Programmation Fonctionnelle
Functors

Michael Lampis

2023-2024

Functor – Version dure

- La classe `Functor` représente une notion importante pour la programmation fonctionnelle.
- C'est quoi un `Functor` (foncteur) ?

Définition selon Wikipedia :

Un **foncteur** $F : \mathcal{C} \to \mathcal{D}$ d'une catégorie $\mathcal{C}$ dans une catégorie $\mathcal{D}$ est la donnée

1. d'une fonction qui, à tout objet $X \in \mathcal{C}$, associe un objet $F(X) \in \mathcal{D}$
2. d'une fonction qui, à tout morphisme $f : X \to Y$ de $\mathcal{C}$, associe un morphisme $F(f) : F(X) \to F(Y)$ de $\mathcal{D}$

- Donc un functor est quelque chose qui "map" une catégorie (??) sur une autre catégorie, tout en préservant les morphismes (???)
- ???

Functor – Version douce

Rappelez–vous des types (paramétriques) suivants :

- `Maybe a`
- `[a]`
- `BinTree a`

Quels sont les points communs des opérations `Maybe`, `[]`, `BinTree` ?

- Aucun de trois n'est un type !
- Tous les trois sont de "constructeurs de type"
  - Étant donné un type `a` ils produisent un nouveau type `Maybe a, [a], BinTree a`
- Conceptuellement, tous les trois donnent un nouveau type "container"
  - Chaque élément du nouveau type est une collection d'éléments de type `a`
- Pratiquement, on aura envie d'appliquer des fonctions (morphismes!) sur les contenus d'une telle collection pour faire quoi que ce soit.

Be kind, rewind

- Point à souligner : `Maybe, [], BinTree` ne sont pas de types.
- Alors ils sont quoi pour Haskell ?
- On peut poser cette question directement au compilateur :

```
*Main> :kind Maybe
Maybe :: * -> *
```

- Commande `:kind` == c'est quoi ce truc ?
- Dans ce contexte : `*` signifie "type concret"

```
*Main> :kind []
[] :: * -> *
*Main> :kind [Int]
[Int] :: *
*Main> :kind Int->Int
Int->Int :: *
```

Question avancée : `:kind (->)` ?

Back to Functors

- Plus pertinent : les trois “types de types” (`Maybe`, `[]`, `BinTree`) jouent les rôles de “containers”.
- Ce rôle est représenté en Haskell par la classe `Functor`

```
*Main> :info Maybe
instance Functor Maybe -- Defined in 'GHC.Base'
*Main> :info []
instance Functor [] -- Defined in 'GHC.Base'
```

- Donc, c’est quoi cette classe ?

```
*Main> :info Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  {-# MINIMAL fmap #-}
```

Functors – Décryptage

```
*Main> :info Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  {-# MINIMAL fmap #-}
```

- Un functor a kind `*->*`
- Donc, si `a` est un type et `f` un functor, `f a` est un type.
- Un objet `f` qui a kind `*->*` est un functor s'il implémente la fonction `fmap`
  - qui prend une fonction `a->b` et un objet de type `f a` et retourne un `f b`
  - Traduction : prend une fonction `a->b` et un "container" de `a` et retourne un container de `b`
- Leçon : **functor** == constructeur de types sur lesquels on peut utiliser `map` (`fmap`)

Functors – Pourquoi ?

- On a vu plusieurs niveaux d'abstraction
  - `f:: Num a=> a->a` et pas `f::Int->Int`, `f2::Double->Double`,... (fonctions polymorphiques)
  - `BinTree a` et pas `BinTreeInt, BinTreeDouble`,... (types paramétriques)
  - `map, filter, foldr`,... (fonctions d'ordre supérieur)
- Dans tous les cas, l'intérêt est de localiser ce que ces objets/opérations ont en commun, **le distiller, et le programmer une seule fois**.
- La motivation est similaire pour `Functor`. On peut écrire une fonction qui prend comme paramètre un "container", sans se soucier de son type précis, et fait quelque chose avec tous ses éléments.

Functors – Exemple

Comparer les exemples suivants :

```
-- Sol 1
doubleAll :: [Int] -> [Int]
doubleAll [] = []
doubleAll (x:xs) = (2*x):doubleAll xs

-- Sol 2
doubleAll :: Num a => [a] -> [a]
doubleAll = map (2*)

-- Sol 3
doubleAll :: (Num a, Functor f) => f a -> f a
doubleAll = fmap (2*)
```

- `map` (Sol2) nous permet d'unifier toutes les opérations du même type
- `Functor` (Sol3) nous permet d'aller encore plus loin...

Functors – Exemple

```
doubleAll :: (Num a, Functor f) => f a -> f a
doubleAll = fmap (2*)
```

Exemples d'utilisation :

```
*Main> doubleAll [1,2,3]
[2,4,6]
*Main> doubleAll (Just 3)
Just 6
```

- Ça marche pour n'importe quel "container type" qui fait partie de la classe `Functor` !!
- Pas besoin de réécrire la même chose pour `[]` et `Maybe`, puisque `fmap` nous offre une généralisation gratuite.
- Objectif : on aimerait que ça marche aussi pour `BinTree`, sans qu'on réécrive `doubleAll`.

```
*Main>doubleAll (list2tree [1..5]) --Ne marche pas encore
2 (4 () (6 () ())) (8 () (10 () ()))
```

Notre propre Functor

Comment faire `BinTree` une instance de `Functor` ?

```
data BinTree a = Leaf
               | Branch (BinTree a) a (BinTree a)
```

Rappel :

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  {-# MINIMAL fmap #-}
```

Donc, il faut implémenter `fmap`

```
instance Functor BinTree where
 fmap _ Leaf = Leaf
 fmap f (Branch l x r) = Branch
                           (fmap f l) (f x) (fmap f r)
```

Functors qui existent

- D'ailleurs, la bibliothèque de Haskell utilise la même méthode pour que `Maybe,[]` soient instances de `Functor`

```
instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)

instance Functor [] where
    fmap = map
```

- Dans les deux cas, on donne au compilateur une fonction qui permet d'appliquer `f` sur tous les éléments du container et obtenir un container avec les résultats.
- Rappel : on a donné une implémentation récursive directe pour `map :: (a->b)->[a]->[b]`

Notre propre Functor

Rappel :

```
doubleAll :: (Num a, Functor f) => f a -> f a
doubleAll = fmap (2*)

instance Functor BinTree where
 fmap _ Leaf = Leaf
 fmap f (Branch l x r) = Branch
                           (fmap f l) (f x) (fmap f r)
```

Donc on peut maintenant faire

```
*Main>doubleAll (list2tree [1..5]) -- OK!
2 (4 () (6 () ())) (8 () (10 () ()))
```

Functors inutiles

- Pourrait–on écrire la définition suivante pour `fmap` de `BinTree` ?

```
instance Functor BinTree where
  fmap _ _ = Leaf
```

- Esc-que ça compile ?
- Quel est l'intérêt ?

Functors inutiles

- Pourrait–on écrire la définition suivante pour `fmap` de `BinTree` ?

```
instance Functor BinTree where
 fmap _ _ = Leaf
```

- Rappel : dans ce contexte
  `fmap :: (a->b) -> BinTree a -> BinTree b`.
- Cette définition respecte la signature de type !
- Donc, malheureusement, ça compile...
- Informellement, ce n'est pas du tout une bonne implémentation de la classe `Functor` pour `BinTree`
- Quelles sont les règles que chaque "bonne implémentation" devrait respecter ?

Les lois des Functors

- Toutes les bonnes implémentation de `fmap` doivent respecter les conditions suivantes :

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

- Appliquer la fonction `id` sur tous les éléments d'un functor devrait retourner le même functor.
- Appliquer la fonction h et puis la fonction g devrait avoir le même résultat qu'appliquer leur composition.

- Les lois semblent tout à fait basiques et raisonnables
- Or, on peut prouver que, pour n'importe quel functor, il existe une définition **unique** qui respecte les lois.
- Donc, si on arrive a implémenter une bonne version de `fmap`, on a **la bonne** version.
- **Attn**: c'est impossible pour le compilateur de vérifier si notre implémentation respecte les lois !

Functors – Version pas si dure

Un **foncteur** $F : \mathcal{C} \to \mathcal{D}$ ~~d'une catégorie $\mathcal{C}$ dans une catégorie $\mathcal{D}$~~ a **kind** $* \to *$, donc de la catégorie de types dans la catégorie de types, est la donnée de

1. d'une fonction qui, ~~à tout objet $X \in \mathcal{C}$, associe un objet $F(X) \in \mathcal{D}$~~ pour chaque type `a` donne un type `f a`
2. d'une fonction qui, ~~à tout morphisme $f : X \to Y$ de $\mathcal{C}$, associe un morphisme $F(f) : F(X) \to F(Y)$ de $\mathcal{D}$~~ pour chaque fonction `a->b` donne une fonction `f a -> f b`

- Donc, un functor de Haskell correspond assez bien à la définition mathématique de la notion de foncteur.
- En fait, si vous lisez le reste de la définition donnée par Wikipedia vous trouverez qu'un foncteur
  1. respecte les identités : pour tout $X \in \mathcal{C}$ on a $F(\mathrm{Id}_X) = \mathrm{Id}_{F(X)}$
  2. respecte la composition : pour tous objets $X, Y, Z \in \mathcal{C}$ et morphismes $f : X \to Y$ et $g : Y \to Z$ on a $F(g \circ f) = F(g) \circ F(f)$
- Qui correspondent aux lois des Functors de Haskell !