Programmation Fonctionnelle
Fonctions d'ordre supérieur

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

24 septembre 2013

Plan

Fonctions d'ordre supérieur : introduction

Fonctions d'ordre supérieur : exemples

`foldr`

D'autres fonctions d'ordre supérieur

Exercices

Plan

Fonctions d'ordre supérieur : introduction

Fonctions d'ordre supérieur : exemples

`foldr`

D'autres fonctions d'ordre supérieur

Exercices

Introduction

Une fonction est dite *d'ordre supérieur* si elle prend (au moins) une fonction parmi ses arguments.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

> `twice` *est une fonction d'ordre supérieur car elle prend une fonction comme son premier argument.*

Pourquoi ces fonctions ?

- Les idiomes communs des langages peuvent s'encorder comme des fonctions dans le langage même.
- les langages spécifiques à un domaine peuvent se définir comme des collections de fonctions d'ordre supérieur.
- Les propriétés algébriques des fonctions d'ordre supérieur peuvent s'utiliser pour raisonner sur les programmes.

Plan

Fonctions d'ordre supérieur : introduction

Fonctions d'ordre supérieur : exemples

`foldr`

D'autres fonctions d'ordre supérieur

Exercices

La fonction `map`

La fonction d'ordre supérieur appelée `map` applique une fonction à tous les éléments d'une liste :

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

Par exemple :

```
> map (+1) [1,3,5,7]
[2,4,6,8]
```

Cette fonction peut se définir de façon très simple en utilisant la compréhension :

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

Sinon – et de façon plus utiles pour les preuves – `map` peut se définir par récursion :

```
map f [] = []
map f (x:xs) = f x:map f xs
```

La fonction `filter`

Cette fonction sélectionne (filtre) tous les éléments d'une liste satisfaisants un prédicat.

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

Par exemple :

```
> filter even [1..10]
[2,4,6,8,10]
```

Définition de `filter` par compréhension :

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

. . . ou par récursion :

```
filter p [] = []
filter p (x:xs)
    | p x = x:filter p xs
    | otherwise = filter p xs
```

Plan

Fonctions d'ordre supérieur : introduction

Fonctions d'ordre supérieur : exemples

`foldr`

D'autres fonctions d'ordre supérieur

Exercices

La fonction `foldr`

Des nombreuses fonctions sur les listes peuvent se définir avec le même schéma de récursion :

```
f []     = v
f (x:xs) = x @ f xs
```

> *f mappe la liste vide vers un valeur v,*
> *et toute autre liste vers le résultat d'appliquer*
> *l'opérateur binaire `@` à la tête et à f de la queue.*

Par exemple :

```
sum [] = 0
sum (x:xs) = x + sum xs
```

> *v=0* et *@=+*

```
product [] = 1
product (x:xs) = x * product xs
```

> *v=1* et *@=**

```
and [] = True
and (x:xs) = x && and xs
```

> *v=True* et *@=&&*

La fonction d'ordre supérieur `foldr` (« *fold right* ») capture ce simple schéma de recursion, avec la fonction `@` et l'argument `v` comme arguments.

Par exemple :

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

La fonction `foldr` peut elle même se définir par récursion :

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

Par ailleurs, c'est utile de penser à cette fonction de façon non récursive, comme le résultat de l'évaluation de l'expression obtenue de la liste en remplaçant chaque symbole `:` par l'opérateur binaire `@` passé en argument, et `[]` par la valeur `v`.

Par exemple :

```
sum [1,2,3] = foldr (+) 0 [1,2,3]
            = foldr (+) 0 (1:(2:(3:[])))
            = (1+(2+(3+0)))
            = 6
```

Remplace tout `(:)` par `(+)` et `[]` par `0`.

Par exemple :

```
product [1,2,3] = foldr (*) 1 [1,2,3]
                = foldr (*) 1 (1:(2:(3:[])))
                = 1*(2*(3*1))
                = 6
```

> *Remplace tout `(:)` par `(*)` et `[]` par `1`.*

Autres exemples avec `foldr`

Même si `foldr` code un simple schéma de récursion, il peut être utilisé pour définir beaucoup plus de fonctions de ce qu'on pourrait s'imaginer.

length de *foldr*

Rappellez vous la fonction `length` :

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Par exemple :

```
length [1,2,3] = length (1:(2:(3:[])))
               = 1+(1+(1+0))
               = 3
```

> *Remplace tout* `(:)` *par* `(\_ n -> 1+n)`
> *et* `[]` *par* `0`.

Par conséquence, on a

```
length = foldr (\_ n -> 1+n) 0
```

reverse de *foldr*

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Par exemple :

```
reverse [1,2,3]
= reverse (1:(2:(3:[])))
= 1 r (2 r (3 r [])
= (([] ++ [3]) ++ [2]) ++ [1]
= [3,2,1]
```

> *Remplacer tout* `(:)` *par* `r` *et* `[]` *par* `[]`*, où*
> `(r) = \x xs -> xs ++ [x]`

Donc :

```
reverse = foldr (\x xs -> xs ++ [x]) []
```

`(++)` de `foldr`

Pour finir, la fonction `(++)` peut se définir de façon très compacte en utilisant `foldr`.

Exemple :

```
[1,2,3] ++ [4,5,6]
= 1:2:3:[] ++ [4,5,6]
= 1:2:3:[4,5,6]
```

> *Remplacer tout `(:)` par `(:)` et `[]` par `ys`.*

Donc :

```
(++ ys) = foldr (:) ys
```

Pourquoi utiliser `foldr` ?

- La définition de certaines fonctions récursives sur les listes – comme `sum` – est plus simple en utilisant `foldr`.
- Propriétés algébriques des fonctions ainsi définies peuvent se prouver en utilisant les propriétés algébriques de `foldr` – comme la règle de la fusion et de la banana split.
  Voir par exemple
  `http://www.randomhacks.net/articles/2007/02/10/map-fusion-and-haskell-performance`
- On peut obtenir des programmes optimisés si on utilise `foldr` à la place de la récursion explicite.

Plan

Fonctions d'ordre supérieur : introduction

Fonctions d'ordre supérieur : exemples

`foldr`

D'autres fonctions d'ordre supérieur

Exercices

D'autres fonctions d'ordre supérieur de `Prelude`

La fonction `(.)` retourne la composition de deux fonctions :

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Par exemple :

```
odd :: Int -> Bool
odd = not . even
```

all

La fonction `all` décide si tout élément d'une liste satisfait un prédicat.

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
```

Par exemple :

```
> all even [2,4,6,8,10]
True
```

any

La fonction `any` décide si au moins un élément d'une liste satisfait un prédicat.

```
any :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]
```

Par exemple :

```
> any isSpace "abc def"
True
```

takeWhile

La fonction `takeWhile` sélectionne les éléments d'une liste, tant que un prédicat est vrai de ses éléments :

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

Par exemple :

```
> takeWhile isAlpha "abc def"
"abc"
```

dropWhile

La fonction `dropWhile` enlève les éléments d'une liste, tant que un prédicat est vraie de ces éléments.

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

Par exemple :

```
> dropWhile isSpace "   abc"
"abc"
```

Plan

Fonctions d'ordre supérieur : introduction

Fonctions d'ordre supérieur : exemples

`foldr`

D'autres fonctions d'ordre supérieur

Exercices

Exercices

1. Si une fonction retourne une autre fonction comme résultat, on dit que cette fonction est ... ?
2. Utilisez les fonctions `map` et `filter` pour donner un équivalent de l'expression `[f x | x <- xs, p x]`
3. Redéfinissez `map f` et `filter p` à l'aide de `foldr`.
4. Proposez une définition de la fonction `foldl`.