

TD 8 : Functors

1 Compréhension des Kinds, Tuples, Functors

On a parlé pendant le cours de la notion de kind. Le kind est une forme de meta-type pour le système de types de Haskell. Faisons un rappel rapide des règles de ce système.

On utilise le symbole `*` pour signifier qu'un objet est un type concret. Par exemple, si on écrit au compilateur `:kind Int` il répond `Int :: *` (`Int` est un type concret). On a vu qu'on a le droit de définir de types paramétriques (p.ex. `Maybe`). Dans ce cas, un tel type n'a pas comme kind `*`, car il n'est pas concret si on ne lui donne pas les paramètres requis. Le kind d'un tel type nous indique combien de paramètres il faut donner pour concrétiser le type. Par exemple `:kind Maybe` donne `*->*`, ce qui signifie que `Maybe` est un objet qui prend un type comme argument (p.ex. `Int`) et retourne un autre type (p.ex. `Maybe Int` qui est un type concret, donc `:kind Maybe Int` donne `Maybe Int :: *`). De la même façon, `:kind (->)` retourne `(->) :: * -> * -> *`, parce que l'opérateur `->` prend deux types concrets `a, b` et construit le type `a->b` (fonction qui prend `a` et retourne `b`).

Regardons maintenant de plus près l'opérateur `(,)` (virgule). On a déjà utilisé cet opérateur pour construire des tuples, par exemple on peut écrire `:type (True, 'a')` et le compilateur nous répond `(True, 'a') :: (Bool, Char)`. Donc, l'opérateur `(,)` n'est rien de spécial! En fait, il est explicitement défini dans la bibliothèque de Haskell avec la définition `data (,) a b = (,) a b`.

On pose maintenant la question si cet opérateur fait partie de la classe `Functor`. On a expliqué que les Functors représentent des containers, qui nous permettent de faire `fmap` sur leurs éléments. Est-ce que c'est le cas pour `(,)`?

```
*Main> :info (,)
data (,) a b = (,) a b -- Defined in 'GHC.Tuple'
..
instance Functor ((,) a) -- Defined in 'GHC.Base'
```

Il paraît que c'est bien le cas. Jouons donc un peu avec `fmap`. Exécutez les lignes suivantes et essayez de comprendre la réponse de Haskell.

```
*Main> fmap (+2) (2,3)
*Main> fmap (+2) (2,3,4)
```

Solution :

```
*Main> fmap (+2) (2,3)
(2,5)
*Main> fmap (+2) (2,3,4)
```

```
<interactive>:62:1: error:
  Non type-variable argument in the constraint: Functor ((,,) t1 t)
..
```

Explication :

Pour la première ligne, on pourrait s'attendre à un résultat comme `(4,5)`. Pourquoi est-ce que `fmap` applique la fonction seulement au deuxième composant de la paire? Si on vérifie la source de la bibliothèque de Haskell (par exemple : <http://hackage.haskell.org/package/base-4.12.0.0/docs/src/GHC.Base.html>) on voit bien qu'on a

```
instance Functor ((,) a) where
  fmap f (x,y) = (x, f y)
```

Donc, cela explique pourquoi on obtient cette réponse : la `fmap` des paires a été programmé en sorte que `f` est appliquée seulement au deuxième composant. Mais pourquoi a-t-on programmé `fmap` comme ça ? Pour comprendre ce choix on regarde de plus près les types et les kinds impliqués. Tout d'abord, le kind de `(,)` est `(,) :: * -> * -> *`, or `Functor` est une classe d'objets de kind `* -> *`. Donc, c'est normal que la bibliothèque de Haskell dit `instance Functor ((,) a)`, où `a` est une variable de type, et pas `instance Functor (,)`, où les kinds ne correspondent pas. Mais quel est le kind de `((,) a)` ? On a par exemple : `((,) Int) :: * -> *`.

Bon, donc les kinds correspondent comme ça. Mais pourquoi on applique `fmap` seulement au deuxième composant ? Il faut maintenant inspecter les types, ou simplement poser une question plus simple : est-ce qu'on peut écrire l'expression `fmap (+) ('a', 5)` ?

```
*Main> fmap (+2) ('a', 5)
('a', 7)
```

L'opérateur `(,)` prend deux paramètres de type, `a, b`. Notre définition dit : pour chaque `a` le type `(a, b)` est un `Functor b` (`b` est la variable partagée de `Functor` et `((,) a)`). Autrement dit, on applique `f` seulement au deuxième composant, car sinon on aurait imposé la contrainte que les deux composants de la paire aient le même type, ce qui n'est pas forcément le cas.

Pour la deuxième ligne l'explication est assez directe : les tuples de taille trois ne sont pas construits avec l'opérateur `(,)`, mais avec un autre opérateur `(,,)`. Vous pouvez vérifier qu'il s'agit bien d'un opérateur différent si vous écrivez `:info (,,)` qui donne

```
*Main> :info (,,)
data (,,) a b c = (,,) a b c    -- Defined in 'GHC.Tuple'
```

En fait, la bibliothèque de Haskell définit un opérateur différent pour des tuples jusqu'à une taille fixe. (Si vous avez besoin des tuples plus longs, vous pouvez définir l'opérateur correspondant, mais évidemment ce n'est pas recommandé, car votre programme deviendra illisible avec des tuples au-delà d'une certaine taille). Dans les informations retournées par le compilateur vous pouvez vérifier que `(,,)` ne fait pas partie de la classe `Functor`, donc c'est normal que le compilateur ne nous permet pas d'utiliser `fmap`.

2 Tableaux Associatifs

Pour cet exercice on va programmer une structure de données appelée Tableau Associatif (associative array), ou dictionnaire, ou simplement `map`. Une version beaucoup plus sophistiqué de cette structure existe déjà dans la bibliothèque de Haskell, dans la module `Data.Map` (<https://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map.html>). Notre objectif est de programmer une version simple avec assez de fonctionnalité pour être utile et pour implémenter un `Functor`.

Un tableau classique de type `t` (ou en Haskell une liste `[t]`) est une structure de données qui étant donné une indice de type `Int` retourne une valeur. On peut donc dire que cette structure associe de valeurs de type `t` avec des clefs de type `Int` (avec la restriction additionnelle que les clefs utilisées forment un intervalle `0..length-1`). Un tableau associatif généralise cette idée en nous permettant d'utiliser d'autres types à la place de `Int` pour représenter les clefs.

Q : Est-ce qu'on peut utiliser n'importe quel type à la place de `Int` pour les clefs ?

A : Il faut qu'on puisse faire de comparaisons entre les clefs. Donc, on a la contrainte `Eq k`, si on veut utiliser des clefs de type `k`

Pour cet exercice on vous demande tout d'abord d'implémenter une version de cette structure de données en utilisant la forme la plus basique : une liste de "key-value" pairs, qui contient comme éléments les pairs de chaque clef avec sa valeur associée. On commence avec la définition suivante :

```
data MyMap k v = MyMap [(k,v)]
```

Notez que le type `MyMap` est un type paramétrique qui prend deux paramètres : le type des clefs `k` et le type des valeurs `v`.

Q : Quel est le kind de `MyMap` ?

A : `MyMap :: * -> * -> *`, c'est-à-dire, `MyMap` est une fonction qui prend deux types concrets (`*`), et retourne un autre type.

Programmer les fonctions suivantes pour le type `MyMap`

1. Une fonction `insert` qui prend un `MyMap`, une clef `k`, et une valeur `v`, et ajoute l'association `k->v` dans le `MyMap`. Si la clef `k` est déjà associée à une valeur, cette valeur est remplacée par `v`. La fonction retourne le `MyMap` mis à jour.
2. Une fonction `find` qui prend un `MyMap` et une clef et retourne un objet de type `Maybe v`. Cet objet est `Nothing` si aucune valeur n'est associée avec la clef donnée, sinon c'est `Just v`, où `v` est la valeur associée avec `k`.
3. Des fonctions `list2map` et `map2list` qui font la conversion entre `MyMaps` et listes. (Pour `list2map` vous pouvez utiliser `insert` et `foldr` et vous pouvez supposer que chaque élément de la liste donnée est une paire `(k, v)`)
4. Faire en sorte que `MyMap` soit une instance de `Show`, et `show` affiche une association par ligne, sous la forme `k --> v`
5. Faire en sorte que `MyMap` soit une instance de `Eq`.
6. Faire en sorte que `MyMap` soit une instance de `Functor`.

Pour toutes les fonctions demandées ci-dessus, essayer de trouver les contraintes de types les plus faibles qui vous permettent de les implémenter. **NB** : il faut ici faire un peu d'attention sur les kinds des types dont on parle. Pour dire que `Blabla` est une instance de `Bloblo`, il faut que `Blabla`, `Bloblo` aient les bons kinds. On ne peut pas dire `instance Show MyMap`, car `Show` est une classe de types concrets (`*`), alors que le kind de `MyMap` est `*->*->*`. De la même façon, `Functor` est une classe d'objets de kind `*->*` qui ne correspond pas non plus au `MyMap`.

Pour votre implémentation de la classe `Functor`, vérifiez si votre version obéit aux lois des `Functors`. Pour rappel il faut que

1. `fmap id = id`
2. `fmap f . fmap g = fmap (f.g)`

Exemples : voici quelques lignes de l'utilisation d'une telle structure pour clarifier ce qui est attendu.

```
*Main> map1 = list2map [("John", "Lennon"), ("Han", "Solo"), ("James", "Bond")]
*Main> map1
"James" -> "Bond"
"Han" -> "Solo"
"John" -> "Lennon"
*Main> find map1 "James"
Just "Bond"
*Main> find map1 "Ben"
Nothing
*Main> find (insert map1 "James" "Lebron") "James"
Just "Lebron"
*Main> fmap reverse map1
"James" -> "snoB"
"Han" -> "oS"
"John" -> "nonneL"
*Main> (list2map [(2, "aa"), (3, "bbb")] ) == (list2map [(3, "bbb"), (2, "aa")] )
True
```

Solution :

```

data MyMap k v = MyMap [(k,v)]

find :: Eq k => MyMap k v -> k -> Maybe v
find (MyMap []) _ = Nothing
find (MyMap ((a,b):ms)) a'
  | a==a' = Just b
  | otherwise = find (MyMap ms) a'

insert :: Eq k => MyMap k v -> k -> v -> MyMap k v
insert (MyMap []) a b = MyMap [(a,b)]
insert (MyMap ((a1,b1):ms)) a b
  | a/=a1 = MyMap ((a1,b1):ms')
  | otherwise = MyMap ((a,b):ms)
where (MyMap ms') = insert (MyMap ms) a b

map2list :: MyMap k v -> [(k,v)]
map2list (MyMap ms) = ms

list2map :: Eq k => [(k,v)] -> MyMap k v
list2map xs = foldr (\ (x,y) m -> insert m x y) (MyMap []) xs

instance (Show k, Show v) => Show (MyMap k v)
where
  show (MyMap []) = ""
  show (MyMap ((a,b):[])) = show a ++ " _-->_" ++ show b
  show (MyMap ((a,b):ms)) = show a ++ " _-->_" ++ show b ++ "\n"
                                ++ show (MyMap ms)

instance (Eq k, Eq v) => Eq (MyMap k v)
where
  MyMap m1 == MyMap m2 = and [ find (MyMap m2) a == Just b | (a,b)<-m1 ]
                        && and [ find (MyMap m1) a == Just b | (a,b)<-m2 ]
--Attn: pas m1 == m2, car l'ordre n'est par pertinent

instance Functor (MyMap k)
where
  fmap f (MyMap m) = MyMap (map (\(a,b) -> (a, f b)) m)
-- equivalent:
-- fmap f (MyMap m) = MyMap (fmap (fmap f) m)

```