PLP

Higher Order Functions

26 October 2025

Table des matières

1 Higher-Order Functions

1.1 Introduction

Les fonctions d'ordre supérieur (Higher-Order Functions, HOF) sont des fonctions qui peuvent prendre d'autres fonctions en argument ou retourner des fonctions en résultat. Elles sont un concept fondamental en programmation fonctionnelle et permettent de créer des abstractions puissantes et réutilisables.

En définissant la fonction `sum` qui prend une fonction `f` et deux entiers `a` et `b`, nous pouvons calculer la somme des résultats de `f` appliquée à chaque entier entre `a` et `b`.

```
sum f a b
  | a > b     = 0
  | otherwise = f a + sum f (a + 1) b
```

Cette fonction peut être utilisée pour calculer la somme des carrés ou la somme des cubes en passant les fonctions `square` et `cube` respectivement.

```
sumInts a b = sum id a b
sumSquares a b = sum square a b
sumCubes a b = sum cube a b
sumFactorials a b = sum factorial a b
```

1.2 Lambda expressions

Les expressions lambda sont des fonctions anonymes qui peuvent être définies sans nom. Elles sont souvent utilisées comme arguments pour les fonctions d'ordre supérieur. Si nous souhaitons créer une fonction lambda qui calcule le cube d'un nombre, nous pouvons le faire de la manière suivante :

```
\x -> x * x * x
```

La ou `\x` indique que nous définissons une fonction prenant un argument `x`, et le corps de la fonction est `x * x * x`.

1.3 Fermetures

Une fermeture est une fonction qui capture les variables de son environnement lexical. Cela signifie qu'une fermeture peut accéder aux variables définies en dehors de son propre corps. Par exemple, si nous avons une fonction qui crée une fonction de multiplication par un facteur donné, nous pouvons utiliser une fermeture pour capturer ce facteur.

```
makeMultiplier factor = \x -> x * factor
```

Dans cet exemple, `makeMultiplier` est une fonction qui prend un argument `factor` et retourne une fonction qui multiplie son argument `x` par ce facteur. La fonction retournée est une fermeture car elle « capture » la variable `factor` de son environnement lexical. Pour utiliser cette fonction, nous pouvons faire :

```
double = makeMultiplier 2
triple = makeMultiplier 3

// puis

result1 = double 5  // result1 sera 10
result2 = triple 5  // result2 sera 15
```

2 Currying (Fonctions currifiées)

Le currying est une technique de transformation de fonctions qui permet de convertir une fonction prenant plusieurs arguments en une série de fonctions prenant un seul argument chacune. En Haskell, toutes les fonctions sont currifiées par défaut. Par exemple, une fonction qui prend deux arguments peut être vue comme une fonction qui prend un argument et retourne une fonction qui prend le second argument.

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

D'un point de vue de signature, ces deux fonctions sont équivalentes :

```
add :: Int -> Int -> Int
```

Cela signifie que `add` est une fonction qui prend un `Int` et retourne une fonction qui prend un autre `Int` et retourne un `Int`.

> **i Info**
>
> Le principe de la currification consiste à transformer une fonction prenant un tuple en argument en une fonction prenant le premier élément du tuple et retournant une fonction prenant le second élément du tuple, et ainsi de suite.

- `curry` : transforme une fonction prenant un tuple en une fonction currifiée.
- `uncurry` : transforme une fonction currifiée en une fonction prenant un tuple.

2.1 Partial application (Application partielle)

L'application partielle est le processus de fixation d'un certain nombre d'arguments d'une fonction pour produire une nouvelle fonction avec un nombre réduit d'arguments. En Haskell, grâce au currying, l'application partielle est naturelle et couramment utilisée.

2.1.1 Application operator

L'opérateur `$` est utilisé pour appliquer une fonction à un argument, en évitant la nécessité de parenthèses. Par exemple, au lieu d'écrire `f (g x)`, on peut écrire `f $ g x`.

2.1.2 Flip operator

L'opérateur `flip` prend une fonction à deux arguments et retourne une nouvelle fonction avec les arguments inversés. Par exemple, si nous avons une fonction `subtract x y = x - y`, nous pouvons utiliser `flip` pour créer une fonction qui soustrait dans l'ordre inverse.

3 List processing

Les fonctions d'ordre supérieur sont souvent utilisées pour traiter des listes. Voici quelques fonctions couramment utilisées pour manipuler des listes en Haskell.

- `map` : applique une fonction à chaque élément d'une liste et retourne une nouvelle liste avec les résultats.
- `filter` : filtre les éléments d'une liste en fonction d'une condition donnée.

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

filter even [1..10]  // Résultat : [2,4,6,8,10]
filter (>5) [1..10]  // Résultat : [6,7,8,9,10]
```

4 Folding (Réduction)

Le folding est une technique de réduction d'une liste à une seule valeur en appliquant une fonction binaire de manière récursive. En Haskell, il existe deux fonctions principales pour le folding : `foldl` (fold left) et `foldr` (fold right).

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

- `foldr` : commence à la fin de la liste et applique la fonction binaire de droite à gauche.
- `foldl` : commence au début de la liste et applique la fonction binaire de gauche à droite.

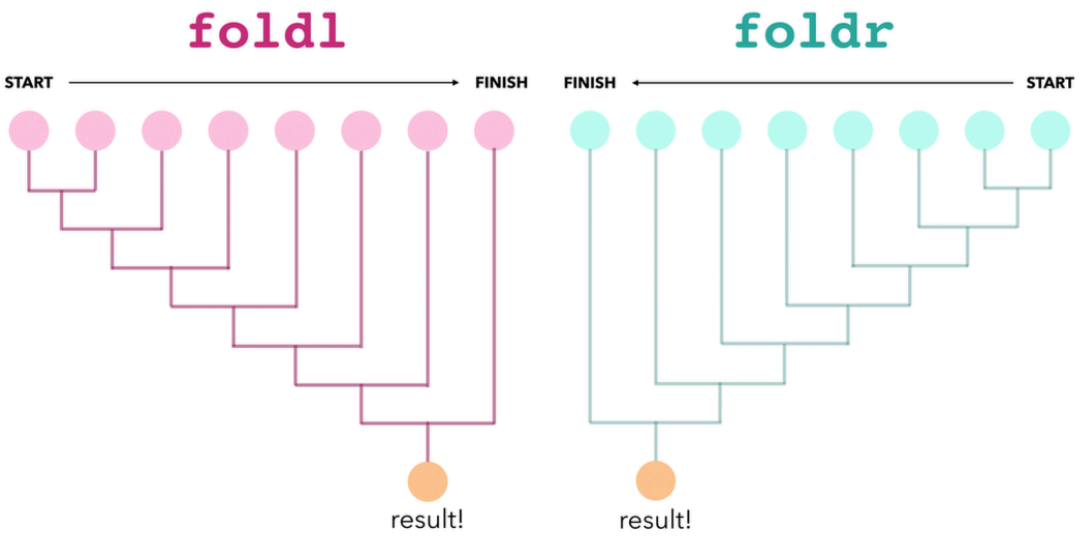

Fig. 1. – Capture des slides du cours – Folding

4.1 Fold right

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

```
foldr (+) 0 [1,2,3,4,5]
1 + (foldr (+) 0 [2,3,4,5])
1 + (2 + (foldr (+) 0 [3,4,5]))
1 + (2 + (3 + (foldr (+) 0 [4,5])))
1 + (2 + (3 + (4 + (foldr (+) 0 [5]))))
1 + (2 + (3 + (4 + (5 + (foldr (+) 0 [])))))
1 + (2 + (3 + (4 + (5 + (0)))))
...
15
```

Fig. 2. – Capture des slides du cours – Fold right

4.2 Fold left

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

```
foldl (+) 0 [1,2,3,4,5]
foldl (+) (0 + 1) [2,3,4,5]
foldl (+) (0 + 1 + 2) [3,4,5]
foldl (+) (0 + 1 + 2 + 3) [4,5]
foldl (+) (0 + 1 + 2 + 3 + 4) [5]
foldl (+) (0 + 1 + 2 + 3 + 4 + 5) []
...
15
```

Fig. 3. – Capture des slides du cours – Fold left

4.3 Associativity and commutativity

L'associativité et la commutativité sont des propriétés importantes des opérations binaires qui influencent le comportement des fonctions de folding.

```
foldr (^) 2 [1..3]
(1 ^ (2 ^ (3 ^ 2)))
(1 ^ (2 ^ 9))
1 ^ 512
1
```

```
foldl (^) 2 [1..3]
((2 ^ 1) ^ 2) ^ 3
(2 ^ 2) ^ 3
4 ^ 3
64
```

Fig. 4. – Capture des slides du cours – Associativity and commutativity

> ⚠ **Warning**
>
> En fonction de notre besoin, nous choisirons `foldl` ou `foldr`. Par exemple, pour élever une liste de nombres à une puissance, nous utiliserons `foldl` car l'opération d'exponentiation n'est pas associative.

Le cas de la soustraction est plus délicat car elle n'est ni associative ni commutative. Par exemple, `foldl (-) 0 [1,2,3]` donne `0 - 1 - 2 - 3 = -6`, tandis que `foldr (-) 0 [1,2,3]` donne `1 - (2 - (3 - 0)) = 2`. Le choix entre `foldl` et `foldr` dépendra donc du résultat souhaité.

5 Function composition

La composition de fonctions est une technique qui permet de combiner plusieurs fonctions en une seule. En Haskell, l'opérateur `(.)` est utilisé pour composer deux fonctions.

```
filter (odd . fst) [(65, "A"), (66, "B"), (67, "C")]
// Résultat : [(65,"A"),(67,"C")]
```