Introduction au λ-calcul

Aurelien Arena

1 Le λ-calcul

Le λ-calcul est un formalisme logique inventé par Church dans les années 1930 pour étudier des notions comme celles de fonction, de calculabilité, de récursion etc... Dans ce cadre, la fonction tient une place prépondérante et prend une acception particulière.

Selon l'approche traditionnelle, une fonction, par exemple $f : X \to Y$, est envisagée sur un mode ensembliste comme une relation binaire $f \subseteq X \times Y$ vérifiant la propriété de fonctionnalité (à chaque élément de X correspond au plus un élément de Y). C'est donc une correspondance "statique" entre éléments.

Au contraire, le λ-calcul met en avant la dimension procédurale et constructive de la fonction en l'assimilant à un opérateur (par exemple U) s'appliquant à un opérande (par exemple V) pour construire un résultat (par exemple UV). Ce formalisme est donc adapté au modèle théorique de l'ordinateur et à son implémentation pratique qui consiste à stocker et exécuter des procédures mécaniques de calcul et des algorithmes de manière effective.

La première approche est dite "extensionnelle" tandis que la seconde est dite "intensionnelle". Cf. Frege Sens et dénotation" [1]. Le langage du λ-calcul

Le λ-calcul repose sur un langage formel (noté ici L et dont les éléments sont appelés des λ-termes) défini à partir d'un ensemble de variables $\{x, y, z, \ldots\}$ et un ensemble de constantes $\{a, b, c, \ldots\}$ ainsi que les règles récursives suivantes:

- Les variables et les constantes sont des λ-termes appelés atomes
- Si U et V sont des λ-termes, alors (UV) est un λ-terme appelé une application
- Si x est une variable et U un λ-terme, alors $(\lambda x.U)$ est un λ-terme appelé une abstraction

Remarque Les λ-termes sont désignés par des lettres majuscules $U, V, X \ldots$ Par ailleurs, si U est un terme bien formé du langage L, on note $U \in L$ puisque la grammaire ci-dessus définit un ensemble.

Remarque Si l'ensemble des constantes est vide, le langage est dit pur. Dans l'abstraction $(\lambda x.U)$, U est le corps et λx un opérateur d'abstraction qui

va abstraire une variable x de U (x devient un paramètre formel) pour en faire une entité opératoire intrinsèquement insaturée (une fonction), c'est-à-dire qui "attend" un argument. Une abstraction ne désigne donc pas le résultat de l'éxécution d'une procédure mais la procédure elle-même.

Convention Afin d'alléger l'écriture, il est possible d'omettre les parenthèses sachant que l'opération d'application est associative à gauche. D'un point de vue pratique, les égalités suivantes seront vérifiées:

- $(UV) = UV$
- $UVXY = (((UV)X)Y)$
- $\lambda x.\lambda y.\lambda z.U = (\lambda x.(\lambda y.(\lambda z.U)))$

Exemple 1 Les expressions suivantes sont des expressions bien formées du langage L ci-dessus.

- $\lambda x.xy$
- $\lambda x.y$
- $\lambda x.\lambda y.(y\lambda z.(zx))$

Exemple 2 Si l'ensemble des constantes servant à construire le langage L contient un opérateur d'addition noté '+', alors les expressions suivantes (en notation infixée) sont également valides.

- $\lambda x.a$ (la fonction constante)
- $\lambda x.\lambda y.(x + y)$ (la fonction somme de deux variables)
- $\lambda x.(x + a)$ (la fonction somme d'une variable avec une constante)

Par exemple, une abstraction $\lambda x.(x + 1)$ s'écrira en notation traditionnelle $x \longmapsto x + 1$

Remarque Un λ-terme U a une occurence dans un autre λ-terme V lorsque pour X, Y des λ-termes (éventuellement vides), on a $V = XUY$

Remarque Si une abstraction $U = \lambda x.Y$ a une occurence dans un terme V, la portée de l'opérateur d'abstraction λx est le corps Y dans le λ-terme U. La portée d'une variable est délimitée par l'opérateur λ qui la précède strictement, indépendamment du contexte plus global. Par exemple, λx.[xλy.[xyλy.[xy]]]. Chaque couleur figure la liaison entre un opérateur d'abstraction et une variable, et les crochets figurent la portée.

Définition L'ensemble $VL(t)$ des variables libres d'un λ-terme t se définit récursivement par les règles suivantes.

- Si $t = x$, alors $VL(t) = x$
- Si $t = UV$, alors $VL(t) = VL(U) \cup VL(V)$
- Si $t = \lambda x.U$, alors $VL(t) = VL(U)/\{x\}$

Un λ-terme qui n'a aucune variable libre est dit clos. Pour un λ-terme les variables qui ne sont pas libres sont dites liées (par un abstracteur).

Il est important de considérer le contexte (c'est-à-dire le λ-terme) dans lequel on qualifie une variable de libre ou liée. Une variable peut-être en effet être libre dans un contexte et liée dans un autre.

Par exemple, $VL(\lambda x.xy) = \{y\}$ mais $VL(x\lambda x.xy) = \{x, y\}$. Le λ-terme $\lambda x.xy$ possède une occurence dans le λ-terme plus global $x\lambda x.xy$. Un λ-terme lorsqu'il est clos est également appelé un combinateur. Les combinateurs et leurs propriétés sont étudiés dans le cadre de la logique combinatoire de Schönfinkel et Curry. Par exemple, voir [2].

2 Notion d'ordre supérieur

De par la définition du langage L, rien n'empêche un opérateur (une fonction) d'opérer sur un autre opérateur et/ou de construire un résultat qui est lui-même de type opérateur. Cette particularité en fait un système dit d'ordre supérieur. Par exemple, l'expression $(\lambda x.x)(\lambda y.y)$ est valide.

Remarque La logique des prédicats de premier ordre est un autre système logique qui a pour domaine de quantification des entités individuelles. Une logique d'ordre deux, elle, peut opérer sur des entités qui elles-mêmes opèrent déjà sur des entités individuelles et ainsi de suite...

3 La curryfication

Les opérateurs évoqués jusqu'à présent sont des opérateurs unaires, c'est-à-dire qu'il ne peuvent s'appliquer qu'à un seul argument à la fois. Or, en programmation il est courant de manipuler des opérateurs n-aires.

En lien avec le caractère d'ordre supérieur de ce langage, un opérateur n-aire à plusieurs arguments peut se définir par des applications successives d'opérateurs unaires. Par exemple, un opérateur binaire prendra d'abord le premier argument puis renverra un opérateur intermédiaire qui prendra le second argument pour enfin construire le résultat final. Ce processus est appelé la curryfication.

Remarque Ce mécanisme n'est pas celui de composition de fonctions car ici ce qui est retourné est une fonction et non pas le résultat d'une fonction comme dans une composition.

Exemple Ainsi, dans la chaine d'applications suivante, 3 et 4 sont respectivement deux arguments "consommés" successivement.

$$\lambda x.\lambda y.(x+y)3\ 4 \longrightarrow_\beta \lambda y.(3+y)4 \longrightarrow_\beta (3+4) \tag{1}$$

Remarque Les notations et le mécanisme d'évaluation des λ-termes seront explicités dans les sections La substitution et La β-réduction.

Illustrons ce mécanisme avec JAVA8.

```
import java.util.function.Function;

public class Test {
    public static void main(String[] args) {
        Integer resultat1 = addNonCur(3, 4);
        Function<Integer, Integer> addTrois = addCur(3);
        Integer resultat2 = addTrois.apply(4);
    }

    /**
     * Version non curryfiee
     */
    public static Integer addNonCur(Integer x, Integer y) {
        return x + y;
    }

    /**
     * Version curryfiee
     */
    public static Function<Integer, Integer> addCur(Integer x) {
        return (Integer y) -> x + y;
    }
}
```

Dans cet exemple, `addTrois` joue le rôle d'opérateur unaire intermédiaire et est formellement représenté par le λ-terme $\lambda y.(3+y)$. Attention! `addCur` est également un opérateur unaire et est représenté par le λ-terme $\lambda x.\lambda y.(x+y)$.

En termes techniques, JAVA8 introduit dans le JDK la notion de type fonctionnel via le mécanisme d'interface fonctionnelle dont java.util.function.Function$<T, R>$ est un exemple. Cette interface utilise le typage paramétrique (les Generics) pour typer les opérateurs.

4 La substitution

Une abstraction est un λ-terme dénotant un opérateur (une fonction) qui s'appliquent potentiellement à un argument. L'évaluation d'une abstraction revient à déterminer la valeur résultante de l'application de l'opérateur correspondant à un opérande particulier.

Exemple L'opérateur $(\lambda x.x + 1)$ appliqué à l'opérande 'a' s'écrira $(\lambda x.x + 1)a$. Cette dernière expression aura pour valeur une fois évaluée $(a + 1)$.

L'évaluation d'une abstraction (cf. La β-réduction) passe par le remplacement de toutes les occurences de ses variables liées. A cette fin, l'opération de substitution est introduite.

Définition La substitution se formalise par une fonction $[V/x] : L \to L$ se définissant pour S, T, U, V des λ-termes et x, y, z des variables, de la manière suivante par récursion:

- Si $U = x$, alors $[V/x]U = V$
- Si U est une constante ou une variable différente de x, alors $[V/x]U = U$
- Si $U = (ST)$, alors $[V/x]U = ([V/x]S[V/x]T)$
- Si $U = \lambda x.S$, alors $[V/x]U = U$
- Si $U = \lambda y.S$ avec $x \neq y$ et $(y \notin VL(V)$ ou $x \notin VL(S))$, alors $[V/x]U = \lambda y.[V/x]S$
- Si $U = \lambda y.S$ avec $x \neq y$ et $y \in VL(V)$ et $x \in VL(S)$, alors $[V/x]U = \lambda z.[V/x][z/y]S$ avec $z \notin VL(V) \cup VL(S)$

La notation $[V/x]U$ dénote le λ-terme résultant de la substitution où V est un λ-terme substituant et x une variable substituée dans le λ-terme U selon les règles énoncées. Cette fonction de substitution agit au niveau du métalangage par rapport à L mais n'est pas un constituant du langage L lui-même (cf. définition de L). Elle met simplement en correspondance des éléments de L.

Remarque Ces règles ont pour but de fournir une gestion systématique du remplacement des variables libres en évitant notamment le téléscopage qui changerait la sémantique de l'expression. Par exemple, le cas 6. renomme la variable liée y en z ce qui ne pose pas de problème puisque le renommage ne change pas la sémantique. Par exemple, $\lambda x.(x + 1)$ et $\lambda y.(y + 1)$ dénotent la même fonction.

Exemple Le λ-terme $(x + 1)$ dans lequel on remplace x par 'a' se note $[a/x](x + 1)$ et dénote le λ-terme $(a + 1)$.

5 La β-réduction

La β-réduction formalise l'opération d'application (ou encore d'évaluation). L'opération de substitution étant établie, il est désormais possible d'écrire la relation existante entre un opérateur appliqué à un opérande et le résultat de cette évaluation.

$$(\lambda x.U)V \longrightarrow_\beta [V/x]U \tag{2}$$

Cette relation est appelée une β-réduction et consiste pour une abstraction $(\lambda x.U)$ appliquée à V à remplacer toutes les occurences de la variable x liées par l'opérateur d'abstraction λx par le λ-terme V dans le corps U.

Attention! le λ-terme U peut lui-même posséder une occurence d'abstraction elle aussi construite à partir d'un second opérateur d'abstraction λx. Les variables x tombant sous la portée de ce sous-terme ne seront pas remplacées en vertu du cas 4 des règles de substitution.

Exemples
$(\lambda x.x\lambda y.\lambda x.x + y)a \longrightarrow_\beta [a/x]x\lambda y.\lambda x.x + y$
$\longrightarrow_\beta [a/x]x[a/x]\lambda y.\lambda x.x + y$ (cas 3)
$\longrightarrow_\beta a\lambda y.[a/x]\lambda x.x + y$ (cas 1, 5)
$\longrightarrow_\beta a\lambda y.\lambda x.x + y$ (cas 4)

Ou encore l'exemple ci-dessus pour illustrer la curryfication,

$$\lambda x.\lambda y.(x + y)3\ 4 \longrightarrow_\beta [3/x]\lambda y.(x + y)4 \longrightarrow_\beta [4/y][3/x](x + y) \tag{3}$$

avec $[4/y][3/x](x + y)$ dénotant le λ-terme $(3 + 4)$.

6 Le typage

La notion de type a été introduite la première fois dans le cadre mathématique au début du XXe siècle. Le typage des expressions est alors utilisé pour évincer du domaine du formulable des expressions paradoxales (paradoxe de Russel en théorie des ensembles).

L'informatique quant à elle, utilise largement ce mécanisme. Asserter qu'une entité est d'un type particulier revient à déclarer qu'elle possède un ensemble de propriétés précises. On note $e : T$ une entité e à laquelle est assigné le type T. Toutes les entités d'un même type forment une classe homogène. Les propriétés impliquées par un type sont par exemple des propriétés intrinsèques comme "être un entier", "être une chaîne de caractères" mais aussi des propriétés combinatoires comme "ne se divise pas par 0 ", "se concatène avec une chaine de caractères" ou bien "s'applique à une entité d'ordre inférieur".

Le typage informatique guide également l'optimisation matérielle du stockage et de la manipulation de variables de différentes sortes (ex: entiers, pointeurs, flottants...), il aide à l'organisation modulaire des programmes, à l'encapsulation

des données, à la détection d'erreurs le plus en amont possible etc... Voir par exemple [3] ou chap. 6 dans [4] pour une présentation approfondie du typage informatique.

Définition De manière générale, un système de types, pour un langage, consiste en un ensemble de types et de règles qui permettent de déterminer pour une entité typable,

- son type s'il est inconnu, ou
- de déterminer une erreur de typage si l'entité est déjà associée à un type.

Remarque Une entité typable peut être une fonction, un argument, un opérateur d'affectation, un relateur de comparaison etc...

La notion de type est également utilisée par d'autres disciplines comme la philosophie ou bien la linguistique formelle. Par exemple, les grammaires catégorielles d'Ajdukiewicz et Bar-Hillel, fondées sur le λ-calcul typé, est un formalisme syntaxique au sein duquel les types sont interprétés comme des types syntaxiques qui régulent les arrangements des unités lexicales.

7 Le λ-calcul simplement typé

Cette section présente une modalité particulière de typage du λ-calcul qui est connue sous le nom de "typage à la Church". Ce dernier procède en trois temps:

1. définition de l'ensemble des types,
2. définition d'un langage, et
3. définition d'une fonction de typage qui assignera (dans ce cadre) un type unique à chaque λ-terme du langage.

Voir "Lambda Calculi with Types" dans [5].

7.1 Les types

Dans le cadre du λ-calcul typé un type particulier est introduit: le type fonctionnel. Il sert à caractériser les entités de nature fonctionnelle que sont les abstractions. Il précise par ailleurs le type de l'entité attendue par l'abstraction et le type de l'entité construite par son application à un opérande. Ainsi, le type fonctionnel $T_1 \rightarrow T_2$ décrit une classe homogène d'entités fonctionnelles prenant un argument de type T_1 pour construire une entité de type T_2.

Définition L'ensemble des types (noté ici $Types$) se définit à partir d'un ensemble de types de base (par exemple, $\{int, bool, string\}$) et de constructeurs de types dérivés selon les règles suivantes:

- Les types de base sont des types
- Si T_1 et T_2 sont des types, alors $T_1 \to T_2$ est un type
- Si $T_1, \ldots, T_n$ sont des types, alors $T_1 \times \ldots \times T_n$ est un type

Par exemple, l'assignation $\langle x_1, \ldots, x_n \rangle : T_1 \times \ldots \times T_n$ signifie que la composante x_i du tuple est de type T_i.

Remarque L'introduction du constructeur de type "produit cartésien" '$\times$' ne change pas fondamentalement la nature du système car comme mentionné précédemment, le produit peut se réduire à la fonctionnalité par l'opération de curryfication.

Exemples
$AND : (bool \times bool) \to bool$ (opérateur booléen ET)
$OR : (bool \times bool) \to bool$ (opérateur booléen OU)
$+ : (int \times int) \to int$ (non curryfié)
$+ : (int \to (int \to int))$ (curryfié)

7.2 Le langage

Cette section se base sur une variante du λ-calcul incluant: 1) le typage simple et, 2) le λ-terme "tuple".

Remarque Le qualificatif "simple" désigne un système de typage qui n'inclut pas les extensions potentielles existantes comme les types polymorphes ou récursifs par exemple.

Définition La syntaxe de ce langage (noté L_T) se définit de la manière suivante, également à partir d'un ensemble de variables et de constantes.

- Les variables et les constantes sont des λ-termes appelés atomes
- Si $U_1, \ldots, U_n$ sont des λ-termes, alors $\langle U_1, \ldots, U_n \rangle$ est un λ-terme appelé un tuple
- Si U est un λ-terme, alors $\pi_i U$ est un λ-terme appelé une projection
- Si U et V sont des λ-termes, alors (UV) est un λ-terme appelé une application
- Si x est une variable et U un λ-terme, alors $(\lambda x : T.U)$ est un λ-terme appelé une abstraction

Remarque Dans la règle 5, l'assignation du type $T \in Types$ à la variable liée x renseigne sur le type de l'argument attendu par le λ-terme général $\lambda x : T.U$ qui est une abstraction. Cette information en fait un langage explictement typé en opposition à un langage implicitement typé qui peut opérer des inférences

pour déterminer le type d'une unité. Au même titre qu'une abstraction, une projection peut faire l'objet d'une évaluation. Ainsi, le λ-terme $\pi_i \langle U_1, \ldots, U_n \rangle$ se réduit au λ-terme U_i. L'opérateur de projection π_i appliqué à un tuple renvoie sa i-ème composante.

Exemple Ci-dessous un exemple de réduction d'expressions utilisant les nouveaux constructeurs de λ-termes.

$$(\lambda x : int \times int.\pi_2 x) \langle 5, 4 \rangle \longrightarrow_\beta \pi_2 \langle 5, 4 \rangle \longrightarrow_\beta 4 \tag{4}$$

7.3 Les règles d'assignation

Les règles d'assignation des types aux expressions sont en général présentées sous forme de règles d'inférence (prémisse / conclusion) pour inscrire ce processus dans un cadre déductif (cf. chap 3, [6]) qui permet d'imbriquer des instances de règles formant des arbres de dérivation permettant d'obtenir l'information recherchée.

Afin de mettre en œuvre les procédures de détermination des types respectifs des λ-termes du langage L_T, il faut disposer d'informations sur les types des variables libres présentes dans ces termes. Les variables libres n'étant pas les mêmes selon le terme considéré, l'ensemble des hypothèses de typage est variable. D'où la notion de contexte de typage.

Définition Dans le cadre du langage typé présenté ici, une assignation $U : T$ implique $T \in Types$ et $U \in L_T$ et signifie que le λ-terme U est de type T.

Définition Un contexte de typage consiste en un ensemble $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\}$ d'hypothèses d'assignation. L'expression $\Gamma \vdash U : T$ signifie que les types des variables libres du λ-terme U sont définis par le contexte de typage Γ, et que dans ce même contexte U est de type T (en terminologie logique, "$U : T$ est une conséquence de Γ").

On note $\Gamma, x : T$ pour $\Gamma \cup \{x : T\}$, qui signifie que le contexte Γ est étendu par l'inclusion de l'assignation $x : T$. Par ailleurs on suppose qu'une variable est typée qu'une fois.

Pour le langage L_T, les règles d'assignation sont les suivantes.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \; [T_{VAR}] \tag{5}$$

$$\frac{\Gamma, x : T_1 \vdash U : T_2}{\Gamma \vdash \lambda x : T_1.U : T_1 \to T_2} \; [T_{ABS}] \tag{6}$$

$$\frac{\Gamma \vdash U : T_1 \to T_2 \quad \Gamma \vdash V : T_1}{\Gamma \vdash UV : T_2} \; [T_{APP}] \tag{7}$$

$$\frac{\textit{pour chaque } i \quad \Gamma \vdash U_i.T_i}{\Gamma \vdash \langle U_1, \ldots, U_n \rangle : T_1 \times \ldots \times T_n} \; [T_{TPL}] \tag{8}$$

$$\frac{\Gamma \vdash U : T_1 \times \ldots \times T_n}{\Gamma \vdash \pi_i U : T_i} \; [T_{PROJ}] \tag{9}$$

Exemple Etant donné un contexte de typage $\Gamma = \{+ : (int \times int) \to int, x : int, y : int\}$ (avec '+' une constante d'opérateur dont le type est donné obligatoirement) et les règles d'assignation mentionnées ci-dessus, il est possible de fournir un arbre de dérivation correspondant au typage du λ-terme $+\langle x, y \rangle$.

$$\dfrac{\dfrac{}{\Gamma \vdash + : (int \times int) \to int} \qquad \dfrac{\dfrac{\frac{x:int \in \Gamma}{\Gamma \vdash x : int}\;[T_{VAR}] \qquad \frac{y:int \in \Gamma}{\Gamma \vdash y : int}\;[T_{VAR}]}{\Gamma \vdash \langle x,y \rangle : int \times int}\;[T_{TPL}]}{\Gamma \vdash + \langle x, y \rangle : int}\;[T_{APP}] \tag{10}$$

On a donc $\Gamma \vdash + \langle x, y \rangle : int$.

De manière générale, un terme bien typé U est un terme pour lequel il existe une assignation de type et que cette dernière est prouvable (c'est-à-dire qu'il existe un arbre de dérivation) à partir des hypothèses de Γ et des règles d'assignations. On écrira alors $\Gamma \vdash U : T$.

Dans le contexte d'un typage à la Church, seuls les termes bien typés auront un sens. Cet ensemble est donc défini de la manière suivante: $L_{BT} = \{U \in L_T \mid \exists \Gamma, T, \; \Gamma \vdash U : T\}$

References

[1] G. Frege, *Ecrits logiques et philosophiques*. Seuil, 1892.

[2] J.-P. Ginisti, *La logique combinatoire*. Paris: Presses Universitaires de France - PUF, 1997.

[3] B. C. Pierce, *Types and Programming Languages*. Cambridge, Mass: MIT Press, 2002.

[4] J. C. Mitchell, *Concepts in Programming Languages*. Cambridge, UK ; New York: Cambridge University Press, Oct. 2002.

[5] S. Abramsky, D. M. Gabbay, and S. E. Maibaum, eds., *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*. New York, NY, USA: Oxford University Press, Inc., 1992.

[6] M. S. Freund, *Logique et Raisonnement*. Paris: Ellipses Marketing, 2011.

[7] M. Bourdeau, "La genèse des grammaires catégorielles et leur arrière-plan logico-philosophique : quelques remarques," *Langages*, vol. 36, no. 148, pp. 13–27, 2002.