Paradigme de programmation fonctionnelle* : Introduction

Le principe général de la programmation fonctionnelle est de concevoir des programmes comme des fonctions mathématiques que l'on compose entre elles. A la différence des programmes impératifs organisés en instructions produisent des effets de bords, les programmes fonctionnels sont bâtis sur des expressions dont la valeur est le résultat du programme. En particulier dans un langage fonctionnel, il n'existe pas d'effet de bord.

Un programme fonctionnel consiste en une expression E (représentant l'algorithme et les entrées). Cette expression E est sujette à des **règles de réécriture** : la réduction consiste en un remplacement d'une partie de programme fonctionnel par une autre partie de programme selon une règle de réécriture bien définie. Ce processus de réduction sera répété jusqu'a l'obtention d'une expression irréductible (aucune partie ne peut être réécrite). L'expression E* ainsi obtenue est appelée forme normale (fn) de E et constitue la sortie du programme.

Exemple : SQR 3+2
représente l'application de la fonction calculant le carré d'un nombre à l'expression 3+2

Evaluation : suite de réductions

```
SQR 3+2  -> SQR 5
         -> 5*5
         -> 25
```

* N. Hameurlain, Licence L3, Université de Pau.

Cette suite de réductions n’est pas la seule possible ; autres réductions :

SQR 3+2 -> (3+2)*(3+2)
-> 5*(3+2)
-> 5*5
-> 25

Certains langages fonctionnels sont purs : LISP, ML, MIRANDA. D’autres comme les différents dialectes de LISP (SCHEME, …) contiennent des constructions impératives (sauts, effets de bord, …). Mais, La base théorique commune à tous les langages fonctionnels est le λ-calcul introduit par Church vers 1930. Le λ-calcul est basé sur trois concepts : les variables, l'abstraction fonctionnelle, permettant de construire des fonctions et l'application de fonctions. Il est possible d'exprimer des fonctions d'ordre supérieur dont le résultat lui même est une fonction, et par conséquent applicable à d'autres fonctions.

Le λ-calcul est le paradigme de programmation fonctionnelle. Il est souvent utilisé comme langage élémentaire de très bas niveau et permet de mettre en évidence des problèmes des langages de programmation sous forme très simple. On démontre qu'il est aussi puissant que les autres formalismes du calcul : machine de Turing, fonctions récursives, primitives, fonctions récursives générales.
Il traite le calcul à partir du concept de fonctions. C'est l'aspect règle de calcul de la fonction (plus informatique) qui intervient ici et non l'approche graphe, c'est à dire ensemble de couples argument-valeur (plus mathématique). Une règle de calcul est l'expression d'un processus décrivent le passage de l'argument à la valeur.

I. Lambda Notation

On doit introduire des formes pour exprimer la notion de fonction et la notion complémentaire d'application

1. Alphabet

   Alphabet= {atomes} $\cup$ {λ} $\cup$ {délimiteurs}

   W= {atomes} ; W = V $\cup$ C ;

   V = {variables} ; C = {constantes}

   délimiteurs = { **)**, **(** }

2. Langage : défini par la grammaire :

   {{S}, alphabet, P, S} avec P :

   i. S $\rightarrow$ v, v $\in$W

   ii. S $\rightarrow$ (SS) application

   iii. S $\rightarrow$ (λx. S) abstraction

L(S) = {λ-expression}={λ-termes}

Exemples: (λx. x) (λy. y) $\in$ L(S)

(λx. (xx)) (λx. (λz. (zx))) $\in$ L(S)

(x (λx. (λy. y))) $\in$ L(S)

(x (x λx.)) $\notin$ L(S)

3. Notations

- x, y, … pour les variables
- a, b, A, B, … pour les constantes
- M, N, P, …, X, Y, … pour les λ-termes
- Le symbole ≡ est un symbole du méta-langage et non du λ-calcul. Il sera utilisé pour indiquer l'identité syntaxique et pour associer à un λ-terme un nom.

- Association à gauche pour l'application : on simplifie ((M1 M2) M3)…Mn) en

M1 M2 M3 …Mn

-Association à droite pour l'abstraction : on simplifie (λx1 (λx2….(λxn M))) en

λx1 λx2….λxn M

<u>Exemples</u> : λx λy λz xz (yz)

≡ (λx (λy (λz ((xz) (yz)))))

≡ S

x (λx. xx) ≡ (x (λx. (xx)))

4. Curryfication

On peut étendre l'abstraction fonctionnelle à la construction des fonctions à plusieurs arguments ;
Exemple : $\lambda x\ \lambda y.\ x+y \equiv \lambda xy.\ x+y$

Plus généralement, la construction $\lambda xy.\ E$ peut être réécrite en $\lambda x.\ (\lambda y.\ E)$,
Ceci transforme une fonction *f* : α x β -> γ en une nouvelle fonction *ff* : : α -> (β -> γ)

Exemple : $(\lambda xy.\ x<y)$ peut être réécrite en
$\lambda x.\ (\lambda y.\ x<y)$, qui représente la fonction à x associe la fonction qui teste si un nombre est supérieur à x.

Le résultat de cette construction appliquée à 0 est la fonction qui teste si un nombre est positif, c à d : $\lambda y.\ 0<y$

Cette opération est appelée *curryfication*. Elle généralise la décomposition d'une fonction n-aire en la composition d'une fonction (n-1) aire avec une fonction unaire.

Intérêt : purement théorique, permet de limiter l'étude des λ-expressions à des fonctions unaires.

5. Notations syntaxiques

Définition 1 : Dans la construction syntaxique $\lambda x.\ S$, S est appelé le *champ* (ou scope) de λx.

Définition 2 : une occurrence de x est *liée* dans un terme si elle est dans le champ d'un λx, sinon cette occurrence est dite *libre*.
Une variable est dite libre (resp. liée) si, au moins une de ses occurrences est libre (resp. liée)

Définition 3 : FV(M) est l'ensemble des variables libres de M et est défini inductivement par :

1. $FV(x) = \{x\}$,
2. $FV(\lambda x.\ M) = FV(M) - \{x\}$
3. $FV(MN) = FV(M) \cup FV(N)$

On note quelquefois $x \in M$ si et seulement si $x \in FV(M)$

6. Substitution

Pour toute λ–expression M et N, pour toute variable x, le résultat de la substitution de N à toute occurrence libre de x dans M est défini inductivement sur la structure syntaxique de M, et est noté [N/x] M :

1.1 $[N/x]\ x \equiv N$;

1.2 $[N/x]\ a \equiv a$, si $a \neq x$;

2. $[N/x]\ (U\ V) \equiv ([N/x]\ U)\ ([N/x]\ V)$

3.1 $[N/x]\ (\lambda x.\ M) \equiv (\lambda x.\ M)$;

3.2 $[N/x]\ (\lambda y.\ M) \equiv \lambda y\ ([N/x]\ M)$ si $y \neq x$ et $y \notin FV(N)$

3.3 $[N/x]\ (\lambda y.\ M) \equiv \lambda z\ ([N/x]\ ([z/y]\ M))$, où z une variable non encore utilisée, si $y \in FV(N)$ (et $x \in FV(M)$)

Exemples :

- [zz/x] (x (λx. λy.x)) = zz (λx. λy.x);
  2. puis 1.1 et 3.1
- [zz /x] (x (λy. xy)) = zz (λy. zzy);
  2. puis 1.1 et 3.2
- [y/x] (λy.yx) = (λz. zy) (3.3)

La règle de substitution nous indique comment transformer une expression en une autre expression équivalente et plus simple.

Remarque : éviter les problèmes qui découlent des conflits de noms

[y/x] (λy. xy) = (λz. yz) ≠ (λy. yy)

II. Règles de conversion/Réduction

On doit exprimer dans le formalisme les règles sur les fonctions unaires :

- Renommage des paramètres des fonctions: le nom des paramètres des fonctions importe peu.
- Une fonction prend tout son intérêt quand on l'applique à des arguments, et appliquer une fonction unaire à un argument consiste à substituer l'argument au paramètre dans le corps de la fonction.

1. α-conversion : renommage des variables liées

On appelle α-radical toute λ-expression de la forme λy. M.

On appelle α-contraction ou α-conversion le remplacement de l'α-radical λy. M par
λv. ([v/y] M) avec v une variable non encore utilisée.

Si on passe de Y à Z par une suite finie de α-contractions, on dit que Y se α-réduit en Z et on note $Y \rightarrow_\alpha Z$

Remarque : Dans 3.3 de la définition de la substitution, on a une α-contraction.

Exemple :

- $(\lambda x.\ xy)\ (\lambda y.\ zy) \rightarrow_\alpha (\lambda u.\ uy)\ (\lambda y.\ zy)$
- $(\lambda x.\ xy)\ (\lambda y.\ zy) \rightarrow_\alpha (\lambda t.\ ty)\ (\lambda x.\ zx)$

- La règle α indique, tant qu'aucun conflit de nom n'est introduit, nous pourrons librement changer les noms des variables liées dans une abstraction.

Dans les langages de programmation, ceci est lié à la possibilité de changer le nom d'un paramètre formel dans une fonction, à condition qu'il soit différent des noms des variables locales et globales de la fonction.

2. β-réductions

On appelle β-radical toute λ-expression de la forme
(λy. M) N ;

On appelle β-contraction le remplacement du β-radical (λy. M) N par [N/y] M.

Une suite finie de β-contractions est dite β-réduction.

Si on passe de Y à Z par une suite finie de α ou β-contractions, on dit que Y se (α) β-réduit en Z, et on note Y-> Z.

Exemples :

- (λx. ((λy.yx) z)) v ->(λx. zx)v
  -> zv
- (λx. ((λy.yx) z)) v ->(λy. yv) z
  -> zv

La règle β formalisme l'approche intuitive que nous avons relativement à l'évaluation des fonctions sur un jeu de paramètres : remplacer les

variables formelles par les paramètres effectifs dans l'abstraction.

3. η-réductions

On appelle η-contraction le remplacement du η-radical λx. Mx par M, si x $\notin$ M. on notera par
λx. M x ->$_{\eta}$ M, si x $\notin$ M.

La règle η permet d'introduire ou d'éliminer les adressages indirectes dans l'évaluation des fonctions : étant donnée qu'il n'y pas d'occurrence libre de x dans M, la règle de conversion s'applique :
(λx. M x) N = [N/x] (M x) = MN,
et en utilisant la propriété, d'extensionalité (voir plus loin) on a : (λx. M x) = M.

En appliquant d'abord la règle η, nous épargnons l'utilisateur des règles de substitution :

Exemple :

- (λx. zx) = z
- (λx. (λy.yt) x) = (λy.yt)

Commentaires:

Alors que la règle α ne permet pas de réduire la complexité d'une abstraction, l'application réitérée de la gauche vers la droite des autres règles permet d'éliminer successivement tous les niveaux d'abstraction, et conduit « normalement » à une expression qui ne peut pas être d'avantage simplifiée. Ce n'est cependant pas une règle générale.

Considérons l'exemple suivant :

$(\lambda x.\ xx)\ (\lambda x.\ xx)$

$\rightarrow_\beta\ (\lambda x.\ xx)\ (\lambda x.\ xx)$

$\rightarrow_\beta\ (\lambda x.\ xx)\ (\lambda x.\ xx)$

$\rightarrow_\beta \ldots$

ou bien

$(\lambda x.\ xxx)\ (\lambda x.\ xxx)$

$\rightarrow_\beta\ (\lambda x.\ xxx)\ (\lambda x.\ xxx)\ (\lambda x.\ xxx)$

$\rightarrow_\beta \ldots$ La complexité s'accroît considérablement.

En général, les expressions que nous manipulerons se simplifieront par l'emploi des règles β et η. Ces deux règles sont également connues sous le nom de règles de réduction, et nous dirons qu'une expression est réductible en une autre par l'une de ces deux règles. Une expression qui peut subir une réduction porte le nom de Redex, ainsi :

$(\lambda x.\ M)\ N$ est un β-redex, et $(\lambda x.\ Mx)$ est un η-redex.

III. Système d'inférence pour la réduction.

Y-> Z se dit « « Y se réduit en Z »

Schémas d'axiomes :

α) $\lambda y.\ M \rightarrow \lambda v\ ([v/y]\ M)$, si v est une variable non encore utilisée

β) $(\lambda x\ M)\ N \rightarrow [N/x]\ M$,

ρ) $M \rightarrow M$ (réflexivité)

Règles d'inférences :

μ) $\dfrac{Y \rightarrow Z}{X\ Y \rightarrow X\ Z}$

ν) $\dfrac{Y \rightarrow Z}{YX \rightarrow Z\ X}$

ξ) $\dfrac{Y \rightarrow Z}{\lambda x\ Y \rightarrow \lambda x\ Z}$

τ) $\dfrac{X \rightarrow Y,\ Y \rightarrow Z}{X \rightarrow Z}$ (transitivité)

δ) $\dfrac{Y \rightarrow Z}{Z \rightarrow Y}$ (symétrie)

La relation est alors notée =, et est une relation d'équivalence (ρ, τ, et δ).

Propriété d'extensionalité :

Définition : une relation = est dite extensionnelle si la règle suivante est valide :Si $\forall X,\ YX=ZX$ alors $Y = Z$.

L'égalité en λ-calcul est rendue extensionnelle en rajoutant l'axiome : η) $\lambda x\ M\ x \rightarrow M$, si $x \notin M$

Remarque : les règles α, β, η sont inversibles.

IV. Propriétés fondamentales

La β-règle est la vraie règle de calcul et la relation -> exprime qu'un λ-terme se réduit en un autre par un nombre fini d'opérations élémentaires.

Plusieurs questions se posent :
- Y a t-il plusieurs chemins de réduction ? si oui convergent-ils ?
- Comment et quand ce processus de réduction s'arrête ?

1.Définitions :
- un terme est en forme normale (fn) s'il ne contient aucun redex.
- Si Y se réduit en Z qui est en fn, on dit que Z est la forme normale de Y.
- Un terme est normalisable s'il a une fn ;
- Un terme est fortement normalisable s'il n'admet pas de réduction infinie.

Exemples :

- zv est le forme normale de (λx.((λy.yx) z)) v et de (λx.zx) v.
- I= (λx. xx) (λx. xx) n'a pas de fn
- (λx. y) I = (λx. y)((λx. xx) (λx. xx)) est normalisable mais pas fortement.

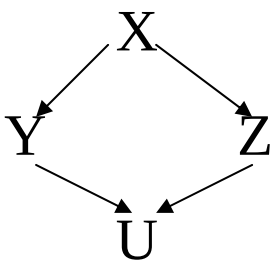
2. Théorèmes.

Théorème 1 : le problème de l'existence de la fn d'un terme est indécidable.

Théorème 2 : (théorème de Church)

Si X-> Y et X-> Z alors, il existe U, Y-> U et Z-> U.

On dit que la relation « -> » a la propriété de confluence ou la propriété de Church-Rosser ou la propriété du losange :



Corollaire : si X a deux fns, elles sont α-convertibles.

Unicité de la fn à l'α-conversion près.

Démo : (absurde) supposons que X a deux fn B et C non convertibles (par α conversion) : X-> B et X-> C.

Comme les règles sont inversibles, donc :
B -> X et C-> X,
Or B et C sont des fn et ne peuvent contenir de redex, et donc non réductibles : contradiction.

V. Stratégies d'évaluation :

Si on veut rendre opératoire le λ-calcul en tant que calcul de la forme normale d'un terme, il faut fixer une stratégie de réduction, i.e., un moyen de déterminer, pour chaque terme, le radical qui doit être contracté. Ceci correspond à déterminer comment sont résolus les appels de fonctions.

1. **Appel par nom à gauche** (ordre normal) :

La stratégie de l'appel par nom consiste à réduire (contracter) le radical le plus à gauche, parmi les redex le plus externes (i.e., dont le λ est le plus à gauche)

Exemple : $(\lambda x.\ x*x)\ ((\lambda x.\ (-x))\ 1)$

$= ((\lambda x.\ (-x))\ 1) * ((\lambda x.\ (-x))\ 1)$

$= (-1) * (-1)$

$= 1$

2. **Appel par valeur** (ordre applicatif)

La stratégie de l'appel par valeur consiste à réduire, dans $(\lambda x.\ M)\ N$ d'abord les redex les plus internes à N avant de réduire le terme $(\lambda x.\ M)\ N$ lui même.

Exemple : $(\lambda x.\ x*x)\ ((\lambda x.\ (-x))\ 1)$

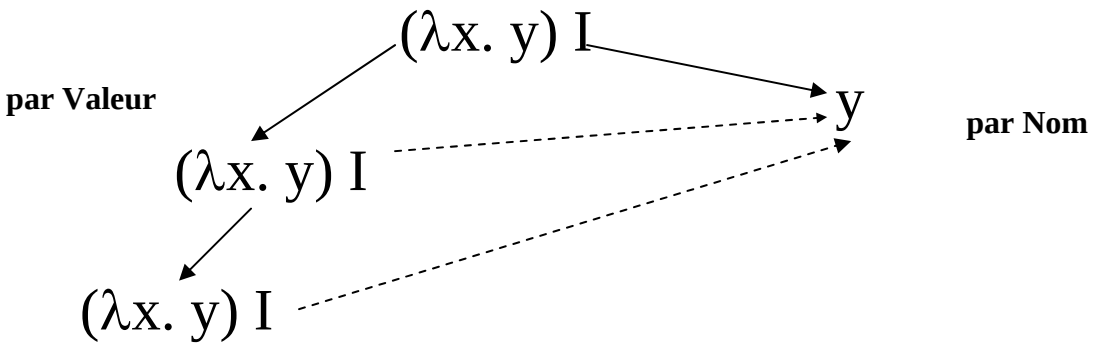
$= (\lambda x.\ x*x)\ (-1)$

$= (-1) * (-1)$

$= 1$

Remarque : cette stratégie (appel par valeur) est plus rapide : elle nécessite moins de β-réductions. Par contre, lorsqu'il existe une forme normale, elle peut nécessiter une suite infinie de β-réductions ; Par exemple :

$(\lambda x.\ y)\ I$ où $I = (\lambda x.\ xx)\ (\lambda x.\ xx)$



Définition (stratégie normalisante)

On dira qu'une stratégie est normalisante (ou correcte) si la réduction qu'elle détermine, pour tout terme normalisable, est finie.

Théorème (de normalisation ou standardisation)

La stratégie de réduction par nom (à gauche) est normalisante.

Exemples :

- $(\lambda x.\ y)\ ((\lambda x.\ xx)\ (\lambda x.\ xx))$
  $\rightarrow_g y$

- $(\ \lambda f\ (\lambda x\ f\ (xx))\ (\lambda x\ f(xx)))\ \ (\lambda x\ \lambda y\ y)$
  $\rightarrow_g (\lambda y\ y)$

- (λf (λx f (xx)) (λx f(xx))) n'a pas de fn.

- (λf (λx f (xx)) (λx f(xx))) (λx λy x) n'a pas de fn.

La stratégie d'appel par valeur n'est pas correcte comme le montre l'exemple ci-dessus (conduit à des réductions infinies)

Commentaires : Si on pense au λ- calcul comme à un système modélisant le calcul :

- un λ-terme représente le calcul,
- La réduction représente à la suite des étapes de calcul,
- Le fn correspond au résultat, i.e., à une expression complètement évaluée.

Questions : comment interpréter les λ-termes non normalisables ?

On peut être tenter de les interpréter comme l'indéfini ; or ils peuvent avoir un contenu opératoire qui peut être extrait en les appliquant à des arguments.

Remarques : le système avec la η-règle possède aussi la propriété de church-Rosser et il y a unicité de la forme normale β-η.

VI. Expressivité du λ-calcul

Le λ-calcul est aussi puissant que les autre formalismes du calcul ; on peut donc exprimer toute entité, objets ou algorithmes du monde informatique.

Pour illustrer ceci, nous allons représenter quelques entités comme les Booléens, l'expression conditionnelle IF (si …. Alors … sinon), les opérateurs AND, OR, NOT, le structures de données élémentaires comme le couple, l'opérateur de construction cons, les entiers, …

Booléens : Représentation de Bool = {True, False}

T ≡ λx. λy. x

F ≡ λx. λy. y

Les booléens doivent servir pour les branchements, et l'on doit représenter l'expression conditionnelle IF (si …alors…sinon) telle que :

IF T a b = a et IF F a b = b

IF ≡ λx λu λv ?

On définit par composition de fonctions, à partir de cet opérateur ternaire les opérateurs booléens :

AND ≡

OR ≡

NOT ≡

Couple: la structure de données élémentaire est le couple : l'opérateur de construction cons permet de rassembler deux entités, et first et snd sont les opérateurs d'extraction.

cons ≡ λa λb λx xab tel que cons a b = λx. xab

first ≡? tel que first (cons ab) = a

snd ≡ ? tel que snd (cons ab) = b

Entiers : la représentation des entiers de Church, $\mathbb{N}$, code les entiers comme des itérateurs de fonctions.

n ≡ λf λy f(f…(f y)…)
n fois

Ce terme est en forme normale et il y a isomorphisme de $\mathbb{N}$ dans $\mathbb{N}$

n /n a b= a (a… (ab)…)

0 ≡ λx λy y

1 ≡ λx λy x y

2 ≡ λx λy x(xy)