TD 02 – λ-calcul, fonctions μ-récursives

Exercice 1. *λ-calcul*

Le λ-calcul a l'avantage d'être extrêmement minimaliste. On définit des λ-termes à base de variables x, de fonctions $\lambda x.t$, et d'application $t\ t'$. Pour vous donner un aperçu, voici comment exprimer le nombre 2 par un λ-terme : $\lambda s.\lambda z.s\ (s\ z)$; et l'addition qui prend en argument les λ-termes pour les nombres a et b : $\lambda a.\lambda b.\lambda s.\lambda z.a\ s\ (b\ s\ z)$. Ensuite ce sont les itérations d'une transformation appelée β-réduction qui déroulent le calcul d'un λ-terme.
Le λ-calcul a été inventé par Alonso Church dans les années 1930.

Définition : tout est fonction

La syntaxe des λ-termes est définie inductivement :
- x est un λ-terme, si x est une variable ;
- $\lambda x.t$ est un λ-terme (λ-abstraction), si t est un λ-terme et x une variable ;
- $t\ s$ est un λ-terme (λ-application), si t et s sont des λ-termes.

Les λ-abstractions permettent de construire des fonctions, et les λ-application permettent de donner des arguments aux fonctions.

Les λ-termes peuvent être réduits en appliquant les règles suivantes :
- $(\lambda x.t)\ t' \mapsto t[x := t']$ (β-réduction) ;
- $\lambda x.t \mapsto \lambda y.t[x := y]$ avec $y \notin \texttt{Free}(t)$ (α-conversion).

Les β-réductions permettent d'appliquer une fonction à un terme, et les α-conversions permettent d'éviter les conflits entre les noms des variables (*capture de variable*). La notation $t[x := t']$ est le terme t dans lequel on a substituée toute occurrence de x par t'. On note $\texttt{Free}(t)$ l'ensemble des variables libres dans t, c'est-à-dire qui ne sont pas liées à une λ-abstraction.

Par exemple, quel que soit s on a $(\lambda x.x)\ s \mapsto x[x := s] = s$ montre que le terme $\lambda x.x$ est la fonction identité, et $(\lambda x.y)\ s \mapsto y[x := s] = y$ montre que le terme $\lambda x.y$ est une fonction constante.

1. Le processus de réduction peut ne jamais terminer. Sauriez-vous trouver un exemple ?

Les β-réductions correspondent aux étapes de calcul. Il peut exister plusieurs façons de réduire un terme, le théorème de Church-Rosser nous dit que toutes les façons d'atteindre une forme normale β (un terme non réductible) mènent au même terme.

Calculer en réduisant des λ-termes

Le lambda calcul est très riche [1], on peut par exemple encoder les nombres, l'addition, la multiplication comme cela :

$$\begin{aligned}\texttt{n} &= \lambda f.\lambda x.(f\ \ldots(f\ x)\ldots)\\ \texttt{+} &= \lambda m.\lambda n.\lambda f.\lambda x.(m\ f\ (n\ f\ x))\\ \texttt{*} &= \lambda m.\lambda n.\lambda f.(m\ (n\ f))\end{aligned}$$

2. Vérifier que `+ 2 3` $= 5$, `* 2 3` $= 6$, et `* 2 0` $= 0$.

On peut encoder l'algèbre de Boole :

$$\begin{aligned}\texttt{t} &= \lambda x.\lambda y.x\\ \texttt{f} &= \lambda x.\lambda y.y\\ \texttt{not} &= \lambda b.\lambda x.\lambda y.(b\ y\ x)\end{aligned}$$

3. Vérifier que `not t` $=$ `f` et `not f` $=$ `t`.

4. Construire le λ-terme d'une fonction `if0` à un argument qui retourne `t` (true) ou `f` (false) suivant si son entrée vaut 0 ou non (conseil : relire la définition des entiers `n`)

Le `if0` nous permet de faire des **branchements conditionnels**, ce qui sera très utile.
On peut créer des paires :

$$\begin{aligned}\texttt{pair} &= \lambda x.\lambda y.\lambda f.(f\ x\ y)\\ \texttt{fst} &= \lambda p.(p\ \lambda x.\lambda y.x)\\ \texttt{snd} &= \lambda p.(p\ \lambda x.\lambda y.y)\end{aligned}$$

5. Que donnent `fst` (`pair` a b) et `snd` (`pair` a b) ?

Grâces aux paires nous pouvons définir une soustraction, en commençant par la fonction prédécesseur `p` qui n'est pas évidente car nous n'avons pas de moyen « d'enlever un f » ou de « remplacer un f par rien ». L'astuce consiste à partir de `pair 0 0` et à avancer durant n pas en copiant la seconde valeur dans la première et en incrémentant la seconde valeur : `pair 0 0` puis `pair 0 1` puis `pair 1 2` puis `pair 2 3` etc, durant `n` étapes, puis de prendre le premier élément de la paire qui sera le terme `n-1`.

6. Construire une fonction `step` qui à `pair m n` associe `pair n n+1`.

7. Construire la fonction prédécesseur `p` (conseil : relire la définition des entiers `n`).

8. En déduire un opérateur de soustraction.

En λ-calcul, les **structures itératives** utilisent la **récursion**, comme en programmation fonctionnelle (Lisp, Haskell, OCaml...).

9. Écrire le pseudo-code d'une procédure sans boucle `for` ni `while` qui calcule la fonction factorielle sur les entiers naturels.

En suivant nos raisonnements, on voudrait calculer la factorielle avec le λ-terme :

$$\texttt{fact} = \lambda n.\texttt{if0}\ n\ \texttt{1}\ (\star\ n\ (\texttt{fact}\ (\texttt{p}\ n))) \qquad \text{⚠}$$

or il faut attendre de définir `fact` avant de pouvoir l'utiliser dans la définition de... `fact`.
Une astuce consiste à donner en entrée à `fact` une copie de lui-même.

10. Écrire `fact` qui commence par $\lambda f.\lambda n...$ et s'utilise avec `fact fact` n.

Ça marche, mais c'est un peu lourd... heureusement, Haskell Curry a introduit le λ-terme :

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

Cet opérateur de point fixe `Y` permet de définir des fonctions récursives dont le nombre d'itération est non borné. Il partage la structure du terme $(\lambda x.x\ x)(\lambda x.x\ x)$ qui se réduit en lui-même (comme une fonction récursive qui se rappelle elle-même). On l'appelle aussi combinateur `Y`.

11. En quoi se réduit (`Y` t) ?

12. On en déduit que (`Y` t) est le point fixe d'une fonction. De quelle fonction ?

Ainsi, en définissant un λ-terme $t = \lambda f.\lambda x.t'$, on obtient que (`Y` t) est équivalent à t (`Y` t) qui donne $\lambda x.t'[f := (\texttt{Y}\ t)]$. En d'autres termes, (`Y` t) donne le λ-terme t où son premier argument f est (`Y` t), que l'on peut appliquer dans t' pour implémenter une procédure récursive. Joie.

13. Utiliser `Y` pour définir la fonction factorielle notée `fact`.

Équivalence avec les machines de Turing

Il est simple de se convaincre que les machines de Turing sont capables de simuler le λ-calcul : on peut écrire le code d'une machine de Turing (un programme) qui applique les règles de réduction à un λ-terme écrit sur le ruban, jusqu'à atteindre une forme normale.
On peut également traduire une machine de Turing M en un λ-terme t_M, tel que t_M appliqué à une configuration c (également un λ-terme) se réduit en t_M appliqué à la configuration c' obtenue en une étape de M à partir de c (c'est-à-dire $c \vdash_M c'$). Pour écrire t_M il faut des branchements conditionnels (`if0`) suivant l'état courant et le symbole sous la tête, et un appel récursif (combinateur `Y`).
Pour construire un λ-terme correspondant à une configuration c, il faut stocker le contenu du ruban dans une liste potentiellement infinie. En suivant l'idée des paires, on peut définir des n-uplets, et des listes :

$$\lambda a_1.\ldots.\lambda a_n.\lambda f.(f\ a_1\ \ldots\ a_n)$$

$$\begin{aligned} [] &= \lambda x.\lambda y.y \\ [a_1, a_2, \ldots, a_n] &= \lambda x.\lambda y.(x\ (\texttt{pair}\ a_1\ [a_2, \ldots, a_n])) \end{aligned}$$

Pour définir les listes infinies, il faut utiliser le combinateur `Y`.

14. Comment représenter une liste infinie de λ-termes `a` ?

Références

[1] J. Garrigues, Cours : *Computability and lambda calculus*, 2013.

Exercice 2. *Fonctions μ-récursives*

Le principe des fonctions μ-récursives est de décrire des fonctions algorithmiquement, c'est-à-dire avec une procédure indiquant comment les calculer [1]. On parle ici de définir les fonctions de $\mathbb{N}$ dans $\mathbb{N}$ qui sont « calculables ». Les fonctions sont construites à partir de fonctions de base, et d'opérateurs pour construire de nouvelles fonctions. Les fonctions primitives récursives sont une étape intermédiaire vers la définition des fonctions μ-récursives.

Fonctions primitives récursives

Les fonctions primitives récursives sont des fonctions de $\mathbb{N} \to \mathbb{N}$, définies inductivement à l'aide des fonctions de base suivantes :

- pour tous p et n, la fonction constante $f(x_1, \ldots, x_p) = n$,
- la fonction successeur, telle que $S(x) = x + 1$,
- pour tous p et $1 \leq i \leq p$, la i^{e} projection $P_i^p(x_1, \ldots, x_p) = x_i$,

et des opérateurs suivants :

- un opérateur de composition $\circ$ des fonctions, qui permet de construire

$$h \circ (g_1, \ldots, g_m) = f \text{ avec } f(x_1, \ldots, x_p) = h(g_1(x_1, \ldots, x_p), \ldots, g_m(x_1, \ldots, x_p)),$$

  à partir d'une fonction h d'arité m, et de m fonctions $(g_i)_{1 \leq i \leq m}$ d'arité p,
- un opérateur de récursion primitive ρ, qui permet de construire

$$\begin{aligned} \rho(g, h) = f \quad \text{avec} \quad & f(0, x_1, \ldots, x_p) = g(x_1, \ldots, x_p) \\ \text{et} \quad & f(y+1, x_1, \ldots, x_p) = h(y, f(y, x_1, \ldots, x_p), x_1, \ldots, x_p), \end{aligned}$$

  à partir d'une fonction g d'arité p, et d'une fonction h d'arité $p + 2$.

Formellement, l'ensemble des fonctions primitives récursives est défini comme le plus petit ensemble de fonctions contenant les fonctions de base et clos par les opérateurs $\circ$ et ρ.

L'intuition derrière l'opérateur de récursion primitive ρ est que :

- g est la condition initiale
- h est l'étape de récursion, dans laquelle on a accès à la valeur calculée précédemment, $f(y, x_1, \ldots, x_p)$.

1. Montrer que l'addition est primitve récursive, en définissant $add : \mathbb{N}^2 \to \mathbb{N}$.

2. Montrer que la multiplication est primitive récursive.

3. Montrer que la soustraction (qui vaut 0 si le résultat est négatif) est primitive récursive.

On pourrait continuer loin, par exemple le test de primalité est primitif récursif.

Ackermann

On peut définir de nombreuses fonctions grâce à la récursion primitive. Cependant, il semble leur manquer quelque chose (qui sera comblé par l'ajout de l'opérateur μ qui donnera les fonctions μ-récursives), car il existe des fonctions qui semblent intuitivement « calculables », mais qui ne sont pas récursives primitives. Un tel exemple est la fonction d'Ackermann, que nous noterons $A : \mathbb{N}^2 \to \mathbb{N}$, et qui est définie comme suit.

$$\begin{aligned} &\text{pour tout } y \in \mathbb{N}, A(0, y) = y + 1 \\ &\text{pour tout } x \in \mathbb{N}, A(x + 1, 0) = A(x, 1) \\ &\text{pour tout } x, y \in \mathbb{N}, A(x + 1, y + 1) = A(x, A(x + 1, y)) \end{aligned}$$

4. Calculer $A(1, 2)$ et $A(4, 3)$.

5. Argumenter du fait que le calcul de la fonction A termine pour tout (x, y).

Il se trouve que la fonction d'Ackermann A croît plus vite que toute fonction primitive récursive, dans le sens suivant.

Théorème. *Pour toute fonction primitive récursive $f : \mathbb{N}^p \to \mathbb{N}$, il existe un entier t tel que pour tous $x_1, \ldots, x_p$ on ait $f(x_1, \ldots, x_p) < A(t, \max_i x_i)$.*

6. Comment peut-on en déduire que la fonction A n'est pas primitive récursive ?

Ce théorème est démontré en considérant l'ensemble de fonctions $\mathcal{A}$ suivant.

$$\mathcal{A} = \{f \mid \exists t : \forall x_1, \ldots, x_p : f(x_1, \ldots, x_p) < A(t, \max_i x_i)\}$$

7. Que représente $\mathcal{A}$?

8. Comment démontrer que $\mathcal{A}$ contient l'ensemble des fonctions primitives récursives ?

Remarque : en récursion primitive tout termine car on a uniquement des « boucle *for* bornées ». L'ajout d'une « boucle *while* » donne les fonctions μ-récursives. Quand vous programmez, essayez d'éviter au maximum les récursions non-bornées.

Fonctions μ-récursives

Les fonctions μ-récursives sont obtenues en ajoutant :

— un opérateur de minimisation μ, qui permet de construire

$$\mu(f)(x_1, \ldots, x_p) = z \iff \begin{array}{l} f(z, x_1, \ldots, x_p) = 0 \text{ et} \\ f(i, x_1, \ldots, x_p) > 0 \text{ pour tout } i \in \{0, \ldots, z-1\}, \end{array}$$

à partir d'une fonction f d'arité $p + 1$;

et en prenant encore une fois le plus petit ensemble de fonctions contenant les fonctions de base et clos par les opérateurs $\circ$, ρ et μ.

L'intuition derrière l'opérateur de minimisation μ est de rechercher, en partant de 0 et en augmentant, le plus petit z tel que f retourne 0. Il est également appelé *unbounded search operator*. Si un tel argument z n'existe pas, alors la recherche ne termine pas. Nous définissons donc désormais des fonctions *partielles* (toutes les fonctions récursives primitives sont *totales*).

9. Donner une fonction partielle μ-récursive dont le domaine est vide.

Équivalence avec les machines de Turing

Toute fonction μ-récursive est calculable par une machine de Turing, et toute fonction calculable par une machine de Turing est μ-récursive. C'est technique, mais on peut effectivement convertir une définition de fonction μ-récursive, en une machine de Turing qui calcule la même fonction, et inversement.

10. « C'est technique » est très évasif... souhaitez-vous en discuter un peu ?

On retrouve nos ensembles récursif et récursivement énumérable de la façon suivante.

Définition. Un langage $A \subseteq \mathbb{N}^p$ est récursif ssi sa fonction caractéristique[1] $\chi(A)$ est μ-récursive et totale[2]. Un langage $A \subseteq \mathbb{N}^p$ est semi-décidable ssi c'est le domaine de définition[3] d'une fonction partielle μ-récursive.

Références

[1] R. Cori et D. Lascar, *Logique mathématique 2 - Fonctions récursives, théorème de Gödel, théorie des ensembles, théorie des modèles*, Dunod, 2003.

1. $\chi(A)(x_1, \ldots, x_p) = 1$ si $(x_1, \ldots, x_p) \in A$, et 0 sinon.
2. C'est-à-dire définie sur tout $\mathbb{N}^p$, ce qui correspond au fait que le calcule termine toujours !
3. En considérant que si une valeur est retournée, c'est 1.