Introduction au lambda-calcul pur

Yves Bertot

Février 2006

1 Le plus petit langage de programmation du monde

Pour l'informaticien, l'étude du λ-calcul (prononcer lambda-calcul) permet de comprendre sur un langage minimal des concepts théoriques qui pourront par la suite se reporter sur des langages de programmation plus riches. Par exemple, nous pourrons raisonner sur la termination des programmes, sur l'équivalence entre deux programmes, sur les relations entre la programmation et la logique. Ce langage est l'exemple le plus simple de langage fonctionnel et toutes les études effectuées sur ce langage se reporteront naturellement dans toute la classe des langages fonctionnels (ML, Haskell, Scheme). Même pour les langages non fonctionnels, le λ-calcul permet de fournir une compréhension de certains phénomènes, comme les appels de procédures et la récursion.

1.1 Syntaxe

On donne habituellement la syntaxe du λ-calcul en disant que c'est l'ensemble des expressions e formées de la façon suivante :

$$e ::= \lambda x.\ e | e_1 e_2 | x$$

Pour une présentation un peu plus précise, on suppose l'existence d'un ensemble V infini de variables (que nous noterons x, y, ..., x_i, y_i, f, g) et l'ensemble des λ-termes est constitué de la façon suivante :

1. Si x est une variable et e est un λ-terme déjà construit, alors $\lambda x.\ e$ est un λ-terme, nous appellerons un tel λ-terme une *abstraction*.
2. Si e_1 et e_2 sont des λ-termes déjà construits alors $e_1 e_2$ (la juxtaposition des termes) est un λ-terme, nous appellerons un tel λ-terme une *application*.
3. Si x est une variable, alors c'est également un λ-terme.

Quelques exemples fréquemment rencontrés de λ-termes :

$\lambda x.\ x$	(on l'appelle **I**),
$\lambda x.\ \lambda\ y.\ x$	(**K**),
$\lambda x.\ x\ x$	(Δ),
$\lambda x.\ \lambda y.\ \lambda z.x\ z\ (y\ z))$	(**S**),
$\lambda x.\ \lambda y.\ x\ y.$	

Conventions de parenthésage On convient d'éviter d'écrire des parenthèses au maximum. Une abstraction commence au symbole λ et se continue aussi loin que possible, tant que ceci n'entre pas en conflit avec une parenthèse.

Ainsi, il n'est jamais nécessaire d'écrire les parenthèses dans la formule $\lambda x.\ (x\ y)$[1]

En revanche, l'application d'une fonction à une variable ne donne en général pas lieu à l'usage de parenthèses. Les parenthèses dans l'expression $(f\ x)\ y$ sont inutiles.

1.2 α-équivalence, variables libres et liées

Intuitivement, il faut voir une abstraction comme la description d'une fonction : $\lambda x.\ e$ représente *la fonction qui à x associe l'expression e*. Ainsi les termes donnés plus haut représentent des fonctions simples à décrire : **I** représente la fonction identité, **K** représente une fonction qui prend un argument et retourne une fonction constante, Δ n'est pas simple à interpréter comme une fonction mathématique : elle reçoit un argument et l'applique à lui-même. L'argument qu'elle reçoit doit donc être à la fois une fonction et une donnée.

Avoir des fonctions qui reçoivent des fonctions en argument n'est pas étranger à la pratique informatique : par exemple, un compilateur ou un système d'exploitation reçoivent des programmes en argument. Appliquer une fonction à elle-même n'est pas complètement absurde non plus : on peut compiler un compilateur avec lui-même.

Dans le terme $\lambda x.\ e$, la variable x peut bien sûr apparaitre dans e. Si la variable y n'apparait pas déjà dans e et si l'on obtient une expression e' en remplaçant toutes les occurrences de x par y, on représente la même fonction. On dit que les deux expressions sont *α-équivalentes*. Pour la majeure partie de nos travaux, toutes les discussions se feront modulo α-équivalence, c'est à dire que nous considérerons généralement que deux termes sont égaux s'ils sont α-équivalents.

La relation d'α-équivalence est une relation d'équivalence et c'est aussi une relation de congruence : si e et e' sont α-équivalents alors $\lambda x.\ e$ et $\lambda x.\ e'$ le sont aussi, $e\ e_1$ et $e'\ e_1$ le sont aussi, et $e_1\ e$ et $e_1\ e'$ le sont aussi.

Quelques exemples et contre-exemples d'α-équivalence :

1. $\lambda x.\ \lambda y.\ y$, $\lambda y.\ \lambda x.\ x$, et $\lambda x.\ \lambda x.\ x$ sont α-équivalents.
2. $\lambda x.\ x\ y$ et $\lambda y.\ y\ y$ ne sont pas α-équivalents.

La construction d'abstraction est une construction liante : les variables de même nom qui apparaissent dans le terme peuvent être renommées en suivant le renommage de la variable qui suit immédiatement un λ qui englobe ce terme. Mais certaines variables apparaissant dans une expression ne sont pas liées, elle sont dites *libres*. Intuitivement, une variable x est libre dans un terme si et seulement si cette variable n'apparait sous aucune expression de la forme $\lambda x.e$. La notion de variable libre est stable par α-équivalence.

[1] cette erreur de style peut apparaître dans ces notes de cours, dont les premières versions furent écrites quand l'auteur n'avait pas compris cette convention.

Par exemple, la variable y est libre dans les termes $\lambda x.\ x\ y$, $\lambda x.\ y$, et dans le terme $\lambda x.\ y\ (\lambda y.\ y)$.

Exercices

1. *Quels sont les termes α-équivalents parmi $\lambda x.\ x\ y$, $\lambda x.\ x\ z$, $\lambda y.\ y\ z$, $\lambda z.\ z\ z$, $\lambda z.\ z\ y$, $\lambda f.\ f\ y$, $\lambda f.\ f\ f$, $\lambda y.\ \lambda x.\ x\ y$, $\lambda z.\ \lambda y.\ y\ z$.*
2. *Trouver un terme α-équivalent au terme suivant dans lequel chaque lieur lie une variable de nom différent :*

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

1.3 Application

L'application correspond à l'application d'une fonction à un argument. Le λ-calcul ne fournit que ce mode d'application. Il n'y a pas d'application d'une fonction à plusieurs argument, car celui-ci peut se décrire à l'aide de l'application à un seul argument pour une raison simple : le résultat d'une fonction peut lui même être une fonction, que l'on appliquera de nouveau.

Ainsi, nous verrons plus tard que l'on peut disposer d'une fonction d'addition dans le λ-calcul. Il s'agit d'une fonction à deux arguments, que l'on appliquera à deux arguments en écrivant (les parenthèses ne sont pas nécessaire mais nous les écrivons pour souligner la structure du terme) :

$$((plus\ x)\ y)$$

Par exemple, on pourra représenter la fonction qui calcule le double d'un nombre en écrivant :

$$\lambda x.((plus\ x)\ x)$$

Dans la suite, on évitera l'accumulation de parenthèses en considérant qu'il n'est pas nécessaire de placer les parenthèses internes lorsqu'une fonction est appliquée à plusieurs arguments. La fonction ci-dessus pourra s'écrire de la façon suivante :

$$\lambda x.plus\ x\ x$$

Cette disparition des parenthèses n'indique pas que l'application est associative : l'application n'est pas associative. Dans la fonction suivante, on ne peut pas enlever les parenthèses :

$$\lambda x.plus(x\ x)$$

Cette fonction n'a pas du tout le même sens que la précédente.

En termes de notations, nous noterons également avec un seul λ les fonctions à plusieurs arguments, de sorte que l'on écrira $\lambda xyz.\ e$ à la place de $\lambda x.\ \lambda y.\ \lambda z.\ e$.

Exercices

3. *Construire le terme qui représente la fonction qui reçoit deux arguments et applique le premier à deux arguments, qui sont tous les deux le second argument,*
4. *Construire le terme qui représente la fonction qui reçoit deux arguments et applique le premier au résultat de l'application du premier argument au second,*
5. *Construire le terme qui représente la fonction qui reçoit deux arguments et applique le deuxième au premier.*

1.4 Substitution

On peut remplacer toutes les occurrences d'une variable liée par un λ-terme, mais il faut faire attention que cette opération soit stable par α-équivalence. Plus précisément, nous noterons $e[e'/x]$ le terme obtenu en remplaçant toutes les occurrences libres de x par e' dans e. L'opération doit en fait se faire en deux temps :

1. d'abord construire un terme α-équivalent à e où aucune des abstractions n'utilise x ou l'une des variables libres de e',
2. ensuite remplacer toutes les occurrences de x par e'.

Les variables libres de $e[e'/x]$ sont alors les variables libres de e et les variables libres de e'.

On peut également décrire récursivement l'opération de substitution par les équations suivantes :

- $x[e'/x] = e'$,
- $y[e'/x] = y$, si $y \neq x$,
- $(e_1\ e_2)[e'/x] = e_1[e'/x]\ e_2[e'/x]$,
- $(\lambda x.\ e)[e'/x] = \lambda x.\ e$,
- $(\lambda y.\ e)[e'/x] = \lambda y.(e[e'/x])$, si y n'est pas libre dans e',
- $(\lambda y.\ e)[e'/x] = \lambda z.((e[z/y])[e'/x])$, si z n'apparait pas libre dans e et e'.

La dernière équation s'applique aussi quand les deux précédentes s'appliquent, mais on appliquera celles-ci de préférence quand c'est possible. Si l'on applique la dernière alors que les précédentes s'appliquent, on obtient simplement un terme α-équivalent.

Voici quelques exemples de substitutions, nous effectuons parfois des renommages et des termes α-équivalents seraient aussi acceptables :

$$\begin{aligned} x\ y[\lambda z.\ x/y] &= x\ (\lambda z.\ x) \\ \lambda x.x\ y[\lambda z.\ x/y] &= \lambda\ t.\ t\ (\lambda z.\ x) \\ \lambda x.\ z\ (\lambda y.\ z\ y)[\lambda x.\ x\ y/z] &= \lambda x.\ (\lambda x.\ x\ y)\ (\lambda t.\ (\lambda x.\ x\ y)\ t) \end{aligned}$$

Notez que l'on a renommé la variable liée dans $\lambda y.\ z\ y$ pour éviter la capture de la variable libre y apparaissant dans le terme substitué.

1.5 Exécution dans le λ-calcul

On définit une notion de β-réduction (prononcer béta-réduction) dans les λ-termes, basée sur la substitution. L'intuition de cette réduction est la suivante : toute fonction appliquée à un argument peut être déroulée. Ce comportement se décrit en définissant une relation binaire notée $\rightarrow$ de la façon suivante :

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

Cette règle doit s'utiliser sur toutes les instances possibles dans un terme, c'est à dire que toute occurrence du membre gauche dans un terme peut être remplacée par l'instance correspondante du membre droit. On a souvent le choix et ceci pose quelques problèmes intéressants.

Toute instance du membre gauche est appelée un *β-redex*, ou simplement un redex. On considère également des enchainements de réductions élémentaires, que nous pourrons appeler des *dérivations* ou même parfois des *réductions* (sous-entendus des réductions non-élémentaires). Nous noterons $\rightarrow^\star$ la relation de dérivation.

Un terme qui ne contient aucun redex est appelé un terme en forme *normale*. Nous dirons qu'un terme possède une forme normale s'il existe une dérivation commençant par ce terme et finissant par un terme en forme normale.

Voici quelques exemples de réduction :

- $((\lambda x.\lambda y.\ x)\lambda x.\ x)\ z \rightarrow (\lambda y.\lambda x.\ x)\ z \rightarrow \lambda x.\ x$,
- $\mathbf{K}\ z(y\ z) = (\lambda xy.\ x)z(y\ z) \rightarrow (\lambda y.\ z)\ (y\ z) \rightarrow z$,
- $\mathbf{S}\ \mathbf{K}\ \mathbf{K} = (\lambda xyz.\ x\ z(y\ z))\mathbf{K}\mathbf{K} \rightarrow (\lambda yz.\ \mathbf{K}z(y\ z))\mathbf{K} \rightarrow (\lambda yz.\ z)\mathbf{K} \rightarrow \lambda z.\ z = \mathbf{I}$
- $\Delta\ \Delta = (\lambda x.\ x\ x)\ \Delta \rightarrow \Delta\ \Delta \rightarrow \Delta\ \Delta$, le terme $\Delta\ \Delta$ est souvent noté Ω.
- $\mathbf{K}\ \mathbf{I}\ \Omega \rightarrow \mathbf{K}\ \mathbf{I}\ \Omega \rightarrow \ldots$,
- $\mathbf{K}\ \mathbf{I}\ \Omega \rightarrow^\star \mathbf{I}$.

Ces exemples montrent qu'il existe des termes sans forme normale et qu'il existe des termes possèdant une forme normale mais d'où peut quand même partir une dérivation infinie.

2 Un langage de programmation

A partir de maintenant nous utiliserons la notation $\lambda x\ y\ z.e$ à la place

$$\lambda x.\ \lambda y.\ \lambda z.\ e.$$

2.1 Nombres naturels

On peut représenter les nombres entiers positifs par les expressions de la forme $\lambda f\ x.f(f \cdots (f\ x) \cdots)$. Par exemple, 0 est le terme $\lambda f\ x.x$, 1 est le terme $\lambda f\ x.\ f\ x$, 2 est le terme $\lambda f\ x.\ f(f\ x)$, et ainsi de suite. On peut alors représenter les fonctions arithmétiques les plus basiques de la façon suivante :

incrémentation $\lambda x\ f\ z.x\ f(f\ z)$,

addition $\lambda x\ y\ f\ z.\ x\ f(y\ f\ z)$,

multiplication $\lambda x\ y\ f\ z.x\ (y\ f)\ z$,

Dans la suite nous noterons *succ*, *plus* et *mult* ces fonctions. Dans la littérature ces nombres sont appelés *entiers de Church*.

Exercices

6. *Donner une représentation alternative de incr.*
7. *Ecrire les nombres 2 et 3, construire la dérivation correspondant à* $2 + 3$*, reconnait-on bien le nombre* 5 *?*
8. *Ecrire la dérivation correspondant à* 2×3*.*
9. *Ecrire la fonction d'exponentiation.*

2.2 Couples de données

On peut représenter la construction d'une paire de valeurs et la récupération des composantes par les fonctions suivantes :

paire $\lambda x\ y\ z.\ z\ x\ y$,

première composante $\lambda p.\ p\ K$

deuxième composante $\lambda p.\ p\ \lambda x\ y.\ y$

Dans la suite nous noterons *pair*, *fst*, *snd* ces trois fonctions.

Maintenant que nous diposons de ces fonctions nous pouvons définir la décrémentation et la soustraction :

décrémentation $\lambda x.fst\ (x\ (\lambda p.\ pair\ (snd\ p)\ (incr\ (snd\ p)))\ (pair\ 0\ 0))$

soustraction $\lambda xy.\ (y\ decr\ x)$, la fonction *decr* est la fonction précédente.

Dans la suite nous noterons *decr* et *sub* ces deux fonctions.

Exercices

10. *Ecrire la dérivation qui montre que* $fst(pair\ x\ y)$ *se réduit en plusieurs étapes en la bonne valeur.*
11. *Ecrire la dérivation qui montre que* $decr(2)$ *se réduit dans la bonne valeur.*

2.3 Valeurs booléennes

Nous pouvons également représenter les valeurs booléennes par $\lambda x\ y.\ x$ (pour T) et $\lambda x\ y.\ y$ pour F. La fonction $\lambda b\ x\ y.b\ x\ y$ peut alors être utilisée pour représenter une expression conditionnelle. Dans la suite nous noterons *if* cette fonction. Nous pouvons définir quelques autres fonctions booléennes.

test à zéro $\lambda x.x\ (K\ F)\ T$ (nous noterons cette fonction *eq0*),

comparaison $\lambda x\ y.\ eq0\ (sub\ x\ y)$,

negation $\lambda x.\ if\ x\ F\ T$ (nous noterons cette fonction *not*),

conjonction $\lambda x\ y.\ if\ x\ y\ x$,
disjonction $\lambda x\ y.\ if\ (not\ x)\ y\ x$.
Dans la suite nous noterons *eq0*, *le*, *not*, *and*, *or* ces fonctions.

Exercices

12. *Quelle est la différence entre F et 0 ? Quelle est la différence entre F et snd ? La valeur T est-elle utilisée ailleurs ?*
13. *Construire la dérivation qui montre que $eq0\ 0 \rightarrow^\star T$,*
14. *Construire la dérivation qui montre que $eq0\ 2 \rightarrow^\star F$,*
15. *Définir le test d'égalité entre deux nombres entiers.*

2.4 Récursivité

L'expression suivante n'admet pas de forme normale, mais elle est intéressante quand même :

$$\Theta = (\lambda z\ x.\ x(z\ z\ x))\lambda z\ x.\ x(z\ z\ x) = ZZ$$

En effet on voit rapidement que cette expression a la propriété suivante :

$$\Theta \rightarrow \lambda x.\ x(Z\ Z\ x) = \lambda x.\ x(\Theta\ x)$$

Si f est une fonction quelconque, alors on a :

$$\Theta\ f \rightarrow^\star f(\Theta\ f)$$

Donc la fonction Θ permet d'associer à n'importe quelle fonction un point fixe de cette fonction. Quel intérêt ?

Si $fx = e$ est la définition récursive d'une fonction, alors cette fonction pourra être décrite dans le λ-calcul par le terme :

$$\Theta(\lambda f\ x.\ e)$$

Par exemple, nous avons défini assez de λ-termes pour décrire la fonction factorielle :

$$fact = \Theta(factF)$$

avec $factF$ définie de la façon suivante :

$$factF = \lambda\ f\ x.(if\ (le\ x\ 1)\ 1\ (mult\ x\ (f\ (decr\ x))))$$

Voyons par exemple comment calculer $fact\ 1$ et $fact\ 3$:

$$\begin{array}{rcl}
fact\ 1 & = & (\lambda\ f\ x.(if\ (le\ x\ 1)\ 1\ (mult\ x\ (f\ (decr\ x)))))(\Theta\ factF)1 \\
& \rightarrow^\star & \lambda x.(if\ (le\ x\ 1)\ 1\ (mult\ x\ ((\Theta\ factF)\ (decr\ x))))1 \\
& \rightarrow^\star & (if\ (le\ 1\ 1)\ 1\ (mult\ 1\ ((\Theta\ factF)\ (decr\ 1)))) \\
& \rightarrow^\star & (if\ T\ 1\ (mult\ 1\ ((\Theta\ factF)\ (decr\ 1)))) \\
& \rightarrow^\star & 1
\end{array}$$

$$
\begin{array}{rcl}
fact\ 3 & = & (\lambda\ f\ x.(if\ (le\ x\ 1)\ 1\ (mult\ x\ (f\ (decr\ x)))))(\Theta\ factF)3 \\
& \rightarrow^\star & \lambda x.(if\ (le\ x\ 1)\ 1\ (mult\ x\ ((\Theta\ factF)\ (decr\ x))))3 \\
& \rightarrow^\star & (if\ (le\ 3\ 0)\ 1\ (mult\ 3\ ((\Theta\ factF)\ (decr\ 3)))) \\
& \rightarrow^\star & (if\ F\ 1\ (mult\ 3\ ((\Theta\ factF)\ (decr\ 3)))) \\
& \rightarrow^\star & (mult\ 3\ ((\Theta\ factF)\ (decr\ 3))) \\
& \rightarrow^\star & (mult\ 3\ ((\Theta\ factF)\ 2))
\end{array}
$$

Le langage ainsi obtenu est très expressif : il a la puissance des machines de Turing. Ce langage a déjà été étudié dans la littérature et a été nommé *PCF* (pour *Programming language for Computable Functions*).

Exercices

16. *Construire la fonction qui représente*

$$sum = n \mapsto \Sigma_{i=0}^{n} i,$$

construire la dérivation qui calcule la valeur sum 2.

17. *On peut proposer une autre représentation pour les nombres entiers, où 0 est représenté par*

$$pair\ F\ F$$

et succ est représentée par

$$\lambda n.\ pair\ T\ n.$$

En gros, le nombre n est représenté par la liste contenant n occurrences de T et terminée par une occurrence de F. Construire, pour cette représentation les fonctions d'addition, multiplication, soustraction, test à zéro, comparaison.

3 Théorèmes de confluence

Il y a un choix non déterministe dans le redex que l'on prend pour commencer une dérivation. On est donc amené à se poser la question si ces choix influent sur la terminaison des dérivations ou sur la valeur de la forme normale atteinte. Le résultat est que la forme normale ne dépend pas des choix, mais que la terminaison en dépend. Nous ne ferons pas toutes les démonstrations, mais nous en ébaucherons la structure principale.

3.1 Confluence locale

Si on peut effectuer deux réductions différentes sur un terme, alors il existe un terme qui peut être atteint par une dérivation à partir des deux termes

obtenus. Mathématiquement cela s'écrit de la façon suivante :

$$e \to e_1 \wedge e \to e_2 \Rightarrow \exists e'.\ e_1 \to^\star e' \wedge e_2 \to^\star e'$$

Pour démontrer ce résultat, il suffit d'utiliser une notion de résidu : il faut réduire dans e_1 tous les rédex résidus de celui qui a été réduit pour obtenir e_2, et dans e_2 tous les résidus de celui qui a été réduit pour obtenir e_1.

Si on définit une nouvelle notion de réduction, où l'on s'autorise à réduire tous les redex pris dans un ensemble à chaque étape et que nous noterons $\to'$, alors on a même une confluence forte :

$$e \to' e_1 \wedge e \to' e_2 \Rightarrow \exists e'.\ e_1 \to' e' \wedge e_2 \to' e'$$

3.2 Confluence globale

La confluence globale correspond à l'expression suivante :

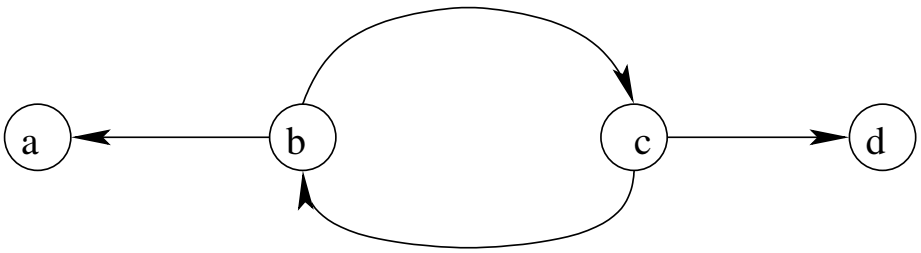
$$e \to^\star e_1 \wedge e \to^\star e_2 \Rightarrow \exists e'.\ e_1 \to^\star e' \wedge e_2 \to^\star e'$$

Elle exprime que si l'on a fait n'importe quel calcul et que le voisin a fait n'importe quel calcul, on peut continuer tous les deux d'une façon qui nous amènera au même endroit. Cette propriété se généralise en une propriété connue sous le nom de *propriété de Church-Rosser*, qui a l'énoncé suivant :

$$e_1 \leftrightarrow^\star e_2 \Rightarrow \exists e', e_1 \to^\star e' \wedge e_2 \to^\star e'$$

La confluence locale ne suffit pas pour assurer la confluence globale, car il faut se méfier des dérivations infinies (la confluence locale dans un système ne contenant aucune dérivation infinie suffirait à assurer la confluence globale).

Pour une relation arbitraire, on peut définir de la même manière que pour la β-réduction les notions de forme normale, confluence locale, et confluence forte. La relation représentée par le schéma suivant possède la propriété de confluence locale, mais pas la propriété de confluence globale.



En revanche la confluence locale forte suffit à assurer la confluence globale. Comme la relation $\to'$ est fortement localement confluente, nous en déduisons qu'elle est globalement confluente et ceci impose que la relation $\to$ est globalement confluente.

L'un des corollaires les plus importants de ce théorème de confluence globale est que la forme normale, lorsqu'elle existe, est unique.

4 Théorème de standardisation

Il existe une stratégie qui permet d'atteindre toujours la forme normale, lorsque celle-ci existe. Cette stratégie peut être qualifiée de paresseuse, car elle impose de ne pas évaluer les arguments d'une fonction avant de savoir si ces arguments seront vraiment utilisés dans le calcul. Dans le λ-calcul elle présente en revanche l'inconvénient d'être très inefficace. Le motto est donc : évaluer le redex le plus à gauche. Cette stratégie est parfois également qualifiée d'*appel par nom*.

Une autre stratégie, utilisée dans la majeure partie des langages de programmation est de toujours évaluer les arguments au moment d'appeler les fonctions, cette stratégie est qualifiée d'*appel par valeur*. En fait, les langages de programmation ne font pas seulement de l'appel par valeur, car cette statégie est certaine de partir dans une dérivation infinie si une telle dérivation infinie est possible. En général, il existe au moins une fonction qui n'évalue pas tous ses arguments et c'est souvent la construction conditionnelle.

Malgré sa tendance à diverger, la stratégie d'appel par valeur est souvent bien plus efficace que la stratégie d'appel par nom, car les calculs sont effectués avant d'être dupliqués.

Exercices

16. *Comparer la dérivation en appel par nom et en appel par valeur des expressions* $\mathbf{KI}\Omega$, $\mathbf{SKK}$, $\mathbf{\Theta}\lambda x\ y.y$.
17. *Comparer la dérivation en appel par nom et en appel par valeur de* $fact\ 3$, *(pour l'appel par valeur, on supposera que l'opérateur if fonctionne toujours en appel par valeur seulement sur son premier argument).*

5 Un langage encore plus simple

Curry a également proposé un calcul minimal, en fait encore plus simple que le le λ-calcul, car il ne comporte pas de variables, c'est celui de la logique combinatoire (nous verrons plus tard pourquoi on l'appelle *logique* et pas calcul).

Ce calcul repose sur l'application comme dans le λ-calcul et deux constantes $S\ K$ dont le comportement est le suivant :

$$\begin{aligned} S\ x\ y\ z &\rightarrow x\ z\ (y\ z) \\ K\ x\ y &\rightarrow x \end{aligned}$$

Ces deux constantes sont également appelées des combinateurs. Toutes les expressions du λ-calcul peuvent être représentée par des combinaisons de ces combinateurs, de telle manière que les notions de réduction coïncident. On aurait donc pu également étudier les notions de confluence, standardisation, représentation des langages de programmation à l'aide de ce langage.

Pour transformer une expression du λ-calcul en une expression de la logique combinatoire, il suffit de s'autoriser à faire cohabiter les constante S et K et d'appliquer les remplacements suivants autant que possible :

$$\begin{array}{rcll} \mathbf{S} & \rightarrow & S & \\ \mathbf{K} & \rightarrow & K & \\ \lambda x.\ x & \rightarrow & SKK & \\ \lambda x.\ y & \rightarrow & Ky & (\text{si } x \neq y) \\ \lambda x.\ S & \rightarrow & KS & \\ \lambda x.\ K & \rightarrow & KK & \\ \lambda x.\ e_1\ e_2 & \rightarrow & S(\lambda x.\ e_1)(\lambda x.\ e_2) & \end{array}$$

Si on applique ces transformations au maximum, on obtient un terme qui ne contient aucune abstraction. Ce nouveau terme est équivalent au λ-terme initial à plus d'un titre, mais nous n'aurons pas le temps d'étudier cette équivalence.

6 Conclusion

Le λ-calcul a été inventé par A. Church dans les années 1930, et a surtout été conçu pour étudier les fondements de la logique et des mathématiques. Néanmoins, son importance du point de vue informatique est apparue très rapidement et depuis les années 1970, ce sont surtout des informaticiens qui s'y sont intéressés.

Les livres d'introduction au λ-calcul sont assez fréquents, on en trouve même de très bons en Français. Le livre de J.L. Krivine [2] en est un très bon exemple. Ce livre s'adresse à un public de D.E.A. et je conseille plutot de n'y prendre que les deux premiers chapitres. Le livre de René Lalement [3] traite un sujet plus large, puisqu'il se consacre à la logique. Néanmoins on pourra lire avec intérêt les chapitres se reportant au λ-calcul. Ici encore, je conseille de ne lire que les deux premiers chapitres pour les besoins de ce cours. Un petit inconvénient de ce livre est qu'il contient quelques erreurs typographiques, par exemple R. Lalement donne la définition de *pair* que j'ai donnée plus haut de façon erronée. La copie dont je dispose (celle de la bibliothèque de l'INRIA Sophia) contient des corrections manuscrites, mais probablement pas pour l'ensemble du livre.

Vous trouverez également un cours complet sur "la théorie des langages de programmation", rédigé par G. Dowek et disponible sur internet à l'adresse suivante :

`http://pauillac.inria.fr/~dowek/Cours/tlp.ps.gz`

On y trouvera une description du langage PCF.

Enfin, l'ouvrage de référence sur le λ-calcul est un ouvrage en anglais, écrit par H. Barendregt [1]. Cet ouvrage couvre l'ensemble des recherches sur le λ-calcul telles qu'elles étaient connues dans les années 1980. Il reste l'ouvrage de référence sur le sujet. Ici encore, le contenu de ce livre dépasse largement l'introduction que ce cours voudrait présenter.

Références

[1] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic. North-Holland, 1984.

[2] Jean-Louis Krivine. *Lambda Calcul, types et modèles*. Masson, 1990.

[3] René Lalement. *Logique, réduction, résolution*. Masson, 1990.