Lambda calcul

Alexandre Boisseau
boisseau@dptmaths.ens-cachan.fr

Le lambda-calcul a été développé en 1930 par Church dans un but double :

- d'une part décrire un formalisme de représentation des fonctions mathématiques,
- d'autre part lui adjoindre quelques principes de logique afin de développer une théorie qui puisse faire office de fondements aux mathématiques.

Actuellement, les objets que manipule le mathématicien ne sont que des ensembles. Par exemple un nombre réel est un ensemble de rationnels (précisément, l'ensemble des rationnels qui le minorent), un nombre rationnel est lui-même un couple de deux entiers, or le couple (a,b) est défini comme étant l'ensemble $\{a,\{a,b\}\}$. Enfin un entier est aussi un ensemble, 0 étant par définition l'ensemble vide et si $0,\ldots,n$ sont définis, on pose $n+1=\{0,\ldots,n\}$. Ces ensembles sont les objets de base et pour les manipuler le mathématicien dispose des ce qu'on appelle communément les «axiomes de la théorie des ensembles». L'alternative recherchée était donc de fonder les mathématiques non plus sur les ensembles mais sur les fonctions. On verra en effet plus loin comment on peut représenter les objets mathématiques usuels (couples, entiers, etc.) par des termes du lambda-calcul.

Kleene et Rosser ont montré un 1935 que le système était inconsistant et le second but n'a par conséquent pas pu être atteint. Par contre il s'est avéré que les fonctions obtenues de cette manière décrivaient exactement la classe des fonctions «effectivement calculables» (*i.e.*calculables par une machine de Turing). Ce que nous allons faire maintenant, c'est tout d'abord expliquer ce que sont les termes du lambda-calcul puis ensuite nous verrons comment les utiliser pour programmer de manière effective.

Construction des termes

On dispose à la base d'un ensemble X (infini mais dénombrable) dont les éléments sont appelés des variables. Ces variables sont des termes du lambda-calcul :

- $X \subset \lambda[X]$.

Si u et v sont deux termes, (uv) est un terme qui représente l'application du terme u au terme v :

- $u,v \in \lambda[X] \Rightarrow (uv) \in \lambda[X]$.

Si u est un terme et $x \in X$, $\lambda x.u$ est un terme qui représente la fonction qui à x associe l'objet décrit par la formule u :

- $x \in X$ et $u \in \lambda[X] \Rightarrow \lambda x.u \in \lambda[X]$.

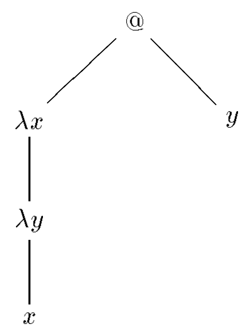
Il n'y a pas d'autres termes dans $\lambda[X]$.

Pour l'instant, les termes ne sont ni plus ni moins que des mots (*i.e.*des suites de symboles) écrits sur l'alphabet $X \cup \{(,),.,\lambda\}$.

La syntaxe

Substitutions dans les termes

Tout d'abord, les termes peuvent être représentés par des arbres. Par exemple au terme $(\lambda x.\lambda y.x)y$ on associe l'arbre suivant (dans lequel le symbole @ désigne l'opération binaire d'application) :



On dit qu'une occurence d'une variable x dans un terme u est libre si en remontant à la racine de l'arbre à partir de cet x on ne rencontre aucun λx. La notation $u[v/x]$ désigne alors le terme u dans lequel les occurences libres de x ont été remplacées par v. Il faut toutefois prendre garde au phénomène de "capture des variables" : considérons les termes $u=\lambda x.(xy)$ et $v=x$, on ne peut pas supposer que $u[v/y]=\lambda x.(xx)$ car, en utilisant la convention usuelle de renommage des variables (que l'on verra plus loin) u est équivalent à $\lambda z.(zy)$ et donc $u[v/y]$ doit être équivalent à $\lambda z.(zx)$. Cette convention de renommage des variables s'appelle alpha-conversion. En pratique, on l'applique sans la mentionner. Lors des substitutions, on s'arrangera à appliquer de telles conversions dans le but d'éviter les captures de variables.

Beta-réduction

Maintenant on peut appliquer la notion de substitution pour décrire comment opèrent les fonctions :

si u et v sont des termes et $x \in X$, alors les termes $(\lambda x.u)v$ et $u[v/x]$ sont équivalents, ce que l'on note :
- $(\lambda x.u)v \equiv u[v/x]$.

On prolonge cette relation de la manière suivante : si les termes u et u' sont équivalents, alors $(uv) \equiv (u'v)$,$(vu) \equiv (vu')$ et $\lambda x.u \equiv \lambda x.u'$ quels que soient $v \in \lambda[X]$ et $x \in X$. C'est en fait la clôture symétrique, réflexive et transitive de cette relation que l'on considèrera.

Quelques conventions

Une loi binaire, par exemple l'addition des entiers, peut être vue :
- d'une part comme l'application (à deux arguments) :
$$f: \begin{array}{ccc} \mathbf{N}^2 & \to & \mathbf{N} \\ (x,y) & \mapsto & x+y, \end{array}$$
- d'autre part comme l'application (à un argument) :
$$g: \begin{array}{ccccc} \mathbf{N} & \to & \mathbf{N}^{\mathbf{N}} & & \\ x & \mapsto & \mathbf{N} & \to & \mathbf{N} \\ & & y & \mapsto & x+y \end{array}$$

le passage de f à g s'effectuant par une transformation évidente. La première forme est appelée forme curryfiée et la seconde forme décurryfiée. Comme on ne dispose pas (pas encore) de couples en lambda-calcul, c'est plutôt cette seconde forme que l'on va utiliser. Pour $x,y \in \mathbf{N}$, $x+y$ s'écrit alors $(g(x))(y)$ ce que l'on note plutôt $(gx)y$ ou encore gxy, l'application étant par convention associative à gauche : ce sera toujours le cas en lambda-calcul.

On dispose donc d'un langage $\lambda[X]$ muni d'une relation d'équivalence $\equiv$. Un programme (de même que ses données) seront des éléments de $\lambda[X]$ et exécuter le programme u sur les entrées $v_1, \ldots ,v_n$ consistera à (essayer de) trouver une forme normale (*i.e.*la plus simple possible) pour le terme $uv_1 \ldots v_n$. Avant de voir comment calculer de telles formes, on va construire quelques termes qui représentent la base des langages de programmation usuels.

La programmation

On va ici définir des termes particuliers qui simplifient la construction de programmes en lambda-calcul. Pour que ces termes puissent être ensuite utilisés à bon escient, on doit préciser quelles sont les propriétés qu'ils vérifient. Par exemple la terme «identité» défini par :

$$\mathtt{I} = \lambda x.x$$

possède la propriété suivante : quel que soit le terme $u \in \lambda[X]$, $\mathtt{I}u$ est équivalent à u. On notera une telle définition associée à l'énoncé d'une propriété sous la forme suivante :

$$\begin{array}{l} \mathtt{I} = \lambda x.x \\ [\![\forall u \in \lambda[X].\ \mathtt{I}u \equiv u]\!]. \end{array}$$

Les entiers

De manière générale, pour coder un objet mathématique en lambda-calcul, on se pose plutôt la question «à quoi ca sert ?» plutôt que «qu'est-ce que c'est ?» Par exemple les entiers servent (entre autres) à itérer un certain nombre de fois une fonction à partir d'une valeur initiale. 3 prend une fonction f et un argument x et renvoie la valeur $f(f(fx))$, donc le terme qui lui correspond est :

$$\overline{3} = \lambda f.\lambda x.f(f(fx)).$$

Plus généralement, un entier n est représenté par le terme :

$$\begin{array}{l} \overline{n} = \lambda f.\lambda x.f(\cdots(fx)\cdots) \\ [\![\forall u,v \in \lambda[X].\ \forall n \in \mathbf{N}.\ \overline{n}\ uv \equiv u(\cdots(uv)\cdots)]\!] \end{array}$$

(avec n occurences de f), en particulier $\overline{0} = \lambda f.\lambda x.x$. L'entier $\overline{n}$ itère n fois, son successeur doit itérer $n+1$ fois, il est par conséquent équivalent à :

$$\lambda f.\lambda x.f(\overline{n}fx)$$

d'où la forme fonctionnelle du successeur :

$$\begin{array}{l} \mathtt{succ} = \lambda n.\lambda f.\lambda x.f(nfx) \\ [\![\forall n \in \mathbf{N}.\ \mathtt{succ}\ \overline{n} \equiv \overline{n+1}]\!]. \end{array}$$

On obtient maintenant de manière immédiate les fonctions usuelles, par exemple $n+m$ se calcule en itérant n fois le successeur à partir de m, $\overline{n+m}$ doit donc être équivalent à $\overline{n}$ $\mathtt{succ}$ $\overline{m}$ ce qui nous donne la forme fonctionnelle de l'addition :

$$\begin{array}{l} \mathtt{plus} = \lambda n.\lambda m.n\ \mathtt{succ}\ m \\ [\![\forall n,m \in \mathbf{N}.\ \mathtt{plus}\ \overline{n}\ \overline{m} \equiv \overline{n+m}]\!]. \end{array}$$

En appliquant la même méthode, on réalise facilement les opérations de multiplication (`mul`) et d'exponentiation (`exp`).

Les booléens

Les booléens sont essentiellement employés en infomatique dans des expressions conditionnelles du style «si t alors u sinon w». Le booléen t peut alors être vu comme une fonction à deux arguments, *vrai* renvoyant son premier argument et *faux* renvoyant le second, d'où les définitions :

$$\texttt{vrai} = \lambda x.\lambda y.x \text{ et } \texttt{faux} = \lambda x.\lambda y.y.$$

On peut alors définir une forme fonctionnelle de l'expression conditionnelle en posant :

$$\begin{array}{l} \texttt{cond} = \lambda t.\lambda x.\lambda y.(txy) \\ \left[\!\!\left[\begin{array}{l} \forall u,v \in \lambda[X].\ \texttt{cond vrai}\ uv \equiv u \\ \forall u,v \in \lambda[X].\ \texttt{cond faux}\ uv \equiv v \end{array} \right]\!\!\right] \end{array}$$

On peut aussi réaliser les opérations booléennes usuelles au moyen de ces termes :

$$\begin{array}{l} \texttt{et} = \lambda x.\lambda y.\texttt{cond}\ xy\ \texttt{faux} \\ \left[\!\!\left[\begin{array}{l} \texttt{et vrai vrai} \equiv \texttt{vrai} \\ \texttt{et vrai faux} \equiv \texttt{faux} \\ \texttt{et faux vrai} \equiv \texttt{faux} \\ \texttt{et faux faux} \equiv \texttt{faux} \end{array} \right]\!\!\right] \end{array}$$

$$\begin{array}{l} \texttt{non} = \lambda x.\texttt{cond}\ x\texttt{faux vrai} \\ \left[\!\!\left[\begin{array}{l} \texttt{non vrai} \equiv \texttt{faux} \\ \texttt{non faux} \equiv \texttt{vrai} \end{array} \right]\!\!\right] \\ \ldots \end{array}$$

Profitons-en pour écrire une fonction testant si un entier est nul. On cherche en fait deux termes u et v tels que pour tout entier n on ait :

$$\begin{array}{l} \overline{n}uv \equiv \texttt{vrai} \text{ si } n = 0, \\ \overline{n}uv \equiv \texttt{faux} \text{ sinon,} \end{array}$$

or $\overline{0}uv \equiv v$, ce qui impose $v \equiv \texttt{vrai}$. Par ailleurs $\overline{1}uv \equiv (uv)$ et on doit donc avoir $(uv) \equiv \texttt{faux}$ ce qui motive pour u le choix du terme $u = \lambda x.\texttt{faux}$. On pose alors :

$$\begin{array}{l} \texttt{nul} = \lambda n.n(\lambda x.\texttt{faux})\texttt{vrai} \\ \left[\!\!\left[\begin{array}{l} \texttt{nul}\ \overline{0} \equiv \texttt{vrai} \\ \forall n \in \mathbf{N} \setminus \{0\}.\ \texttt{nul}\ \overline{n} \equiv \texttt{faux} \end{array} \right]\!\!\right] \end{array}$$

Couples et projections

Mathématiquement, un élément $(e,f) \in E \times F$ peut être vu comme une application d'un ensemble à deux éléments (par exemple $\{1,2\}$) dans $E \cup F$:

$$\begin{array}{rrcl} c: & \{1,2\} & \to & E \cup F \\ & 1 & \mapsto & e \\ & 2 & \mapsto & f \end{array}$$

La première projection consiste à appliquer c en 1 et la seconde en 2. On va plutôt prendre $\{\texttt{vrai},\texttt{faux}\}$ comme ensemble à deux éléments. Si u et v sont deux termes, le terme suivant est une représentation du couple (u,v) :

$$\lambda t.\texttt{cond}\ tuv$$

En effet, appliqué en $\texttt{vrai}$, il renvoie u et en $\texttt{faux}$, il renvoie v. Par conséquent le terme suivant :

$$\texttt{cons} = \lambda x.\lambda y.\lambda t.\texttt{cond}\ txy$$

prend en arguments deux termes et renvoie le couple formé de ces deux termes. Les projections quant à elles s'écrivent :

$$\begin{array}{l} \texttt{p1} = \lambda x.x\ \texttt{vrai} \\ [\![\forall u,v \in \lambda[X].\ \texttt{p1}(\texttt{cons}\ uv) \equiv u]\!] \end{array}$$

$$\begin{array}{l} \texttt{p2} = \lambda x.x\ \texttt{faux} \\ [\![\forall u,v \in \lambda[X].\ \texttt{p2}(\texttt{cons}\ uv) \equiv v]\!] \end{array}$$

Comme premier exemple de programmation, voyons comment réaliser la fonction prédécesseur, notée $\texttt{pred}$ telle que $\texttt{pred}\ \overline{n+1} \equiv \overline{n}$ pour tout entier n. Comme $\overline{n+1}$ itère $n+1$ fois une fonction, il faut trouver une fonction qui lors de sa première application «ne fait rien» et lors des suivantes prend le successeur :

$$\begin{array}{l} \texttt{aux} = \lambda x.\texttt{cond}(\texttt{p1}\ x) \\ \qquad (\texttt{cons vrai}(\texttt{succ}(\texttt{p2}\ x))) \\ \qquad (\texttt{cons vrai}(\texttt{p2}\ x)) \end{array}$$

appliquée en $\texttt{cons faux}\ v$ (*i.e.*le couple $(\texttt{faux},v)$) elle renvoie $(\texttt{vrai},v)$ (elle ne fait donc rien sur la seconde composante) et appliquée en $(\texttt{vrai},v)$, elle renvoie $(\texttt{vrai},\texttt{succ}\ v)$. La construction du prédécesseur est alors immédiate en utilisant cette fonction auxiliaire :

$$\begin{array}{l} \texttt{pred} = \lambda n.\texttt{p2}(n\ \texttt{aux}(\texttt{cons faux}\ \overline{0})) \\ [\![\forall n \in \mathbf{N}.\ \texttt{pred}\ \overline{n+1} \equiv \overline{n}]\!]. \end{array}$$

Remarquons qu'on ne donne aucune information sur le terme $\texttt{pred}\ \overline{0}$.

Récursivité

C'est la dernière construction à effectuer pour disposer d'un langage de programmation suffisament puissant. Etudions tout d'abord les points fixes : un point fixe pour un terme u est un terme v tel que $uv \equiv v$. Le résultat fondamental est que non seulement tous les termes possèdent un point fixe mais qu'en plus il existe un terme $\texttt{fix}$ tel que pour tout terme u, $\texttt{fix}\ u$ soit un point fixe pour u (*i.e.*: $u(\texttt{fix}\ u) \equiv \texttt{fix}\ u$). On peut trouver de nombreux termes réalisant un tel calcul de point fixe, le plus célèbre étant sans doute :

$$\texttt{fix} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

(combinateur de Curie). Pour tout $u \in \lambda[X]$ on a alors :

$$\begin{array}{rcl} \texttt{fix}\ u & \equiv & (\lambda x.u(xx))(\lambda x.u(xx)) \\ & \equiv & u((\lambda x.u(xx))(\lambda x.u(xx))) \\ & \equiv & u(\texttt{fix}\ u). \end{array}$$

La propriété que satisfait $\texttt{fix}$ est donc :

$$[\![\forall u \in \lambda[X]\ .u(\texttt{fix}\ u) \equiv u]\!].$$

Pour voir maintenant comment construire un opérateur de récursivité, étudions la fonction factorielle définie par :

$$\begin{cases} \text{fact}(0) = 1 \\ \text{fact}(n) = n \times (\text{fact}(n-1)) \text{ si } n \geqslant 1. \end{cases}$$

Si f est une fonction de $\mathbf{N}$ dans $\mathbf{N}$, notons $\hat{f}$ la fonction définie par :

$$\begin{array}{rrcl} \hat{f}: & \mathbf{N} & \to & \mathbf{N} \\ & 0 & \mapsto & 1 \\ & n & \mapsto & n \times f(n-1) \text{ si } n \geqslant 1. \end{array}$$

Soit Γ l'application de $\mathbf{N}^{\mathbf{N}}$ dans lui-même qui à f associe $\hat{f}$, fact est alors un point fixe de Γ (en fait, tout point fixe de Γ réalise la factorielle). Il suffit donc de trouver un terme u qui code la fonction Γ et $\texttt{fix}\ u$ sera la factorielle. Pour le terme u :

$$u = \lambda f.\lambda n.\texttt{cond}(\texttt{nul}\ n)\overline{1}(\texttt{mul}\ n(f(\texttt{pred}\ n)))$$

convient.

Essayons maintenant de généraliser : une définition par récurrence d'une fonction $f : \mathbf{N} \to E$ s'écrit :

$$\begin{cases} f(0) = x_0 \\ f(n) = g(n,f(n-1)) \text{ si } n \geqslant 1 \end{cases}$$

avec $x_0 \in E$ et $g : \mathbf{N}^2 \to \mathbf{N}$. La fonction Γ associée est alors :

$$\begin{array}{rrcl} \Gamma: & \mathbf{N}^{\mathbf{N}} & \to & \mathbf{N}^{\mathbf{N}} \\ & f & \mapsto & \begin{array}{rcl} \mathbf{N} & \to & \mathbf{N} \\ 0 & \mapsto & x_0 \\ n \geqslant 1 & \mapsto & g(n,f(n-1)) \end{array} \end{array}$$

Le codage de cette fonction Γ est réalisé par le terme suivant :

$$\lambda f.\lambda n.\texttt{cond}(\texttt{nul}\ n)x_0(gn(f(\texttt{pred}\ n)))$$

d'où l'on déduit la forme fonctionnelle de Γ (prenant en argument x_0 et g) :

$$\begin{array}{l} \texttt{gamma} = \\ \quad \lambda x.\lambda g.\lambda f.\lambda n. \\ \quad\quad \texttt{cond}(\texttt{nul}\ n) \\ \quad\quad\quad x \\ \quad\quad\quad (gn(f(\texttt{pred}\ n))). \end{array}$$

Le terme suivant est donc un opérateur de récursivité :

$$\texttt{rec} = \lambda x.\lambda g.\texttt{fix}(\texttt{gamma}\ xg)$$

$$\left[\!\!\left[\begin{array}{l} \texttt{rec}\ uv\overline{0} \equiv u \\ \forall n \in \mathbf{N}.\ \texttt{rec}\ \overline{n+1}\ uv \equiv v\ \overline{n+1}\ (\texttt{rec}\ uv\overline{n}) \end{array} \right]\!\!\right]$$

appliqué à des termes x_0 et g qui définissent une fonction f, $\texttt{rec}\ x_0 g$ définit un terme codant f.

Un exemple : reconnaissance du langage de Dick

Nous allons maintenant donner un véritable exemple de programme réalisé en lambda-calcul : on va chercher un terme $u \in \lambda[X]$ tel que $uw \equiv \texttt{vrai}$ si w est un terme représentant un mot bien parenthésé et $uw \equiv \texttt{faux}$ si w est un terme représentant un mot mais qui n'est pas bien parenthésé. Avant d'en arriver à cette construction, réexpliquons brièvement ce que sont les mots et le langage de Dick.

Mots sur un alphabet fini ou dénombrable

Soit A un ensemble (fini ou dénombrable) qu'on appellera alphabet. L'ensemble des mots construits sur A (noté A^*) est tout simplement l'ensemble les suites finies (eventuellement vides) d'éléments de A. On peut munir A d'une structure de monoïde avec la loi de composition interne :

$$(a_1, \cdots, a_p).(b_1, \cdots, b_q) = (a_1, \cdots, a_p, b_1, \cdots, b_q).$$

Dans la suite, $(a_1, \cdots, a_p)$ sera noté plus simplement $a_1 \cdots a_p$. On considère ici l'alphabet à deux lettres $A = \{[,]\}$ et on définit $p : A^* \to \mathbf{Z}$ comme étant l'unique morphisme de monoïdes tel que $p([) = +1$ et $p(]) = -1$. On dit qu'un mot $w = a_1 \cdots a_n \in A^*$ est un mot de Dick (ou encore «bien parenthésé») si $p(a_1 \cdots a_i) \geqslant 0$ pour tout $i \in [\![1,n]\!]$ et $p(w) = 0$ (le lecteur pourra vérifier qu'il s'agit bien là des mots correspondant à la notion usuelle de «bon parenthésage»). Comme on a introduit de nouveaux objets mathématiques (entre autres les mots et les entiers relatifs), on doit tout d'abord chercher comment les représenter en lambda-calcul avant de passer à la phase de reconnaissance. Afin d'alléger les notations, on fera l'abus d'écriture suivant : si $u,v \in \lambda[X]$, la notation (u,v) désignera le terme $\texttt{cons}\ uv$.

Les entiers relatifs

Nous n'aurons besoin que de deux opérations sur les entiers relatifs : un calcul de successeur et un calcul de prédécesseur. Il faudra aussi prévoir un terme pour tester la positivité et un pour la nullité. De manière générale, pour $n \in \mathbf{N}$ on notera $\widetilde{n} = (\mathtt{vrai},\overline{n})$ et $\widetilde{-n} = (\mathtt{faux},\overline{n})$. Remarquons qu'on a défini $\widetilde{0}$ de deux manières différentes. Auparavant on avait associé à chaque entier (naturel) un terme et ceci de manière unique. Ici chaque entier (relatif) possède un ensemble de représentants. Maintenant, lorsqu'on écrira $u = \widetilde{n}$ (ou $u \equiv \widetilde{n}$), cela voudra dire «u est un représentant de $n \in \mathbf{Z}$» (ou «u est équivalent à un représentant de $n \in \mathbf{Z}$») et une propriété du type $u\widetilde{n} \equiv \widetilde{m}$ devra se lire "si t est un représentant de $n \in \mathbf{Z}$, alors ut est équivalent à un représentant de $m \in \mathbf{Z}$". Les mêmes conventions auront cours lorsque, plus loin, on définira ce qu'est un représentant d'un mot $w \in A^*$. Le calcul du successeur se fait alors sans difficulté. Pour plus de commodité, on définit également un terme qui prend l'opposé :

$$
\begin{array}{l}
\mathtt{succz} = \\
\quad \lambda n. \\
\qquad \mathtt{cond}(\mathtt{p1}\ n) \\
\qquad\quad (\mathtt{vrai},(\mathtt{succ}\ (\mathtt{p2}\ n))) \\
\qquad\quad (\mathtt{cond}(\mathtt{nul}(\mathtt{p2}\ n)) \\
\qquad\qquad (\mathtt{vrai},\overline{1}) \\
\qquad\qquad (\mathtt{faux},(\mathtt{pred}(\mathtt{p2}\ n)))) \\
[\![\forall n \in \mathbf{Z}.\ \mathtt{succz}\ \widetilde{n} \equiv \widetilde{n+1}]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathtt{oppz} = \\
\quad \lambda n. \\
\qquad ((\mathtt{non}(\mathtt{p1}\ n)),(\mathtt{p2}\ n)) \\
[\![\forall n \in \mathbf{Z}.\ \mathtt{oppz}\ \widetilde{n} \equiv \widetilde{-n}]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathtt{predz} = \\
\quad \lambda n. \\
\qquad \mathtt{oppz}(\mathtt{succz}(\mathtt{oppz}\ n)) \\
[\![\forall n \in \mathbf{Z}.\ \mathtt{predz}\ \widetilde{n} \equiv \widetilde{n-1}]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathtt{positz} = \\
\quad \lambda n. \\
\qquad \mathtt{ou}(\mathtt{p1}\ n)(\mathtt{nul}(\mathtt{p2}\ n)) \\
\left[\!\!\left[\begin{array}{l} \forall n \in \mathbf{N}.\ \mathtt{positz}\ \widetilde{n} \equiv \mathtt{vrai} \\ \forall n \in \mathbf{N} \setminus \{0\}.\ \mathtt{positz}\ \widetilde{-n} \equiv \mathtt{faux} \end{array} \right]\!\!\right]
\end{array}
$$

Comparaison et moins

Le plus simple pour comparer deux entiers (naturels) n et m consiste à convertir m en entier relatif, iérer n fois le prédécesseur et vérifier si le résultat est positif. Cela s'écrit :

$$
\begin{array}{l}
\mathtt{inf} = \\
\quad \lambda n.\lambda m. \\
\qquad \mathtt{positz}(n\ \mathtt{predz}(\mathtt{vrai},m)) \\
\left[\!\!\left[\begin{array}{l} \forall n,m \in \mathbf{N}.\ n \leqslant m \Rightarrow \mathtt{inf}\ \overline{n}\ \overline{m} \equiv \mathtt{vrai} \\ \forall n,m \in \mathbf{N}.\ n > m \Rightarrow \mathtt{inf}\ \overline{n}\ \overline{m} \equiv \mathtt{faux} \end{array} \right]\!\!\right]
\end{array}
$$

On peut aussi construire la fonction **moins** qui retire m à n (à condition que n soit plus grand que m) :

$$
\begin{array}{l}
\mathtt{moins} = \\
\quad \lambda n.\lambda m. \\
\qquad m\ \mathtt{pred}\ n \\
[\![\forall n,m \in \mathbf{N}.\ \mathtt{moins}\ \overline{n+m}\ \overline{m} \equiv \overline{n}]\!].
\end{array}
$$

Mots

Comme on l'a vu, l'alphabet sur lequel les mots sont construits est supposé dénombrable, on peut donc supposer par exemple que $A = \mathbf{N}$. Un mot $w = a_1 \cdots a_n$ sera représenté par un terme t tel que $t0 \equiv \overline{n}$ et $ti \equiv \overline{a_i}$ pour $i \in [\![1,n]\!]$. On dira qu'un tel terme t est un représentant de w ce que l'on notera $t = \overline{w}$ (et qu'on devrait en toute rigueur noter $t \in \overline{w}$). La fonction donnant la longueur d'un mot se construit alors de manière triviale :

$$
\begin{array}{l}
\mathtt{long} = \lambda w.w\overline{0} \\
[\![\forall w = a_1 \ldots a_n \in A^*.\ \mathtt{long}\ \overline{w} \equiv \overline{n}]\!]
\end{array}
$$

de même que la fonction renvoyant la $i^{\text{ème}}$ lettre :

$$
\begin{array}{l}
\mathtt{lettre} = \lambda w.\lambda i.wi \\
\left[\!\!\left[\begin{array}{l} \forall w = a_1 \ldots a_n \in A^*.\ \forall i \in \{1, \ldots ,n\}. \\ \quad \mathtt{lettre}\ \overline{w}\ \overline{i} \equiv \overline{a_i} \end{array} \right]\!\!\right]
\end{array}
$$

La fonction réalisant la concaténation de deux mots est plus compliquée à définir, le lecteur pourra vérifier (à titre d'exercice) que le terme suivant convient :

$$
\begin{array}{l}
\mathtt{concat} = \\
\quad \lambda u.\lambda v.\lambda n. \\
\qquad \mathtt{cond}(\mathtt{nul}\ n) \\
\qquad\quad (\mathtt{plus}\ (\mathtt{long}\ u)(\mathtt{long}\ v)) \\
\qquad\quad (\mathtt{cond}(\mathtt{inf}\ n\ (\mathtt{long}\ u)) \\
\qquad\qquad (un) \\
\qquad\qquad (v\ (\mathtt{moins}\ n\ (\mathtt{long}\ u)))) \\
[\![\forall w_1,w_2 \in A^*.\ \mathtt{concat}\ \overline{w_1}\ \overline{w_2} \equiv \overline{w_1 w_2}]\!].
\end{array}
$$

Finalement, prenant en argument un entier a, le terme suivant construit le mot à une lettre égal à a :

$$
\begin{array}{l}
\mathtt{base} = \lambda a.\lambda n.\mathtt{cond}\ (\mathtt{nul}\ n)\ \overline{1}\ a \\
[\![\forall a \in A.\ \mathtt{base}\ \overline{a} \equiv \overline{a}]\!].
\end{array}
$$

Mots de Dick

Fixons $w = a_1 \dots a_n \in A^*$, on peut alors associer à w la fonction :

$$\begin{array}{rccl} p_w : & \{0,\dots,n\} & \to & \mathbf{Z} \\ & i & \mapsto & p(a_1 \dots a_i) \end{array}$$

(avec par convention $p_w(0) = 0$) de sorte que $p(w) = p_w(n)$. Cette fonction associée à w possède une définition récursive naturelle :

$$\begin{cases} p_w(0) = 0 \\ p_w(i) = p_w(i-1) + 1 \text{ si } i \in \{1,\dots,n\} \text{ et } a_i = [\\ p_w(i) = p_w(i-1) - 1 \text{ si } i \in \{1,\dots,n\} \text{ et } a_i =] \end{cases}$$

Identifions l'alphabet $A = \{[,]\}$ avec $\{0,1\}$, on peut alors représenter p_w par le terme suivant :

$$\begin{array}{l} \texttt{rec}\ \widetilde{0}\ (\lambda n.\lambda z. \\ \quad \texttt{cond}\ (\texttt{nul}\ (\texttt{lettre}\ wn)) \\ \qquad (\texttt{succ}\ z) \\ \qquad (\texttt{pred}\ z)) \end{array}$$

et si on abstrait ce terme en fonction de w on obtient :

$$\begin{array}{l} \texttt{p} = \\ \quad \lambda w.\texttt{rec}\ \widetilde{0}\ (\lambda n.\lambda z. \\ \qquad \texttt{cond}\ (\texttt{nul}\ (\texttt{lettre}\ wn)) \\ \qquad\quad (\texttt{succ}\ z) \\ \qquad\quad (\texttt{pred}\ z)) \end{array}$$

$$\left[\!\!\left[\begin{array}{l} \forall w = a_1 \dots a_n \in A^*.\ \forall i \in \{1,\dots,n\}. \\ \quad \texttt{p}\ \underline{w}\ \underline{i} \equiv \underline{p(a_1 \dots a_i)} \end{array} \right]\!\!\right]$$

Posons finalement :

$$\begin{array}{rl} D' = & \{w = a_1 \dots a_n \in A^* | \\ & \quad \forall i \in \{1,\dots,n\}.p(a_1 \dots a_i) \geqslant 0\} \end{array}$$

on a alors $D = D' \cap \{w \in A^* | p(w) = 0\}$. Le terme suivant permet de tester si un mot est dans D' :

$$\begin{array}{l} \texttt{testDprime} = \lambda w.\texttt{rec}\ \texttt{vrai} \\ \quad (\lambda n.\lambda z.\texttt{et}\ z\ (\texttt{positz}\ (\texttt{p}\ wn))) \end{array}$$

et donc on en déduit le test d'appartenance à D :

$$\begin{array}{l} \texttt{testD} = \lambda w.\texttt{et} \\ \quad (\texttt{testDprime}\ w\ (\texttt{long}\ w)) \\ \quad (\texttt{nulz}\ (\texttt{p}\ w\ (\texttt{long}\ w))). \end{array}$$

Questions de stratégies

Lorsu'on cherche une forme normale pour un terme, il est commode de remplacer la relation d'équivalence $\equiv$ par une relation de réduction (ou règle de réécriture) que l'on appelle beta-réduction et définie par :

- $(\lambda x.u)v \to u[v/x]$,
- si $u \to u'$, alors $(uv) \to (u'v)$, $(vu) \to (vu')$ et $\lambda x.u \to \lambda x.u'$ quels que soient v et x.

On cherche alors à réduire successivement les termes à l'aide de cette relation pour obtenir une forme irréductible. Deux problèmes peuvent alors se poser. D'une part certains termes n'ont pas de forme normale, par exemple :

$$u = (\lambda x.(xx))(\lambda x.(xx))$$

peut se réduire indéfiniment sur lui-même. Par ailleurs, si on pose $v = \lambda x.y$, alors on peut réduire indéfiniment vu sur lui-même (si on choisit de réduire d'abord dans u) ou bien aboutir immédiatement à la forme normale y si on subsitue u à x dans y. En fait ce principe est général : si dans un terme, on réduit toujours le lambda qui se trouve le plus à gauche, alors on aboutira à une forme normale (si toutefois le terme en possède une).

Pour en savoir plus...

[B] H.P. Barendregt, *The lambda calculus, its syntax and semantics*, Studies in logic and the foundations of mathematics.

[L] R. Lalement, *Logique, réduction, résolution*, Masson.