Introduction à la programmation fonctionnelle

map n00b2pgm audience

Nantes Functional Programming Group

14 mai 2012

Table of Contents

1. La pensée fonctionnelle
   - Le clash des visions
2. Les concepts clefs
   - Les fonctions
   - La transparence référentielle
   - L'évaluation paresseuse
   - La récursion
   - La co-récursion

La programmation fonctionnelle quand on vient de l'impératif, c'est avant tout un autre paradigme.

paradigme @Wikipédia

Un paradigme est une représentation du monde, une manière de voir les choses.

Le paradigme impératif

On dicte à l'ordinateur des modifications à exécuter sur l'état du programme :

- on déclare des variables (allocation de mémoire)
- on les transforme (modification de la mémoire allouée)
- on les détruit (désallocation de mémoire)

Le paradigme fonctionnel

On abstrait une partie de l'implémentation pour se concentrer sur les données : on déclare la relation qui existe entre les données que l'on reçoit et celles que l'on recherche

Intuition de la différence entre les deux

Au lieu de dire comment on fait les choses, on dit simplement que les choses doivent être faites. Le reste relève du talent des codeurs de GHC (ou équivalent pour les autres langages :)).
Par exemple, pour calculer la somme des nombres allant de 1 à n, en C à gauche et en Haskell à droite, on dirait :

```c
int f(int n) {
  int result = 0, i = 1;
  for(; i <= n; i++)
    result += i;
  return result;
}
```

```haskell
f :: Integer -> Integer
f n = sum [1..n]
```

Table of Contents

1 La pensée fonctionnelle
- Le clash des visions

2 Les concepts clefs
- Les fonctions
- La transparence référentielle
- L'évaluation paresseuse
- La récursion
- La co-récursion

Dégrossir le concept de fonction

Une fonction est une boîte noire qui transforme un objet en un autre objet.

Un peu de formalisme

Une fonction se caractérise par :

- son domaine : l'ensemble duquel est issu son argument
- son codomaine : l'ensemble auquel appartient le résultat
- la manière dont elle transforme l'argument en le résultat

Les fonctions : exemple avec la fonction carré

En français

- son argument est un entier naturel
- son résultat est un entier naturel
- elle retourne son argument au carré

En maths

$$\begin{array}{rccc} \texttt{carré} : & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & x^2 \end{array}$$

En Haskell

```
square :: Integer -> Integer
square n = n ^ 2
```

En quoi sont-elles un concept clef

Pour décrire les données que l'on souhaite obtenir à partir des données de départ dans un programme, on exprime des transformations.
Les fonctions sont la manière la plus simple de les spécifier.

Ce qu'est un programme fonctionnel

On pourrait voir un programme fonctionnel comme un enchaînement d'appels de fonctions sur les données de départ :

$$d_i \xrightarrow{f} d_1 \xrightarrow{g} d_2 \xrightarrow{h} d_f$$

La composition

Composer deux fonctions, c'est les "fusionner" pour obtenir la fonction qui se définit par l'application des deux fonctions d'origine.

En version matheuse

si $f : \begin{array}{ccc} B & \to & C \\ b & \mapsto & c \end{array}$ et $g : \begin{array}{ccc} A & \to & B \\ a & \mapsto & b' \end{array}$

alors $f \circ g : \begin{array}{ccc} A & \to & C \\ a & \mapsto & f(g(a)) \end{array}$

Ce qu'est un programme fonctionnel bis

On peut reprendre la vision proposée d'un programme fonctionnel :

$$d_i \xrightarrow{f} d_1 \xrightarrow{g} d_2 \xrightarrow{h} d_f$$

Et en proposer une autre, cette fois en utilisant la composition de fonctions :

$$d_i \xrightarrow{h \circ g \circ f} d_f$$

Modularité : si on a $g = g'$, alors, le programme suivant est équivalent au programme ci-dessus :

$$d_i \xrightarrow{h \circ g' \circ f} d_f$$

Table of Contents

1 La pensée fonctionnelle
- Le clash des visions

2 Les concepts clefs
- Les fonctions
- La transparence référentielle
- L'évaluation paresseuse
- La récursion
- La co-récursion

Ce qu'est la transparence référentielle

La transparence référentielle, c'est le fait de pouvoir interchanger l'appel d'une fonction et son résultat indifférement.

Pourquoi seulement en fonctionnel ?

- nécessité qu'une fonction retourne toujours le même résultat pour un argument donné
- nécessité qu'une fonction n'ait pas d'effets de bord

Ces deux points ne sont souvent pas respectés en impératif. De fait, les fonctions impératives sont plus des procédures.

Ce que ça implique

- plus facile de raisonner sur le comportement d'une fonction
- optimisations à l'exécution (mémoization, cache, parallélisation)
- testabilité garantie

Table of Contents

1 La pensée fonctionnelle
- Le clash des visions

2 Les concepts clefs
- Les fonctions
- La transparence référentielle
- L'évaluation paresseuse
- La récursion
- La co-récursion

Ce qu'est l'évaluation paresseuse

On évalue un bloc que lorsque l'on en a besoin (donc potentiellement jamais).

Ce que ça implique

- un langage paresseux peut manipuler des structures de données infinies
- un language paresseux est **non strict**. L'appel d'une fonction avec un argument non défini n'est pas forcément non défini.

Table of Contents

1 La pensée fonctionnelle
- Le clash des visions

2 Les concepts clefs
- Les fonctions
- La transparence référentielle
- L'évaluation paresseuse
- La récursion
- La co-récursion

Ce qu'est la récursion

La récursion est le fait pour une fonction de s'appeller soi-même (directement ou indirectement).
La définition par récurrence est quant à elle le fait de définir la valeur d'un élément d'une suite en fonction de la valeur de l'élément qui le précède. Il suffit ensuite de connaître la valeur d'un élément pour obtenir la valeur de tous ses successeurs.

Intuition sur la récursion

Pour savoir monter à n'importe quel barreau d'une échelle, il suffit de :

- savoir monter sur le premier barreau
- savoir monter d'un barreau au barreau suivant

La définition par récurrence est analogue : avec une *initialisation* et une *propriété d'hérédité*, on définit toute une suite.

Exemples

La suite des nombres pairs (P_n)

$$\begin{cases} P_0 = 0 \\ \forall n \geq 0, \quad P_{n+1} = P_n + 2 \end{cases}$$

La suite des nombres de Fibonacci (F_n)

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \geq 0, \quad F_{n+2} = F_{n+1} + F_n \end{cases}$$

Exemples - suite

On peut s'amuser avec des exemples un petit peu plus tordus – ici on définit deux suites en même temps :

Les suites des nombres pairs (P_n) et impairs (I_n)

$$\begin{cases} P_0 = 0 \\ \forall n \geq 0, \quad P_{n+1} = I_n + 1 \end{cases} \qquad \forall n \geq 0, \quad I_n = P_n + 1$$

Table of Contents

1 La pensée fonctionnelle
- Le clash des visions

2 Les concepts clefs
- Les fonctions
- La transparence référentielle
- L'évaluation paresseuse
- La récursion
- La co-récursion

Ce qu'est la co-récursion

La co-récursion, on peut dire pour simplifier que c'est la récursion sur le résultat (le co-domaine) plutôt que l'argument (le domaine).

Rappel sur la récursion

Quand on utilise une définition par récurrence, le but est de successivement s'appeller avec des arguments "plus petits" jusqu'à atteindre un cas de base.

La co-récursion

Quand on utilise une définition par co-récurrence, le but est de construire le résultat en s'appellant successivement avec des arguments "plus grands". Il peut y avoir ou non un cas d'arrêt (souvent il n'y en a pas).

Exemples

Rappel de récursion

```
pair :: Integer -> Integer
pair 0 = 0
pair x = pair (x - 1) + 2
```

Co-récursion

```
pairs :: [Integer]
pairs = 0 : map (+ 2) pairs
```

On peut être les deux...

Comme map...

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

... ou zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []
```

Utilité

La co-récursion est extrêmement utile pour spécifier les listes infinies (qu'on appelle aussi streams). Quelques exemples canoniques sont:

Les nombres de Fibonacci

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Les factorielles

```
factorials = [Integer]
factorials = 1 : zipWith (*) [1..] factorials
```