Lambda Calcul et Programmation Fonctionnelle (LCPF)

Zied Bouraoui

CRIL-CNRS & Univ Artois

bouraoui@cril.univ-artois.fr

http://www.cril.univ-artois.fr/~bouraoui/

Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Lambda calcul

Introduction

Le **lambda-calcul** est un langage et un système de réécriture imaginé par le mathématicien Alonzo Church en 1932.

Le lambda-calcul est un langage qui est :

- très petit : il ne comporte que deux constructions syntaxiques.
- très expressif : il est capable d'exprimer toutes les fonctions calculables.

Syntaxe

Les expressions du lambda-calcul sont appelée *lambda-expressions* ou *lambda-termes*.

Une lambda-expression est :

- soit une constante : 4, π, +, … (un nombre, un symbole ou un opérateur).

- soit une variable : x, y, z, ... (typiquement une seule lettre minuscule).

- soit une **abstraction** de la forme : (λx.M)
  où x est une variable et M est une lambda-expression.

- soit une **application** de la forme : (M N)

  où M et N sont des lambda-expressions.

Conventions de parenthésage

Les lambda-expressions sont non ambigües. Mais le grand nombre de parenthèses rend la lecture des expressions difficile :

$$(((\lambda x.(\lambda y.((+\ x)\ y)))\ 1)\ 2)$$

Conventions :

1. Les parenthèses du début et de la fin sont optionnels.
2. L'application est plus prioritaire que l'abstraction. (c.-à-d., la porté du λ va aussi loin que possible)
3. L'application est associative à gauche.

Ceci équivaux à ces quatre règles :

$$(M) \equiv M \quad (1)$$

$$\lambda x.(M\ N) \equiv \lambda x.M\ N \quad (2)$$

$$\lambda x.(\lambda y.M) \equiv \lambda x.\lambda y.M \quad (3)$$

$$(M\ N)\ O \equiv M\ N\ O \quad (4)$$

Exercice

Enlever les parenthèses inutiles de la lambda-expression :

$$(((\lambda x.(\lambda y.((+\ x)\ y)))\ 1)\ 2)$$

Règle (1) :

$$((\lambda x.(\lambda y.((+\ x)\ y)))\ 1)\ 2$$

Règle (2):

$$((\lambda x.(\lambda y.(+\ x)\ y))\ 1)\ 2$$

Règle (3) :

$$((\lambda x.\lambda y.(+\ x)\ y)\ 1)\ 2$$

Règle (4) (deux fois) :

$$(\lambda x.\lambda y.+\ x\ y)\ 1\ 2$$

Signification des lambda-expressions

Signification de l'abstraction

Une abstraction de la forme $\lambda x.M$ exprime la fonction anonyme qui à tout x associe M (M est l'image de x par cette fonction).

Dans ce cas, la lambda-expression M est appelée corps de l'abstraction.

Exemple : L'abstraction λx.x exprime la fonction qui à tout x associe x (fonction identité). (L'abstraction λy.y exprime la même fonction.)

Exemple : La fonction λx.λy.y exprime la fonction d'ordre 2 qui à tout x associe la fonction identité.

Signification de l'application

Une application de la forme M N exprime l'image de N par M.

Exemple : L'application (λx.x) 1 exprime l'image du nombre 1 par la fonction identité.

Exemple : En lambda-calcul, la constante + exprime la fonction d'ordre 2 qui à tout x associe la fonction qui à tout y associe x + y. Donc, l'expression + 1 2 ≡ (+ 1) 2 exprime le résultat de l'application de cette fonction d'ordre supérieur à 1 puis à 2.

Réduction

Notion de réduction

Intuitivement, une *réduction* en lambda-calcul est l'action de **transformer une lambda-expression en une autre plus simple et ayant la même signification**, et de répéter cette opération jusqu'à ce que la lambda-expression ne puisse plus être réduite.

Exemple: $(\lambda x.* \ 2 \ x) \ 3 \rightarrow *2 \ 3 \rightarrow 6$

De manière générale, la réduction d'une lambda-expression s'interprète comme le calcul du résultat.

Une réduction peut comporter plusieurs étapes, ces étapes décrivent le déroulement du calcul.

Exemple: $(\lambda x.\lambda y. + \ x \ y) 1 \ 2$

$$\rightarrow (\lambda y.+ \ 1 \ y) \ 2 \rightarrow + \ 1 \ 2 \rightarrow 3$$

Bêta-réduction

Une expression de la forme $(\lambda x.M)\ N$ se réduit en M où les occurrences de la variable x sont remplacées par N. C'est à dire :

$$(\lambda x.M)\ N \xrightarrow{\beta} M[x := N]$$

Exemple : $(\lambda x.\lambda y.+\ x\ y)\ 1\ 2$

$$\equiv ((\lambda x.\lambda y.+\ x\ y)1)2$$

$$\xrightarrow{\beta} \quad ((\lambda y.+\ x\ y)[x :=1])\ 2 \equiv (\lambda y.1y)\ 2$$

$$\xrightarrow{\beta} \quad (\lambda y.+\ 1y)[y:=2] \equiv +\ 1\ 2$$

Delta-réduction

La δ-réduction modélise les opérations mathématiques classiques. C'est-à-dire :

$$op\ c_1\ c_2\ \ldots\ c_n \xrightarrow{\delta} c_0$$

où op est un opérateur qui exprime une opération n-aire, chaque c_i est une constante qui exprime un argument, et c_0 est une constante qui exprime le résultat de l'opération.

Exemple :

$$+1(*2\ 3) \xrightarrow{\delta} +\ 1 \qquad 6 \xrightarrow{\delta} 7$$

Problème de capture de variable

Soit la lambda-expression (λf.λa.f a). Il s'agit de la fonction d'ordre supérieur qui reçoit une fonction et un argument et applique la fonction à l'argument.

Nous avons :

$$((\lambda f.\lambda a.f\ a)\ g)\ x)$$
$$\xrightarrow{\beta}(\lambda a.g\ a)\ x$$
$$\xrightarrow{\beta}g\ x$$

Pourtant, en remplaçant g par a nous avons :

$$((\lambda f.\lambda a.f\ a)\ a)\ x)$$
$$\xrightarrow{\beta}(\lambda a.a\ a)\ x$$
$$\xrightarrow{\beta}x\ x$$

C'est-à-dire, le résultat est incorrect.

Problème de capture de variable

La variable *a* qui apparaît dans le corps de l'abstraction exprime un autre objet que celui exprimé par la variable *a* en argument, et ces objets ont été confondus !

On dit que *a* a été **capturée** dans le corps de l'abstraction après la β-réduction.

En conséquence, la β-réduction ne peut pas toujours être appliquée.

Il faut que les variables **libres** dans l'argument ne soient pas **liées** dans le corps de l'abstraction.

Variables libres et liées

Une occurrence d'une variable x est liée dans une lambda-expression M si elle apparaît dans M à l'intérieur d'une sous-expressions de la forme λx.E.

Une occurrence d'une variable x est libre dans une lambda-expression M si elle n'est pas liée dans M.

Exemple :

$$(\lambda f.\lambda a.f\ a)\ a$$

La deuxième occurrence de *a* est liée dans l'expression. Cela veut dire que les deux occurrences de *a* désignent le même objet.

L'occurrence de a est libre dans l'expression, elle désigne un autre objet.

Retour à la bêta-réduction

Une expression de la forme (λx.M) N se réduit en M où les occurrences de la variable x sont remplacées par N. C'est à dire :

$$(\lambda x.M)\ N \xrightarrow{\beta} M[x := N]$$

s'il n'existe pas de variable dont une occurrence est libre dans N et une autre occurrence est liée dans M.

Alpha-réduction

La α-réduction est le renommage des variables dans une abstraction :

$$\lambda x.M \xrightarrow{\alpha} \lambda y.(M[x := y])$$

où y est une variable qui n'apparaît pas dans M.

Exemple :

$$\lambda a.f\ a \xrightarrow{\alpha} \lambda b.(f\ a)[a := b] \equiv \lambda b.f\ b$$

Réduction généralisée

$$(\lambda x.(\lambda y.\lambda z. + z\ y)\ 4\ x)\ 3$$

Notez que nous pouvons appliquer les réductions à des sous-expressions.

Exemple : Nous pouvons faire :

$$(\lambda x.(\lambda y.\lambda z. + z\ y)\ 4\ x)\ 3$$
$$\xrightarrow{\beta} (\lambda y.\lambda z. + z\ y)\ 4\ 3)$$

ou bien :

$$(\lambda x.(\lambda y.\lambda z. + z\ y)\ 4\ x)\ 3$$
$$\xrightarrow{\beta} (\ \lambda x.(\ \lambda z. + z\ 4\ )\ x)\ 3$$

Redex

Une sous-expression que l'on peut choisir de réduire par β-réduction (donc de la forme (λx.M) N)) est appelée redex.

Exemple : La lambda-expression de l'exemple précédent comporte donc deux redex :

$$(\lambda x.(\lambda y.\lambda z. + z\ y)_{\wedge}4\ x)_{\wedge}3$$

Le premier redex est plus à l'intérieur de l'expression que le deuxième.

Forme normale

Lorsqu'une lambda-expression **ne peut plus se réduire** autrement que par la α-réduction, alors elle est en **forme normale**.

Lorsqu'une lambda-expression M se réduit en une lambda-expression N et que N est **en forme normale**, alors **N est la forme normale de M.**

Théorème de Church-Rosser : Si une même lambda-expression M se réduit en une lambda-expression M1 (en choisissant certains redex) et en une autre lambda-expression M2 (en choisissant d'autres redex), alors il existe une autre lambda-expression N telle que M1 et M2 se réduisent en N.

Autrement dit, la réduction est **confluente**.

Forme normale

Donc, toutes les réductions d'une même lambda-expression aboutissent à une même forme normale (à des α-réductions près) **si elles terminent.**

Cependant, une réduction peut ne pas terminer :

Exemple : En choisissant toujours le redex le plus à l'intérieur :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_\wedge(\lambda z.z\ z)) \xrightarrow{\beta} (\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_\wedge(\lambda z.z\ z)) \xrightarrow{\beta} \ldots$$

la réduction ne termine pas.

En choisissant toujours le redex le plus à l'extérieur :

$$(\lambda x.\lambda y.y)_\wedge((\lambda z.z\ z)\ (\lambda z.z\ z)) \xrightarrow{\beta} \lambda y.y$$

la réduction termine en une étape.

Stratégies de réduction

Stratégies de réduction

Une stratégie de réduction définie l'ordre dans lequel les redex sont utilisés.

La plupart des langages fonctionnels utilisent l'une de ces deux stratégies (avec quelques variantes) :

- Ordre applicatif de réduction (AOR) : consiste à choisir toujours le redex interne.
- Ordre normal de réduction (NOR) : consiste à choisir toujours le redex externe.

Stratégies de réduction

Exemple : Considérons l'expression :

$$(\lambda x.(\lambda a. * a\ a)\ x)\ ((\lambda y.y)\ 2)$$

Stratégie AOR :

$$(\lambda x.(\lambda a. * a\ a)_{\wedge} x)\ ((\lambda y.y)\ 2) \xrightarrow{\beta} (\lambda x. * x\ x)\ ((\lambda y.y)_{\wedge} 2) \xrightarrow{\beta}$$

$$(\lambda x.* xx)_{\wedge} 2) \xrightarrow{\beta} *22) \rightarrow 4$$

Stratégie NOR :

$$(\lambda x.(\lambda a. * a\ a)\ x)_{\wedge}((\lambda y.y)\ 2) \xrightarrow{\beta} (\lambda a. * a\ a)_{\wedge}((\lambda y.y)\ 2) \xrightarrow{\beta}$$

$$* ((\lambda y.y)_{\wedge} 2)((\lambda y.y)2) \xrightarrow{\beta} *2((\lambda y.y)_{\wedge} 2) \rightarrow *22 \rightarrow 4$$

Passage des arguments

Avec AOR, l'argument est évalué avant l'application de la fonction. Ceci correspond au passage par valeur (le mode utilisé par le langage C, par exemple).

Avec NOR, la fonction est appliquée avant l'évaluation de l'argument. Ceci correspond au passage par nom (le mode utilisé par le langage FORTRAN, par exemple).

Évaluation

Avec AOR l'évaluation des arguments est faite dès que possible. Ceci correspond à l'évaluation affairée (eager evaluation).

Avec NOR l'évaluation des arguments est faite le plus tard possible. Ceci correspond à l'évaluation paresseuse (lazy evaluation).

AOR vs. NOR

AOR est généralement plus rapide que NOR.

La raison est qu'il arrive fréquemment la situation où nous devons réduire une expression de la forme $(\lambda x.M)\ N$ où M est sous forme normale et contient plusieurs occurrences libres de x.

Avec AOR, l'expression N est réduite en premier et donc l'argument est évalué une seule fois.

Exemple : Considérez l'expression :

$$(\lambda a.\ *\ a\ a)\ ((\lambda y.y)\ 2)$$

Avec AOR, la sous-expression $((\lambda y.y)\ 2)$ sera évalué une seule fois.

AOR vs. NOR

Pourtant, AOR ne garanti pas la terminaison :

Exemple : Rappelez-vous de l'expression :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

Nous avons vu que la réduction ne termine pas alors qu'une forme normale existe : $\lambda y.y$.

AOR vs. NOR

NOR est généralement moins efficace que AOR.

Pourtant, elle garanti la terminaison quand une forme normale existe.

Théorème de normalisation de Curry : L'ordre normal de réduction conduit à coup sûr à la forme normale lorsqu'elle existe.

Exemple : Nous avons vu que la forme normale de l'expression ci-dessous peut être trouvé avec NOR :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

AOR vs. NOR

Il existe certains cas où NOR est plus efficace que AOR.

Exemple : Soit (λx.M) N où M est sous forme normale et il n'y a aucune occurrence libre de x dans M, alors NOR permet d'attendre la forme normale en une étape.

Stratégie du langage Haskell

Le langage Haskell utilise NOR avec une variation : l'argument d'un redex est évalué au plus une fois.

Pour éviter d'évaluer plusieurs fois l'argument la β-réduction est réalisée avec partage de l'argument.