λ-calcul et typage

Nicolas Barnier, Pascal Brisset

ENAC

Avril 2009

Qu'est-ce qu'une fonction ?

Classiquement

Pas de notation uniforme/standard en mathématiques classiques :

$$f,\ f(x),\ \frac{\partial f}{\partial x},\ \int_0^1 f(x)dx,\ \sum_{i=1}^{10} x_i \ldots$$

Sauf : $x, y \longmapsto \sin x + \cos x$
Confusion classique : f' *vs* $f'(x)$
Pas de calcul possible !

λ-calcul (A. Church - 1930's)

Langage permettant d'écrire des *expressions* fonctionnelles (ou non) et muni d'un *calcul*

Langage

Les *termes* (ou *expressions*) du λ-calcul s'écrivent avec 'λ', '.', '(', ')' et des *variables* ($V = x, y, z...$). Nous les écrirons ici en utilisant la syntaxe du langage ML.

Définition

L'ensemble Λ des termes du λ-calcul (pur, i.e. sans type) est le plus petit ensemble vérifiant les propriétés suivantes :

- variable : $V \subset \Lambda$ (toute variable est un λ-terme) ;
- abstraction : $\forall x \in V, \forall A \in \Lambda$ alors $(fun\ x \rightarrow A) \in \Lambda$ (Church : « $\lambda.x\ a$ ») ;
- application : $\forall A, B \in \Lambda$ alors $(A\ B) \in \Lambda$ (« A appliqué à B »)

L'opération d'application est prioritaire sur l'abstraction.

Curryfication

Exemples : x, $x\ y$, $fun\ x \rightarrow x, (f\ x)\ y$, $fun\ x \rightarrow x\ x$, ...
Intuitivement, $fun\ x \rightarrow Y$ est la fonction qui à x (paramètre) associe Y (« corps » de la fonction) ; $F\ A$ est l'application de F (fonction) à A (argument).
On n'a besoin que de fonctions à **un** argument.
Comparer

$$f : (x, y) \mapsto x * y$$
$$f : x \mapsto \underbrace{(y \mapsto x * y)}_{f_x}$$

Donc $f(2, 3) = f_2(3) = 6$.
Notations :
Les parenthèses à gauche sont inutiles, $((A\ B)\ C)$ est identique à $(A\ B\ C)$
Un seul $fun \rightarrow$ pour des abstractions successives, $fun\ x\ y\ z \rightarrow A$ pour $fun\ x \rightarrow fun\ y \rightarrow fun\ z \rightarrow A$

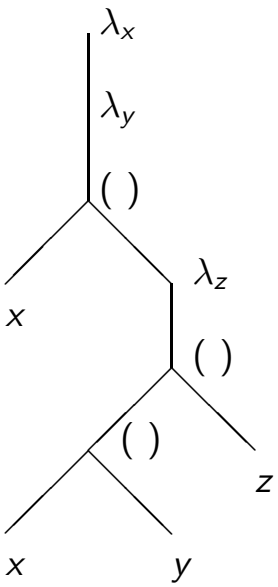
Occurrences libres et liées

Les *occurrences* d'une variable x dans un terme A sont les endroits où x apparaît dans A, excepté en position de paramètre.

$$\mathit{fun}\ x\ y\ \rightarrow\ \underline{x}\ (\mathit{fun}\ z\ \rightarrow\ (\underline{x}\ y)\ z)$$

Structure arborescente d'un λ-terme : une occurrence est notée par le *chemin* depuis la racine : $\lambda_x\lambda_y G$ et $\lambda_x\lambda_y D\lambda_z GG$

On dit qu'une occurrence de x est **libre** si λ_x n'apparaît pas dans le chemin : p.ex. dans $\mathit{fun}\ z\ \rightarrow\ (x\ y)\ z$.
On dit qu'elle est **liée** sinon.



Combinateur

Définition : un *combinateur* est un λ-terme sans variable libre.
Quelques fameux :

$\mathcal{I} = \mathit{fun}\ x\ \rightarrow\ x$	$\Delta = \mathit{fun}\ x\ \rightarrow\ x\ x$
$\mathcal{T} = \mathcal{K} = \mathit{fun}\ x\ y\ \rightarrow\ x$	$\Omega = \Delta\ \Delta$
$\mathcal{F} = \mathit{fun}\ x\ y\ \rightarrow\ y$	$\mathcal{Y} = \mathit{fun}\ f\ \rightarrow\ (\mathit{fun}\ u\ \rightarrow\ f\ (u\ u))\ (\mathit{fun}\ u\ \rightarrow\ f\ (u\ u))$
$\mathcal{S} = \mathit{fun}\ x\ y\ z\ \rightarrow\ (x\ z)\ (y\ z)$	$\mathcal{IF} = \mathit{fun}\ x\ y\ z\ \rightarrow\ x\ y\ z$

Notation de de Bruijn

L'occurrence d'une variable est notée par un entier qui est la profondeur relative de l'abstraction de cette variable.
Exemple :
$\mathit{fun}\ x \to \mathit{fun}\ y \to x\ (\mathit{fun}\ z \to (x\ y)\ z)$ est noté
$\mathit{fun} \to \mathit{fun} \to 2\ (\mathit{fun} \to (3\ 2)\ 1)$
De même

$$\begin{aligned} \mathit{fun}\ x \to \mathit{fun}\ x \to x &\equiv \mathit{fun} \to \mathit{fun} \to 1 \\ \mathit{fun}\ x\ y \to (\mathit{fun}\ x \to y)\ x &\equiv \mathit{fun} \to \mathit{fun} \to (\mathit{fun} \to 2)\ 2 \end{aligned}$$

Remarques

Pas besoin de nommer les variables, mais...
Pas de notation pour les variables libres
Peu intuitif (illisible !)

Calcul : α-équivalence

Intuitivement, $x \mapsto \cos x$ et $y \mapsto \cos y$ désignent la même fonction.
Ou encore, $(\mathit{fun}\ x \to x)$ et $(\mathit{fun}\ y \to y)$ sont les même λ-termes.
Les *noms* n'importent pas.

Définition

Deux termes sont *α-équivalents* ($=_\alpha$) s'ils ont la même représentation de de Bruijn.

$$\begin{aligned} \mathit{fun}\ x \to x\ (\mathit{fun}\ y\ z \to z\ y) &\equiv \\ \mathit{fun}\ z \to z\ (\mathit{fun}\ x\ z \to z\ x) &\equiv \end{aligned}$$

Propriété : l'α-équivalence est une relation d'équivalence.
Dorénavant, on écrira $=$ pour $=_\alpha$.

Calcul : Substitution

Définition

Soient A et B deux λ-termes, x une variable. On note $A[x \leftarrow B]$ le terme A où toutes les occurrences libres de x ont été remplacées par B.

$$\begin{aligned}(\mathit{fun}\ y \rightarrow x\ (\mathit{fun}\ z \rightarrow x\ y\ z))[x \leftarrow B] &= \\ (\mathit{fun}\ y \rightarrow x\ (\mathit{fun}\ z \rightarrow x\ y\ z))[z \leftarrow B] &= \\ (\mathit{fun}\ y \rightarrow x)[x \leftarrow y] &=\end{aligned}$$

Algorithme :

$$\begin{aligned} x[x \leftarrow X] &= X \\ y[x \leftarrow X] &= y \\ (A\ B)[x \leftarrow X] &= (A[x \leftarrow X]\ B[x \leftarrow X]) \\ (\mathit{fun}\ x \rightarrow A)[x \leftarrow X] &= \mathit{fun}\ x \rightarrow A \\ (\mathit{fun}\ y \rightarrow A)[x \leftarrow X] &= \mathit{fun}\ y \rightarrow A[x \leftarrow X] \end{aligned}$$

Calcul : β-réduction & β-équivalence

Intuition : application d'une fonction à un argument, i.e. *la règle de calcul* fondamentale.

Définition *β-réduction* ($\Rightarrow_\beta$)

$$(\mathit{fun}\ x \rightarrow A)\ B \Rightarrow_\beta A'[x \leftarrow B]$$

où $A' =_\alpha A$ et A' ne contient pas de variables libres de B.

Un terme de la forme $(\mathit{fun}\ x \rightarrow A)\ B$ est appelé *radical* ou *redex*.
On étend $\Rightarrow_\beta$ à la structure des λ-termes : si $M \Rightarrow_\beta M'$ alors

$$\begin{aligned}(\mathit{fun}\ x \rightarrow M) &\Rightarrow_\beta (\mathit{fun}\ x \rightarrow M') \\ (M\ N) &\Rightarrow_\beta (M'\ N) \\ (N\ M) &\Rightarrow_\beta (N\ M')\end{aligned}$$

Par clôture symétrique, réflexive et transitive de $\Rightarrow_\beta$, on engendre la relation d'équivalence $=_\beta$.

β-équivalence

On a $A =_\beta B$ s'il existe $A_0, \ldots, A_n$ (avec $n \geq 0$) tel que

- $A_0 = A$ et $A_n = B$ (éventuellement $n = 0$, c'est-à-dire $A = B$) ;
- $\forall i, 0 \leq i \leq n-1$, $A_i \Rightarrow_\beta A_{i+1}$ ou $A_{i+1} \Rightarrow_\beta A_i$.

Exemples :

$$\begin{aligned}
(\mathit{fun}\ x \to x)\ (\mathit{fun}\ y \to y) &\Rightarrow_\beta \\
(\mathit{fun}\ x \to x\ x)\ (\mathit{fun}\ y \to y) &\Rightarrow_\beta \\
\mathit{fun}\ y \to ((\mathit{fun}\ x\ y \to x)\ y) &\Rightarrow_\beta \\
(\mathit{fun}\ x\ y \to x\ (y\ x))\ (\mathit{fun}\ u \to u\ u)\ (\mathit{fun}\ v\ w \to v) &\Rightarrow_\beta \\
(\mathit{fun}\ u \to u\ u)\ (\mathit{fun}\ v \to v\ v) &\Rightarrow_\beta
\end{aligned}$$

Dorénavant, on écrira $=$ pour $=_{\alpha\beta}$.

η-équivalence

Définition : *Extensionnalité*

Deux fonctions f et g sont égales ssi pour tout x, on a $f\ x = g\ x$.

Exemple : $x \mapsto x + x$ et $x \mapsto 2x$ sont extensionnellement égales (mais n'ont pas le même procédé/algorithme de calcul).
Pour les λ-termes : $A =_{\text{ext}} B$ si $\forall X, (A\ X) = (B\ X)$.
Considérons A et $(\mathit{fun}\ x \to A\ x)$ où x n'est pas libre dans A. Pour tout X, on a $(\mathit{fun}\ x \to A\ x)\ X =_\beta (A\ X)$ donc $A =_{\text{ext}} (\mathit{fun}\ x \to A\ x)$.

Définition : η-équivalence

Pour tout A, $(\mathit{fun}\ x \to A\ x) \Rightarrow_\eta A$ si x n'est pas libre dans A. On note $=_\eta$ la relation d'équivalence associée.

En OCaml : `let f = fun x -> g x ; ;` $\equiv$ `let f = g ; ;`
Dorénavant, on écrira $=$ pour $=_{\alpha\beta\eta}$.

Forme normale

Définition

On dit qu'un terme est sous *forme normale* s'il ne contient pas de radical. On dit qu'un terme M est *normalisable* s'il existe un terme N sous forme normale tel que $M =_{\alpha\beta\eta} N$.

Exemple : $(\mathit{fun}\ u \rightarrow u\ u)\ (\mathit{fun}\ v \rightarrow v\ v)$ n'est pas normalisable.

Définition

Soient $\Rightarrow$ une relation binaire, $\Rightarrow^*$ sa clôture transitive. $\Rightarrow$ est *confluente* ssi

$$\forall x, x_1, x_2 \text{ tels que } x \Rightarrow^* x_1 \text{ et } x \Rightarrow^* x_2$$

implique

$$\exists x' \text{ tel que } x_1 \Rightarrow^* x' \text{ et } x_2 \Rightarrow^* x'$$

Forme normale (cont)

Propriété

Soient $\Rightarrow$ une relation binaire, $\Rightarrow^*$ sa clôture transitive, $\equiv$ la relation d'équivalence associée. Si $\Rightarrow$ est confluente, alors

$$x \equiv y \text{ si et seulement si } \exists z \text{ tel que } x \Rightarrow^* z \text{ et } y \Rightarrow^* z$$

Théorème de Church-Rosser

Théorème de Church-Rosser

La β-réduction $\Rightarrow_\beta$ est confluente.

Corollaire : Pour tout A, si A est normalisable alors il existe $A*$ sous forme normale *unique* tel que $A \Rightarrow_\beta^* A*$.
Conséquence : La *stratégie* de réduction a *peu* d'importance : si on obtient une forme normale, c'est la bonne.
Attention : $(\mathit{fun}\ x\ y\ \rightarrow\ x)\ \mathcal{I}\ \Omega \Rightarrow_\beta$???
Théorème : la stratégie de réduction gauche est normalisante : si un terme est normalisable, on peut obtenir sa forme normale en réduisant itérativement le radical le plus à gauche.
Remarque : c'est la stratégie d'un langage fonctionnel *paresseux* (p.ex. Haskell).

Programmer en λ-calcul

- Booléens :

$$\texttt{True} = \mathcal{T} = \mathit{fun}\ x\ y\ \rightarrow\ x$$
$$\texttt{False} = \mathcal{F} = \mathit{fun}\ x\ y\ \rightarrow\ y$$

- Conditionnelle : $\texttt{If\_Then\_Else} = \mathcal{IF} = \mathit{fun}\ p\ y\ z\ \rightarrow\ p\ y\ z$
- Couples :

$$\texttt{Couple} = \mathcal{C} = \mathit{fun}\ x\ y\ z\ \rightarrow\ z\ x\ y$$
$$\texttt{Fst} = \mathit{fun}\ x\ \rightarrow\ x\ \mathcal{T}$$
$$\texttt{Snd} = \mathit{fun}\ x\ \rightarrow\ x\ \mathcal{F}$$

- Entiers :

$$0 = \mathcal{I} = \mathit{fun}\ x\ \rightarrow\ x$$
$$\texttt{Succ} = \mathcal{C}\ \mathcal{F}$$
$$\texttt{Pred} = \mathit{fun}\ x\ \rightarrow\ x\ \mathcal{F} \quad (\text{Rq. } \texttt{Pred}\ 0 = \mathcal{F})$$
$$\texttt{Eq\_zero} = \mathcal{Z} = \mathit{fun}\ x\ \rightarrow\ x\ \mathcal{T}$$

λ-calcul typé

Définition

Soit $A = \{\text{int}, \text{bool}, \ldots\}$ ensemble de symboles ; T l'ensemble des types est défini par

- $A \subset T$: tout élément de A est un type (*atomique*) ;
- pour tout $\alpha, \beta \in T$, $\alpha \to \beta$ est un type.

Intuition :

- un type est un domaine ;
- $\alpha \to \beta$ est le domaine des fonctions de α dans β.

Notation : $\to$ est associative à droite.

$$\alpha_1 \to (\alpha_2 \to (\ldots(\alpha_n \to \beta)\ldots)) = \alpha_1 \to \alpha_2 \to \cdots \to \beta$$

Typage

Le *typage* consiste à associer un type à un λ-terme.
On va écrire des *jugements* : $\Gamma \vdash t$ « t est typé dans le contexte Γ ».
Un *contexte* est un ensemble de paires variable-type $(x : \tau)$
On écrira $\vdash x : \tau$ si le contexte est vide.

Système de *règles*

$$\frac{\Gamma \vdash A : \alpha \to \beta \qquad \Gamma \vdash B : \alpha}{\Gamma \vdash (A\ B) : \beta}\text{App} \qquad \frac{\Gamma, x : \alpha \vdash A : \beta}{\Gamma \vdash (\mathit{fun}\ x \to A) : \alpha \to \beta}\text{Abs}$$

$$\frac{}{\Gamma \vdash x : \alpha}\text{Var, si } (x : \alpha) \in \Gamma$$

Exemple : $A = \{i, b\}$, $\vdash \ \mathit{fun}\ x \ \to \ x\ (\mathit{fun}\ y \ \to \ y) : ((b \to b) \to i) \to i$

$$\dfrac{\dfrac{}{\{x : ((b \to b) \to i)\} \ \vdash \ x : ((b \to b) \to i)}\text{Var} \quad \dfrac{\dfrac{}{\{x : \ldots, y : b\} \ \vdash \ y : b}\text{Var}}{\{x : \ldots\} \ \vdash \ (\mathit{fun}\ y \ \to \ y) : b \to b}}{\dfrac{\{x : ((b \to b) \to i)\} \ \vdash \ (x\ (\mathit{fun}\ y \ \to \ y)) : i}{\dfrac{}{\vdash \ (\mathit{fun}\ x \ \to \ x\ (\mathit{fun}\ y \ \to \ y)) : ((b \to b) \to i) \to i}\text{Abs}}\text{App}}$$

On peut aussi *inférer* un type :

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \alpha' \ \vdash \ x : \beta'}\alpha = \beta'}{x : \alpha \ \vdash \ \mathit{fun}\ y \ \to \ x : \beta}\beta = \alpha' \to \beta'}{\vdash \ \mathit{fun}\ x \ \to \ \mathit{fun}\ y \ \to \ x : \tau}\tau = \alpha \to \beta$$

On conclut : $\tau = \alpha \to \beta = \beta' \to (\alpha' \to \beta')$.
D'où : $\vdash \ \mathit{fun}\ x\ y \ \to \ x : \mathit{int} \to (\mathit{bool} \to \mathit{int})$ ou
$\vdash \ \mathit{fun}\ x\ y \ \to \ x : \mathit{bool} \to (\mathit{bool} \to \mathit{bool})$

Propriétés

La β-réduction conserve le typage

$$((\underbrace{\mathit{fun}\ \underbrace{x}_{\alpha} \ \to \ \underbrace{A}_{\beta}}_{\alpha \to \beta}\ \underbrace{B}_{\alpha}) \ \Rightarrow_\beta \ A[\ \underbrace{x \to B}\ ]$$

même type pour x et B

Prop. : Toute β-réduction d'un λ-terme typé termine.
Cor. : Tout λ-terme typé possède une forme normale.
Cor. : Tout λ-terme non normalisable n'est pas typable (la réciproque est fausse : $\mathit{fun}\ x \ \to \ (x\ x)$).
Un aperçu du paysage normalisable ...

ML

λ-calcul typé + définitions récursives

Le système Ocaml

Le langage est **fortement** typé : pas de coercion (*casting*).
Le système **infère** les types : l'utilisateur n'écrit pas les types de ses expressions.

```
#1729;;
- : int = 1729

#(+);;
- : int -> int -> int = <fun>

#let succ = fun x -> x + 1;;
val succ : int -> int = <fun>

#1 + 1.5;;
Characters 4-7:
  1 + 1.5;;
      ^^^
This expression has type float but is here used with type int
```


Le typage est **polymorphe** : l'inférence est la plus générale possible.

```
#fun x -> x;;
- : 'a -> 'a = <fun>

#fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

#(=);;
- : 'a -> 'a -> bool = <fun>
```

Le `let` permet de **généraliser**

```
#let id = fun x -> x in (id 1, id 2.0);;
- : int * float = (1, 2.)

#(fun id -> (id 1, id 2.0)) (fun x -> x);;
Characters 21-24:
  (fun id -> (id 1, id 2.0)) (fun x -> x);;
                      ^^^
This expression has type float but is here used with type int
```