

Programmation Fonctionnelle en Haskell

*Licence « Sciences et Technologies »
Parcours « Informatique » – 2ème année*

UFR Sciences – Université de Caen
Patrice Enjalbert
Département d'Informatique
Patrice.Enjalbert@info.unicaen.fr

1

Chapitre 3 (Fin)

1. Encore un itérateur : `foldl`. Application : vecteurs revisités
2. Currification
3. Evaluation paresseuse

2

1. itérateur foldl

```
foldl :: (a -> b -> b) -> b -> [a] -> b
foldl f init [e1,e2,...,en] =
    (f ... (f (f init e1) e2)... en)
```

NB. Souvent $a=b$ et $init$ = neutre pour f

```
> foldl (+) 0 [1..5]      -- == (...((0+1) + 2) + ...)+ 5
15
> sum [1..5]
15
> foldl (*) 1 [1..5]
120
```

3

- zip et unzip

```
> :t zip
zip :: [a] -> [b] -> [(a, b)]
> :t unzip
unzip :: [(a, b)] -> ([a], [b])

> zip "haskel" [1..(length "haskel")]
[('h',1),('a',2),('s',3),('k',4),('e',5),('l',6)]
> unzip it
("haskel",[1,2,3,4,5,6])
```

4

- Application : produit scalaire

```
type Vect = [Float]
w1 = [1, -3, -5] :: Vect
w2 = [-3, 1, -2] :: Vect

> zip w1 w2
[(1.0,-3.0),(-3.0,1.0),(-5.0,-2.0)]
> map (\(x,y)-> x*y) it
[-3.0,-3.0,10.0]
```

Il faut en faire la somme...

5

```
mysum l = foldl (\x y -> x+y) 0 l
ou :
mysum l = foldl (+) 0 l
Ou
mysum = foldl (+) 0

scalaire v1 v2 =
    mysum (map (\x y -> x*y) (zip v1 v2))

> scalaire w1 w2
4
```

6

2. Notation «curryfiée »

Question : quelle est la différence entre

$f :: a \rightarrow (b \rightarrow c)$
 et $f' :: (a, b) \rightarrow c$?

Réponse :

- Intuitive :
 f prend les arguments « l'un après l'autre »
 f' les prend « en couple »
- Précise
 f va d'un ensemble a vers l'ensemble des fonctions de b dans c .
 f' va du produit cartésien $a \times b$ vers l'ensemble c .

7

Fait :

Pour toute $f :: a \rightarrow b \rightarrow c$
 Il existe $f' :: (a, b) \rightarrow c$ « équivalente ».

Et *vice versa* :

$f'(x, y) = f \ x \ y$	$f' = \lambda(x, y) \rightarrow (f \ x \ y)$
$f \ x \ y = f'(x, y)$	$f \ x = \lambda y \rightarrow (f' \ (x, y))$
	$f = \lambda x \rightarrow (\lambda y \rightarrow (f' \ (x, y)))$

8

Dans « l'équivalence »

entre $f :: a \rightarrow b \rightarrow c$

Et $f' :: (a,b) \rightarrow c$

Le passage de f' à f est la *curryfication*
(du nom du logicien... Haskell Curry !!)

f est la *curryfié* de f'

f' est la *décurryfiée* de f

Les 2 opérations sont définies en Haskell :

9

```
> :t modulo
modulo :: Int -> Int -> Int
> modulo 3 11
2
> :t modulo 3
modulo 3 :: Int -> Int
> :t (uncurry modulo)
(uncurry modulo) :: (Int, Int) -> Int
> (uncurry modulo) (3,11)
2

> modulo' = (uncurry modulo)
> :t (curry modulo')
(curry modulo') :: Int -> Int -> Int
> (curry modulo') 3 11
2
```

10

- Application : vecteurs (reformulation)

```
w1 = [1, -3, -5] :: Vect
w2 = [-3, 1, -2] :: Vect

> map (\(x,y)-> x*y) (zip w1 w2)
[-3.0,-3.0,10.0]

Ou :
> map (uncurry (*)) it
[-3.0,-3.0,10.0]

scalaire v1 v2 =
  sum (map (\x y -> x*y) (zip v1 v2))
Ou
  sum (map (uncurry (*)) (zip v1 v2))
Ou
  foldl (+) 0 (map (uncurry (*)) (zip v1 v2))
```

11

3. Evaluation paresseuse

Evaluation d'une fonction

- Appel par valeur (« standard »)

```
f x y z = ...
> f a b c
  a est évalué, puis b, puis c
  Puis f est évalué pour ces « valeurs »
g l = ... :: [a] -> b
> g [x,y,z,...]
  La structure (liste [x,y,z..]) est construite, puis g est évaluée
```

12

3. Evaluation paresseuse

Evaluation d'une fonction

- Appel par nécessité (évaluation paresseuse)
- > f a b c
a est évalué *si nécessaire*, puis b *si nécessaire* ... pour pouvoir produire une valeur pour f
- > g 1
La structure (liste [x,y,z..]) n'est construite qu'autant que nécessaire

Fondamental pour l'efficacité des langages fonctionnels

13

Chapitre 4 : Types « data »

1. Présentation
2. Arbres binaires d'entiers
3. Arbres binaires polymorphes
4. La clause « deriving »
5. Autres exemples : arbres généraux, types énumérés

14

1. Présentation

- Pour représenter des données, Haskell propose des types prédéfinis :
 - **types simples** : Bool, Int, Float, Char, String...
 - types **structurés** :
 - o Liste de a : [a]
 - o Paires et t-uples : (a,b) (a,b,c) ...
- De plus, l'utilisateur peut définir ses propres types pour structurer, organiser, des ensembles de données :
 - déclaration **data**, un mode générique « universel »
 - Ces types sont structurés par des constructeurs de types.

15

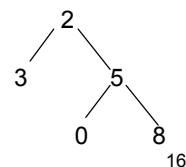
2. Arbres binaires d'entiers

```
data ArbreBinEnt =  
    Vide | ABE Int ArbreBinEnt ArbreBinEnt
```

Un arbre binaire d'entier est :

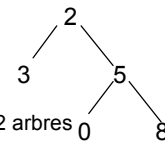
- Soit l'arbre vide `Vide`
 - Soit la donnée, structurée par `ABE`
- de
- 1 entier `:: Int`
 - 2 arbres binaires entiers `:: ArbreBinEnt`

```
a1 = ABE 2 (ABE 3 Vide Vide)  
        (ABE 5 (ABE 0 Vide Vide)  
              (ABE 8 Vide Vide))  
      :: ArbreBinEnt
```



```
data ArbreBinEnt =
    Vide | ABE Int ArbreBinEnt ArbreBinEnt
```

```
a1 = ABE 2 (ABE 3 Vide Vide)
      (ABE 5 (ABE 0 Vide Vide)
              (ABE 8 Vide Vide))
```

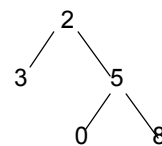


- Vide et ABE sont des constructeurs de types
 - ABE : construit un arbre à partir d'un entier et de 2 arbres
 - Vide : constante d'arbre
- Notez la majuscule des identificateurs de type et constructeurs de type

17

```
data ArbreBinEnt =
    Vide | ABE Int ArbreBinEnt ArbreBinEnt
```

```
a1 = ABE 2 (ABE 3 Vide Vide)
      (ABE 5 (ABE 0 Vide Vide)
              (ABE 8 Vide Vide))
```



- L'accès aux constituants d'un arbre (d'une « data ») se fait par filtrage :

```
gauche, droit :: ArbreBinEnt -> ArbreBinEnt
racine :: ArbreBinEnt -> Int
```

```
gauche (ABE _ g _) = g
droit (ABE _ _ d) = d
racine (ABE r _ _) = r
```

```
> droit a1
ABE 5 (ABE 0 Vide Vide) (ABE 8 Vide Vide)
```

18

```

data ArbreBinEnt =
    Vide | ABE Int ArbreBinEnt ArbreBinEnt

a1 = ABE 2 (ABE 3 Vide Vide)
      (ABE 5 (ABE 0 Vide Vide)
              (ABE 8 Vide Vide))

```

• Recherche d'un élément

```

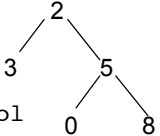
memberIntTree :: Int -> ArbreBinEnt -> Bool
memberIntTree _ Vide = False
memberIntTree x (ABE r g d) =
    r == x || memberIntTree x g || memberIntTree x d

```

```

> memberIntTree 5 a1
True
> memberIntTree 7 a1
False

```



```

      2
     / \
    3   5
     \  / \
      0 8

```

19

```

data ArbreBinEnt =
    Vide | ABE Int ArbreBinEnt ArbreBinEnt

a1 = ABE 2 (ABE 3 Vide Vide)
      (ABE 5 (ABE 0 Vide Vide)
              (ABE 8 Vide Vide))

```

• Somme des noeuds

```

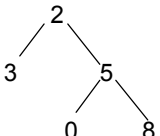
sumTree :: ArbreBinEnt -> Int
sumTree Vide = 0
sumTree (ABE r g d) = r + (sumTree g) + (sumTree d)

```

```

> sumTree a1
18

```



```

      2
     / \
    3   5
     \  / \
      0 8

```

20

3. Arbres binaires paramétrés (polymorphes)

```
data ArbreBin a =  
    Null | AB a (ArbreBin a) (ArbreBin a)
```

En général, on veut des arbres de « n'importe quoi » : nombres, caractères, strings...

De même que le type « liste » est paramétré par le type (a) de ses éléments : [a]

On aura un type « arbre binaire » paramétré par le type a de ses nœuds
ArbreBin a
Par exemple : ArbreBin Int, ArbreBin Char...

21

```
data ArbreBin a =  
    Null | AB a (ArbreBin a) (ArbreBin a)
```

- Recherche d'un élément

```
memberTree :: (Eq a) => a -> (ArbreBin a) -> Bool
```

```
memberTree _ Null = False  
memberTree x (AB r g d) =  
    r == x || memberTree x g || memberTree x d
```

22

- Schéma récursif général

```
fonc :: ArbreBin a -> b
```

```
fonc Null = v0          -- Cas de base : arbre vide
fonc (AB r g d) = (gonc r (Fonc g) (Fonc d))
    -- Récursion sur les sous arbres Gche et Dt
    -- Combinaison avec la racine
```

- Rappel schéma récursif sur les listes

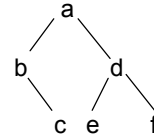
```
foo :: [a] -> b
foo [] = v0
foo (x:l) = bar x (foo l)
    -- Récursion sur la queue de liste
    -- Combinaison avec la tête
```

23

24

- **Construction d'un arbre.**

Exemple : arbre parfaitement équilibré
"abcdef"



```
data ArbreBin a =
  Null | AB a (ArbreBin a) (ArbreBin a)
```

Rappel : un arbre est dit *parfaitement équilibré* ssi pour chacun de ses nœuds, les sous-arbres gauche et droit ont le même nombre de nœuds, à 1 près.

```
liste2arbre :: [a] -> ArbreBin a
liste2arbre [] = Null
liste2arbre (x:xs) =
  (AB x (liste2arbre lg) (liste2arbre ld))
  where
    partage :: [a] -> ([a],[a])
    partage l = let p = div (length l) 2
                  in ((take p l), (drop p l))
    (lg,ld) = (partage xs)
```

25

4. Clause « deriving »

```
data ArbreBin a =
  Null | AB a (ArbreBin a) (ArbreBin a)
```

- **Problème. En fait :**

```
> liste2arbre "abcdef"
```

Top level:

No instance for (Show (ArbreBin Char)) arising from use of `print' at Top level

Probable fix: add an instance declaration for (Show (ArbreBin Char))...

- **Traduction (approximative)**

Le type ArbreBin Char ne possède pas de méthode d'affichage

Il faut le faire hériter d'une classe Show qui est capable d'inférer (dériver) cette méthode pour tout type « data »

26

```

data ArbreBin a =
    Null | AB a (ArbreBin a) (ArbreBin a)
    deriving (Show)

> liste2arbre "abcdefg"
AB 'a' (AB 'b' (AB 'c' Null Null) (AB 'd' Null
    Null)) (AB 'e' (AB 'f' Null Null) (AB 'g'
    Null Null))

-- NB. La fonction show produit un string pour tous
    les types « showable »
> show [1..5]
"[1,2,3,4,5]"

```

27

- Idem pour avoir une égalité entre arbres de ce type : classe Eq

```

> liste2arbre "abcdefg"
AB 'a' (AB 'b' (AB 'c' Null Null) (AB 'd' Null Null))
    (AB 'e' (AB 'f' Null Null) (AB 'g' Null Null))
> it == it
**** Erreur .....

Avec :
data ArbreBin a =
    Null | AB a (ArbreBin a) (ArbreBin a)
    deriving (Show, Eq)

> liste2arbre "abcdefg"
AB 'a' (AB 'b' (AB 'c' Null Null) (AB 'd' Null Null))
    (AB 'e' (AB 'f' Null Null) (AB 'g' Null Null))
> it == it
True

```

28