

TD 03 – 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.

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 $y \notin \text{Free}(t)$ (α -conversion)¹.

La notation $t[x := t']$ est le terme t dans lequel on a substituée toute occurrence de x par t' . 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.

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 :

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

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

On peut encoder l'algèbre de Boole :

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

1. $\text{Free}(t)$ est l'ensemble des variables libres dans t (contrairement à celles liées par une lambda abstraction), et cette condition sert à éviter un phénomène appelé *capture de variable*.

3. Vérifier que $\text{not } t = f$ et $\text{not } f = t$.
4. Construire le λ -terme d'une fonction qui retourne t (true) ou f (false) suivant si son entrée vaut 0 ou non.

On peut créer des paires :

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

5. Que donnent $\text{fst} (\text{pair } a \ b)$ et $\text{snd} (\text{pair } a \ b)$?

Grâce 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 $\text{pair } 0 \ 0$ et à avancer durant n pas en copiant la seconde valeur dans la première et en incrémentant la seconde valeur : $\text{pair } 0 \ 0$ puis $\text{pair } 0 \ 1$ puis $\text{pair } 1 \ 2$ puis $\text{pair } 2 \ 3$ 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 $\text{pair } m \ n$ associe $\text{pair } n \ n+1$.
7. Construire la fonction prédécesseur p (indice : revoir la définition des entiers n).
8. En déduire un opérateur de soustraction.

En suivant l'idée des paires, on peut définir des n -uplets, et des listes :

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

Pour définir les listes infinies, nous avons besoin de l'opérateur de point fixe

$$Y = (\lambda f. \lambda x. (x \ (f \ f \ x))) \ (\lambda f. \lambda x. (x \ (f \ f \ x)))$$

9. En quoi se réduit $(Y \ t)$?
10. Comment représenter une liste infinie de λ -termes a ?

L'opérateur de point fixe Y permet aussi de définir des fonctions récursives dont le nombre d'itération est non borné, comme la fonction factorielle $Y \ (\lambda f. \lambda n. \text{if0 } n \ 1 \ (* \ n \ (f \ (p \ n))))$. L'opérateur de point fixe Y 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 également combinateur Y .

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 ce que cela ne soit plus possible.

On peut également traduire une machine de Turing M s'exécutant à partir d'une entrée w en un λ -terme, dont la réduction correspond à l'exécution de M sur le mot w . Une telle traduction applique (λ -application) les transitions (qui sont des λ -abstractions) à des configurations pour donner de nouvelles configurations (une telle construction demande de bien comprendre l'opérateur de point fixe Y).

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 $\mathbb{N} \rightarrow \mathbb{N}$, définies inductivement à l'aide des fonctions de base suivantes :

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

et des opérateurs suivants :

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

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

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 réursion primitive ρ , qui permet de construire

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

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 réursion primitive ρ est que :

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

1. Montrer que l'addition est primitive récursive, en définissant $add : \mathbb{N}^2 \rightarrow \mathbb{N}$.
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 réursion 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 $A : \mathbb{N}^2 \rightarrow \mathbb{N}$, 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 réursive $f : \mathbb{N}^p \rightarrow \mathbb{N}$, il existe un entier t tel que pour tous x_1, \dots, x_p on ait $f(x_1, \dots, x_p) < A(t, \max_i x_i)$.

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

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

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

7. Que représente \mathcal{A} ?

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

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

Fonctions μ -réursives

Les fonctions μ -réursives sont obtenues en ajoutant :

— un opérateur de minimisation μ , qui permet de construire

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

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 réursives primitives sont *totales*).

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

Équivalence avec les machines de Turing

Toute fonction μ -réursive est calculable par une machine de Turing, et toute fonction calculable par une machine de Turing est μ -réursive. C'est technique, mais on peut effectivement convertir une définition de fonction μ -réursive, 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 réursif et réursivement énumérable de la façon suivante.

Définition. Un langage $A \subseteq \mathbb{N}^p$ est réursif ssi sa fonction caractéristique² $\chi(A)$ est μ -réursive et totale³. Un langage $A \subseteq \mathbb{N}^p$ est semi-décidable ssi c'est le domaine de définition⁴ d'une fonction partielle μ -réursive.

Références

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

2. $\chi(A)(x_1, \dots, x_p) = 1$ si $(x_1, \dots, x_p) \in A$, et 0 sinon.

3. C'est-à-dire définie sur tout \mathbb{N}^p , ce qui correspond au fait que le calcul termine toujours !

4. En considérant que si une valeur est retournée, c'est 1.