Fonctions d'ordre supérieur

Programmation Fonctionnelle
Master 2 I2L apprentissage

Sébastien Verel
verel@lisic.univ-littoral.fr
http://www-lisic.univ-littoral.fr/~verel

Université du Littoral Côte d'Opale
Laboratoire LISIC
Equipe OSMOSE

Octobre 2017

Plan

Paradigme fonctionnel

- Réfuter la notion d'instruction,
- Pas de notion de temps,
- Pas de modification (comme en mathématique)

Principe

Eléments de base = fonctions

cf. lambda calcul

Paradigme fonctionnel

Différent impératif / fonctionnel

Impératif : $\{I_1 \ ; \ I_2\}$

Fonctionnel : $f_1 \ o \ f_2$

Approximativement

- Programmation impérative :

  Exécuter un programme : évaluer un arbre, dont la valeur n'est pas importante.

  Au fur et à mesure de l'évaluation de l'arbre, des actions sont produites sur le monde extérieur.

  Sens au programme = actions

- Programmation fonctionnelle :

  Exécuter un programme : évaluer un arbre,
  et le résultat du programme est la valeur de l'arbre.

  Sens au programme = valeur

Paradigme fonctionnel

- S'appuie sur des mécanismes mentaux primitifs :
  Définition par récurrence
- Pas d'effet de bord, pas d'opérations d'affectation :
  En interne, pile pour stocker les informations temporaires (*notion d'environnement*)
- Calcul consiste en l'évaluation d'une fonction pour éviter toute modification "d'états"
- Résultat dépend seulement des entrées et non pas de l'état du programme

Paradigme fonctionnel

- S'appuie sur des mécanismes mentaux primitifs :
  Définition par récurrence
- Pas d'effet de bord, pas d'opérations d'affectation :
  En interne, pile pour stocker les informations temporaires (*notion d'environnement*)
- Calcul consiste en l'évaluation d'une fonction pour éviter toute modification "d'états"
- Résultat dépend seulement des entrées et non pas de l'état du programme
- **transparence référentielle** : remplacement des entrées par des fonctions qui ont les mêmes valeurs
- **Auto-référent** : capable de parler de lui-même
  Fonction, type de base
  Construction au cours de l'exécution

Transparence référentielle

Remplacement des entrées par des fonctions
qui ont les mêmes valeurs

A tester (ex04.hs)

```
ajout x y = x + y

one = 1

two = 2

main = do
   print $ ajout 1 2
   print $ ajout one two
```

Fonctions, types de base

```
bOne True = 1
bOne False = 0

bTwo True = 2
bTwo False = 0

ajoutF :: (Bool -> Int) -> (Bool -> Int) -> Int
ajoutF x y = x True + y True

main = do
   print $ ajoutF one two
   print $ ajoutF bOne bTwo
```

Attention : parenthèses obligatoire `(Bool -> Int)`

Fonctions, types de base

```
bOne True = 1
bOne False = 0

bTwo True = 2
bTwo False = 0

ajoutF :: (Bool -> Int) -> (Bool -> Int) -> Int
ajoutF x y = x True + y True

main = do
   print $ ajoutF one two
   print $ ajoutF bOne bTwo
```

Attention : parenthèses obligatoire `(Bool -> Int)`

`Couldn't match expected type...`

Normal il faut préciser que one et two soient bien des fonctions.

Fonction d'ordre supérieur

Fonction d'ordre 1

Définie à partir de fonctions d'ordre zéro.

Remarque : les fonctions d'ordre zéro sont les fonctions de base de Erlang.

Fonction d'ordre $n + 1$

Définie à partir de fonctions d'ordre n.

Fonction d'ordre supérieur

Synthaxe

```
functionName f1 f2 ... =
    ... f1 ...
```

Appel de fonction avec fonctions en paramètres

```
functionName f1 f2
```

Exemple

```
ajoutF :: (Bool -> Int) -> (Bool -> Int) -> Int
ajoutF x y = x True + y True

ajoutF bOne bTwo
3
```

Petits exos express

- Définir une fonction qui ajoute 1 à tous les nombres d'une liste d'entier
- Définir une fonction qui soustrait 1 à tous les nombres d'une liste d'entier
- Définir une fonction qui inverse toutes les listes d'une liste de listes. (Utiliser `reverse`)

Un peu de recul

Comment généraliser toutes ces fonctions
qui répondent à la même structure ?

Fonction map

```
mymap :: (a -> b) -> [ a ] -> [ b ]
mymap _ [] = []
mymap f (h:t) = f h: map f t
```

`f` est un paramètre de type `(a -> b)`

Fonction map

```
mymap :: (a -> b) -> [ a ] -> [ b ]
mymap _ [] = []
mymap f (h:t) = f h: map f t
```

`f` est un paramètre de type `(a -> b)`

```
incr x = x + 1

decr x = x - 1

main = do
   print $ mymap incr [1..6]
   print $ mymap decr [1..6]
   print $ mymap reverse [ [1..3], [1..4], [1..6] ]
```

Application spécifique de fonction

Exemple

```
increment l = map incr l

decrement l = map decr l

increment [1..6]
decrement [1..6]
```

Remarque : `map` est une fonction prédéfinie

Opération algébrique entre fonctions

Composition

La fonction entre fonctions $f \circ g$ se définit en Haskell par le point :

```
f . g
```

Exemple

```
rien = incr . decr
rien 3
3
```

Fonction anonyme

les *lambda*

Définition

Fonction sans nom

Synthaxe

```
\ pat1 pat2 ... patn -> expression
```

Avec $n \geq 1$

La notation \ doit rappeler la lettre λ

Exemples

```
\x -> x + 1
```

ou

```
\x y -> x + y
```

Exécution :

```
(\x -> x + 1) 2
(\x y -> x + y) 1 2
```

Que l'on peut utiliser avec map :

```
mymap (\x -> x + 1) [1..6]
mymap (\x -> x - 1) [1..6]
```

Fonction anonyme avec pattern matching

Synthaxe

```
\ x1 x2 ... xn -> case (x1, x2, ..., xn) of
    (pat1, pat2, ..., patn) -> expression
```

Exemple

```
\x -> case x of
  True -> 1
  False -> 0
```

Metaprogrammation

Possibilité qu'a une fonction de déterminer à l'exécution quelle autre fonction appliquée

Possibilité d'écrire des programmes qui crée d'autres programmes.

Fonction comme résultat

Il est possible que le résultat d'une fonction soit une fonction

Exemple

```
addOne =
  \x -> x + 1

addOne 3
4
```

La curryfication, Haskell Curry (1900 - 1982)

Les fonctions en programmation fonctionnelle (et en Haskell) n'ont qu'un seul argument.

Forme non curryfiée de la fonction `add` :

```
uncurry_add (x, y) = x + y
```

Formes non curryfiée : un seul paramètre qui peut être un tuple

Curryfication

Forme curryfiée de la fonction `add` :

```
curry_add x y = x + y
```

La forme curryfiée est en fait équivalente à :

```
curry_add = \x -> \y -> x + y
```

`curry_add` est une définition de fonction
qui a `x` associe une fonction
qui a `y` associe `x + y`.

Application partielle

Tester les types de :

```
:type curry_add
:type curry_add 3
(max 4) 5
```

Notation pour opérateurs binaires (operator section)

En Haskell, on peut utiliser l'application partielle pour simplifier l'écriture des opérateurs binaires

Si # est un opérateur binaire, alors :

```
(#) = \x -> \y -> x # y

(x#) = \y -> x # y

(#y) = \x -> x # y
```

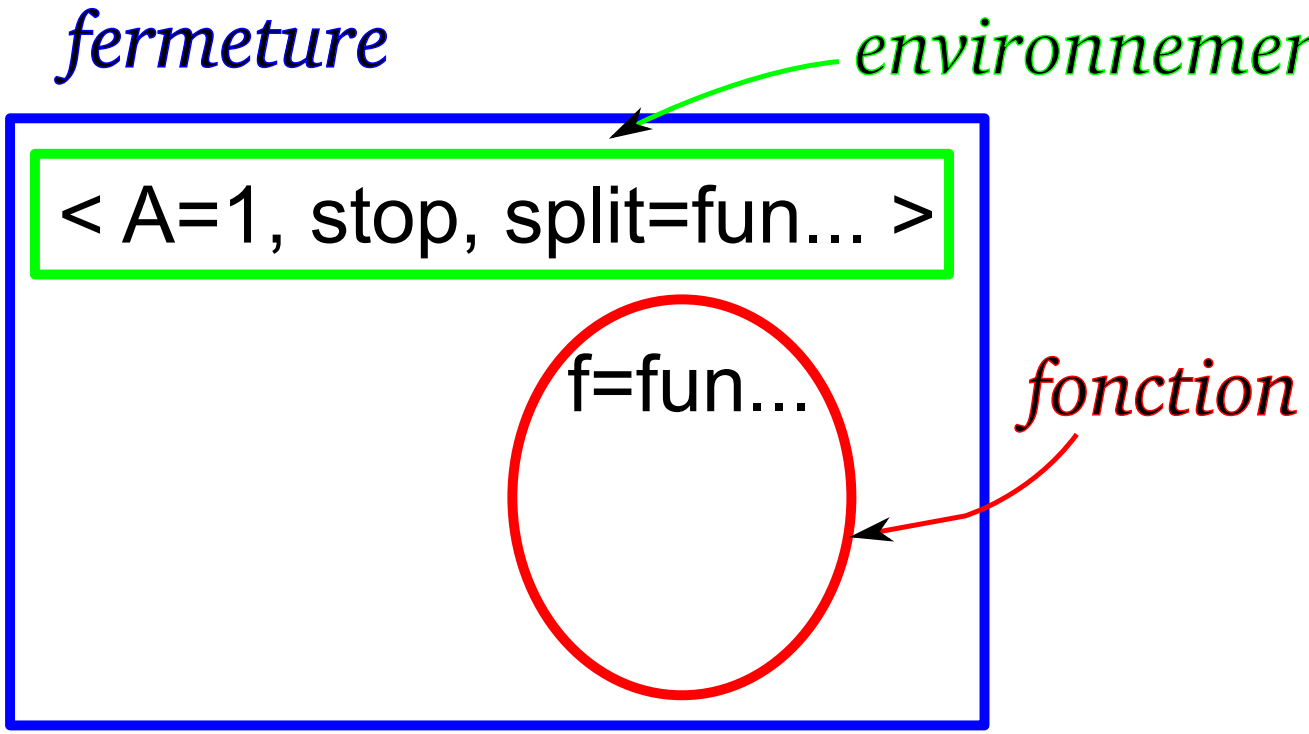
exemple

```
mymap (/2) [1..6]
[0.5,1.0,1.5,2.0,2.5,3.0]
```

Fermeture (closure)

Notion importante !

Une fermeture est une fonction avec son environnement d'exécution



Fermeture

Exemples

```
add a =
  \x -> x + a

main = do
   print $ add 4 10
   let add1 = add 1
       add4 = add 4 in do
     print $ add1 10
     print $ add4 20

mymap (add 1) [1..6]
mymap (add (-1)) [1..6]
```

Fermetures

Exemple

```
add a =
  \x -> x + a

addB =
  \x -> x + b
  where b = 3

print $ addB 5
```

Les variables a dans add et b dans addB appartiennent à l'environnement de chaque fonction résultat que l'on appelle des fermetures.

Fermetures, définitions locales

Exemple

```
cacheX a =
   let x = 1
   in
     let f = \x -> x + a
     in f 2
```

La définition locale de `x` est masquée par la fonction lambda.

Les lambda récursives

Problème !

Les `lambda` n'ont pas de nom pour être récursivement appelée...

Il faut donc faire appel aux travaux sur le lambda calcul :
Alonzo Church, Haskell Curry

Petite réflexion (ha ha ha) :

"Si cette phrase est vraie, alors le Monstre existe"

Fix point combinator

Que fait la fonction `y` ? Quel est son type ?

```
y f = f (y f)
```

Fix point combinator

Que fait la fonction `y` ? Quel est son type ?

```
y f = f (y f)
```

A tester

```
y (+2)
y (1:)

premier a b = a
y (premier 42)

take 12 (y (1:))
```

Point fixe

Définition

L'élément x est un point fixe de la fonction g ssi $g(x) = x$

Fix

```
f (y f) == y f
```

La fonction `y` calcule un point de la fonction donnée en paramètre.

Remarque : la fonction `y` est déjà définie par la fonction `fix` dans le module `Control.Monad.Fix`

Fonction récursive anonyme avec fix point combinator

```
\x -> if x == 0 then 1 else x * fac (x-1)
```

Problème on utilise `fac` qui n'existe pas ou est externe

Fonction récursive anonyme avec fix point combinator

```
\x -> if x == 0 then 1 else x * fac (x-1)
```

Problème on utilise `fac` qui n'existe pas ou est externe

Si la fonction `rec` qui calcule la factorielle existe
Alors elle vérifie la fonction :

```
\rec x -> if x == 0 then 1 else x * rec (x-1)
```

`rec` est une fonction de type :

```
Integer -> Integer
```

La lambda est de type :

```
(Integer -> Integer) -> Integer -> Integer
```

Fonction récursive anonyme avec fix point combinator

Définissons temporairement pour l'explication :

```
fact' = \rec x -> if x == 0 then 1 else x * rec (x-1)
```

Alors, `f = y fact'` est un point fixe vérifiant :

```
f = fact' f
```

C'est-à-dire que `y fact'` est la fonction factorielle !

```
factorial = y fact'
factorial 5
120
```

On pourrait développer les récursions...

Fibonacci en récursivité terminale

```
fiboT 0 un unM1 = un
fiboT 1 un unM1 = un
fiboT n un unM1 = fiboT (n-1) (un + unMoins1) un)

fibona n = fiboT n 1 0
```

Fibonacci en récursivité terminale

```
fiboT 0 un unM1 = un
fiboT 1 un unM1 = un
fiboT n un unM1 = fiboT (n-1) (un + unMoins1) un)

fibona n = fiboT n 1 0

fibonacci n =
   let f = y (\rec n un unMoins1 ->
      if n == 1
      then un
      else rec (n-1) (un + unMoins1) un)
   in f n 1 1
```

Petits exos

Exercice 1 : Fonction dérivée

Ecrire une fonction calculant une fonction qui approxime la fonction dérivée.
On rappelle que $(f(x+h)-f(x))/h$ est une valeur approchée de la fonction dérivée de f en x lorsque h est petit.

Exercice 2 : Les lambdas en récursivité terminale

Ecrire une fonction terminale enveloppée (qui contient une fonction anonyme récursive terminale) pour calculer la longueur d'une liste.

Exercice 3 : Fonction puissance

Ecrire une fonction qui calcule, pour n entier positif, la fonction puissance qui à x associe x^n.