Lambda Calcul

Gabriel Scherer

Résumé

Dans cet article, nous présenterons le λ-calcul non typé, quelques méthodes d'encodage des données dans ce langage, et quelques résultats de calculabilité. Nous décrirons ensuite le λ-calcul simplement typé (et d'autres résultats de calculabilité), puis le λ-calcul typé de second ordre ou System F, pour conclure sur un théorème de complétude des codages de structures de données.

On pourrait décrire le lambda-calcul (ou λ-calcul) comme "le premier et le plus simple des langages de programmation fonctionnels". C'est un formalisme de description de fonctions et de calculs, développé au départ par Alonzo Church dans les années 30.

Comme les machines de Turing, il a été conçu dans le but de donner un sens précis aux termes "algorithme" et "fonction calculable" ; la notion de calculabilité qui en découle est équivalente à celle de Turing, mais l'approche est très différente. Le λ-calcul est depuis devenu un champ de recherche actif, au delà des considérations de calculabilité : il est utilisé en théorie des types, sémantique des langages de programmation, etc.

1 Lambda-calcul non typé

Un terme du λ-calcul est décrit inductivement par la grammaire suivante, où v représente un symbole de variable quelconque

$$E ::= v \mid E\ E \mid \lambda v.\, E$$

Les termes de la forme EE sont des *applications*, et ceux de la forme $\lambda v.\, E$ des *abstractions*. Une variable x est *liée* quand elle est placée sous le champ d'une abstraction selon x (dans la partie E d'un terme $\lambda x.\, E$) ; ses autres occurrences sont dites *libres*, et un λ-terme sans variable libre est *clos*.

Intuitivement, on peut penser que $\lambda v.\, E$ désigne une fonction qui à v associe E, et que MN est l'application de l'argument N à la fonction M. L'application des fonctions est décrite par une opération de *substitution*.

Note syntaxique La représentation textuelle des λ-termes imposant des parenthèses pour lever les ambiguïtés, quelques conventions sont utiles pour soulager l'écriture.

L'application a priorité sur l'abstraction : $\lambda a.\, bc = \lambda a.\, (bc)$. Nous utiliserons de plus la syntaxe d'application des fonctions curryfiées (comme dans les langages OCaml/Haskell/etc.) : $abc = a\ b\ c = (a\ b)\ c$. Pour éviter l'alignement des

rangées de λ, on pourra placer plusieurs paramètres après un lambda, au lieu de plusieurs lambdas à la suite : $\lambda abc.\, E = \lambda a.\, \lambda b.\, \lambda c.\, E$, etc.

1.1 Curryfication

L'abstraction en λ-calcul se fait selon une seule variable. Il n'y a pas à proprement parler de "fonctions à plusieurs variables". En mathématiques, la situation est identique, et on utilise à la place d'une fonction selon les variables a et b une fonction sur le couple (a, b) (on a alors un seul paramètre, le couple). Les couples ne sont pas accessible dans le λ-calcul de base, donc la solution choisie est différente : on représente une fonction à deux arguments comme une fonction selon le premier argument, qui renvoie une fonction selon le deuxième argument, qui renvoie le résultat final. On appelle cette transformation la *curryfication* (en mathématiques, pour A, B et C des ensembles, elle correspond à l'isomorphisme entre $(A \times B) \to C$ et $A \to (B \to C)$).

Si on ajoutait à notre λ-calcul une primitive d'addition $+$ (on verra plus tard comment décrire l'addition en λ-calcul pur), on pourrait écrire la fonction qui prend deux variables et les additionne

$$\text{somme} := \lambda a.\, \lambda b.\, a + b$$

C'est là que la simplification syntaxique des λ successifs prend tout son sens : on peut écrire $\lambda ab.\, a + b$, ce qui représente le *même* λ-terme, mais suggère plutôt une fonction à deux arguments. Par exemple, si l'on veut prendre deux paramètres, puis les appliquer à une fonction quelconque dans l'ordre inverse, on peut écrire cette opération $\lambda x.\, \lambda y.\, \lambda f.\, fyx$, ou $\lambda xyf.\, fyx$, ou encore $\lambda xy.\, \lambda f.\, fyx$, la troisième version ayant ma préférence personnelle.

1.2 Substitution, β–réduction

La substitution à la variable v du λ-terme t dans le λ-terme E, notée $E[t/v]$ (penser "E avec t pour v"), correspond à un remplacement des occurrences de v par t dans E.

Il faut cependant prendre garde au phénomène de *capture* qui rend la définition formelle de la substitution un peu plus délicate : on veut que seules les variables libres de E soient remplacées, et que des variables libres de t ne deviennent pas liées dans E. Par exemple, si on remplace naïvement x par y dans $\lambda y.\, x$ (intuitivement, la fonction qui à y associe x), on obtient$\lambda y.\, y$, qui est la fonction identité : l'occurrence libre de y est devenue liée (on dit qu'elle a été capturée), ce qui change la signification du λ-terme.

On définit la substitution par induction :

- $v[t/v] = t$
- $x[t/v] = x$ où x est une variable différente de v
- $(A\ B)[t/v] = A[t/v]\ B[t/v]$
- $(\lambda v.\, E)[t/v] = \lambda v.\, E$
- $(\lambda x.\, E)[t/v] = \lambda y.\, E'[t/v]$ quand x est différente de v, où y et E' sont définies ainsi : si x n'est pas libre dans t (pas de risque de capture), alors $y = x$ et $E' = E$, sinon on choisit une variable z qui n'est pas libre dans t, on pose $y = z$ et $E' = E[z/x]$.

On peut alors définir la relation β correspondant à une application de fonction (β–réduction) :

$$(\lambda v.\, E)t \quad \xrightarrow{\beta} \quad E[t/v]$$

La β–réduction est définie au renommage des variables capturantes près (plus précisément on peut définir une relation d'équivalence entre λ-termes, l'α-équivalence, qui indique que deux termes sont égaux à renommages bénins près).

1.3 Terminaison, forme normale

On peut β–réduire les λ-termes qui ne sont pas directement l'application d'une abstraction et d'un terme, mais qui contiennent des sous-termes réductibles. Par exemple on écrira, même si la réduction se fait à l'intérieur d'une abstraction

$$\lambda x.\, (\lambda y.\, yy)x \quad \xrightarrow{\beta} \quad \lambda x.\, xx$$

Un λ-terme peut contenir plusieurs sous-termes β–réductibles. On peut alors réduire n'importe lequel. Quand un λ-terme ne contient aucun sous-terme réductible, on dit qu'il est *sous forme normale*. Intuitivement, cela correspond à un λ-terme évalué "jusqu'au bout".

La question de savoir quels sous-termes réduire en premier n'est pas forcément évidente : quand on veut réduire "mécaniquement" les sous-termes au lieu de le faire à la main (par exemple si l'on code un interprète ou compilateur pour un λ-calcul ou langage dérivé), il faut choisir une procédure décidant du sous-terme à réduire le premier. On appelle ces procédures des *stratégies* de réduction.

Le choix de la stratégie de réduction peut être très important dans un langage de programmation, mais elle ne nous préoccupera pas ici. On dira qu'un terme est *normalisable* s'il existe [1] une suite de réductions qui permet d'en obtenir une forme normale (un théorème de Church-Rosser indique que la forme normale, si elle existe, est unique : deux stratégies différentes qui terminent donneront le même résultat). On verra par la suite qu'il existe des termes non normalisables.

2 Codage de données simples

Tel quel, le λ-calcul a l'air d'être un langage théorique d'étude des applications de fonctions. Il semble difficile d'y programmer (ou plutôt d'y décrire des programmes) concrètement, du fait du manque d'opérations primitives. Il n'y a même pas de symboles de constantes !

Quand on étudie un langage particulier, il arrive d'étendre le λ-calcul de base avec de nouvelles constructions, ou des opérations et des constantes supplémentaires. Cependant, le langage donné suffit en fait à construire la plupart des objets "classiques" des langages de programmation : il s'agit de représenter des concepts (ici, des structures de données : booléens, entiers naturels, etc.) sous forme de λ-termes. On parle de *codage* ; les codages présentés portent le nom général de *church encoding*.

1. cela revient à utiliser une stratégie de réduction non déterministe

2.1 Booléens

Pour commencer à représenter des programmes en lambda-calcul, on aimerait disposer d'une instruction conditionnelle. On voudrait encoder une structure de la forme (IF p THEN a ELSE b), qui vaut a si le booléen p est vérifié, et b sinon. Il s'agit de choisir une représentation des booléens en λ-calcul qui rende cette forme naturelle (et qui existe !).

L'idée la plus naturelle est d'utiliser une application de fonctions. On voudrait pouvoir écrire ce test tout simplement $p\ a\ b$, ce qui revient à représenter les deux valeurs booléennes, "vrai" et "faux", par des fonctions à deux arguments.

$$\begin{aligned} \text{vrai} &:= \lambda ab.\, a \\ \text{faux} &:= \lambda ab.\, b \end{aligned}$$

Cet encodage permet effectivement de représenter les booléens. On peut alors écrire avec les fonctions booléennes usuelles. La forme (si $p\ a\ b$), équivalente à (IF p THEN a ELSE b), est par construction particulièrement simple.

$$\text{si} := \lambda pab.\, p\ a\ b$$

On peut aussi représenter la disjonction, la conjonction et la négation.

$$\begin{aligned} \text{et} &:= \lambda p.\, \lambda q.\, p\ q\ \text{faux} \\ \text{ou} &:= \lambda p.\, \lambda q.\, p\ \text{vrai}\ q \\ \text{non} &:= \lambda p.\, \lambda a.\, \lambda b.\, p\ b\ a \end{aligned}$$

Il faut noter que cet encodage n'est pas unique : on pourrait par exemple inverser les définitions de vrai et faux, et on pourrait toujours définir ces opérations (il suffirait d'inverser et et ou, ainsi que si et non).

2.2 Entiers

On peut aussi représenter les entiers naturels. L'idée sous-jacente est de représenter l'entier n par l'opération d'itération n fois d'une fonction :
$f \mapsto f^n(x)$

$$\begin{aligned} 0 &:= \lambda fx.\, x \\ 1 &:= \lambda fx.\, fx \\ 2 &:= \lambda fx.\, f(fx) \end{aligned}$$

Plus généralement, on peut définir une fonction succ qui prend en paramètre le code d'un entier, et renvoie le code de son successeur.

$$\text{succ} := \lambda n.\, \lambda fx.\, f(nfx)$$

Enfin, on peut coder un prédicat iszero qui renvoie vrai si l'entier fourni est nul, et faux sinon. Il suffit d'itérer n fois la fonction qui renvoie toujours faux, en lui donnant vrai comme valeur initiale : si $n > 0$, la fonction sera appliquée et renverra faux, sinon f^n est l'identité et renvoie vrai.

$$\text{iszero} := \lambda n.\, n(\lambda x.\, \text{faux})\ \text{vrai}$$

Enfin, on peut coder facilement diverses fonctions numériques usuelles.

$$\begin{aligned} \text{plus} &:= \lambda nm.\, \lambda fx.\, nf(mfx) \\ \text{mult} &:= \lambda nm.\, \lambda f.\, n(mf) \end{aligned}$$

Pour des raisons de lisibilité, on notera $(a+b)$ et $(a \times b)$ au lieu de (plus a b) et (mult a b) respectivement, mais cela reste des termes du λ-calcul pur.

2.3 Paires

Enfin, on voudrait pouvoir représenter des couples de λ-termes. On sait déjà représenter des entiers, donc on pourrait imaginer d'implémenter l'usuelle bijection de $\mathbb{N}^2$ vers $\mathbb{N}$, pour représenter un couple d'entier par un simple entier. Pour généraliser à tout λ-terme, il suffit de choisir un Gödel-codage sur les λ-termes (qui sont finis) pour les représenter par des entiers. Cependant, il faudrait alors pouvoir reconstruire le lambda-terme correspondant à son Gödel-codage, soit en substance implémenter un méta-évaluateur (de λ-calcul en λ-calcul). Pour commencer, on ne sait même pas coder la fonction pred (qui à 0 associe 0, et à n associe $n-1$) pour l'instant : vous pouvez essayer, c'est un exercice qui n'est pas évident.

Il existe une solution beaucoup plus simple et naturelle. Il suffit de représenter les couples comme une application partielle attendant les deux membres du couple comme arguments :

$$\text{pair} := \lambda xy.\, \lambda f.\, fxy$$

(pair ab) construira donc un λ-terme qui prend en paramètre une fonction f et l'applique aux deux éléments du couple [2]. En particulier, si l'on veut obtenir l'un des deux éléments, il suffit de lui passer une fonction de sélecteur qui renvoie soit le premier, soit le second de ses arguments. Par chance, nous avons déjà rencontré ces fonctions, ce sont les booléens.

$$\begin{aligned} \text{fst} &:= \lambda p.\, p \text{ vrai} \\ \text{snd} &:= \lambda p.\, p \text{ faux} \end{aligned}$$

Pour des raisons de lisibilité, on notera (a, b) au lieu de (pair a b).

Il devient alors possible de coder plutôt simplement la fonction pred. L'idée est d'associer par itérations un couple valant (pred n, n) à l'entier n : pour 0 il suffit de donner $(0, 0)$, et ensuite d'itérer n fois l'opération $(a, b) \mapsto (b, b+1)$.

$$\begin{aligned} \text{succpair} &:= \lambda p.\, ((\lambda n.\, (n, \text{succ } n))(\text{snd } p)) \\ \text{pred} &:= \lambda n.\, \text{fst } (n \text{ succpair } (0, 0)) \end{aligned}$$

On peut aussi généraliser le codage des couples à des dimensions supérieures. Pour n entier, on peut représenter le constructeur de n-uplet (couple à n membres) et les projections.

$$\begin{aligned} \text{prod}_n &:= \lambda f.\, \lambda x_1 \dots x_n.\, fx_1 \dots x_n \\ \text{proj}_i^n &:= \lambda p.\, p(\lambda x_1 \dots x_n.\, x_i) \end{aligned}$$

2. en terme d'implémentation de langages de programmation, on stocke le couple dans une fermeture

2.4 Fonctions récursives primitives

Grâce au codage de Church des entiers naturels, on peut définir une notion de calculabilité des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ selon le λ-calcul : une fonction $\bar{f} : \mathbb{N} \to \mathbb{N}$ est λ-calculable s'il existe un λ-terme f tel que pour tout entier naturel $\bar{n}$ de code n, le résultat de l'évaluation (par β–réductions successives) du terme $(f\ n)$ est le codage de l'entier $\bar{f}(\bar{n})$.

Les fonctions récursives primitives sont λ-calculables : on sait représenter les fonctions constantes, n-uplets, et projections. Il est clair que l'on peut écrire les projections de fonctions (pour les fonctions d'arité 1, $f \circ g = \lambda x.\, f(gx)$). Il reste à prouver que l'on peut représenter en λ-calcul la fonction h, définie par récursion primitive à partir de deux fonctions f et g :

$$\begin{aligned} h(0, x_1, \ldots, x_n) &= f(x_1, \ldots, x_n) \\ h(n+1, x_1, \ldots, x_n) &= g(h(n, x_1, \ldots, x_n), x_1, \ldots, x_n) \end{aligned}$$

Nous traiterons le cas des fonctions d'arité 1, la généralisation ne posant pas de difficulté. Si le terme c représente une constante entière (ou fonction d'arité 0), et g une fonction $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ λ-calculable, on voudrait un λ-terme h vérifiant $h(0) = c$ et $h(n+1) = f(h(n), n)$.

Si on avait $h(n+1) = f(h(n))$, il s'agirait en fait de l'itérée de la fonction h, que l'on peut représenter facilement en λ-calcul. Il manque cependant le paramètre n supplémentaire. On va utiliser la même méthode que pour pred : on transmet le couple $(h(n), n)$ à la fonction itérée.

$$\begin{aligned} \text{succf} &:= \lambda p.\, p(\lambda x n.\, (f\ x\ n,\ \text{succ}\ n)) \\ h &:= \lambda n.\, \text{fst}\ (n\ (c, 1)\ \text{succf}) \end{aligned}$$

3 Récursion

3.1 Combinateur de point fixe

On voudrait maintenant pouvoir définir des fonctions récursives. L'exemple le plus courant est la fonction factorielle :

$$\text{fac} \stackrel{?}{=} \lambda n.\, (\text{iszero}\ n\ 1\ (n \times \text{fac}(\text{pred}\ n))))$$

Une telle définition ne serait pas correcte puisqu'elle utilise le terme fac qui n'est pas défini. On peut la modifier légèrement pour remplacer l'appel récursif de fac par l'appel d'une fonction f passée en paramètre.

$$\text{fac'} := \lambda f n.\, (\text{iszero}\ n\ 1\ (n \times f(\text{pred}\ n)))$$

Cette fonction fac' n'est pas la factorielle : pour obtenir une factorielle, il faudrait l'appeler en lui donnant "elle-même" en argument :

$$\text{fac} = \text{fac'}(\text{fac}) = \text{fac'}(\text{fac'}(\text{fac})) = \ldots$$

Autrement dit, on cherche le point fixe de l'opération $f \to \text{fac'}\ f$. Plus généralement, on voudrait (afin de pouvoir écrire toute fonction récursive, et pas seulement

fac) disposer d'un *combinateur de point fixe*, c'est-à-dire un λ-terme Y vérifiant l'équation suivante :

$$Y(f) = f(Y(f))$$

Il se trouve qu'il est possible de définir ce terme en λ-calcul.

$$\begin{aligned} \text{auto} &:= \lambda x.\, x\, x \\ Y &:= \lambda f.\, \text{auto}\ (\lambda x.\, f(\text{auto}\ x)) \end{aligned}$$

Le terme vérifie bien l'équation demandée :

$$\begin{aligned} Y(f) &= (\lambda f.\, \text{auto}\ (\lambda x.\, f(\text{auto}\ x)))f \\ &=_\beta \text{auto}\ (\lambda x.\, f(\text{auto}\ x)) \\ &=_\beta f(\text{auto}\ (\lambda x.\, f(\text{auto}\ x))) \\ &= f(Y(f)) \end{aligned}$$

On peut alors définir fac .

$$\text{fac} := Y(\text{fac'}\ )$$

Il est en fait possible de définir fac plus simplement, comme une fonction récursive primitive.

3.2 Fonctions récursives

Tous les λ-termes ne possèdent pas une forme normale : il est facile de voir par exemple que la β–réduction du terme $Y(\lambda x.\, x)$ donne le même terme, qui n'est pas une forme normale (Y est une abstraction placée dans une application). On peut dire par analogie aux langages de programmation que l'évaluation de $Y(\lambda x.\, x)$ "boucle à l'infini".

On a donc aussi en λ-calcul une notion de *fonctions partielles* : ce sont des fonctions dont l'application à certains arguments est un λ-terme ne possédant pas de forme normale.

Il est facile d'implémenter le schéma μ non borné par une fonction récursive. On peut donc représenter en λ-calcul toutes les fonctions récursives (donc toutes les fonctions calculables au sens de Turing).

Résultat. *Toutes les fonctions calculables sont λ-calculables*

À l'inverse, on peut aussi montrer (mais ce n'est pas le but du présent article) que toutes les fonctions λ-calculables sont calculables. Il suffit en fait de se convaincre qu'on peut implémenter un "évaluateur" de λ-termes sur un ordinateur, et en dériver un programme équivalent pour machine de Turing (cela revient à écrire un compilateur).

Résultat admis. *La réduction d'un λ-terme sous forme normale, si elle existe, est calculable.*

Une fois cela admis, le résultat est clair : soit f une fonction λ-calculable, il suffit pour montrer qu'elle est calculable de construire une machine de Turing qui, étant donné un entier $\bar{n}$:

– construit le terme fn, où n est le Church-encoding de $\bar{n}$

– réduit ce terme en forme normale
– transforme le Church-encoding du résultat en un format admis pour la machine de Turing

Le passage d'un entier à son encodage de Church, ou inversement, ne pose pas de difficulté (même si la manipulation d'une machine de Turing est si fastidieuse que l'explicitation de ce point n'est pas envisageable).

Résultat. *Toutes les fonctions λ-calculables sont calculables. Le modèle de calculabilité du λ-calcul est équivalent à celui des machines de Turing.*

4 Lambda-calcul simplement typé

Le résultat de la section précédente peut être vu comme un résultat positif (le λ-calcul est aussi puissant que les autres modèles de calcul), mais aussi comme un résultat négatif : il existe des λ-termes non normalisables.

Il est possible d'éviter ce désagrément en enrichissant le langage d'un système de typage : on essaie d'attribuer un type aux λ-termes, de manière à ce que tous les termes typables soient normalisables (mais il existe des termes au typage non valide, entre autres les termes non normalisables).

Il existe en fait de nombreux λ-calculs typés, correspondant à de nombreux systèmes de typages différents. On présentera ici le λ-calcul *simplement typé*, et plus tard le λ-calcul *de second ordre*.

4.1 Typage simple

On enrichit le langage du λ-calcul par une nouvelle classe d'objets, les *types*. Ils sont définis par la grammaire suivante, où τ représente n'importe quelle variable (de type) :

$$T ::= \tau \mid T \to T$$

Le type $A \to B$ est le type des abstractions qui prennent une variable de type A en paramètre et renvoient une variable de type B. Par analogie avec les opérations ensemblistes, on parle d'*exponentiation*. La flèche d'exponentiation est associative à droite : $A \to B \to C = A \to (B \to C)$.

On modifie aussi légèrement la grammaire des λ-termes pour que le type des variables apparaisse explicitement dans les abstractions (on le note en exposant) :

$$E ::= v \mid E\ E \mid \lambda v^T.\, E$$

Ainsi, le terme $\lambda x^{\tau_1}.\, x$ est de type $\tau_1 \to \tau_1$: c'est l'identité sur les λ-termes de type τ_1. Si l'on note $T[x]$ le type du λ-terme clos x, on peut définir T récursivement :

– $T[MN] = B$ si $T[M] = A \to B$ et $T[N] = A$, MN n'est pas typable sinon
– $T[\lambda x^t.E.]$ est le type de E, sachant que les occurrences liées de x sont de type t
– $T[v]$ est le type connu de v (puisque le terme initial était clos, l'occurrence v apparaît sous le champ d'une abstraction, et son type est donc connu)

On notera $x : \tau$ pour dire "x est de type τ".

4.2 Récursion

On ne s'autorise désormais qu'à écrire des λ-termes correctement typés. Par exemple, le terme $\lambda x^\tau y^\tau . xy$ n'est pas valide, car le type de x n'est pas de la forme $t_A \to t_B$.

En particulier, on ne peut pas donner un type au λ-terme non typé auto $= \lambda x. xx$: puisque x est appliqué à lui-même, son type est de la forme $t_1 \to t_2$, mais son argument est x, donc $t_1 = t_1 \to t_2 = (t_1 \to t_2) \to t_2 = \dots$: si x était typable, son type serait infini, ce qui est absurde. Le combinateur de point fixe Y n'est donc pas typable (puisqu'il contient un terme non typable).

On a montré que notre définition de Y n'était pas typable, mais à priori il pourrait exister un autre λ-terme vérifiant l'équation de Y, typable. On peut montrer que tout λ-terme typé est normalisable, donc en particulier qu'il n'existe pas de combinateur de point fixe en λ-calcul simplement typé.

4.3 Codages avec typage simple

On parle pour les types de la forme τ de *variables*, mais en réalité on les manipule comme des constantes : il n'y a pas au niveau des types de mécanisme permettant d'assigner une valeur à une variable, donc elle sera toujours utilisée dans sa forme "libre", pour la comparer à une autre variable.

Il reste possible de manipuler des entiers. Il suffit de choisir ν une variable de type, et de poser pour $n \in \mathbb{N}$:

$$\begin{aligned} Nat &:= (\nu \to \nu) \to \nu \to \nu \\ n &:= \lambda f^{\nu\to\nu} x^\nu . \underbrace{f(f \dots (f}_{n fois} x) \dots) \end{aligned}$$

Les définitions des opérations usuelles fonctionnent comme précédemment.

Étant donné un type α, on peut construire un type booléen $Bool_\alpha : \alpha \to \alpha \to \alpha$, avec $\text{vrai}_\alpha := \lambda x^\alpha y^\alpha . x$, etc. Il n'est plus possible d'appliquer un booléen à n'importe quels λ-termes : les branches doivent être de type α. On a donc besoin d'une famille de booléens couvrant tous les types que l'on veut manipuler.

De même, la définition précédente des paires ne convient pas : pour trois types α, β et γ on peut construire un terme $p : \lambda x^\alpha y^\beta . \lambda f^{\alpha\to\beta\to\gamma} fxy.$, mais ce type est beaucoup moins général que les paires du λ-calcul non typé : on ne peut utiliser ses paires qu'avec des opérations de type de retour γ. En particulier, on ne peut pas (sauf si $\alpha = \beta$) accéder aux deux éléments du couple, et plus généralement il faut connaître, au moment de la création de la paire, le contexte dans lequel il sera utilisé (qui indique le type γ pertinent).

Pour compenser le manque d'aisance qui découle de ces restrictions, on peut étendre le langage en rajoutant des constructions. On peut ajouter des types *sommes* et *produits* ; pour cela, on enrichit le langage des types :

$$T ::= \tau \mid T \to T \mid T + T \mid T * T$$

Le type produit est l'analogue du produit cartésien ensembliste : un terme de type $\alpha * \beta$ contient un élément de type α *et* un élément de type β. On ajoute les constructions suivantes au langage des λ-termes :

– Pour $x : \alpha$ et $y : \beta$, la construction de couple $(x, y) : \alpha * \beta$

– Pour $p : \alpha * \beta$, les projections $(\pi_1\ p) : \alpha$ et $(\pi_2\ p) : \beta$

On rajoute aussi deux règles de β–réduction de la forme

$$\pi_k(x_1, x_2) \xrightarrow{\beta} x_k$$

Le type somme est similaire à l'union disjointe de deux types : un élément de type $\alpha + \beta$ contient un élément de type α *ou* un élément de type β. On ajoute les constructions suivantes au langage des λ-termes :

– Pour $x : \alpha$ ou $y : \beta$, les injections $(i_1\ x) : \alpha + \beta$ et $(i_2\ y) : \alpha + \beta$

– Pour $s : \alpha + \beta$, $f_1 : \alpha \to \gamma$, $f_2 : \beta \to \gamma$, le filtrage de somme $(case_{f_2}^{f_1}\ s) : \gamma$

On rajoute encore deux règles de β–réduction de la forme

$$case_{f_2}^{f_1}\ (i_k\ t) \xrightarrow{\beta} f_k\ t$$

Le type produit permet de créer facilement des paires, indépendamment de leur contexte d'utilisation, et le filtrage de motifs correspondant à la construction *case* permet d'exprimer des conditions.

Ces ajouts représentent un compromis : on a perdu en simplicité (et donc en facilité de manipulation formelle) du langage, mais gagné en expressivité. A-t-on fait une bonne affaire ?

Il se trouve que les deux constructions ajoutées, somme et produit, permettent d'exprimer une grande quantité de types, la famille qu'on appelle les "types algébriques" (qui s'expriment comme des polynomes en une ou plusieurs variables de types). La pratique a montré que ces constructions suffisaient [3] à exprimer une très grande partie des types dont ont naturellement besoin les programmeurs (on en retrouve d'ailleurs des formes masquées dans tous les langages de programmation, comme les *struct* et *union* en C) : dans le cadre du λ-calcul simplement typé, qui a ses limitations fondamentales (absence de polymorphisme), ces deux ajouts suffisent à obtenir un modèle réellement utilisable.

5 Normalisation

5.1 Théorème de normalisation

Nous allons maintenant montrer que les λ-termes typés sont normalisables. C'est une conséquence de la différence fondamentale entre le lambda-calcul non typé et les lambda-calculs typés.

On peut voir cela comme une propriété négative (le λ-calcul ainsi défini ne permet pas d'exprimer la récursion, donc est strictement moins puissant au niveau de ses capacités de calcul), mais c'est une propriété très intéressante. En particulier, en λ-calcul simplement typé, le problème de l'arrêt est décidable : tous les programmes ($\approx$ λ-termes) bien typés terminent. Concrètement, il suffirait à un compilateur de vérifier que le programme est bien typé, et l'utilisateur pourrait l'exécuter en ayant la garantie qu'il s'arrête (... un jour).

Idée de la preuve On va exprimer une fonction donnant une mesure de la "complexité" d'un λ-terme, et montrer qu'on peut la faire décroître strictement par β–réductions successives. Cette preuve provient de [Girard89] ; la preuve du

3. à priori, on peut même se contenter de seulement l'une des deux

théorème de limitation sera elle aussi fortement inspirée de cet ouvrage.

On définit le *degré* d'un type par induction sur sa structure :
- $\delta(\tau) = 1$
- $\delta(T \to T') = 1 + \max(\delta(T), \delta(T'))$

Le degré d'un type est exactement sa hauteur en tant qu'arbre.

Quand un λ-terme n'est pas en forme normale, on appelle *redexes* ses sous-termes directement réductibles. Un redexe t est nécessairement de la forme $(\lambda x.\, A)B$ [4]. On définit son degré (où $T[t]$ désigne encore le type du terme t) :

$$d_r((\lambda x.\, A)B) = \delta(T[\lambda x.\, A]) = \delta(T[B] \to T[A])$$

En particulier on a toujours $\delta(T[B]]) < d_r(t)$.

Enfin, on définit le degré $d(t)$ d'un terme t par le degré maximal de ses redexes, et 0 s'il est en forme normale.

Lemme. *Pour t, u des λ-termes et v une variable :*

$$d(t[u/v]) \leqslant \max(d(t), d(u), \delta(T[u]))$$

Il suffit de constater que les redexes de $t[u/v]$ sont :
- les redexes de t, dont la substitution n'affecte pas le degré ;
- les redexes de u, potentiellement dupliqués par la substitution, ce qui n'augmente pas leur degré ;
- de nouveaux redexes si u est une abstraction et si t contenait des sous-termes de la forme $v\ x$; il y a maintenant le redexe $u\ x$ dont le degré est, par définition de d_r, $\delta(T[u])$.

Lemme. *Si $t \xrightarrow{\beta} t'$, alors $d(t') \leqslant d(t)$*

L'un des redexes de t, noté r, a été β–réduit en le terme r' dans le terme t'. Les redexes affectés par la réduction sont :
- un potentiel redexe de la forme $r\ x$, qui est devenu $r'\ x$:

$$d(r'\ x) = \delta(T[r']) = \delta(T[r]) < d_r(r) \leqslant d(t)$$

- les redexes de r', qui sont obtenus par une réduction (ie. substitution) à l'intérieur de r : si r est de la forme $(\lambda v.\, r_1)r_2$, alors $r' = r_1[r_2/v]$ donc :

$$d(r') \leqslant \max(d(r_1), d(r_2), \delta(T[r_2])) \leqslant \delta(T[r_2]) < d(r) \leqslant d(t)$$

Pour un λ-terme t, on considère les redexes de degré maximal $d(t)$. Nécessairement, l'un de ces redexes ne contient que des redexes de degré strictement inférieur (car il n'y a qu'un nombre fini de redexes) ; si on le β–réduit, le nombre de redexes de degré maximal décroît strictement (car les redexes qu'il contient, bien que potentiellement dupliqués par la substitution, n'augmentent pas en degré). on peut donc, en un nombre fini de β–réductions, faire décroître strictement le degré maximal de t, donc, en répétant le processus, en obtenir une forme normale (car les formes normales sont les seuls termes de degré nul).

4. Dans le cadre d'un λ-calcul enrichi de sommes et de produits, il existe d'autres redexes liés aux nouvelles possibilités de β–réduction de ces constructions. Ils se comportent de manière semblable, et ne seront pas explicitement mentionnés dans les preuves.

Résultat. *Tout λ-terme simplement typé est normalisable.*

On peut en fait obtenir un résultat encore plus fort : on a montré que pour tout λ-terme, il existe une suite de réductions aboutissant à sa forme normale. On peut montrer que *toute* stratégie de réduction aboutit à la forme normale. C'est ce qu'on appelle la normalisation *forte*, et nous ne le prouverons pas.

5.2 Théorème de limitation

D'après le théorème de normalisation, les fonctions non totales ne sont pas représentables en λ-calcul simplement typé. On peut en fait obtenir un résultat plus contrariant : il existe des fonctions *totales* qui ne sont pas représentables.

On va exhiber une fonction non représentable, par le biais d'un argument diagonal. Ce qui est satisfaisant, c'est que la fonction utilisée n'est pas très artificielle, mais plutôt naturelle (du moins pour un informaticien) : on va montrer qu'il n'est pas possible de coder un *évaluateur* (dans un sens à préciser) d'un λ-calcul avec normalisation à l'intérieur de ce même λ-calcul.

On commence par choisir un Gödel-codage λ-calculable sur les λ-termes : un λ-terme t est un arbre fini, donc peut être représenté par un entier $\sharp t$.

La notion d'*évaluation* correspond en λ-calcul à une réduction en forme normale. On définit donc la fonction $\text{eval} : \mathbb{N} \to \mathbb{N}$:

- $\text{eval}(\sharp a) = \sharp b$ où b est la forme normale de a
- $\text{eval}(n) = 0$ si n n'est pas de la forme $\sharp t$

D'après le théorème de normalisation, cette fonction de réduction est totale. D'après le résultat admis en 3.2, elle est de plus calculable [5] ; c'est une fonction récursive totale. Supposons par l'absurde que cette fonction est λ-calculable.

On s'en sert pour construire une deuxième fonction λ-calculable, 'apply', qui applique une fonction λ-calculable en un argument. Si n est l'encodage de Church de l'entier $\bar{n}$, et que $\text{int}(t)$ désigne l'entier correspondant à l'entier de Church t (ou 0 si c'est un autre λ-terme),

$$\text{apply}(\sharp f, \bar{n}) = \text{int}(\text{eval}(\sharp(f\ n)))$$

En particulier, si f est la λ-représentation d'une fonction $\bar{f} : \mathbb{N} \to \mathbb{N}$ et $\bar{n} \in \mathbb{N}$, on a par construction

$$\text{apply}(\sharp f, \bar{n}) = \bar{f}(\bar{n})$$

On peut alors écrire une fonction $\text{evil} : \mathbb{N} \to \mathbb{N}$:

$$\text{evil}(c) = 1 + apply(c, c)$$

5. il y a une légère subtilité ici : le résultat admis portait sur le λ-calcul non typé, un autre langage. Cependant, comme un terme du λ-calcul typé peut aussi être interprété (en "oubliant" les informations de typage) comme un terme non typé valide, on peut réutiliser le résultat précédent. Il nous fournit une machine de Turing (à priori non déterministe) qui termine pour chaque terme s'il existe une suite de β–réductions vers une forme normale. Le théorème de normalisation nous garantit que ce chemin existe toujours, ce qui rend la machine totale dans le cas du λ-calcul typé. On pourrait aussi construire une machine de Turing spécifique, mettant en oeuvre l'algorithme exhibé par la preuve de normalisation, qui donne explicitement une bonne stratégie de réduction : on aurait alors une machine déterministe, n'essayant qu'un seul chemin de réductions, et garantissant la terminaison.

Puisque eval est supposée λ-calculable, evil l'est aussi, et correspond à un λ-terme e. On peut enfin laisser le serpent se mordre le queue :

$$\begin{aligned}\text{evil}(\sharp e) &= 1 + \text{apply}(\sharp e, \sharp e)\\ &= 1 + \text{evil}(\sharp e)\end{aligned}$$

Résultat. *Il existe des fonctions récursives totales qui ne sont pas représentables en λ-calcul typé.*

La preuve ne repose pas en fait spécifiquement sur le λ-calcul simplement typé, mais sur la propriété de normalisation. On peut donc l'étendre à d'autres langages fonctionnels totaux.

6 Lambda-calcul de second ordre : System F

Avec le λ-calcul simplement typé, la puissance du langage des types n'est pas vraiment satisfaisante. Par exemple, on doit écrire une fonction identité pour chaque type : si τ et τ' sont des types différents, on a $\lambda x^{\tau}.\, x$ et $\lambda x^{\tau'} x.$, deux fonctions identité non compatibles. C'est le *statu quo* en mathématiques, où les fonctions ayant des domaines de départ et d'arrivée distincts sont différentes, mais cela rend l'écriture de programme assez inconfortable.

6.1 Typage de second ordre

On voudrait pouvoir représenter un objet qui, *étant donné* un type t, fournit la fonction identité $\lambda x^{t}.\, x$. C'est l'idée sous-jacente au λ-calcul de second ordre. On ajoute une abstraction au niveau des types : un λ-type peut dépendre d'autres λ-types.

$$T ::= \tau \mid T \to T \mid \Delta\tau.\, T$$

Par exemple, le type de notre "identité sur tous les types" sera $\Delta\tau.\, \tau \to \tau$.

Il faut alors traduire cet ajout au niveau des λ-termes : un terme peut dépendre d'un type, et on peut lui appliquer un type.

$$E ::= v \mid E\ E \mid \lambda v^{T}.\, E \mid \Lambda\tau.\, E \mid E\ T$$

Les λ-termes de type $\Delta\tau.\, T$ sont ceux de la forme $\Lambda\tau.\, E$ quand T est le typede E. Par exemple, le λ-terme correspondant au type $\Delta\tau.\, \tau \to \tau$ est $\Lambda\tau.\, \lambda x^{\tau}.\,$. Si t est un type, on obtient la fonction identité sur t en l'appliquant à ce λ-terme :

$$\text{id}_t = (\Lambda\tau.\, \lambda x^{\tau}.\, x)\ t$$

Pour interpréter les applications de types, on étend la substitution, puis la β–réduction aux types.

- $(\lambda x^{T}.\, E)[t/\tau] = \lambda x^{T[t/\tau]}.\, E[t/\tau]$
- $(\Lambda\tau.\, E)[t/\tau] = \Lambda\tau.\, E \qquad (\Lambda\tau'.\, E)[t/\tau] = \Lambda\tau'.\, E[T/\tau]$
- $(T \to T')[t/\tau] = T[t/\tau] \to T'[t/\tau]$
- $\tau[T/\tau] = T \qquad \tau'[T/\tau] = \tau'$

$$(\Lambda\tau.\, E)T \quad \xrightarrow{\beta} \quad E[T/\tau]$$

Pour une description plus en profondeur des systèmes de typage de F_1 et F_2, vous pouvez vous réferer par exemple à [Cardelli97].

Les résultats de normalisation peuvent être étendus à F_2, même si la preuve (présentée par exemple dans [Girard89]) est nettement plus compliquée : les termes de F_2 sont fortement normalisables. À l'inverse, la preuve du théorème de limitation s'étend immédiatement : la fonction récursive totale de réduction en forme normale de F_2 n'est pas exprimable dans F_2.

6.2 Produit et somme

On a vu que le λ-calcul simplement typé était sévèrement limité dans sa capacité à exprimer des structures de données, ce qui nous a forcé à ajouter des constructions au langage. Le λ-calcul de second ordre, grâce à la possibilité d'abstraction des types, est beaucoup plus expressif : il ne sera pas nécessaire de lui rajouter des constructions utilitaires.

Le problème avec la représentation usuelle des paires de type A et B en λ-calcul simplement typé était la dépendance de leur type, $A \to B \to \Gamma$, à un type externe Γ. On peut maintenant le surmonter en généralisant

$$\begin{aligned} A * B &:= \Delta\Gamma.\ A \to B \to \Gamma \\ (a, b) &:= \Lambda\Gamma.\, \lambda f^{A\to B\to\Gamma}.\, fab \\ \pi_1(p : A * B) &:= pA(\lambda x^A y^B.\, x) \end{aligned}$$

On peut de même représenter les types sommes

$$\begin{aligned} A + B &:= \Delta\Gamma.\ (A \to \Gamma) \to (B \to \Gamma) \to \Gamma \\ i_1(x : A) &:= \Lambda\Gamma.\, \lambda f_1^{A\to\Gamma} f_2^{B\to\Gamma}.\, f_1 a \\ case^{f_1:A\to\Gamma}_{f_2:B\to\Gamma}\ s &:= s\Gamma f_1 f_2 \end{aligned}$$

6.3 Définition d'une structure de données

On va maintenant essayer de généraliser les codages effectués jusqu'à présent. On voudrait pouvoir encoder plus de données, et surtout obtenir une méthode "universelle" d'encodage, c'est-à-dire qui aboutit pour toute une famille de structures de données. On n'a pour l'instant aucun procédé systématique permettant de donner le λ-codage d'un type de données particulier.

Les structures de données que l'on veut représenter sont de la forme suivante : elles partagent toutes un même type, et sont obtenues en combinant des *constructeurs*, un ensemble de constantes et de fonctions permettant de construire des structures à partir de structures plus simples, ou de paramètres (par exemple les constructeurs d'une liste d'entiers prendront des entiers en paramètres). Les constructeurs sont typés, et seules les structures les utilisant correctement sont valides. C'est le modèle utilisé par la plupart des langages de programmation, ou encore la plupart des définitions inductives en mathématiques.

Il existe différentes manières de formaliser cette description. L'article qui a fourni le résultat important de cette partie, [Böhm-Berarducci85], utilise des structure algébriques libres, les algèbres de termes hétérogènes. Afin de ne pas

introduire un formalisme supplémentaire, nous verrons une définition différente (inspirée de leurs *noms* qui sont des λ-termes) : on définira un type de données comme une extension du λ-calcul, où le type de la structure est une nouvelle constante de type, et les constructeurs sont représentés par des λ-termes constants typés. C'est ce que nous avons fait pour ajouter somme et produit au λ-calcul simplement typé.

Un type de données est déterminé par :

- une constante de type D^c, le *type résultat*
- une famille de constantes de type $(A_p)_{p\leqslant n_a}$, les *types paramètres*
- une famille $(C_i^c)_{i\leqslant n_c}$ de "constructeurs", des symboles de constantes de termes, auxquels sont associés les types (T_i^c) d'arité $a(i)$ vérifiant :
  - $T_i^c = T_{i,1}^c \to T_{i,2}^c \to \cdots \to T_{i,a(i)}^c \to D$
  - Pour $k < a(i)$, $T_i^k \in \{D^c\} \bigcup \{A_p\}_{p\in P}$

On définit alors un *élément* de la structure de données comme un λ-terme clos, en forme normale, de type D^c.

Par exemple, on peut représenter les "listes" par le type de donnée dont le type résultat est L^c, qui possède un seul type paramètre A et les constructeurs suivants :

- $nil : L^c$, la liste vide [ ]
- $cons : A \to L^c \to L^c$, qui à $a_1 : A$ et $[a_2\ a_3\ \ldots\ a_n]$ associe $[a_1\ a_2\ \ldots\ a_n]$

6.4 Codage d'une structure de donnée

Étant donné une structure $D^c, (A_p), (C_i^c)$, on souhaite représenter cette structure par un type D dans le λ-calcul pur (sans ajouts), et une famille $(C_i)_{i\leqslant n_c}$ de λ-termes correspondant aux constructeurs (C_i).

Il faut commencer par choisir des types du λ-calcul pur correspondant aux types paramètres (A_p), et des éléments de ces types pour les $a_1 \ldots a_n$. Pour alléger les notations (qui en ont bien besoin, comme vous le constaterez sans doute), on laissera cette transformation implicite : quand on écrit a_i, on sous-entend qu'il s'agit d'un λ-terme du type correspondant au type paramètre. Pour l'exemple des listes, si on veut des listes d'éléments de type Γ, on aura $A := \Gamma$ et $a_i : \Gamma$.

On pose alors, si X n'apparaît dans aucun des A_p,

$$D := \Delta X.\ T_1^X \to T_2^X \to \cdots \to T_{n_c}^X \to X$$
$$\text{avec } T_i^X := T_i^c[X/D^c][A_p/A_p^c]_{p\leqslant n_a} \quad (i \leqslant n_c)$$

L'idée (semblable à celle utilisée pour les sommes et produits) est d'abstraire les parties qui étaient fixées dans la définition de la structure. Ici, on a remplacé le type D^c par un type X sur lequel on abstrait. On fait la même chose au niveau des termes : pour chaque constructeur C_i^c on choisit une variable c_i, et on encode le terme t ainsi :

$$\Lambda X.\, \lambda c_1^{T_1^X} \ldots c_{n_c}^{T_{n_c}^X}.\, t[c_i/C_i^c]_{i\leqslant n_c}$$

Par exemple, la liste $[a_1\ a_2]$, représentée dans notre type de données par $cons\ a_1\ (cons\ a_2\ nil)$), est encodée selon cette méthode par

$$\Lambda X.\, \lambda n^X c^{A\to X\to X}.\, c\ a_1\ (c\ a_2\ n)$$

Le type de données de ces listes est $List_A := \Delta X.\ X \to (A \to X \to X) \to X$.

6.5 Structures de données et manipulations usuelles

6.5.1 Structures courantes

Ce codage représente une manière systématique d'obtenir un codage en λ-calcul pur d'une structure de données courante. En particulier, on peut retrouver mécaniquement les encodages exhibés en partie 2.

Entiers Les entiers sont caractérisés par l'élément nul, et la fonction successeur : on part donc de la structure de données de type N et des constructeurs $succ : N \to N$ et $zero : N$. Le codage donne alors

$$\begin{aligned} Nat &:= \Delta N.\ (N \to N) \to N \to N \\ 2 &:= \Lambda N.\ \lambda s^{N \to N} z^{N}.\ s(sz) \end{aligned}$$

On peut retrouver de façon similaire les autres encodages (booléens, paires). On peut de plus produire facilement d'autres encodages, comme par exemple les arbres binaires, de type T et de constructeurs $leaf : T$ et $node : T \to T \to T$:

$$Tree := \Delta T.\ T \to (T \to T \to T) \to T$$

6.5.2 Constructeurs

On a présenté les constructeurs comme des constantes de base, mais on peut aussi les représenter comme des fonctions agissant sur la structure. On peut bien sûr coder ces fonctions.

À chaque

$$C_i^c : T_{i,1}^c \to T_{i,2}^c \to \cdots \to T_{i,a(i)}^c \to D$$

on associe un C_i

$$C_i := \lambda x_1^{T_{i,1}} \ldots x_{a(i)}^{T_{i,a(i)}}.\ \Lambda X.\ \lambda c_1 \ldots c_{n_c}.\ c_i\ (x_1 X c_1 \ldots c_{n_c})\ \ldots\ (x_{a(i)} X c_1 \ldots c_{n_c})$$

$$\text{avec} \quad T_{i,j} := T_{i,j}^c[D/D^c] \quad \text{et} \quad T_i := T_i^c[D/D^c]$$

L'expression est un peu lourde, mais une fois que l'on a ces constructeurs il est possible de décrire des valeurs dans le λ-calcul pur directement, sans avoir à repasser par la phase de traduction (par exemple on peut construire le λ-terme pur représentant une liste à deux éléments uniquement avec les constructeurs purs des listes).

6.5.3 Catamorphismes

On a vu en 2.4 qu'il est possible d'utiliser la représentation de Church des entiers pour écrire des fonctions primitives récursives. Le codage présenté ici permet en fait de généraliser cette propriété : on peut l'utiliser pour exprimer simplement tous les *catamorphismes* de structures de données. Les catamorphismes, souvent nommés *fold* dans les langages de programmation fonctionnels, sont des classes d'algorithme parcourant des structures de données récursives de manière prédeterminée. Ce sont les fonctions itératives définies dans [Böhm-Berarducci85].

On a déjà exprimé 'pred' et 'fac' comme catamorphismes. Voici trois autres exemples, 'nodes', qui compte les noeuds d'un arbre binaire, 'concat' qui concatène deux listes, et 'list' qui à a et n associe la liste $[a \ \ldots a]$ de taille n.

$$\begin{aligned} \text{nodes} &:= \lambda t^{Tree}.\, t\ Nat\ 0\ (\lambda a^{Nat} b^{Nat}.\, a + b) \\ \text{concat}_A &:= \lambda a^{List_A} b^{List_A}.\, a\ List_A\ b\ cons_A \\ \text{list} &:= \lambda a^{A} n^{Nat}.\, n\ List_A\ nil_A\ (cons_A\ a) \end{aligned}$$

6.6 Un théorème de complétude

Étant donné une structure de données D^c, on a montré qu'on pouvait construire un λ-type D, et coder tous les éléments de D^c à l'intérieur de ce type. On peut se poser une autre question : tous les λ-termes de type D correspondent-ils bien à des éléments de D^c ? Il se trouve que la réponse est "oui, à équivalence des λ-termes près".

Soit en effet un terme t de type D. On veut montrer que t correspond à un terme de notre λ-calcul étendu ayant servi à définir D^c. Mais on peut plonger notre terme t dans le λ-calcul étendu, ce qui nous donne un terme t'.

On pose

$$t" := t'\ D^c\ C_1^c\ \ldots\ C_{n_c}^c$$

Comme $t' : \Delta X.\ T_1^X \to \cdots \to T_{n_c}^X \to D$, on a $t" : D^c$. Il suffit de montrer que la forme normale de $t"$ est un élément valide de notre structure, c'est-à-dire que $t"$ est sans abstraction :

- il n'est pas de la forme $\lambda x.\, E$ puisque D_c n'est pas une exponentielle ;
- il ne contient aucun redexe $(\lambda x.\, E)F$ puisqu'il est en forme normale ;
- il ne peut contenir de sous-terme de la forme $E(\lambda x.\, F)$: si c'était le cas, on considérerait le plus à gauche de ces sous-termes. Alors E ne contient aucune abstraction, donc est construit à partir des C_i^c et des éléments des A_p uniquement. Par définition des types des C_i, ils n'admettent que des arguments de type D^c ou A_p, or $(\lambda x.\, F)$ a un type de la forme $A \to B$, ce qui est absurde.

Résultat. *Si D est le type encodé d'un type de données D^c, alors tous les λ-termes de type D correspondent à un élément de D^c.*

Références

[Böhm-Berarducci85] Automatic Synthesis of Typed Λ-Programs on Term Algebras
C. Böhm and A. Berarducci, *Theoretical Computer Science*, 1985.

[Cardelli97] Type Systems
L. Cardelli, *Handbook of Computer Science and Engineering*, 1997.

[Girard89] Proofs and Types
J.-Y. Girard, traduction par P. Taylor et Y. Lafont, 1989.