Logique et λ-calcul

Cours du M1 STIC ISC

Tom Hirschowitz

1 Lambda-calcul

1.1 Motivation et définition des programmes

Le λ-calcul est conçu comme le langage de programmation le plus simple permettant de définir des fonctions et de les appliquer à des arguments.

Dans ce langage, comme dans presque tous les langages de programmation, il y a des variables. Comme il n'y a pas grand-chose d'autre, on peut prendre presque n'importe quoi pourvu qu'il y en ait assez. Ici, cela signifie au moins $\mathbb{N}$. Comme on veut faire des maths avec, on va au plus simple et on prend $\mathbb{N}$ — la plupart des auteurs font d'autres choix, mais ça ne change pas grand-chose.

Bon, donc premier point : toute variable (c'est-à-dire ici tout entier) est un programme. Dans l'idée, c'est le programme qui renvoie la valeur de la variable.

Dans le λ-calcul, comme on veut des fonctions, on veut pouvoir former, pour tout programme P et toute variable $x \in \mathbb{N}$, la fonction $x \mapsto P$. Dans l'idée, c'est la fonction qui, quand on l'appelle avec une valeur, disons V, comme argument, exécute le programme $P[x \mapsto V]$ et renvoie sa valeur. Ici, $P[x \mapsto V]$ est une notation désignant P, dans lequel x est partout remplacé par V.

D'où, deuxième point : pour tout programme P et toute variable x, le couple (x, P) forme un nouveau programme. Quand on voit un tel couple comme un programme, on le note $\lambda x.\, P$ — c'est le λ du λ-calcul.

Enfin, comme on veut pouvoir utiliser ces fonctions, pour tous programmes P et Q, on veut pouvoir considérer P comme une fonction et l'appliquer à l'argument Q.

D'où, troisième et dernier point : pour tous programmes P et Q, le couple (P, Q) forme un nouveau programme. Quand on voit un tel couple comme un programme, on le note $P\,Q$.

Comme on cherche à définir un langage minimal offrant ces possibilités, on pose : les programmes du λ-calcul sont définis par la grammaire

$$P ::= x \mid \lambda x.\, P \mid P\,P,$$

où x désigne une variable.

Remarque 1. Pour le lecteur non familier avec la notation BNF, cela signifie simplement qu'un programme est une suite de variables, de caractères « λ », « . », « » (espace), formée par les règles suivantes :

- toute suite de longueur 1 formée par une variable est un programme ;
- pour tous programmes P et variable x, la suite commençant par $(\lambda, x, .)$ et finissant par P est un programme ;
- pour tous programmes P et Q, la suite commençant par P, continuant par une espace et finissant par Q est un programme.

Le langage obtenu est ambigu. Par exemple, $\lambda x.\, P\,Q$ peut se comprendre soit comme $(\lambda x.\, P)\,Q$, soit comme $\lambda x.\, (P\,Q)$. Pour résoudre ce problème, on étend la grammaire comme suit :

Définition 1. les programmes du λ-calcul sont définis par la grammaire

$$P ::= x \mid \lambda x\,.\,P \mid P\,P \mid (P).$$

On note Λ l'ensemble des programmes.

Ainsi, on peut lever les ambiguïtés grâce aux parenthèses.

Pour éviter un usage excessif des parenthèses, on définit des *priorités*, ce qui revient à adopter la convention que certaines parenthèses peuvent rester implicites :

- $P\,Q\,R$ est interprété comme $(P\,Q)\,R$;
- $\lambda x\,.\,P\,Q$ est interprété comme $\lambda x\,.\,(P\,Q)$.

Remarque 2. Il se passe exactement la même chose quand on décrète que $x + y \times z$ doit être interprété comme $x + (y \times z)$.

Notation 1. Dans la suite, on n'utilisera quasiment pas le fait que les variables sont des entiers. Par exemple, on écrira des programmes comme $\lambda x\,.\,x\,x$. Ceci est à comprendre comme : on prend une variable arbitraire x, peu importe laquelle, et on forme le programme $\lambda x\,.\,x\,x$. Cela peut ainsi désigner $\lambda 1.11$, $\lambda\,2.2\,2$, etc. J'espère que le lecteur sent déjà que tous ces programmes définissent en fait la même fonction, puisque de l'une à l'autre on ne fait que renommer l'argument formel.

Exercice 1. Insérer les parenthèses implicites pour les programmes suivants : $\lambda x\,.\,\lambda y\,.\,x\,y\,z$, $\lambda x\,.\,(\lambda y\,.\,x)\,y$.

2 Amélioration de la définition des programmes

Mathématiquement, notre définition des programmes n'est pas très efficace. Etant donnée une suite de symboles, il est en général loin d'être trivial de déterminer si c'est un programme ou pas. Autre problème, plusieurs suites de symboles représentent moralement le même programme. Par exemple, rien n'interdit une utilisation excessive des parenthèses : les programmes x, (x), $((x))$, $(((x)))$, ... devraient clairement être identifiés.

2.1 Syntaxe abstraite

Une manière d'améliorer la situation consiste à représenter les programmes comme des arbres étiquetés. On ne précise pas ici le sens formel de « arbre étiqueté », l'intuition devant être claire (le faire nous prendrait trop de temps). On utilise comme ensemble d'étiquettes

$$E = \mathbb{N} + \mathbb{N} + 1.$$

Ici, 1 désigne le singleton $\{0\}$ et le $+$ désigne l'union disjointe d'ensembles, qu'on peut définir comme

$$(\{0\}\times\mathbb{N}) \cup (\{1\}\times\mathbb{N}) \cup (\{2\}\times 1).$$

Une étiquette est donc

- soit un couple $(0, x)$, pour $x \in \mathbb{N}$,
- soit un couple $(1, x)$,
- soit le couple $(2, 0)$.

La convention est que les nœuds étiquetés par $(0, x)$ seront des nœuds unaires représentant $\lambda x.\, P$, dont l'unique fils représentera P. Les nœuds étiquetés par $(1, x)$ seront des feuilles et représenteront le programme x. Enfin, les nœuds étiquetés par $(2, 0)$ seront des nœuds binaires représentant $P\,Q$ et dont les fils représenteront P et Q. En conséquence, on pose :

Notation 2. L'étiquette $(0, x)$ sera notée λx, l'étiquette $(1, x)$ sera notée x et $(2, 0)$ sera notée $@$.

On peut donc interpréter chaque programme-suite de symboles par un programme-arbre étiqueté : on définit une fonction $P \mapsto [\![P]\!]$

$$\begin{array}{ccc} \text{Suites de symboles} & \to & \text{Arbres étiquetés} \\ P & \mapsto & [\![P]\!] \end{array}$$

définie inductivement par les règles décrites en figure 2.1. Dans le premier cas, le membre de droite désigne l'arbre constitué d'une seule feuille. Dans le deuxième cas, le membre de droite est l'arbre dont la racine est étiquetée par $@$ et dont les fils gauche et droit sont respectivement $[\![P]\!]$ et $[\![Q]\!]$.

Enfin, dans le troisième cas, c'est l'arbre dont la racine est étiquetée par λx et dont l'unique fils est $[\![P]\!]$.
Deux avantages immédiats de cette représentation est qu'on évite certaines redondances, ainsi que les parenthèses.
Exemple 1. Tous les programmes x, (x), ... du début de partie sont interprétés par le même arbre, le premier de la figure *2.1*.

Exemple 2. Les programmes $x\,y\,z$ et $x\,(y\,z)$ sont respectivement interprétés comme les arbres :

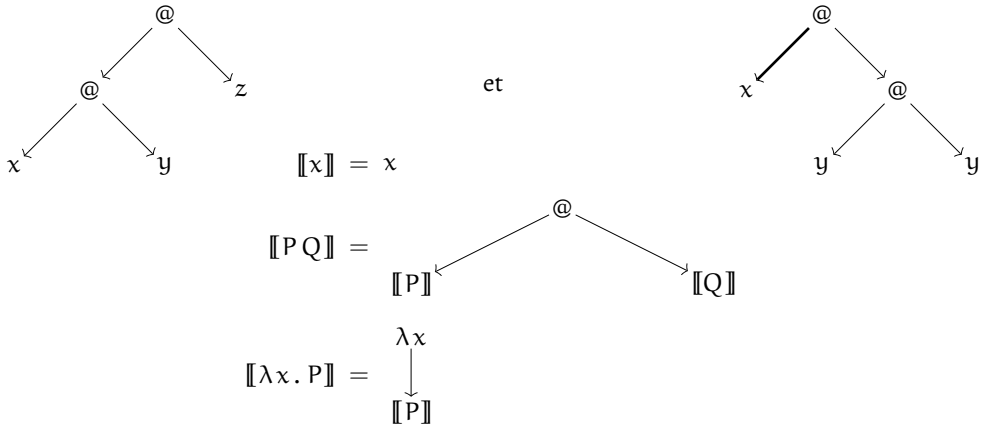
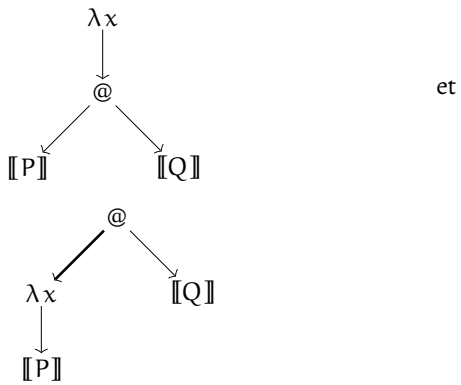


Figure 2.1 – Interprétation des programmes-suites de symboles en programmes-arbres étiquetés

On note que dans les deux cas les feuilles sont x, y et z, de gauche à droite, la structure de l'arbre remplaçant l'usage des parenthèses.
Exemple 3. Les programmes $\lambda x\,.\,P\,Q$ et $(\lambda x\,.\,P)\,Q$ sont respectivement représentés par



Comme dans l'exemple précédent, les feuilles de l'arbre sont les mêmes pour tous $[\![P]\!]$ et $[\![Q]\!]$, dans le même ordre, et c'est la structure de l'arbre qui fait la différence.
Définition 2. On appelle ces arbres étiquetés des *arbres de syntaxe abstraite*.

Si on a éliminé certaines redondances, il en reste une, très importante :

Exemple 4. Dans un langage de programmation, le nom d'un argument formel n'est pas significatif. En λ-calcul, cela indique par exemple que l'on devrait considérer les programmes $\lambda 1.1$, $\lambda 2.2$, … comme équivalents. Attention, ceci ne porte pas sur les « variables libres ». Par exemple, les programmes $x\,y$ et $x\,z$ ne sont pas équivalents en général (si $y \neq z$). Le premier applique la fonction x à l'argument y, alors que le second l'applique à z.

2.2 Lambda-graphes

Pour éliminer la dernière redondance évoquée en fin de partie précédente, on remplace les occurrences d'arguments formels par des « pointeurs en arrière » vers la fonction qui les introduit.

Exemple 5. Le programme $\lambda x.\,x$, qui correspond à l'arbre de syntaxe abstraite 

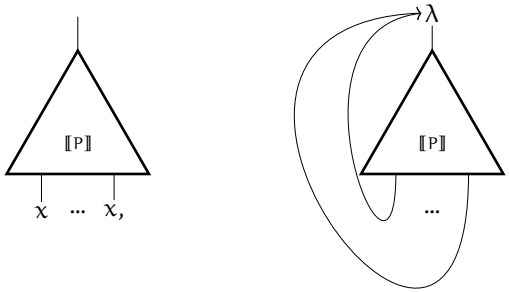
, devient le λ-graphe

L'interprétation devient plus difficile à définir, parce que quand on rencontre une variable x, il faut savoir si elle se situe sous un λx ou pas :

- si oui, on la remplace par un pointeur en arrière vers le dernier λx rencontré;
- si non, on renvoie un nœud variable x.

On ne va pas définir l'interprétation formellement, c'est au-delà des buts de ce cours. En revanche, on peut la décrire assez précisément en disant que sur les variables et les applications, on fait comme avec les arbres de syntaxe abstraite, alors que pour les λ, on procède comme suit.

Si $[\![P]\!]$ est le λ-graphe de gauche ci-dessous, où on fait apparaître les occurrences de x, alors $[\![\lambda x.\,P]\!]$ est celui de droite :



Exercice 2. Donner la représentation en λ-graphes de $\lambda x.\,(x\,(x\,y))$ et $(\lambda x.\,x)\,(x\,y)$.

Dans la suite on utilisera principalement la notation syntaxique, mais implicitement, elle désignera plutôt les λ-graphes.

3 Quasi définition de l'exécution

Dans les parties précédentes, on a défini l'ensemble des programmes du λ-calcul. On s'attache maintenant à décrire l'exécution de ces programmes. On va le faire sous la forme d'une *relation*, dite de réduction, entre eux, c'est-à-dire un ensemble de couples de programmes. Cette relation sera notée $\longrightarrow$ est formera donc un sous-ensemble de $\Lambda \times \Lambda$.

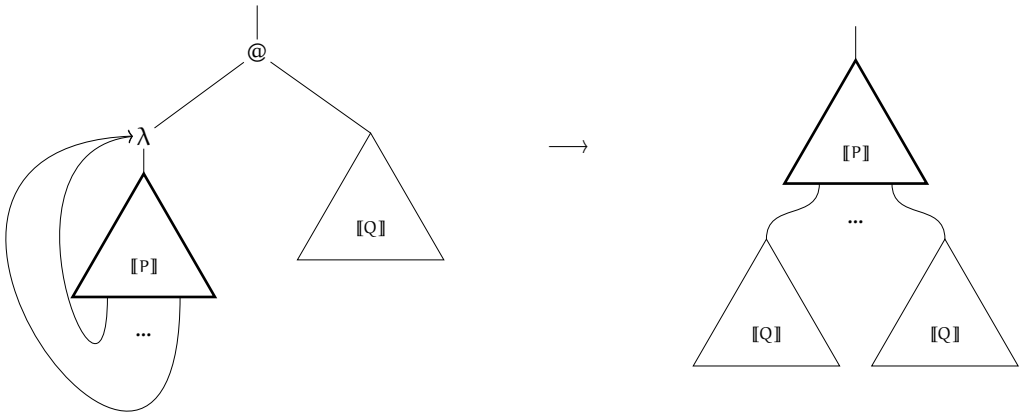
Notation 3. On notera cette relation de manière infixe, c'est-à-dire que pour dire que le couple (P, Q) est dans la relation, on écrira $P \longrightarrow Q$.

Définition 3. Cet ensemble est à nouveau défini par des règles de formation :

- pour tous programmes P et Q et pour toute variable x, on a $(\lambda x.\, P)Q \longrightarrow P[x \mapsto Q]$;
- pour toute variable x et tous programmes P, P' et Q tels que $P \longrightarrow P'$, on a

$$PQ \longrightarrow P'Q, \qquad QP \longrightarrow QP' \qquad \text{et} \qquad \lambda x.\, P \longrightarrow \lambda x.\, P'.$$

Ici $P[x \mapsto Q]$ désigne la *substitution* de x par Q dans P, qu'il nous faut définir formellement. On représente en général de telles règles de manière plus concise au moyen de *règles d'inférence*, comme en figure 3.2 : la barre horizontale de chaque règle représente un « si… alors… ». Les hypothèses, qu'on appelle dans ce cas des *prémisses*, sont au-dessus, la conclusion étant en-dessous. Les indications β, ξ, G et D sont les noms des règles ; on les utilisera pour y faire référence. Attention, la première règle, β, mentionne un argument formel x. Elle ne peut donc pas directement s'appliquer à la représentation en λ-graphes. En fait, elle se représente assez intuitivement dans ce dernier formalisme :



Cette représentation peut en fait nous guider dans la définition de $P[x \mapsto Q]$.

Exemple 6. Considérons $(\lambda x.\,(\lambda y.\,x))\,y$. Par β, ce programme ce réduit en $(\lambda y.\,x)[x \mapsto y]$. Si on remplace bêtement x par y, on tombe sur $\lambda y.\,y$, alors que si on se fie à la représentation par λ-graphes, on tombe plutôt sur quelque chose comme $\lambda y'.\,y$. C'est cette dernière réponse qui est la bonne : comment la refléter dans la définition de $P[x \mapsto Q]$?

$$\frac{}{(\lambda x.\,P)\,Q \longrightarrow P[x \mapsto Q]}\,\beta \qquad \frac{P \longrightarrow P'}{\lambda x.\,P \longrightarrow \lambda x.\,P'}\,\xi$$

$$\frac{P \longrightarrow P'}{P\,Q \longrightarrow P'Q}\,G \qquad \frac{P \longrightarrow P'}{Q\,P \longrightarrow Q\,P'}\,D$$

Figure 3.2 – Relation de réduction