Le λ-calcul

1 Folklore historique

Le λ-calcul a été inventé par Alonzo Church dans les années 1930, qui publia un premier article sur la partie calculatoire (celle que je vous ai présentée) en 1936, puis en second en 1940 sur un sous-système, le λ-calcul simplement typé, qui présente un certain nombre de propriétés intéressantes, notamment le fait de terminer (plus de souci avec $\delta\delta$). La motivation première était, à l'époque, de réfléchir à la notion de calculabilité, à laquelle d'autres réponses furent apportées (les machines de Turing notamment). Le λ-calcul est en effet un système dit *Turing-complet*, ce qui signifie qu'il permet de calculer tout ce qui est calculable. À la louche, c'est capable de faire tout ce qu'un langage de programmation pourra faire, donc par extension un ordinateur. Bref, c'est puissant.

Depuis, on a aussi trouvé tout un tas de résultats importants en logique grâce et avec ça, ce qui en fait un outil assez prisé par les chercheurs aujourd'hui.

2 Un univers fonctionel

En λ-calcul, il n'y a que des fonctions. Tout est fonction. A fortiori, les fonctions sont des valeurs. C'est le point clé pour comprendre. Par rapport à ce que vous connaissez, il est par exemple crucial d'être capable de faire la différence entre f (fonction) et $f(2), f(x)$ (valeurs). La notation $f : x \mapsto f(x)$ aide pas mal : f est la fonction qui à une variable x associe la valeur $f(x)$.

Le deuxième point clé, dont vous disposez déjà dans votre bagages mathématiques, c'est l'idée de *substitution*. En effet, si je vous donne le polynôme $P : x \mapsto x^2 + 3 * x + 5$, vous saurez tous mécaniquement calculer $P(2)$ en remplaçant chaque occurence de la variable x par 2, ce que l'on notera $P(x)\{x := 2\} \equiv P(2) = 2^2 + 3*2+5 = 42$.

Ce sont les deux idées principales du λ-calcul, chaque fonction explicitera ses attentes d'arguments, et la seule opération de calcul sera de remplacer une variable par l'argument lui correspondant.

3 C'est parti

3.1 Syntaxe

Le λ-calcul est défini inductivement, chose dont vous n'avez pas forcément l'habitude. Ou tout du moins pas formellement. Pourtant, c'est comme ça que l'on défini les entiers par exemple, un entier est soit 0 soit le successeur d'un autre entier. Dans la vie de tous les jours, si vous deviez définir vos ancètres, vous pourriez dire que c'est soit vos parents, soit les parent de vos parents, etc... mais très vite vous en auriez marre, et diriez que c'est soit vos parents, soit les parents d'un de vos ancètres. Pour ceux qui sont familiers de la récurrence, c'est exactement pareil. Plus formellement, on définit un objet par des briques de bases, et des constructeurs sur les objets de la façon suivante :

$$objet ::= brique_1 \mid brique_2 \mid \ldots \mid Constructeur_1(objet_1, objet_2, \ldots) \mid Constructeur_2(objet_1, objet_2) \mid \ldots$$

Cela se lit comme ça : un objet est soit la brique1, soit la brique2, etc, soit le constructeur1 appliqué aux objet1, objet2 déjà construits par induction. Si l'on reprend les exemples, pour les entiers, la seule brique de base est 0, le seul constructeur est la fonction successeur. Pour les aïeux, les briques de base, ce sont vos parents, et les deux constructeurs seraient d'être le père ou la mère d'un ancètre. On écrirait formellement :

$$n ::= 0 \mid successeur(n)$$

$$ancetre ::= papa \mid maman \mid papa(ancetre) \mid maman(ancetre)$$

.

Normalement, à ce stade, les définitions inductives ne devraient plus trop vous faire peur. On peut donc y aller pour le λ-calcul :

T, U	$::=$	x	$\mid$	$\lambda x.(T)$	$\mid$	TU
(*termes*)		(*variables*)		(*abstraction*)		(*application*)

Il est à noter qu'il est nécessaire de s'être auparavant doter d'un alphabet pour les variables. Pour plus de facilité, je noterai dorénavant les variables en minuscules et les termes en majuscules.

Quelques exemples :

$Id \equiv \lambda x.x$
$\delta \equiv \lambda x.(xx)$
$swap \equiv \lambda x.(\lambda y.(yx))$
$T \equiv \lambda b.(\lambda f_1.(\lambda f_2.((bf_1)f_2)))$
$comp \equiv \lambda f.(\lambda g.(\lambda x.(f(gx))))$

Pour plus de lisibilité, je noterai maintenant $\lambda xyz.t$ à la place de $\lambda x.(\lambda y.(\lambda z.t)))$, le dernier exemple devient ainsi : $comp \equiv \lambda fgx.(f(gx))$

3.2 La β-réduction : $\longrightarrow_\beta$

C'est le joli nom qu'on donne à la seule opération de calcul existant. Elle correspond en fait à appliquer une fonction à son argument, c'est à dire remplacer la variable abstraite par son argument. On la définit comme suit :

$$(\lambda x.T)u \longrightarrow_\beta T\{x := U\}$$

La notation $T\{x := U\}$ est la même que pour le polynôme auparavant. Moralement, ça veut juste dire qu'à chaque occurence de x dans T, on effectue la substitution par U. Tant qu'on y est, on peut introduire une notation toute bête pour exprimer le fait qu'on effectue plusieurs pas de réduction : $\xrightarrow{*}_\beta$. Par exemple, si $T \longrightarrow_\beta T_1 \longrightarrow_\beta T_2 \longrightarrow_\beta T_3 \longrightarrow_\beta T*$, on peut noter $T \xrightarrow{*}_\beta T*$.

Pour ceux qui voudraient aller plus loin, on peut définir la substitution par induction de la manière suivante (on déconstruit le λ-terme) :

$$\begin{array}{rcl|rcl} y\{x := U\} & = & y & (\lambda y.T)\{x := U\} & = & \lambda y.(T\{x := U\}) \\ x\{x := U\} & = & U & (T_1T_2)\{x := U\} & = & (T_1\{x := U\})(T_2\{x := U\}) \end{array}$$

Exercice 1. *Pour vous assurer d'avoir bien compris, essayer de voir sur des exemples ce que comp, swap et δ réalisent comme opérations.*

3.3 Propriétés

À titre plus ou moins culturelles, les deux propriétés les plus importantes du λ-calcul.

Définition 1 (**Forme normale**)**.** On appelle *forme normale* un terme U tel que l'on ne peut plus le réduire, i.e. qu'il n'existe pas de terme U' tel que $U \longrightarrow_\beta U'$. Une relation de réduction est dite *normalisante* (ou termine [1]) si tout terme admet une forme normale, c'est-à-dire que pour tout T, il existe $T*$ en *forme normale* tel que $T \xrightarrow{*}_\beta T*$.

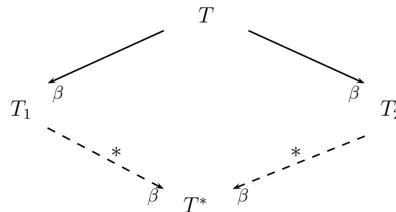
Exercice 2. *Les termes suivants sont-ils en forme normale ? Si non, ont-ils une forme normale ?*
$\lambda x.(x(xx))$, $(\lambda x.x)\lambda y.y$, $\lambda x.((\lambda y.y)x), (\lambda xy.yx)\delta, \delta\delta$.

Propriété 2 (**Terminaison**)**.** *Le λ-calcul ne garantit en aucun cas la terminaison du calcul, i.e. la relation $\longrightarrow_\beta$ n'est pas normalisante.*

Démonstration. Essayez par exemple de réduire $\Omega \equiv \delta\delta$... □

Définition 3 (**Confluence**)**.** Une relation $\longrightarrow$ est dite confluente si, pour tout termes T,T_1,T_2 tels que $T \longrightarrow T_1$ et $T \longrightarrow T_2$, il existe $T*$ tel que $T_1 \xrightarrow{*} T*$ et $T_2 \xrightarrow{*} T*$.

On comprend tout de suite mieux sur un dessin :



Propriété 4 (**Confluence**)**.** *Le λ-calcul est confluent pour les termes admettant une forme normale*

Le gros intérêt de cette propriété est que si l'on l'a le choix sur plusieurs réductions, on peut choisir n'importe laquelle, si le calcul s'arrête, ce sera de toute façon sur une unique valeur. Je la mentionne surtout parce que c'est une question cruciale en réécriture, le domaine où l'on joue beaucoup avec des relations de réductions.

1. *Tout comme en français, ça veut juste dire que le calcul s'arrête un jour*

4 Retrouvons $\mathbb{N}$

Au début de cet atelier, j'avais mentionner que l'on pouvait tout calculer ou presque en λ-calcul, la première naturelle est donc de retrouver les entiers ainsi que l'addition, la multiplication, etc... nous allons donc voir comment les encoder à la manière de Church.

4.1 Les entiers

On s'appuie sur la définition de $\mathbb{N}$ et des entiers vue plus haut. En notant S la fonction successeurs, on définit les entiers de la façon suivantes :

$$n ::= 0 \mid S(n)$$

Cependant, en λ-calcul, tout est fonction. L'idée va donc être de demander un zéro, de demander une fonction successeur, et de renvoyer l'entier correspondant :

$$n \equiv \lambda sz.\underbrace{s(s(\ldots s(s(}_{\text{n fois}}z))\ldots))$$

Ainsi on a facilement :

$$0 \equiv \lambda sz.z,\ 1 \equiv \lambda sz.(s(z)),\ 2 \equiv \lambda sz.(s(s(z))), etc\ldots$$

Attention à ne pas confondre alors 1 avant et après qu'on lui ait donné ses arguments ! Si vous passez par exemple à 1 la fonction "+2" au lieu du successeur, vous obtiendrez moralement le 2 construit avec le même zéro et le bon successeur.

4.2 L'addition

L'idée est essentiellement de dire que pour construire 2+3=5, on applique 3 fois le successeur à 0, puis on réapplique 2 fois le successeur au résultat obtenu :

$$5 = 2 + 3 \equiv s(s(s(s(s(z)))))$$

Cela revient donc juste à construire 2, en partant non pas de 0 mais de 3. On va donc, pour effectuer l'addition des deux entiers n_1 et n_2, avoir besoin des deux entiers en question, du successeur, du zéro, et l'on retournera n_2 entier à qui l'on aura donné en arguments la fonction successeur et n_1 au lieu de zéro :

$$add \equiv \lambda n_1 n_2 sz.((n_1\ s)((n_2\ s)z))$$

En considérant le parenthésage à gauche (convention usuelle, peut-être trouverez-vous ça plus lisible, on voit mieux qui prend quel argument), cela donne :

$$add \equiv \lambda n_1 n_2 sz.(n_1\ s\ (n_2\ s\ z))$$

Exemple : Effectuons le calcul effectif de 2+3 (ça sera en plus l'occasion de revoir un peu la β-réduction). On a $2 \equiv \lambda sz.(s(s(z)))$, $3 \equiv \lambda sz.(s(s(s(z))))$. D'où :

$$\begin{aligned}
add\ 2\ 3 &= (\lambda \underline{n_1} n_2 sz.\underline{n_1}\ s\ (n_2\ s\ z))\ \underline{\lambda sz.(s(s(z)))}\ \lambda sz.(s(s(s(z)))) \\
&\longrightarrow_\beta (\lambda \underline{n_2} sz.(\lambda sz.(s(s(z))))\ s\ (\underline{n_2}\ s\ z)))\ \underline{\lambda sz.(s(s(s(z))))} \\
&\longrightarrow_\beta \lambda sz.(\lambda \underline{s} z.(\underline{s}(\underline{s}(z)))\ \underline{s}\ (\lambda sz.(s(s(s(z))))\ s\ z)) \\
&\longrightarrow_\beta \lambda sz.(\lambda \underline{z}.(s(s(\underline{z})))\ \underline{(\lambda sz.(s(s(s(z))))\ s\ z)}) \\
&\longrightarrow_\beta \lambda sz.(s(s((\lambda \underline{s} z.(\underline{s}(\underline{s}(\underline{s}(z))))\ \underline{s}\ z)))) \\
&\longrightarrow_\beta \lambda sz.(s(s((\lambda \underline{z}.(s(s(s(\underline{z}))))\ \underline{z})))) \\
&\longrightarrow_\beta \lambda sz.(s(s(s(s(s(z)))))) \ \equiv 5
\end{aligned}$$

Si vous avez compris ce calcul (comprendre signifiant bien entendu êtes capable de le refaire avec 1 et 2 par exemple), c'est que c'est tout bon !

4.3 Multiplication

Et le plus dur pour la fin. Pour la multiplication, on va s'inspirer un petit peu de ce que l'on a fait pour l'addition, en remarquant que $3 * 2 = \underbrace{2 + 2 + 2}_{3\text{ fois}}$. En fait, ce qui se passe, c'est qu'on effectue 3 fois l'opération "+2", en partant de 0. Ça tombe bien, en λ-calcul, 3 attend qu'on lui donne son successeur, il suffit donc de lui donner le terme qui fait l'opération "+2" à la place, et ça sera gagner. Or ce terme n'est ni plus ni moins que 2 à qui l'on a donné le successeur mais qui attends encore un 0. Cette fois-ci, pour plus de clarté, je commence par l'exemple avant de généraliser. On a donc :

- $3 \equiv \lambda sz.(s(s(s(z))))$
- $2 \equiv \lambda sz.(s(s(z)))$
- 2 à qui l'on a passé le successeur : $(\lambda sz.(s(s(z))))s \longrightarrow_\beta \lambda z.s(s(z))$
- et donc 3 avec ce terme en guise de successeur :

$$\begin{aligned}
(\lambda sz.(s(s(s(z)))))\lambda z.s(s(z)) &\longrightarrow_\beta \lambda z.(\lambda z.s(s(z))(\lambda z.s(s(z))(\lambda z.s(s(z))(z)))\\
&\longrightarrow_\beta \lambda z.(\lambda z.s(s(z))(\lambda z.s(s(z))(s(s(z)))\\
&\longrightarrow_\beta \lambda z.(\lambda z.s(s(z))(s(s(s(s(z))))))\\
&\longrightarrow_\beta \lambda z.(\lambda z.s(s(z))(s(s(s(s(z))))))\\
&\longrightarrow_\beta \lambda z.(s(s(s(s(s(s(z))))))) \equiv 6
\end{aligned}$$

Dans le cas général, on a donc :

$$mult \equiv \lambda n_1 n_2 sz.(n_1\ (n_2\ s)\ z)$$

Si cela ne vous semble pas transparent, encore une fois, on ne comprend jamais aussi bien qu'en essayant, donc choisissez vous deux petits entiers n_1 et n_2, et calculez $mult\ n_1\ n_2$. À l'inverse, si c'est complétement transparent et beaucoup trop facile, vous pouvez essayez de définir *power* tel que $power\ n_1\ n_2 \equiv n_1^{n_2}$. C'est un peu plus compliqué !

5 Miscellanées

On peut de même s'amuser à encoder tout un tas de structures en λ-calcul, notamment les listes, les arbres, les booléens et de façon générale tout ce qui est cher aux programmeurs.
Pour les plus téméraires d'entre vous, il existe un certain nombre d'articles sur le λ-calcul, et même des livres (notament ceux de Barendregt et Krivine). Si vous voulez bosser votre anglais en bonus, demandez-moi des références !
Sinon, si vous voulez juste apprendre le λ-calcul à vos petits frères et soeurs (ou à vos enfants), les alligators sont là pour vous : http://worrydream.com/AlligatorEggs/

To be continued... ?