Programmation fonctionnelle
Lambda-calcul

D.E ZEGOUR
École Supérieure d'Informatique
ESI

Programmation fonctionnelle / Lambda-calcul

Sommaire

A. Introduction
- Définition
- Caractéristiques
- Schéma fonctionnel
- Quelques langages fonctionnels

B. Lambda-calcul
- Variables et fonctions en mathématiques
- La λ -notation,
- Fonction à plusieurs arguments
- Les systèmes λ-applicatifs,
- Grammaire du λ-calcul
- Notion de variables libres et liées,
- Substitution
- Changement de variable : α-conversion
- Contraction β-conversion
- Réduction
- Théorie propre du λ-calcul

Introduction

Introduit en 1930 par Alonzo Church (systeme formel pour les fonctions)

Alan Turing (étudiant de Church) a montré que lambda calcul est equivalent à la machine de Turing (puissance).

Lambda calcul = pivot de tous les langages fonctionnels

Applications : intelligence artificielle , systèmes de preuves, ...

Plusieurs langages (C++,Java, Python, ...) utilisent des lambda expressions pour exprimer des fonctions anonymes

Programmation fonctionnelle / Lambda-calcul

Définition, caractéristiques

Définition :

Le traitement est décrit par application de fonctions.

Exemple : f (g(x, h(x)), f(x-1, g(h(x-1), 2)))

Caractéristiques :

Facilité d'exprimer et de prouver les programmes.

Basé sur un système formel (Lambda-calcul)

Absence d'affectation

Absence de séquentiabilité

Absence de contrôle explicite (GOTO, EXIT). Les langages fonctionnels ne sont pas impératifs.

Calcul basé sur les fonctions.

Programmation fonctionnelle / Lambda-calcul

Schéma de programme fonctionnel

Un seul type de contrôle implicite : Si Cond alors inst1 sinon inst2 Fsi.

Instructions : conditionnelle et appel de fonction.

Exemple

Fact(n) : Si n = 0 alors 1 sinon mult(n, Fact(n-1)) Fsi

Programmation fonctionnelle / Lambda-calcul

Quelques langages fonctionnels

LISP (Processor of Listes, Mac Carthy 1960)

ISWIN (Landin 1966)

ML (1973)

FP (Backus 1978)

HOPE (1980)

MIRANDA(1985)

HASKEL (80 ->90)

OCAML (1996)

F# (2013)

Programmation fonctionnelle / Lambda-calcul

Variables et fonctions en mathématiques : Inconvénient de la notation usuelle

En mathématiques, confusion dans la notation usuelle f(x)

- la fonction elle-même
- valeur de la fonction pour x donné

Plus d'ambiguité dans les fonctionnelles (fonctions de fonctions)

f (g(x)) ? f(g(x+1) ?

La λ-notation

λx.M

A l'argument x associer la fonction M

désigne donc la fonction elle-même.

C'est donc une fonction anonyme

(Abstraction fonctionnelle)

(λx.M v) : valeur de la fonction pour la valeur v

(Application)

Ainsi, $\lambda x(x^2)$ est la fonction carré ($f(x)=x^2$)

$\lambda x(x^2)$ 2 c'est 4 (f(2))

Programmation fonctionnelle / Lambda-calcul

Fonctions à plusieurs arguments

Lambda-calcul ne peut representer que des fonctions à une seule variable

Cas des fonctions à plusieurs arguments : $\boldsymbol{\lambda x_1.\lambda x_2.\quad \lambda x_n.M}$

Notée aussi $\boldsymbol{\lambda^n x_1 x_2....x_n.M}$ ou encore $\boldsymbol{\lambda x_1 x_2....x_n.M}$

$\lambda x.\lambda y.\ (+\ x\ y) = \lambda^2 xy.\ (+\ x\ y) = \lambda xy.\ (+\ x\ y)$ --> notation usuelle $F(x, y) = x+y$

$\lambda y(+\ x\ y)$: désigne l'opération d'addition de l'argument y à x. (x est alors fixe).
c'est $f(y) = x+y$

Programmation fonctionnelle / Lambda-calcul

Les systèmes λ-applicatifs

Système λ applicatif : les seules opérations sont l'abstraction fonctionnelle (**λx.M**)et des applications (**(λx.M v)**)

L'application est représentée par une simple juxtaposition

fab signifie f appliqué à a, puis le résultat appliqué à b (fab c'est (fa)b)

fabc c'est ((fa)b)c

$(\lambda x_1.\lambda x_2. \quad \lambda x_n.M)\ a_1\ a_2 \ldots.a_n$

$(\lambda x_1.(\lambda x_2. \ldots \ (\lambda x_{n-1}.(\lambda x_n.M))..)\ a_1\ a_2 \ldots.a_n$ (fonction a un seul argument : x1)

$= (\lambda x_2. \ldots \ (\lambda x_{n-1}.(\lambda x_n.M2))..)\ a_2 \ldots.a_n$ (fonction a un seul argument : x2)

$= (\lambda x_3. \ldots \ (\lambda x_{n-1}.(\lambda x_n.M3))..)\ a_3 \ldots.a_n$ (fonction a un seul argument : x3)

…

Les fonctions à plusieurs arguments sont construites progressivement : Curryfication

Les systèmes λ-applicatifs

Principe de curryfication (Haskell Curry) : toutes les fonctions deviennent à un seul argument

Exemple 1 : $f(x, y, z) = x + y + z$

$f(x, y, z) = f'(x)(y)(z)$

Pour $x = 3$ $f'(3) = g$ $g(y)(z) = 3 + y + z$

Pour $y = 5$ $g(5) = h$ $h(z) = 3 + 5 + z$

Exemple 2 : (+ x y) c'est ((+x) y)

(+ 3 4) c'est ((+3) 4)

+3 : fonction qui rajoute 3 à son argument

Programmation fonctionnelle / Lambda-calcul

Grammaire du λ-calcul

<u>Syntaxe</u>

V = { x, y, ...} ensemble des variables.

C = { a, b, ...} ensemble des constantes.

Op = { +, -, or, and, ... }

L --> C / Op

L --> V

L --> (L L)   Associativité à gauche (Application)

L --> λV. L   Associativité à droite (Abstraction fonctionnelle)

(L L L) c'est ((L L) L)

λx.λy.L c'est λx. (λ.yL)

Exemple :

λx.x est obtenu par abstraction : f(x)=x

((λx.x)3) est obtenu par application : f(3)

Programmation fonctionnelle / Lambda-calcul

Notion de variables libres et liées

Dans l'écriture λv.e, les occurrences de la variable v dans l'expression e sont dites liées. Toutes les autres occurrences de variables sont dites libres.

Soit X, Y dans L et soit x dans V et a dans C.

Variables libres	Variables liées
Varlib(a) = {}	Varlié(a) = {}
Varlib(x) = {x}	Varlié(x) = {}
Varlib(XY)= Varlib(X) U Varlib(Y)	Varlié(XY)= Varlié(X) U Varlié(Y)
Varlib(λx.X)= Varlib(X) - {x}	Varlié(λx.X)= Varlié(X) U {x}

Programmation fonctionnelle / Lambda-calcul

Notion de variables libres et liées : Exemple

X = (λx.(λy.y) λz.x) , de la forme X = (M N) avec M = λx(λy.y) et N = λz.x

Varlib(X) = Varlib(λx.(λy.y)) U Varlib(λz.x)

<u>Varlib(λy.y)</u> - {x} U <u>Varlib(x)</u>-{z}

<u>Varlib(y)</u>- {y} - {x} U {x} - {z}

{y} - {y} - {x} U {x} - {z}
{} U {x}
{x}

Varlié(X) =Varlie(λx.(λy.y)) U Varlie(λz.x)

<u>Varlié(λy.y</u>) +{x} U <u>Varlié(x)</u>+ {z}

<u>Varlié(y)</u> + {y} + {x} U {} + {z}

{} + {y} + {x} U {} + {z}
{ x, y, z }

Remarquons que x est liée dans M mais libre dans N.
Nous pourrions écrire indifféremment X = ((λt(λy.y)) λz.x))
(changement du nom de la variable liée)

Programmation fonctionnelle / Lambda-calcul

Substitution (Sémantique)

Soit N et M dans L et x une variable libre dans M.

[N/x]M : substituer N aux occurrences libres de x dans M.

[N/x]a = a

[N/x]x = N

[N/x]y = y si y # x

[N/x](M1M2) = [N/x]M1 [N/x]M2

[N/x](λxM) = (λxM)

[N/x](λyM) =
x # y
y non libre dans N
c'est (λy [N/x] M)
y libre dans N
c'est λz [N/x] ([z/y]M)
(avec z non libre dans M)

Exemple :

Prenons M = λy.xy

Si N=t : [t/x] λy.xy = λy.ty

Si N=y : [y/x] λy.xy = λy.yy (confusion)

=[y/x] λz.xz (changement de variable)

=λz.yz

Programmation fonctionnelle / Lambda-calcul

Changement de variable liée : (Alpha-conversion)

(Alpha) : Si y n'est pas libre dans M, alors $\lambda x.M = \lambda y[y/x]M$.

Congruence :

X est congruent à Y ssi Y est le résultat d'application d'une série de changements de variables liées.

X Ξ Y ssi X--α-->Y1, Y1--α-->Y2,........Yn--α-->Y (n≥1)

La congruence est une relation d'équivalence.

Contraction : Béta-conversion

Une λ-expression de la forme **(λ x.M) N** s'appelle un Béta-radical ou Béta-rédex ou rédex.

(Beta) : (λx.M) N = [N/x] M

si λ x.M est une fonction, son application à n'importe quel N doit être vue comme le résultat de la substitution de x par N dans M

Les règles (Alpha) et (Béta) prises ensemble constituent la *Beta-conversion*.

La règle (Alpha) prise seule constitue la *Alpha-conversion*.

Programmation fonctionnelle / Lambda-calcul

Réduction

X se réduit à Y (=>) ssi Y est le résultat d'une série de alpha et beta-conversions.
=> est réflexive et transitive.

Exemples :

1. $(\lambda x.xy)F \Rightarrow Fy$
2. $(\lambda x.y)F \Rightarrow y$
3. $(\lambda x.(\lambda y.yx)z)v) \Rightarrow [v/x]((\lambda y.yx)z) == (\lambda y.yv)z \Rightarrow [z/y]\ (yv] == zv$
4. $(\lambda x.xxy)(\lambda x.xxy) \Rightarrow (\lambda x.xxy)(\lambda x.xxy)y$

$\Rightarrow (\lambda x.xxy)(\lambda x.xxy)yy \quad \Rightarrow \quad$ ect...

Programmation fonctionnelle / Lambda-calcul

Réduction [Autres exemples]

$(\lambda x.\ + \underline{x}\ 1)\ 4$

=> 5

$(\lambda x.\ + \underline{x}\ \underline{x})\ 5$

=> 10

$(\lambda x.\ (\lambda y.\ -\ y\ \underline{x}\ )\ )\ 4\ 5$

=> $(\lambda y.\ -\ \underline{y}\ 4\ )\ )\ 5$

=> 1

$(\lambda x.\ (\lambda x.\ +\ (\ -\ x\ 1\ )\ )\ \underline{x}\ 3\ )\ 9$

=> $(\lambda x.\ +\ (\ -\ \underline{x}\ 1\ )\ )\ 9\ 3$

=> $+\ (\ -\ 9\ 1\ )\ \ 3$

=> $+\ 8\ 3$

=> 11

$(\lambda x.\ (\lambda y.\ +\ \ \underline{x}\ \ ((\ \lambda x.\ -\ x\ 3)\ y\ )))\ 5\ 6$

=> $(\lambda y.\ +\ \ 5\ \ ((\ \lambda x.\ -\ x\ 3)\ \underline{y}\ ))\ \ 6$

=> $+\ \ 5\ \ (\ \lambda x.\ -\ \underline{x}\ 3)\ 6$

=> $+\ \ 5\ \ (-\ 6\ 3)$

=> $+\ 5\ )$

=> 8

Théorie propre du λ-calcul

L'ensemble des énoncés de la forme X => Y qui sont vrais.

Règles : Alpha, Béta, Axiome(X => X)

et les règles de déductions :

1. X => Y $\rightarrow$ ZX => ZY
2. X => Y $\rightarrow$ XZ => YZ
3. X => Y $\rightarrow$ λx.X => λx.Y
4. X => Y et Y => Z $\rightarrow$ X => Z

Formellement, nous dirons que X => Y ssi il existe une preuve de cet énoncé utilisant uniquement les axiomes et les règles ci dessus.

Théorie propre du λ-calcul

Autres règles (simplifications)

(Eta) : *Si x n'est pas libre dans M,alors $\lambda x.(Mx) = M$*

(Appliquer les deux membres à Y)

(Tau) : *Si x n'est pas libre dans M et dans N alors*

$Mx = Nx \rightarrow M = N$

(Prendre M= λy. (+ y 1); N= λz. (+ z 1))