\lambda u v . S v) && \text{par } \beta\text{-réduction} \\ &= r \underline{n} && \text{par } r\text{-réduction} \\ &= \underline{n} \end{aligned}$$
- supposons que $add \underline{m} \underline{n} =_T m + n$ et montrons $add \underline{m+1} \underline{n} =_T m + 1 + n$

$$\begin{aligned} add \underline{m+1} \underline{n} &= r \underline{m+1} \underline{n} (\lambda u v . S v) && \text{par } \beta\text{-réduction} \\ &= r (\lambda u v . S v) \underline{m} (r \underline{m} \underline{n} (\lambda u v . S v)) && \text{par } r\text{-réduction} \\ &= r (\lambda u v . S v) \underline{m} \underline{m+n} && \text{par hypothèse de récurrence} \\ &= r (S \underline{m+n}) && \text{par } \beta\text{-réduction} \\ &= \underline{m+1+n} && \text{par définition des codes des entiers} \end{aligned}$$

On termine en donnant la traduction en Ocaml du programme d'addition ci-dessus. On invite le lecteur à tester ce programme.

```

let add = fun x y -> r x y (fun u v -> S v)

```

iteration

Soit $it = \lambda n f x . r n x (\lambda u v . f v)$. On calculera le type de it en s'aidant d'Ocaml ou du programme fourni au paragraphe 3.1.2.

Montrons par récurrence sur l'entier naturel n que l'itérateur it vérifie : $(it \underline{n} f x) =_T f^n x$.

- prouvons que $(it \underline{0} f x) =_T x$

$$\begin{aligned} (it \underline{0} f x) &= r \underline{0} x (\lambda u v . f v) && \text{par } \beta\text{-réduction} \\ &= r x && \text{par } r\text{-réduction} \end{aligned}$$

³le mot fonction est ici employé dans le sens de programme à données et résultat sur les entiers naturels, une preuve de terminaison est une preuve d'arrêt d'un programme

– supposons $(it\ n\ f\ x) =_T f^n\ x$ et montrons que $(it\ \underline{n+1}\ f\ x) =_T f^{n+1}\ x$

$$\begin{aligned} (it\ \underline{n+1}\ f\ x) &=_{T\ r\ \underline{n+1}} x(\lambda u\ v . f\ v) && \text{par } \beta\text{-réduction} \\ &=_{T} (\lambda u\ v . f\ v)\ \underline{n}\ (r\ \underline{n}\ x\ (\lambda u\ v . f\ v)) && \text{par } r\text{-réduction} \\ &=_{T} (\lambda u\ v . f\ v)\ \underline{n}\ f^n\ x && \text{par hypothèse de récurrence} \\ &=_{T} f^{n+1}\ x && \text{par } \beta\text{-réduction} \end{aligned}$$

On fait la traduction en Ocaml du programme d'itération ci-dessus.

```
let it = fun n f x -> r n x (fun u v -> f v)
```

fonction d'Ackerman

Dans ce paragraphe, nous ne distinguons plus les entiers naturels et leurs représentations dans le type nat. Les variables m et n désignent des éléments de type nat.

La fonction d'Ackerman est représentée en Ocaml par le programme suivant.

```
let rec a = function
  0 -> (fun n -> (S n))
| (S m) -> (function
              0 -> (a m (S 0))
            | (S n) -> (a m (a (S m) n)))
```

La fonction d'Ackerman croit plus vite que toute fonction récursive primitive. On montre que cette fonction est aussi programmable dans le système T (ce qui en illustre l'expressivité) en remplaçant la récursivité générale par la récursivité très contrôlée de la fonction it.

1. Par récurrence sur n on prouve que :
 $a\ (S\ m)\ n = (a\ m)^n\ ((a\ m)\ (S\ O))$.
2. D'après les propriétés de it, il en résulte que :
 $a\ (S\ m)\ n = it\ n\ (a\ m)\ ((a\ m)\ (S\ O))$.
3. Soit b la fonction définie dans T par : $b = \lambda f\ n . it\ n\ f\ (f\ (S\ O))$.
 Par définition de b et la propriété (2) on a :
 $a\ (S\ m)\ n = b\ (a\ m)\ n$.
 Par récurrence sur n , on en déduit que :
 $a\ m\ n = b^m\ (a\ O)\ n$.
4. Soit a' la fonction définie dans T par $a' = \lambda m . it\ m\ b\ S$.
 Il est clair que les programmes a et a' définissent la même fonction. En effet dans T, mais pas dans OCaml qui distingue constructeur et fonction, $(a\ 0) = S$, donc : $a'\ m\ n = it\ m\ b\ (a\ 0)\ n = b^m\ (a\ O)\ n = a\ m\ n$

On donne la traduction en Ocaml du programme a' écrit dans le système T. L'unique différence avec le programme ci-dessus est due à la distinction que fait Ocaml entre constructeur et fonction.

```
let b = fun f n -> it n f (f (S 0));;
let a' = fun m -> it m b (fun n -> (S n));;
```

3.3 Système F

Dans ce système, on ne fait aucune extension aux λ -termes mais on modifie les règles de typage.

3.3.1 Définition du typage

Nouveaux types

L'ensemble des types est enrichi en autorisant un quantificateur universel sur les types. Dans ce paragraphe, pour être proche du logiciel de Raffali (cf [raf]) qui permet de programmer dans le système F, les variables de types sont écrites avec des majuscules et on adopte ses conventions.

L'ensemble des types est défini par la syntaxe suivante :
 $\text{type} ::= \text{variable_de_type} \mid \text{type} \rightarrow \text{type} \mid (\text{type}) \mid \bigwedge \text{variable_de_type} \text{ type}.$

Le quantificateur est prioritaire sur la flèche, par suite le type $\bigwedge X X \rightarrow X$ est égal au type complètement parenthésé $((\bigwedge X X) \rightarrow X)$.

Règles

On ne rappelle pas les axiomes et règles non modifiées et on présente uniquement les règles nouvelles concernant le quantificateur. Soit Γ un contexte, t un λ -terme, A et B deux types.

1. généralisation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : (\bigwedge X A)} \text{ GEN}$$

où X est une variable de type non libre dans le contexte Γ .

2. instanciation

$$\frac{\Gamma \vdash t : (\bigwedge X A)}{\Gamma \vdash t : A[X := B]} \text{ INST}$$

$A[X := B]$ est la substitution avec renommage déjà définie pour le λ -calcul (cf 1.2.5) : les occurrences libres de X sont remplacées par B et les variables liées de A , qui capturent des variables libres de B , sont renommées.

Théorème 3.3.1 *Tout terme typable est fortement normalisable.*

On trouvera la preuve dans [3] et dans [1].

Théorème 3.3.2 *Le typage est indécidable.*

Dans le contexte des langages de programmation, la quantification des types existe aussi sous le nom de polymorphisme. Dans Ocaml, l'usage des règles du système F est contrôlé de façon à ce que le typage puisse être calculé automatiquement. Les restrictions adoptées sont les suivantes :

1. Les types du système F sont classés en types simples (sans quantificateur) et schémas de types.

2. Les règles d'application et d'abstraction ne peuvent être utilisées qu'avec des types simples.
3. L'usage du polymorphisme est limitée au typage des termes de la forme $(let\ x = e1\ in\ e2)$. Ce terme est réduit comme $((\lambda x . e2)\ e1)$ mais son typage est défini en Ocaml par la règle suivante :

$$\frac{\Gamma \vdash e1 : \sigma \quad \Gamma, x : \sigma \vdash e2 : \tau}{\Gamma \vdash (let\ x = e1\ in\ e2) : \tau} \text{ LET}$$

Dans cette règle, τ est un type simple et σ est un schéma de type.

Exemple 3.3.1 *On montre comment déduire avec les règles d'Ocaml le type du terme $(let\ x = \lambda y . y\ in\ x\ x)$. Cette déduction est aussi indirectement un exemple du typage de $((\lambda x . x\ x)\ \lambda y . y)$ dans le système F.*

- | | | |
|-----|---|-----------------|
| (1) | $y : X \vdash y : X$ | <i>axiome</i> |
| (2) | $\vdash \lambda y . y : X \rightarrow X$ | <i>abs 1</i> |
| (3) | $\vdash \lambda y . y : \bigwedge X(X \rightarrow X)$ | <i>gen 2</i> |
| (4) | $x : \bigwedge X(X \rightarrow X) \vdash x : \bigwedge X(X \rightarrow X)$ | <i>axiome</i> |
| (5) | $x : \bigwedge X(X \rightarrow X) \vdash x : (X \rightarrow X)$ | <i>inst 4</i> |
| (6) | $x : \bigwedge X(X \rightarrow X) \vdash x : (X \rightarrow X) \rightarrow (X \rightarrow X)$ | <i>inst 4</i> |
| (7) | $x : \bigwedge X(X \rightarrow X) \vdash (x\ x) : (X \rightarrow X)$ | <i>app 6, 5</i> |
| (8) | $\vdash ((\lambda x . x\ x)\ \lambda y . y) : (X \rightarrow X)$ | <i>let 3, 7</i> |

On recommande au lecteur de vérifier que ce terme est bien typable en Ocaml.

3.3.2 Programmation dans le système F

Le système F est au moins aussi expressif que le système T, car on peut coder dans F, la récursion primitive fonctionnelle. De plus, ainsi qu'il est montré dans [1] et [3], on peut représenter dans le système F tous les types inductifs (couple, listes, arbres,...). Tous les programmes ci-dessus ont été écrits et testés grâce au logiciel de Mr Raffali ([5]). Ses conventions ont été adoptées, en particulier λ est remplacé ci-dessous par \backslash .

Représentations de booléens et des couples

On prend la même représentation que dans le λ -calcul pur, en vérifiant que le typage dans F est possible.

```
# les booleens
# T = vrai, F = faux, IF est le conditionnelle
type Bool = \X (X ->X -> X);
let T = \x y.x :Bool;
let F = (\x y.y):Bool;
let IF = (\b x y.b x y):\verb+\+X (Bool->X->X->X);# les couples

# les couples
# PAIR construit un couple
# FST projette un couple sur sa première composante
# SND projette un couple sur sa deuxième composante
```

```

type P[A,B] = /\X ((A -> B -> X) -> X);
let PAIR = (\x y f. f x y): /\A /\B (A->B->P[A,B]);
let FST = (\p.p(\x y.x)):\A/\B(P[A,B]->A);
let SND = (\p.p(\x y.y)):\A/\B(P[A,B]->B);

```

Représentations des entiers

On code les entiers avec la représentation de Church. Rappelons que nous avons noté c_n l'entier de Church codant l'entier naturel n . On donne sans commentaires les programmes réalisant les opérations successeur, addition, multiplication et prédécesseur.

```

# les entiers
type Nat = /\X((X->X)->X->X);
# L'entier 0
let 0 = (\f x.x):Nat;
# La fonction successeur
let S = (\n f x.f (n f x)): Nat -> Nat;
# L'addition
let add = (\n m f x. n f (m f x)): Nat -> Nat ->Nat;
# La multiplication
let mul = (\n m f.n (m f)): Nat -> Nat ->Nat;
# La fonction predecesseur
let pred = (\n.(n(\p x y.p(S x)x)(\x y.y) 0 0)):Nat-> Nat;

```

Le programme `pred` calcule c_0 comme prédécesseur de c_0 et c_n comme prédécesseur de c_{n+1} . On recommande de tester puis de prouver la correction de ces programmes.

Programmes de l'itération et de la récursion primitive fonctionnelle

Le programme pour l'itération est particulièrement simple, à cause de la représentation des entiers par des entiers de Church. On le donne sans commentaire.

```

# L'iteration : it n f u = f^n u
let it = (\n f u.n f u): /\X (Nat ->(X->X)->X ->X);

```

La récursion r est programmable à partir de l'itération. On explique la démarche qui conduit au programme avant de conclure par ce programme.

Pour obtenir $(r\ c_n\ u\ v)$ où n est un entier naturel, on calcule les suites $x_{(i:0\leq i\leq n)}$ et $y_{(i:0\leq i\leq n)}$ ci-dessous :

- $x_0 = u$
- $y_0 = (PAIR\ c_0\ x_0)$
- $x_{i+1} = v\ c_i\ u_i$
- $y_{i+1} = (PAIR\ c_{i+1}\ u_{i+1})$

Par définition de r , on a : $r\ c_n\ u\ v = SND\ y_n$.

Soit h la fonction définie par : $h\ v\ x = PAIR\ (S\ (FST\ x))\ (v\ (FST\ x)\ (SND\ x))$.

Par définition de h , on a : $y_{i+1} = (h\ v)^i\ y_0$.

Donc d'après la spécification de it , on a : $y_n = (h v)^n y_0 = it n (h v)(PAIR c_0 u)$.
 Puisque $r c_n u v = SND y_n$, on en déduit immédiatement le programme pour r écrit dans le système F.

```
let h = (\v x.(PAIR (S (FST x))(v (FST x) (SND x)))) :
  /\X ((Nat->X-> X)->P[Nat,X] -> P[Nat,X]);
let r = (\n u v. SND (it n (h v) (PAIR 0 u))) :
  /\X (Nat -> X -> (Nat -> X -> X) -> X);
```


Bibliographie

- [1] J.L.Krivine : Lambda-calcul, types et modèles.
Masson (1990)
- [2] H.P.Barendregt : The Lambda Calculus, Its Syntax and Semantics.
Elsevier Science Publisher (1985)
- [3] J.Girard, Y.Lafont, P.Taylor : Proof and Types.
Cambridge University Press (1989)
- [4] JP.Jouannaud, C.Kirchner : Solving equations in abstract algebras, a rule-based survey of unification. Computational Logic : Essays in Honor of Alan Robinson.
MIT Press (1991)
- [5] C.Raffali [http ://www.lama.univ-savoie.fr/sitelama/Membres/pages_web//RAFFALLI/normaliser.html](http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web//RAFFALLI/normaliser.html)