lambda-calcul et types

Yves Bertot

▶ **To cite this version:**

Yves Bertot. lambda-calcul et types. Ecole Jeunes Chercheurs en Programmation (CNRS-INRIA) Toulouse, 2006. inria-00083975

HAL Id: inria-00083975
https://cel.hal.science/inria-00083975

Submitted on 5 Jul 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

inria-00083975, version 1 - 5 Jul 2006

lambda-calcul et types

Yves Bertot

Mai 2006

Dans ces notes de cours, nous survolons plusieurs aspects du λ-calcul et du typage des langages fonctionnels. Les connaissances abordées ici permettent une meilleure compréhension des langages de programmation fonctionnels comme Lisp, ML [12], OCaml [15] et des systèmes de preuve comme Coq [5].

Paradoxalement, nous commençons par décrire un modèle réduit de langage de programmation sans types, puis nous montrons plusieurs extensions de ce langage avec des notions de typage. En particulier, nous montrons comment les types peuvent être utilisés pour représenter des formules logiques et les termes typés comme des preuves.

L'un des points les plus difficiles de ce cours est l'introduction de types dépendants, qui permettent de définir des fonctions dont le type exprime très précisément les propriétés requises des entrées et les propriétés assurées pour les résultats. Un tel système de type permet de construire des programmes pour lesquels beaucoup d'erreurs de programmation peuvent être détectées au moment de la compilation. Vu comme un outil pour faire de la logique, un tel système de type peut servir à construire des vérificateurs de démonstrations sur ordinateur, comme le système Coq. Puisqu'il s'agit de décrire ensemble un langage de programmation et système logique, nous obtenons un outil avec lequel nous pouvons prouver que les programmes ne contiennent pas certaines formes d'erreurs.

1 Un rappel sur le lambda-calcul pur

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 terminaison 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). De plus, le λ-calcul permet de comprendre certains phénomènes comme les appels de procédures et la récursion même pour les langages non fonctionnels,

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*. Intuitivement, $\lambda x.e$ est la fonction qui envoie x vers e.
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**),
$(\lambda f.\ \lambda x.(f(((xx)f))))(\lambda f.\ \lambda x.(f(((xx)f))))$	($\mathbf{Y}_T$)
$\lambda x.\ \lambda y.\ (x\ y)$.	

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 apparaître dans e. On peut remplacer x par une autre variable y, à condition de remplacer toutes les occurrences de x par y, et à condition que y n'apparaissent pas déjà dans e. La nouvelle expression représent alors 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[1] : 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 instances de x qui apparaissent dans le terme $\lambda x.\ e$ sont liées à cette abstraction. On peut changer le nom de ces occurrences en même temps que celui introduit par l'abstraction. 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.

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 introduit 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 arguments, car ceci 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 :

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

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

[1] La relation d'α-conversion elle passe au contexte.

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

1.4 Substitution

On peut remplacer toutes les occurrences d'une variable libre 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. Une approche possible est de faire l'opération en deux temps :

1. d'abord construire un terme e'', 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' dans e''.

Les variables libres de $e[e'/x]$ sont alors les variables libres de e et les variables libres de e', en excluant x, qui disparait dans le procédé.

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.

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 $\rightsquigarrow$ de la façon suivante :

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

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 enchaînements 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 :

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

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.

Nous n'avons pas le temps de le décrire ici, mais le lambda-calcul contient assez de primitives pour représenter les constructions habituelles de la programmation : en effet, il est possible d'adopter des conventions pour représenter les nombres entiers, les listes, les valeurs booléennes, et de fournir des fonctions qui permettent de tester des valeurs booléennes, de parcourir des listes, de faire de l'arithmétique sur les nombres entiers, et ainsi de suite. Ceci montre que le λ-calcul est un langage de programmation complet.

2 Le λ-calcul simplement typé

Comme langage de programmation, le λ-calcul pur a plusieurs défauts. Le premier défaut est que l'on peut trop facilement faire des erreurs de programmation. Le deuxième défaut est que l'on utilise pas intelligemment les structures de données fournies sur les ordinateurs, comme les nombres.

Nous allons maintenant décrire comment le λ-calcul a été étendu pour ajouter une notion de typage aux fonctions. Cette notion de typage permet d'exprimer quelles sont les données attendues par chaque fonction et quelles sont les données retournées par les calculs. Ceci permet de réduire les erreurs que l'on peut faire à la programmation et d'inclure des fonctions qui peuvent prendre en compte les types natifs de l'ordinateur. Nous verrons également que le λ-calcul typé permet de garantir que les calculs terminent toujours.

2.1 Un langage de types

Nous considérons que nous disposons d'un ensemble P de types primitifs. Par exemple, P pourra contenir les types `int`, `bool`, `float`. Nous considérons le langage de types T défini de la façon suivante :

- tout type primitif de P est un type de T,
- si t_1 et t_2 sont deux types alors $t_1 * t_2$ est un type,
- si t_1 et t_2 sont deux types alors $t_1 \rightarrow t_2$ est un type.

Les types de la forme $t_1 * t_2$ seront utilisés pour décrire le type des couples, nous utiliserons la notation $\langle e_1, e_2 \rangle$ pour décrire la construction qui permet d'obtenir un couple. Les types de la forme $t_1 \rightarrow t_2$ seront utilisés pour décrire le type des fonctions prenant en argument des valeurs de type t_1 et retournant des valeurs de type t_2.

Par exemple, le type $\texttt{int} * \texttt{int}$ représente le type des couples de valeurs entières, tandis que le type $\texttt{int} \rightarrow \texttt{int}$ représente le type des fonctions qui prennent un entier en argument et retournent un entier. Une fonction à deux argument pourra être décrite alternativement comme une fonction recevant un argument de type "couple d'entier" et retournant un entier ou comme une fonction recevant un entier et retournant une nouvelle fonction de type entier vers entier. La fonction $\texttt{curry}_{int}$ permet de passer d'une forme à l'autre. Elle a le type suivant :

$$((\texttt{int} * \texttt{int}) \rightarrow \texttt{int}) \rightarrow (\texttt{int} \rightarrow (\texttt{int} \rightarrow \texttt{int}))$$

En programmation fonctionnelle, il est fréquent de définir des fonctions à plusieurs arguments sans utiliser de couples. On obtient alors des fonctions dont le type est constitué d'une longue séquence de flèches. L'usage est d'enlever les parenthèses autour de ces flèches lorsque ces parenthèses sont à droite, ainsi le type de $\texttt{curry}_{int}$ s'écrit mieux de la façon suivante :

$$((\texttt{int} * \texttt{int}) \rightarrow \texttt{int}) \rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$$

2.2 Annotation des λ-termes avec des types

Nous allons maintenant nous intéresser à une variante du λ-calcul ou chaque fonction définie par abstraction contient une information sur le type attendu. Les expression du λ calcul auront donc la forme suivante :

$$e ::= x | \langle e_1, e_2 \rangle | \lambda x : t.\ e | e_1\ e_2 | \mathit{fst} | \mathit{snd}$$

Nous allons chercher à déterminer le type des expressions de ce langage, mais pour le faire nous serons obligés de disposer d'informations sur le type des variables libres dans ces expressions. Pour parler de ces informations nous utiliserons un contexte, constitué d'une séquence de couples composés de variables et de types. Ces contextes seront notés avec la variable Γ, le contexte vide sera noté $\emptyset$ et le contexte auquel on ajoute le couple associant la variable x avec le type t sera noté $\Gamma, x : t$.

L'usage est de décrire les règles de typage en utilisant le style de règle d'inférence emprunté à la logique. Les règles 1 à 7 se retrouvent sous la forme suivante :

$$\frac{}{\Gamma, x : t \vdash x : t}\ (1) \qquad \frac{\Gamma \vdash x : t \qquad x \neq y}{\Gamma, y : t \vdash x : t}\ (2)$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \langle e_1, e_2 \rangle : t_1 * t_2}\ (3)$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.\ e : t \to t'}\ (4)$$

$$\frac{\Gamma \vdash e_1 : t \to t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'}\ (5)$$

$$\frac{}{\Gamma \vdash \mathit{fst} : t_1 * t_2 \to t_1}\ (6) \qquad \frac{}{\Gamma \vdash \mathit{snd} : t_1 * t_2 \to t_2}\ (7)$$

Lorsque l'on représente les règles de typage de cette manière, il est possible de décrire également le procédé complet de typage d'une expression par une figure, que l'on appelle un arbre de dérivation.

Par exemple le typage de $\lambda f : \texttt{int} * \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle$ peut être représenté par la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \cdots, x : \texttt{int} \vdash x : \texttt{int}}\ (1)}{\Gamma, \cdots \vdash x : \texttt{int}}\ (2) \quad \dfrac{}{\Gamma, \cdots, y : \texttt{int} \vdash y : \texttt{int}}\ (1)}{\Gamma, \cdots \vdash \langle x, y\rangle : \texttt{int} * \texttt{int}}\ (3)}{\Gamma, \cdots \vdash f : \texttt{int} * \texttt{int} \to \texttt{int}}\ (2)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int} \vdash f\langle x, y\rangle}\ (5)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int} \vdash \lambda y : \texttt{int}.\ f\langle x, y\rangle : \texttt{int} \to \texttt{int}}\ (4)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int} \vdash \lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle : \texttt{int} \to \texttt{int} \to \texttt{int}}\ (4)}{\begin{array}{l}\Gamma \quad \vdash \lambda f : \texttt{int} * \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle \\ \quad : (\texttt{int} * \texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}\end{array}}\ (4)$$

2.3 Utilisation logique

Les expressions du langage des types peuvent être lues comme si elles étaient des formules logiques : les types primitifs de P sont des variables propositionnelles, on lit le type flêche comme une implication et le type produit cartésien comme une conjonction. Par exemple, la formule *si A et B est vrai alors B et A est aussi vrai* sera représenté par le type suivant :

$$A * B \to B * A.$$

Cette façon de lire les types comme des formules logiques est justifiée par une remarque supplémentaire : s'il existe une expression ayant le type t dans le contexte vide, alors cette formule logique est une tautologie. Lorsque cette propriété est satisfaite, on dit que le type t est habité, un habitant du type est aussi appelé une preuve de la formule logique. Par ailleurs, lorsque l'on efface

les termes du λ-calcul qui apparaissent dans une dérivation de typage et que l'on considère les contextes simplement comme des listes de formules prouvées, on retrouve simplement une preuve du calcul des séquents.

Par exemple, le type $A * B \to B * A$ est habité par le terme suivant :

$$\lambda x : A * B.\langle snd\ x, fst\ x\rangle.$$

C'est cette utilisation des types comme des formules logiques qui est à la base des systèmes de preuves en théorie des types comme Coq. La correspondance entre les types et les formules logiques d'une part et les programmes fonctionnels et les preuves d'autre part est souvent appelée la *correspondance de Curry-Howard*.

Tous les types habités sont des formules logiques tautologiques, mais toutes les tautologies ne sont pas des types habités. On sait par exemple que la formule suivante est vérifiable à l'aide d'un tableau de vérité (cette formule est appelée la formule de Peirce).

$$((A \to B) \to A) \to A.$$

La différence entre les formules prouvables à l'aide du λ-calcul typé et les formules prouvables par des tables de vérité fait l'objet de débats depuis le début du vingtième siècle. Un école de pensée estime que l'on devrait ne considérer comme vraies que les formules prouvable par des termes typés : on parle alors de logique *intuitionniste*. Par opposition, on appelle logique *classique* l'approche qui accepte les tables de vérités comme moyen de preuve. La différence fondamentale est qu'en logique classique, le tiers-exclus est admis. Ce tiers-exclus exprime simplement que *toute formule est soit vraie soit fausse*.

Exercices

3. *Construisez une preuve de* $(A \to B \to C) \to (B \to A \to C)$,
4. *Construisez une preuve de* $(A \to B \to C) \to A * B \to C$,
5. *Construisez une preuve de* $((((A \to B) \to A) \to A) \to B) \to B$, *notez que cette formule est proche de la loi de Peirce.*

2.4 Réduction typée

Pour disposer d'une réduction dans le λ-calcul typé, nous allons nous contenter de considérer que les termes du λ-calcul typé se réduisent comme les termes du λ-calcul non typé, par β-réduction. Une propriété notable de la réduction est un théorème de stabilité : la réduction d'un terme se fait à type constant, ce qui peut se démontrer par récurrence sur la taille des expressions considérées.

Nous n'allons pas détailler cette preuve, mais seulement en vérifier deux lemmes.

Premièrement, si e_1 a le type $t \to t'$ et se réduit en e_1' de même type, alors l'expression $e_1\ e_2$ est bien typée si et seulement si $e_1'\ e_2$ est bien typée.

Deuxièmement, si e_1 a le type t' dans le contexte $\Gamma, x : t$ et si e_2 a le type t dans le contexte Γ, alors l'expression $(\lambda x : t.\ e_1)e_2$ est bien typée et a le type t'

dans le contexte Γ. Il reste à vérifier que $e_1[e_2/x]$ est bien typé dans le contexte Γ, ce qui est naturel pour les deux raisons suivantes :

1. Dans $e_1[e_2/x]$ la variable x n'apparait plus, donc la déclaration $x : t$ n'est plus nécessaire dans le contexte de typage,
2. on remplace la variable x de type t par une expression e_2 de type t.

2.5 Terminaison des réductions

Si on observe la règle 5 qui régit le typage de l'application d'une fonction on s'aperçoit que le type de la fonction est un terme strictement plus grand que le type de l'argument. Pour cette raison, il n'est pas possible de typer une expression de la forme $x\ x$. L'expression $\Delta \equiv \lambda x.\ x\ x$ n'est pas typable et donc pas l'expression $\Omega \equiv \Delta\Delta$, ni l'expression $Y = (\lambda zx.\ x(z\ z\ x))\lambda zx.\ x(z\ z\ x)$. Les expressions que nous avons vues dans le cours sur le λ-calcul pur qui présentent des dérivations infinies ne sont pas typables. En fait ceci se généralise en un théorème très important : *toutes les expressions typables du λ-calcul simplement typé sont fortement normalisantes.* Dit autrement, les exécutions de programmes écrits en λ-calcul simplement typé terminent. Une démonstration de ce théorème peut être trouvée dans [7], on trouve également une généralisation de ce résultat dans [10].

Le fait d'avoir éliminé les causes de non-terminaison peut sembler un grand progrès, mais en chemin on perd la possibilité de définir des fonctions récursives. Une première solution est de ré-introduire la récursion tout en conservant le typage en ajoutant une constante Y au langage avec une règle de réduction :

$$(Y\ f) \rightsquigarrow f(Y\ f)$$

Pour conserver la stabilité du typage, Y doit être une fonction de type $\theta \to \psi$. D'après le membre gauche, l'argument doit être une fonction donc $\theta = \sigma \to \sigma'$. Il faut également assurer que le type de $Y\ f$ coincide avec le type de $f(Y\ f)$ donc $\sigma' = \psi$. Enfin, pour que $f(Y\ f)$ soit bien typé il faut en outre que le type de $(Y\ f)$ coïncide avec le type en entrée de f, soit $\sigma = \psi$. Tout réuni, nous trouvons que Y doit avoir le type $(t \to t) \to t$ pour tout type t.

Par exemple, nous pouvons travailler dans le contexte suivant :

Γ, *plus*, *sub*, *mult* : $\texttt{int} \to \texttt{int} \to \texttt{int}$, *le* : $\texttt{int} \to \texttt{int} \to \texttt{bool}$,
if : $\texttt{bool} \to \texttt{int} \to \texttt{int} \to \texttt{int}$

et définir la fonction factorielle, de façon bien typée, en utilisant la même approche que pour le λ-calcul pur. On construit d'abord une fonctionnelle $factF$ de type

$$factF : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$$

et définie de la manière suivante :

$$factF \equiv \lambda f : \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\ \texttt{if}(\texttt{le}\ x\ 0)\ 1\ (\texttt{mult}\ x\ (f\ (\texttt{sub}\ x\ 1)))$$

La fonction factorielle est alors le terme $fact \equiv Y\,factF$.

L'opérateur Y apporte donc le retour de la récursion générale et de la possibilité pour les programmes de « boucler ». Cette approche avec opérateur de point fixe est celle fournie dans la majeure partie des langages fonctionnels typés, dont `ML`, `OCaml` et `Haskell` sont les exemples les plus connus.

2.6 Types récursifs et récursion structurelle

Une autre approche permet d'introduire de la récursion sans perdre la propriété que les calculs terminent toujours. Il s'agit d'introduire des structures de données représentant des arbres et de n'autoriser que les fonctions récursives qui calculent sur ces arbres en n'autorisant les appels récursifs que sur les sous-termes directs de leur argument initial. On introduit donc simultanément un type de données récursif et une fonction qui décrit la récursion sur ce type. La contrainte que les appels récursifs ne sont autorisés que pour les sous-arbres peut être exprimée à l'aide du typage.

Commençons par un exemple. Le type des nombres naturels peut être décrit comme une structure de donnée nommée `nat` avec trois constantes `O` :`nat` et `S` :`nat`→`nat` et une constante `rec_nat` qui peut recevoir le type suivant, pour tout type t :

$$\texttt{rec_nat} : t \to (\texttt{nat} \to t \to t) \to \texttt{nat} \to t.$$

Avec ces constantes, on ajoute au λ-calcul typé les règles de réduction suivantes :

$$\texttt{rec_nat}\ v\ f\ \texttt{O} \rightsquigarrow v \qquad \texttt{rec_nat}\ v\ f\ (\texttt{S}\ x) \rightsquigarrow f\ x\ (\texttt{rec_nat}\ v\ f\ x).$$

Chaque nombre n est représenté par le terme `S(S(S...O))` où `S` est répété n fois. La constante `rec_nat` permet de construire les fonctions usuelles de l'arithmétique, tout en conservant la propriété que les calculs terminent toujours. En effet, l'expression `rec_nat` $v\ f\ x$ qui apparait dans la deuxième règle de réduction correspond au seul appel récursif autorisé, qui doit nécessairement avoir lieu sur le sous-terme immédiat de l'argument initial `S` x. Ainsi, lorsqu'une fonction définie à l'aide `rec_nat` calcul sur un nombre n, elle ne peut se rappeler récursivement que sur $n-1$, puis sur $n-2$ et ceci s'arrête nécessairement lorsque l'on arrive à 0. De plus, on a la garantie que toute valeur en forme normale et bien typée de type `nat` dans le contexte vide est uniquement composée d'un nombre fini de `S` appliqués à `O`.

Par exemple, l'addition de deux nombres entiers est décrite par la fonction suivante :

$$\lambda x\ y.\texttt{rec_nat}\ y\ (\lambda p\ r.\texttt{S}\ \texttt{r})x.$$

En effet, si l'on ajoute 0 à y, le résultat est y et si l'on ajoute `S` p à y, le résultat doit être `S` $(p + y)$.

Les constantes `S` et `O` sont appelées les constructeurs du type `nat` et nous appellerons `rec_nat` le récurseur associé à ce type.

Il est possible de définir d'autres types supportant de la récursion. Il suffit à chaque fois de donner une collection de constructeurs, qui sont toujours des fonctions ou des constantes dont le type final est le type récursif que l'on veut

définir. Les arguments des constructeurs peuvent être dans le type que l'on est en train de définir (ce qui fait alors apparaitre de la récursion) ou dans un type déjà défini. Le récurseur associé à un type à n constructeurs est une fonction à $n+1$ argument. Les n premiers arguments correspondent à un traitement par cas sur le dernier argument. Le premier argument indique comment le calcul s'effectue si le dernier argument a été obtenu à l'aide du premier constructeur, le deuxième argument traite le cas du second constructeur, et ainsi de suite. Dans chaque cas, si le constructeur considéré a k arguments dont l arguments dans le type récursif lui-même, la fonction indiquant comment ce cas est traité est une fonction à $k+l$ arguments. Par exemple, le deuxième constructeur de `nat` a 1 argument dont 1 dans le type `nat` lui-même. Le type attendu pour le deuxième argument de `rec_nat` est donc une fonction à 2 arguments (1+1). Les arguments supplémentaires correspondent aux valeurs retournées par les appels récursifs de la fonction définie sur les sous-terme du constructeur qui sont dans le type lui-même.

Voici un autre exemple. On considère le type `bin` des arbres binaires avec deux constructeurs `leaf` et `node` de la façon suivante :

1. `leaf` : `bin`,
2. `node` : `nat` $\rightarrow$ `bin` $\rightarrow$ `bin` $\rightarrow$ `bin`.

Puisque ce type a deux constructeurs, le récurseur `rec_bin` prend trois arguments, le premier est une constante comme `leaf`, le second est une fonction à 5 arguments, parce que `node` a trois arguments dont 2 dans le type `bin`. Les deux arguments supplémentaires correspondent aux résultats d'appels récursifs sur les sous-termes de type `bin`. Le type de `rec_bin` est donné de la façon suivante :

$$\texttt{rec_bin} : t \rightarrow (\texttt{nat} \rightarrow \texttt{bin} \rightarrow t \rightarrow \texttt{bin} \rightarrow t \rightarrow t) \rightarrow \texttt{bin} \rightarrow t$$

et les règles de réduction sont les suivantes :

$$\texttt{rec_bin}\ v\ f\ \texttt{leaf} \rightarrow v$$

$$\texttt{rec_bin}\ v\ f\ (\texttt{node}\ n\ t_1\ t_2) \rightarrow f\ n\ t_1\ (\texttt{rec_bin}\ v\ f\ t_1)\ t_2\ (\texttt{rec_bin}\ v\ f\ t_2)$$

Intuitivement, on autorise seulement les appels récursifs sur les sous-arbres d'un arbre binaire. Les sous-arbres d'un arbre binaire sont nécessairement plus petit que lui, en n'autorisant ainsi que des appels récursifs qui décroissent dans la structure des arbres, on assure que les calculs récursifs terminent toujours. Cette approche de la récursion où l'on impose que les appels récursifs n'aient lieu que sur des sous-termes directs de l'argument initial est appelée récursion structurelle.

Par exemple on peut décrire la fonction qui additionne toutes les valeurs dans un arbre par la fonction suivante :

$$\texttt{rec_bin}\ 0\ (\lambda n : \texttt{int}, t_1 : \texttt{bin}, v_1 : \texttt{int}, t_2 : \texttt{bin}, v_2 : \texttt{int}.(\mathit{plus}\ n(\mathit{plus}\ v_1\ v_2)))$$

Il existe une correspondance assez naturelle entre les fonctions récursive que l'on peut définir avec le récurseur associé à un type récursif et les définitions de fonctions récursives définies en OCaml ou Haskell à l'aide du filtrage. Par exemple on peut définir le type `bin` en OCaml par la commande suivante :

```
type bin = Leaf | Node of nat*bin*bin
```

La fonction $g =$ `rec_bin` v f correspond à la définition OCaml suivante :

```
let rec g x = match x with
   Leaf -> v
 | Node n t1 t2 -> f n t1 (g t1) t2 (g t2)
```

3 Inférence de types

3.1 Abstraction non typée

La programmation fortement typée apporte une assurance supplémentaire, mais la nécessité de fournir le type de toutes les variables a rapidement été ressenti comme une contrainte trop lourde. Pour alléger cette contrainte on peut étendre le langage avec une λ-abstraction non typée $\lambda x.\ e$ et lui donner la règle de typage : *pour tous types* t *et* t'*, si* e *a le type* t' *dans le contexte* $\Gamma, x : t$ *alors* $\lambda x.\ e$ *a le type* $t \to t'$. En fait, nous généralisons ici aux abstractions un procédé que nous nous étions déjà autorisé pour les projecteurs des couples *fst* et *snd*, pour l'opérateur de point fixe Y et pour les récurseurs associés à des types récursifs.

Lorsque l'on considère une expression composée avec plusieurs abstractions non typées, la question que l'on se pose est de choisir un type pour chacune de ces abstractions typées de façon que l'expression entière soit bien typée. Une solution est de se reposer sur un algorithme d'unification, après avoir donné à chaque abstraction non typée un type variable. Le programme de vérification de type remplace la vérification d'égalité entre deux types par l'ajout d'une équation dans un jeu de contraintes à résoudre. Bien sûr on donne également un type à toutes les occurrences de constantes polymorphes apparaissant dans le terme, par exemple toute occurrence de *fst* reçoit le type $T_1 * T_2 \to T_1$, où T_1 et T_2 sont de nouvelles variables, on fait de même avec *snd* et Y, si cette constante fait partie du langage. Ensuite on effectue le typage comme d'habitude, sauf que l'on ajoute une contrainte dans une liste de contraintes pour chaque application, où il faut vérifier que l'argument de l'application a bien un type égal au type attendu par la fonction.

Par exemple observons le typage de l'expression $\lambda x.(plus\ x\ x)$, l'annotation avec des variables de types donne l'expression suivante, $\lambda x : T.\ (plus\ x\ x)$ et le typage avance de la manière suivante :

1. on démarre avec une liste de contraintes vides,
2. *plus* a le type $\texttt{int} \to \texttt{int} \to \texttt{int}$ dans le contexte $\emptyset, x : T$,
3. x a le type T dans le contexte $\emptyset, x : T$,
4. l'application *plus* x est bien typée dans le contexte $\Gamma, x : T$ si $\texttt{int} = T$ (on ajoute donc cette contrainte à la liste de contraintes), cette expression a le type $\texttt{int} \to \texttt{int}$,
5. l'application $plus\ x\ x \equiv (plus\ x)\ x$ est bien typée dans le contexte $\Gamma, x : T$ si $\texttt{int} = T$, cette expression a le type `int`

6. l'expression $\lambda x : T.\ (plus\ x\ x)$ est bien typée si les contraintes $\{\texttt{int} = T, \texttt{int} = T\}$ ont une solution σ et a alors le type $\sigma(T) \to \texttt{int}$.

La solution d'un système de contraintes, lorsqu'elle existe est une subtitution d'un type pour chacune des variables de types, ici la solution est la substitution qui remplace T par `int`. L'expression considérée a donc le type $\texttt{int} \to \texttt{int}$.

La résolution des contraintes se fait à l'aide d'un algorithme d'unification, déjà connu dans le domaine de la démonstration sur ordinateur et de Prolog.

3.2 Typage polymorphe

Lorsque l'on dispose d'une valeur de type t et d'une fonction de type $t \to t$, on peut vouloir appliquer cette fonction deux fois sur cette valeur, ce que l'on écrira de la façon suivante :

$$\lambda v\ f.f(f\ v).$$

Cette fonction travaille toujours de la même façon, quel que soit le type t considéré, qui ne joue au fond aucun rôle. On pourrait donc vouloir appliquer la même fonction en deux endroits différents, une fois pour le type `int` et une fois pour le type `bool`, par exemple, l'expression choisie aurait la forme suivante :

$\lambda c.(\lambda g.\lambda b : \texttt{bool}, f_1 : \texttt{bool} \to \texttt{bool}, n : \texttt{int}, f_2 : \texttt{int} \to \texttt{int}.c\ (g\ b\ f_1)\ (g\ n\ f_2))$
$\lambda v\ f.f(f\ v).$

Cette expression n'est bien typée, car l'une des utilisation de g impose que le type de v soit `bool` tandis que l'autre utilisation impose que le type de v soit `int`. Ici, il semble que le typage impose une duplication de code.

La solution de ce problème est de généraliser à du code écrit par l'utilisateur la solution fournie pour *fst*, *snd* ou Y. Plutôt que d'exprimer que la fonction $\lambda v\ f.\ f(f\ v)$ a le type $T \to (T \to T) \to T$ pour un T donné, nous voulons exprimer que cette fonction a le type $t \to (t \to t) \to t$ pour tout type t. Ainsi, certaines fonctions peuvent avoir un type quantifié universellement, on parle alors de type *polymorphe*.

Nous allons maintenant ajouter une nouvelle construction dans notre langage, qui aura la syntaxe suivante :

$$\mathsf{let}\ x = e\ \mathsf{in}\ e'$$

Opérationnellement, le sens à donner à une telle expression est celui d'un redex, c'est à dire que le comportement à l'exécution des deux expressions suivantes devrait être similaire :

$$\mathsf{let}\ x = e\ \mathsf{in}\ e' \sim (\lambda x.\ e')e$$

Pour le typage néanmoins, le typage polymorphe s'exprime par la règle suivante :

$$\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash e'[e/x] : t}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : t}$$

Donc pour typer une expression $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ dans un contexte Γ il faut d'abord vérifier si l'expression e est bien typée dans ce contexte et ensuite vérifier

si chacune des utilisations de e est bien typée, indépendamment des autres utilisations. Cette règle de typage effectue donc la duplication de code avant de faire la vérification de typage, ce qui permet d'éviter aux programmeurs d'effectuer cette duplication eux-mêmes.

Cet algorithme de typage n'est pas très efficace, car il fait appel à une substitution qui peut provoquer une croissance excessive de la taille du terme à typer. On trouvera dans [6] un algorithme plus efficace, que G. Dowek décrit dans son cours[2].

4 Types dépendants

La théorie des types ne devient vraiment intéressante qu'avec les types dépendants, qui autorisent à considérer des familles de types indexées par des données. Avec cette extension du λ-calcul typé, on pourra exprimer que certaines fonctions ne peuvent être appliquée que sur des arguments qui vérifient certaines propriétés. Par exemple, une fonction de recherche dans une liste ne peut être appliquée qu'à une liste et un entier plus petit que la longueur de cette liste.

4.1 Extension syntaxique

La première étape est de permettre d'indexer les types par des valeurs. Il s'agit de définir des fonctions qui prennent des valeurs en argument et retournent des types : leur type d'arrivée est donc un type de types ; on appelle aussi cela une *sorte*. Ici, je supposerai qu'il existe une sorte `Type`. Nous décrirons une famille de types indexée par un type A sous la forme d'une fonction de type $A \to$ `Type`.

Lorsque $f : A \to$ `Type` est une famille de types indexée par des éléments de A, il est intéressant de considérer des fonctions qui prennent en entrée un élément x de A et produisent comme résultat un élément du type $f\ x$ indexé par x. La notation basée sur les flêches est inadaptée pour ce besoin : il faut donner un nom à l'argument de la fonction lorsque l'on décrit le type de cette fonction, pour pouvoir utiliser ce nom lorsque l'on décrit le type du résultat. Les auteurs ont dû introduire une nouvelle notation. La plus populaire utilise une notion de produit indexé : $\Pi x : A.\ f\ x$.

Cette notation a une explication intuitive assez simple : Le produit cartésien $A_1 \times A_2$ de deux types, contient des couples qui allient des valeurs de A_1 et A_2, donc des valeurs dans des types différents. Mais un couple peut aussi être compris comme une fonction sur $\{1, 2\}$: lorsqu'on lui donne en argument 1, on obtient la première composante, de type A_1, lorsqu'on lui donne en argument 2, on obtient la seconde composante, de type A_2. La notation de produit indexé généralise naturellement cette interprétation des produits cartésiens : une fonction de type $\Pi x : A.\ f\ x$ permet d'obtenir des valeurs de type $f\ x$ pour tous

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

les indices possibles dans A, de la même manière qu'un produit cartésien indexé par A.

4.2 Extension du typage

En présence de types dépendants, les règles de typage pour les λ abstractions et les applications doivent être modifiées pour traduire le fait qu'une fonction peut retourner une valeur dont le type dépend de l'argument. Les nouvelles règles prennent la forme suivante :

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.e : \Pi x : t.\ t'}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : t.t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'[e_2/x]}$$

En fait, la notation $A \to B$ peut encore être utilisée lorsque le type n'est pas réellement dépendant.

Avec l'aide de ces règles de typage, on comprend que si B et C sont des familles de types indexées par A, et $f : \Pi x : A.\ P\ x$ et $g : \Pi x :\ P\ x \to Q\ x$ sont des fonctions, alors l'expression suivante est bien typée et a le type annoncé :

$$\lambda x : A.g\ x\ (f\ x) : \Pi x :\ Q\ x.$$

4.3 Interprétation logique

Pour utiliser les types dépendants pour faire de la logique, il est pratique de se donner une deuxième sorte, que l'on utilisera explicitement pour écrire des formules logiques. Nous appellerons cette sorte `Prop`. Nous avons vu dans la première section que les variables de types correspondaient à des variables propositionnelles. Lorsque les propositions sont indexées par les éléments d'un type `A`, elle deviennent des prédicats sur ce type. Considérons par exemple le prédicat $P : A \to$ `Prop` : pour deux valeurs différentes x et y de type A, on dispose de deux types $P\ x$ et $P\ y$ différents, dont l'un peut être habité et l'autre pas, ce qui signifie que l'une des formules est prouvable et l'autre pas.

Pour interpréter la nouvelle construction de produit dépendant, il faut se rappeler que toutes les fonctions que nous considérons terminent. On a alors la propriété suivante : si f a le type $\Pi x : A.\ P\ x$, alors pour tout élément x de A, cette fonction va fournir un élément de $P\ x$. En d'autres termes, $P\ x$ sera toujours habité, toujours prouvable. On peut donc lire le produit dépendant comme une quantification universelle. Dans la suite, j'utiliserai très souvent la notation $\forall x : A,\ P\ x$ lorsque P a le type $A \to$ `Prop`.

Les expressions dont le type est une une formule logique sont des théorèmes. Le calcul typé permet de composer ces expressions, pour obtenir des preuves de nouveaux théorèmes.

Par exemple, supposons que l'on dispose d'un prédicat `even`[3] et de deux constantes `even0` et `even2` avec les types suivants :

$$\begin{array}{lcl} \texttt{even0} & : & \texttt{even O} \\ \texttt{even2} & : & \forall x : \texttt{nat}.\texttt{even}\ x \rightarrow \texttt{even}(\texttt{S}\ (\texttt{S}\ x)) \end{array}$$

Le terme `even2 (S (S O))` (`even2 O even0`) est une preuve de la proposition `even (S (S (S (S O))))`. En d'autres termes, c'est une preuve que 4 est pair.

4.4 Types dépendants et types récursifs

Les types récursifs peuvent aussi être définis de telle manière que l'on obtienne en fait des familles de types. Par exemple, les listes de données peuvent être paramétrées par le type des éléments : on obtient alors une famille de type représentée par une fonction `list : Type` $\rightarrow$ `Type`. Pour construire un élément de l'un des types de cette famille, on peut choisir de construire une liste vide ou d'ajouter un élément à une liste existante. Même pour une liste vide, il faut choisir le type des éléments pour savoir dans quel type parmi les différents types indexés se trouvera le résultat, donc le constructeur de liste vide n'est pas une constante, mais une fonction qui prend un type A en argument et retourne un élément du type des listes indexées par A :

$$\texttt{nil} : \forall A : \texttt{Type}.\texttt{list}\ A$$

Pour ajouter un élément dans une liste existante, il faut fixer le type A puis prendre un élément de A et une liste déjà existante d'éléments de A. Le constructeur a donc la forme suivante :

$$\texttt{cons} : \forall A : \texttt{Type}.A \rightarrow \texttt{list}A \rightarrow \texttt{list}A.$$

Si l'on veut définir une fonction récursive avec un type dépendant Πx : `nat`.$A\ n$ à l'aide de `rec_nat` il faut que la valeur prévue pour `O` soit de type A `O` et que la fonction prévue pour `S` x sache utiliser une valeur la valeur x et le résultat de l'appel récursif sur x, qui est de type $A\ x$ pour construire une valeur de type A (`S` x). Ceci s'exprime par le type suivant : dans le type

$$\texttt{rec_nat} : \forall A : \texttt{nat} \rightarrow \texttt{Type}.A\ \texttt{O} \rightarrow (\Pi n : \texttt{nat}.A\ n \rightarrow A\ (\texttt{S}\ n)) \rightarrow \Pi n : nat.A\ n$$

Lorsque l'on lit le type de `rec_nat` comme une formule logique, on s'aperçoit que ce type est une formule logique très connue. C'est le principe de récurrence que l'on peut utiliser pour démontrer une formule sur les entiers : pour tout prédicat P, si P est satisfait en 0 et si pour tout n $P\ n$ implique $P\ (n+1)$, alors P est satisfait pour tous les entiers :

$$\forall P : nat \rightarrow \texttt{Type}.P\ \texttt{0} \rightarrow (\forall x : \texttt{nat}.P\ n \rightarrow P\ (\texttt{S}\ n)) \rightarrow \forall x : \texttt{nat}.P\ n$$

[3] en anglais, `even` veut dire pair.

Il y a encore bien d'autres interactions entre types dépendants et types récursifs. En particulier, les types dépendants permettent d'étendre la définition de fonctions récursives en maintenant la propriété de terminaison mais en relâchant la contrainte que les appels récursifs ne peuvent avoir lieu que sur des sous-termes directs. Nous n'aurons pas le temps de tout explorer ici. Une bonne façon de les explorer et d'utiliser un système de démonstration basé sur la théorie des types, comme le système Coq [5], qui utilise une théorie des types particulière : le calcul des constructions inductives.

5 Pour en savoir plus

Ces notes ne donnent qu'un aperçu grossier de ce que l'on peut faire avec la théorie des types. Il existe plusieurs ouvrages qui présentent une étude plus en profondeur. Le lambda-calcul pur est étudié de façon intensive dans [1] qui est resté la référence majeure sur ce sujet. Une étude historique des systèmes de types pour le λ-calcul et leur utilisation en logique est fournie dans [9].

Plusieurs langages de programmation sont dérivés du λ-calcul typé avec quelques extensions [3, 15, 14].

De nombreux systèmes de preuve utilisent à des degrés divers le λ-calcul typé comme ossature pour le langage des formules logiques ou des preuves [8, 13, 4, 11] L'un des systèmes basés sur la théorie des types les plus avancés est le système Coq pour lequel nous avons rédigé un ouvrage [2].

Références

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

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art :the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[3] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly and associates, 2000.

[4] Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[5] Coq development team. *The Coq Proof Assistant Reference Manual, version 8.0*, 2004. `http://coq.inria.fr`.

[6] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *ninth ACM symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[7] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*. Cambridge University Press, 1989.

[8] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[9] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory From its Origins Until Today*. Kluwer Academic Publishers, 2004.

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

[11] Lawrence C. Paulson. *Logic and computation, Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[12] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[13] Lawrence C. Paulson and Tobias Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[14] Simon Thompson. *Haskell, the craft of functional programming*. Addison-Wesley, 1996.

[15] Pierre Weis and Xavier Leroy. *Le Langage Caml*. Dunod, 1999.