Programmation Fonctionnelle
Les Listes

Michael Lampis

2023-2024

Les Listes

Rappel : les listes en Haskell

`[a]` signifie “liste de `a`”. Exemple : [1,2,3], [‘a‘]

- Éléments de type uniforme.
- Taille variable entre 0 et ∞.

Concéption : chaque liste a une “tête” (premier élément) et une “queue” (le reste)

- Les listes sont construites avec l’opération `:`
- `(:) :: a -> [a] -> [a]`
- Explication : `(:) h t` (plus souvent écrit `h:t`) retourne la liste dont le premier élément est `h` et le reste est la liste `t`
- Exemples : `1:[2,3] == [1,2,3]`, `[1]:[] == [[1]]`

Les Listes – Opérations

- Opérations principales
  - `(:)` construit une liste, étant donné le premier élément et le reste
  - `head :: [a] -> a` retourne le premier élément
  - `tail :: [a] -> [a]` supprime premier élément, retourne reste
  - `last :: [a] -> a` retourne dernier élément
  - `(++) :: [a] -> [a] -> [a]` concaténation de deux listes
  - `(!!) :: [a] -> Int -> a` random access
  - `length :: Foldable t => t a -> Int` taille (??)
  - `take :: Int -> [a] -> [a]` retourne n premiers éléments
  - `drop :: Int -> [a] -> [a]` supprime n premiers éléments
  - `reverse :: [a] -> [a]`
  - `repeat :: a -> [a]` liste infinie qui répéte le même élément
  - `[i..j]` liste de `Int`(?) entre i et j

Exemple:

```
> (take 3 (repeat 5)) ++ (take 2 (repeat 4))
[5,5,5,4,4]
```

Listes en compréhension

- Pour construire une sous-liste (ou un sous-ensemble) on utilise souvent une notation “mathématique”
- Opérateur <- (c’est-à-dire $\in$)

Exemple :

```
Prelude> [ x | x<-[2..5] ]
[2,3,4,5]
Prelude> [ x*3 | x<-[2..5] ]
[6,9,12,15]
```

On peut utiliser plusieurs variables :

```
Prelude> [ (x,y) | x<-[1..3], y<-[5..7] ]
[(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),(3,7)]
Prelude> [ (x,y) | y<-[5..7], x<-[1..3] ]
[(1,5),(2,5),(3,5),(1,6),(2,6),(3,6),(1,7),(2,7),(3,7)]
```

Listes en compréhension

D'autres exemples :
On peut utiliser `x` pour définir l'intervalle de `y` (mais pas l'inverse, l'ordre de définition est importante)

```
Prelude> [ (x,y) | x<-[1..3], y<-[(x+2)..(x+4)] ]
[(1,3),(1,4),(1,5),(2,4),(2,5),(2,6),(3,5),(3,6),(3,7)]
```

On peut ajouter des conditions de sélection

```
Prelude> [ (x,x*x) | x<-[1..10], x `mod` 3 == 0 ]
[(3,9),(6,36),(9,81)]
```

Que font ces fonctions ? (On utilise le pattern matching pour les conditions de sélection)

```
concat' :: [[a]] -> [a]
concat' xss = [ x | xs <- xss, x<-xs]


firsts :: [(a,b)] -> [a]
firsts xs = [ x | (x,_) <- xs ]
```

Exemple : Nombres Premiers

La fonction suivante calcule tous les facteurs de n

```
factors :: Int -> [Int]
factors n = [ x | x<-[1..n], n `mod` x == 0]
```

Donc, une façon de décider si un nombre est premier est

```
prime :: Int -> Bool
prime n = (factors n) == [1,n]
```

```
*Main> factors 24
[1,2,3,4,6,8,12,24]
*Main> factors 17
[1,17]
*Main> prime 17
True
*Main> take 15 [ x | x<-[1..], prime x ]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

Ranges

- On a utilisé la notation `[1..10]` pour signifier une intervalle.
- Syntaxe générale : `[a,b,..,c]` où l'élément `b` est facultatif
  - Sens : si `b=a+s` on a la liste `[a,a+s,a+2s,..,]`
  - La suite arithmétique qui commence par `a,b` et contient tous les termes $\leq$`c`.

Exemples :

```
*Main> [1,4..15]  -- donne [1,4,7,10,13]
*Main> [20..1] -- donne []
*Main> [20,19..1] -- donne
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
*Main> [1,2,4,8..64]
 -- Error!
*Main> ['a'..'g'] -- donne "abcdefg" (cf. Enum class)
*Main> [0.1,0.3..1.0]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
 -- ??? déconseillé pour les Float..
```

L'opération zip

Opération élémentaire pour combiner deux listes

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

- Étant donné deux listes, produit leur combinaison.
- Si listes de taille différente, on arrête quand la liste la plus courte est finie.

```
*Main> zip [1..5] [1..4]
[(1,1),(2,2),(3,3),(4,4)]
*Main> zip [1..5] "Haskell"
[(1,'H'),(2,'a'),(3,'s'),(4,'k'),(5,'e')]
```

(Rappel : String = [Char])

L'opération zip

Que fait la fonction suivante ?

```
testme :: Ord a => [a] -> Bool
testme xs = pairs == pairs'
  where
   pairs = zip xs (tail xs)
   pairs' = [ (a,b) | (a,b) <- pairs, a<=b ]
```

- `Ord a =>` signifie qu'on peut comparer les éléments de type `a`
- `pairs` est une liste de taille `(length xs)-1` (pourquoi?)
- Chaque élément de `pairs` est une paire de deux éléments consécutifs de `xs`
- `pairs'` est une sous-liste de `pairs`
- **NB**: la variable `a` de la dernière ligne n'a rien à voir avec la variable `a` de la déclaration de type.

Fonctions d'ordre supérieur

Rappel : fonction d'ordre supérieur == fonction qui prend comme argument fonction

```
applytwice :: (Int -> Int) -> Int -> Int
applytwice f x = f (f x)

*Main> applytwice (+2) 5
9
```

Q: Peut–on donner un type plus générique à la fonction `applytwice` ?

- `(a->b)->c->d` ?
- `(a->b)->a->b` ?
- `(a->a)->a->a` ?

NB: on va appliquer `f` sur ce qui est retourné par `f`.

Fonctions d'ordre supérieur et listes

Pour manipuler des listes on utilise souvent des fonctions d'ordre supérieur comme :

- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `foldl::Foldable t=>(b -> a -> b) -> b -> t a -> b`
  - C'est quoi `Foldable` ??
  - Peu importe. Pour le moment on va supposer que
    `foldl :: (b->a->b) -> b -> [a] ->b`
- `foldr` (presque même type que `foldl`)

Explications :

- `map` applique fonction sur chaque élément, retourne liste des résultats.
- `filter` supprime éléments qui ne satisfont pas une condition.
- `foldl` applique opération binaire sur un élément initiale et chaque élément de la liste (on verra comment).

Map

- La fonction `map` est déjà définie.
- Cependant, c'est facile de donner une implémentation récursive.

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

- Pensez à bien vérifier les types de toutes les expressions de cette définition...

Map

Exemples d'utilisation :

```
*Main> map even [1..4]
[False,True,False,True]
*Main> map (+2) [1..4]
[3,4,5,6]

*Main> map length ["I", "love", "Haskell"]
[1,4,7]

*Main> map (map (+2)) [ [1,2], [3], [4] ]
[[3,4],[5],[6]]
```

Filter

- La fonction `filter` est déjà définie.
- Voici deux implémentations possibles

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

ou plus facilement

```
filter :: (a->Bool) -> [a] -> [a]
filter f xs = [ x | x<-xs, f x ]
```

Filter

Exemples d'utilisation :

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

*Main> filter (>5) [1..10]
[6,7,8,9,10]

*Main> filter (/= 'e') "I love Haskell"
"I lov Haskll"

*Main> map (^2) (filter even [1..10])
[4,16,36,64,100]

*Main> map (2^) (filter even [1..10])
[4,16,64,256,1024]
```

Fold

On considère les fonctions suivantes : (que font–elles ?)

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs

product :: Num a => [a] -> a
product [] = 1
product (x:xs) = x * product xs

or :: [Bool] -> Bool
or [] = False
or (x:xs) = x || or xs

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

Fold – Motivation

Dans tous les cas on a :

- Une valeur de base
  - 0, 1, `False`, `True`
- Une opération binaire
  - $+$, `*`, `||`, `&&`

À partir de la valeur de base a_0, l'opération binaire $\oplus$, et la liste $[a_1, a_2, \ldots, a_n]$ on calcule

$$(((a_0 \oplus a_1) \oplus a_2) \oplus a_3) \ldots \oplus a_n) \ldots)$$

ou

$$(a_1 \oplus (a_2 \oplus (a_3 \oplus \ldots (a_n \oplus a_0) \ldots)$$

- Les fonctions `foldl` et `foldr` représentent ces deux opérations génériques.

Foldr

Une définition de `foldr`

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

- Ça fait quoi ??

Rappel : listes sont construites avec `:`

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

L'opération `foldr`

- Remplace les `:` par la fonction `f`
- Remplace le `[]` qui termine la liste par `v`

```
foldr (+) 0 [1,2,3,4,5] == 1+2+3+4+5+0
```

Foldr – Exemples

```haskell
sum :: Num a => [a] -> a
sum = foldr (+) 0
-- ou
sum xs = foldr (+) 0 xs

product :: Num a => [a] -> a
product = foldr (*) 1

or :: [Bool] -> Bool
or = foldr (||) False
```

Foldr – Exemples

```
length :: [a] -> Int
length = foldr ignore 0

ignore :: a -> Int -> Int
ignore _ n = n+1

reverse' :: [a] -> [a]
reverse' = foldr cons' []

cons' :: a -> [a] -> [a]
cons' x xs = xs++[x]
```

NB: `cons'` est l'inverse de l'opération `(:)`, car elle met `x` à la fin de la liste.

Foldl

Et `foldl` ?

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

La différence est dans l'ordre des opérations

`fold? f a0 [a1, a2, a3]`

- Avec `foldl` on fait
  - `f (f (f a0 a1) a2) a3`
- Avec `foldr` on fait
  - `f a1 (f a2 (f a3 a0))`
- Pour les opérations commutatives et associatives, peu importe (sauf questions de complexité !!).

Foldl et Foldr

Exemples :

```
*Main> foldl (+) 0 [1..5]
15
*Main> foldr (+) 0 [1..5]
15
*Main> foldl (-) 0 [1..5]
-15
*Main> foldr (-) 0 [1..5]
3
```

Pour le dernier exemple on a :

$$(1 - (2 - (3 - (4 - (5 - 0))))) = (1 - (2 - (3 - (4 - 5)))) = 3$$

La composition

- Quand on utilise des fonctions d'ordre supérieur, c'est souvent utile de les composer.
- Exemple :

```
odd x = not (even x)
```

- Pour simplifier ce type d'expression, et pour éviter de définir des fonctions inutiles, on peut utiliser l'opérateur `.`

```
*Main> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
*Main> map ( (2+) . (3*) ) [1..5]
[5,8,11,14,17]
```

- Donc `odd = not . even` est une définition équivalente (plus concise).
- Cette définition a l'avantage qu'on n'est pas obligés de nommer la fonction `odd`
- L'expression `not . even` est légale et a type `Int -> Bool`

Composition – Exemple

- Quels sont les types des expressions suivantes ?
- Quel est le résultat de la dernière ligne ?

```
compose = foldr (.) id
```

```
id x = x
```

```
*Main> map (compose [ (+1), (*2), (+1) ] ) [1..5]
```

Composition – Exemple

- Quels sont les types des expressions suivantes ?
- Quel est le résultat de la dernière ligne ?

```
compose :: [a->a] -> a -> a
compose = foldr (.) id

id :: a -> a
id x = x

*Main> map (compose [ (+1), (*2), (+1) ] ) [1..5]
```

Composition – Exemple

- Quels sont les types des expressions suivantes ?
- Quel est le résultat de la dernière ligne ?

```
compose :: [a->a] -> a -> a
compose = foldr (.) id

id :: a -> a
id x = x

*Main> map (compose [ (+1), (*2), (+1) ] ) [1..5]
[5,7,9,11,13]
```

Expressions Lambda

- Comme on a vu, très souvent on a besoin de définir une fonction (temporaire) qui va être passée comme paramètre vers une fonction d'ordre supérieur.
- Une façon générale de le faire (sans nommer la fonction) utilise les **expressions lambda**

Exemple:

```
*Main> map (\x -> x + 5) [1..4]
[6,7,8,9]
```

- L'opérateur `\` représente la lettre grecque λ.
- Origine : le système de logique appelé **lambda calcul**.
  - On en reparlera...
- Syntaxe :
  ```
  \x1 x2 x3.. xn -> <corps de la fonction>
  ```
- Donc `id = \x -> x` est une autre définition de la fonction `id`.

Foldr, Foldl, et Map – Exemple avec Lambda

Les fonctions `foldr` et `foldl` sont plus générales que `map`. En fait, `map` peut être implémenté en utilisant ces deux fonctions :

```
myMapWithFoldr :: (a -> b) -> [a] -> [b]
myMapWithFoldr f xs = foldr (\x p -> f x : p) [] xs

myMapWithFoldl :: (a -> b) -> [a] -> [b]
myMapWithFoldl f xs = foldl (\p x -> p ++ [f x]) [] xs

Prelude> myMapWithFoldr length ["abc", "de", "f"]
[3,2,1]
Prelude> myMapWithFoldl length ["abc", "de", "f"]
[3,2,1]
```

Foldr, Foldl, et Map – Exemple avec Lambda

Que font les lignes suivantes ?

```
Prelude> foldr (\x xs -> xs ++ [x]) [] [1,2,3]
```

```
Prelude> foldl (\xs x -> x : xs) [] [1,2,3]
```

```
Prelude> foldr (:) [1,2,3] [4,5,6]
```

```
Prelude> foldl (\xs x -> xs++[x]) [1,2,3] [4,5,6]
```

Foldr, Foldl, et Map – Exemple avec Lambda

Que font les lignes suivantes ?

```
Prelude> foldr (\x xs -> xs ++ [x]) [] [1,2,3] -- reverse
[3,2,1]

Prelude> foldl (\xs x -> x : xs) [] [1,2,3] -- reverse !
[3,2,1]

Prelude> foldr (:) [1,2,3] [4,5,6] -- ++ !
[4,5,6,1,2,3]

Prelude> foldl (\xs x -> xs++[x]) [1,2,3] [4,5,6] -- ++ !
[1,2,3,4,5,6]
```

zipWith

- `zipWith` est une autre fonction d'ordre supérieur de la bibliothèque de Haskell.
- Pouvez–vous deviner son type ?
- Pouvez–vous redéfinir `zip` en utilisant `zipWith` ?

Exemple:

```
*Main> zipWith (*) [1..5] [2..6]
[2,6,12,20,30]
```

zipWith

- `zipWith` est une autre fonction d'ordre supérieur de la bibliothèque de Haskell.
- Pouvez–vous deviner son type ?
- Pouvez–vous redéfinir `zip` en utilisant `zipWith` ?

Exemple:

```
*Main> zipWith (*) [1..5] [2..6]
[2,6,12,20,30]

*Main> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

zipWith

- `zipWith` est une autre fonction d'ordre supérieur de la bibliothèque de Haskell.
- Pouvez–vous deviner son type ?
- Pouvez–vous redéfinir `zip` en utilisant `zipWith` ?

Exemple:

```
*Main> zipWith (*) [1..5] [2..6]
[2,6,12,20,30]

*Main> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

*Main> zipWith (\x y -> (x,y)) [1..] "Haskell"
[(1,'H'),(2,'a'),(3,'s'),(4,'k'),(5,'e'),(6,'l'),(7,'l')]

zip' = zipWith \x y -> (x,y)
-- ou
zip' = zipWith (,)
```

Pour une poignée de dollars

- Un autre opérateur qui est souvent utile quand on définit une expression–fonction : `$`
- Syntaxe : `f $ x` veut dire qu'on applique la fonction `f` sur l'argument `x`.
- `:t ($)` donne `(a->b)->a->b`
- Pourquoi a–t–on besoin de ça ???
  - Rappel : `f x` veut dire on applique `f` sur `x`
  - Différence : l'opération `f x` a toujours la priorité et associe à gauche.
  - `a b c d` veut dire `((a b) c) d`
  - C'est l'inverse pour l'opérateur `$`
  - `a $ b $ c $ d` veut dire `a (b (c d))`
  - `a $ b c` veut dire `a (b c)`

Exemples – Dollars

```
*Main> sum $ filter (> 10) $ map (*4) [1..5]
48
*Main> sum (filter (> 10) (map (*4) [1..5]))
48
```

- Les deux expression sont équivalentes
- Le `$` nous permet de supprimer les parenthèses.

```
*Main> map ($ 3) [ (* x) | x<-[2..5] ]
[6,9,12,15]
```

- Le `$` nous a permis d'exprimer l'application d'une fonction, comme fonction.
- Avec expression lambda on aurait écrit :

```
*Main> map (\x -> x 3) [ (* x) | x<-[2..5] ]
[6,9,12,15]
```