Autre Paradigme

Patrice Boizumault

Université de Caen - Normandie

Lundi 27 janvier 2020

Fonctions anonymes (λ-abstraction)

- La notation $\lambda x.f(x)$ désigne la fonction : $x \to f(x)$
    - successeur sur les entiers : $\lambda x.(x+1)$
    - multiplier un nombre par 2 : $\lambda x.(2.x)$
    - être un entier strictement positif : $\lambda x.(x > 0)$
- La notation `Haskell`

```
\x -> f x
\x -> x + 1
\x -> x * 2
\x -> x > 0
```

- Utilisation

```
> (\x -> x + 1) 2    ==> 3
> (\x -> x * 2) (7+5)    ==> 24
> (\x -> x > 0) (head [1, 2, 7])    ==> True
```

Fonctions anonymes (λ-abstraction)

- Fonctions avec plusieurs paramètres

```
\x y -> [x, y]          liste avec 2 e'l'ements
\x y -> (x+y)/2         moyenne de 2 nombres
\x y -> (x, y)          couple
```

- Utilisation

```
> (\x y -> [x, y] ) 6 85   ==> [6,85]
> (\x y -> [x, y] ) 'a' 'b'   ==> "ab"
> (\x y -> (x+y)/2) 7 14   ==> 10.5
> (\x y -> (x, y)) 7 12   ==> (7,12)
> (\x y -> (x, y)) 'a' 27   ==> ('a',27)
> (\x y -> (x, y)) "azerty" True   ==> ("azerty",True)
```

Notation préfixée/infixée des opérateurs binaires

- notation infixée par défaut

```
> (7 * (8 + 2))      ==> 70
> 'a' : "zertyop"      ==> "azertyop
```

- utilisation en préfixé (en "parenthèsant" l'opérateur)

```
> (*) 7 (8 + 2)       ==> 70
> (*) 7 ((+) 8 2)      ==> 70
> (:) 'a' "zertyop"      ==> "azertyop"

> "ab" ++ "1zaw"      ==> "ab1zaw"
> (++) "ab" "1zaw"      ==> "ab1zaw"
```

Application partielle[1] : opérateur (+)

- (+) opérateur binaire de somme
- (+ 2) fonction qui ajoute 2 à un nombre
- (+ 2) et (\x -> x+2) sont 2 fonctions égales

```
> 2 + 7        ==> 9
> (+ 2) 7        ==> 9
> (\x -> x+2) 7        ==> 9
```

- les différents types (parenthèsage implicite à droite)

```
(+) :: Int -> (Int -> Int)
(+ 2) :: Int -> Int
(+ 2) 7 :: Int
```

[1] la notion de curryfication (cas général) sera présentée à la séance suivante.

Application partielle : opérateur (==)

- `(==)` opérateur binaire
- `(== 'a')` fonction qui détermine si un caractère vaut `'a'`
- `(== 'a')` et `(\x -> x=='a')` sont 2 fonctions égales

```
> 'b' == 'a'        ==> False
> (== 'a') 'b'        ==> False
> (\x -> x=='a') 'b'        ==> False
```

- les différents types

```
(==) :: Char -> Char -> Bool
(== 'a') :: Char -> Bool
(== 'a') 'b' :: Bool
```

Multiples applications d'une fonction

- `(map f xs)` construit la liste des applications de la fonction `f` à tous les éléments de la liste `xs`

```
> map square [1..4]   ==> [1,4,9,16]
> map odd [1..4]     ==> [True, False, True, False]
> map (* 2) [1..4]    ==> [2, 4, 6, 8]
> map even [1..4]    ==> [False, True, False, True]
```

- typage

```
map :: (a -> b) -> [a] -> [b]
```

- somme des carrés des n premiers entiers

```
sumsquare :: Int -> Int
sumsquare n = sum (map square [1..n])
```

Multiples applications d'une fonction

- définition récursive de la fonction `map`

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = (f x) : (map f xs)
```

- autre définition à l'aide d'une ZF-expression

```
map f xs = [f x | x <- xs]
```

Multiples applications d'une fonction

- distributivité de `map` par rapport à la composition de fonctions

```
map (f.g) = (map f) . (map g)
```

- distributivité de `(map f)` par rapport à la concaténation

```
map f (xs ++ ys) = (map f xs) ++ (map f ys)
```

Fonctionnelles

Filtrer les éléments d'une liste satisfaisant une propriété

- (filter p xs) construit la liste des éléments de la liste xs satisfaisant le prédicat p

```
> filter even [1..10]   ==>  [2,4,6,8,10]
> filter odd [1..10]   ==> [1,3,5,7,9]
> filter (== 'a') "azertayaap"   ==> "aaaa"
> filter (/= 'a') "azertayaap"   ==> "zertyp"
```

- typage

```
filter :: (a -> Bool) -> [a] -> [a]
```

- somme des carrés des nombres pairs entre 1 et n

```
sumEvenSquare :: Int -> Int
sumEvenSquare n = sum (map square (filter even [1..n]))
```

Filtrer les éléments d'une liste satisfaisant une propriété

- définition récursive de `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs)
   | (p x)      = x:(filter p xs)
   | otherwise = filter p xs
```

- autre définition à l'aide d'une ZF-expression

```
filter p xs = [x | x <- xs, p x]
```

Filtrer les éléments d'une liste satisfaisant une propriété

- commutativité par rapport à la composition de fonctions

```
(filter p) . (filter q) = (filter q) . (filter p)
```

- distributivité de `(filter p)` sur `(++)`

```
filter p (xs ++ ys) = (filter p xs) ++ (filter p ys)
```

Retour sur les fonctions anonymes (λ-abstraction)

- Eviter de définir des fonctions de peu d'intérêt général ou bien utilisées ponctuellement

```
> map (\x->x+1) [1..10]  ==> [2,3,4,5,6,7,8,9,10,11]

> filter (\x->(5<x && x<10)) [1..20]  ==> [6,7,8,9]
```

- Les fonctions anonymes peuvent utiliser le filtrage

```
> let xs = [("Jean",23),("Marie",25),("Paul",30),("Anne",18)]

-- les moins de 25 ans
> filter (\(nom,age) -> (age <= 25)) xs
[("Jean",23),("Marie",25),("Anne",18)]
```

Retour sur les fonctions anonymes (λ-abstraction)

- Les fonctions anonymes peuvent utiliser le filtrage (suite)

```
> let xs = [("Jean",23),("Marie",25),("Paul",30),("Anne",18)]

-- les moins de 25 ans
> filter (\(_, age) -> (age <= 25)) xs
[("Jean",23),("Marie",25),("Anne",18)]

-- Les couples d'entiers ordonnés
> filter (\(x,y) -> (x<y)) [(1,4), (5,2), (2, 9)]
[(1,4), (2,9)]

lesordonnes :: [(Int, Int)]  ->  [(Int, Int)]
lesordonnes xs = filter (\(x,y) -> (x<y)) xs
```

Folders (à droite)

- généraliser à une liste l'application d'un opérateur binaire
- `foldr` associe à droite i.e. l'application commence par la fin de liste

```
foldr op e [x1, x2, ..., xn] =
         x1 op (x2 op (x3 ... op (xn op e) ...))

foldr (+) 0 [2,5,7,9] = 2 + (5 + (7 + (9 + 0))) = 23
foldr (*) 1 [2,5,7,9] = 2 * (5 * (7 * (9 * 1))) = 630
```

- fonctions usuelles exprimées à l'aide de `foldr`

```
sum     = foldr (+) 0
product = foldr (*) 1
and     = foldr (&&) True
or      = foldr (||) False
```

Fonctionnelles

La fonction `concat` généralise l'opérateur `(++)`

- ```
  > concat [[1,2],[2,3],[3,4]]  ==> [1,2,2,3,3,4]
  > [1,2] ++ [2,3] ++ [3,4]    ==> [1,2,2,3,3,4]

  > concat ["nous ", "formons ", "une ", "liste"]
                               ==> "nous formons une liste"
  ```
- typage

  ```
  concat :: [[a]] -> [a]
  ```
- `concat` définie par un `foldr`

  ```
  concat xss = foldr (++) [] xss
  ```
- `concat` définie par une ZF expression

  ```
  concat xss = [x | xs <- xss, x <- xs]
  ```
  ```

Folders (à droite)

- `foldr` associe à droite : l'application commence par la fin de liste

```
foldr op e [] = e
foldr op e [x1,...,xn] = x1 op (x2 ... op (xn op e)...)
```

- typage (exemples étudiés jusque là)

```
(op) :: (a -> a -> a)
foldr :: (a -> a -> a) -> a -> [a] -> a
```

- typage (cas général)

```
(op) :: (a -> b -> b)
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Folders (à droite)

- dans les exemples considérés, les opérateurs sont de type `(a -> a -> a)`, associatifs et possèdant un neutre `e`. Le couple `(op, e)` forme un **monoïde**.

- si `(op, e)` forme un monoïde, alors

```
foldr op e [] = e
foldr op e [x1,x2,...,xn] = x1 op x2 ... op xn
```

- mais tous les couples `(op, e)` ne forment pas un monoïde

```
length :: [a] -> Int
length = foldr op 0
         where op x n = n + 1
```

Folders (à gauche)

- `foldl` associe à gauche : on commence l'application par le début de liste

```
foldl op e [] = e

foldl op e [x1,...,xn] = (...((e op x1) op x2)  ... op xn)
```

- exemples :

```
reverse :: [a] -> [a]
reverse = foldl op []
          where op xs x = x:xs

pack :: [Int] -> Int
pack = foldl op 0
       where op n x = 10*n + x
```

Folders (propriétés)

(P_1) si `(op,e)` forme un monoïde, alors : `(foldr op e) = (foldl op e)`

(P_2) soient op1 et op2 tq

(i) `x op1 (y op2 z) = (x op1 y) op2 z`
(ii) `x op1 e = e op2 x`

alors `(foldr op1 e) = (foldl op2 e)`

(P_3) soient op et e, alors il existe op' tq

```
foldr op e xs = foldl op' e (reverse xs)
```

Pour cela, définir op' par `op' x y = op y x`

Folders (propriétés)

(P_4) deux autres propriétés par rapport à l'opérateur (++)

```
foldl op e (xs ++ ys) = foldl op (foldl op e xs) ys

foldr op e (xs ++ ys) = foldr op (foldr op e ys) xs
```

Folders (définitions récursives)

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr op e []     = e
foldr op e (x:xs) = op x (foldr op e xs)


foldl :: (b -> a -> b) -> b -> [a] -> b

foldl op e []  = e
foldl op e xs  = op (last xs) (foldl op e (init xs))


-- rappel P4
foldr op e xs = foldl op' e (reverse xs)
           where op' x y = op y x
```