Programmation Fonctionnelle
Monads

Michael Lampis

2023-2024

Functor et Applicative – Rappel

- On a vu la classe (paramétrique) Functor

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

- . . . et sa sous–classe Applicative

```
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  {-# MINIMAL pure, (<*>) #-}
```

Functor – Rappel

- `Functor a` représente un containeur contenant des objets de type `a`
  - Exemple : `[a]`, `Maybe a`
- En déclarant qu'un type est une instance de `Functor`, on déclare que ce type offre une version de la fonction `fmap`.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

- `fmap` (également écrite `<$>`) nous permet d'accéder indirectement au contenu du Functor
- ... elle nous permet d'appliquer une fonction sur chaque élément du Functor
- ... et obtenir un Functor (du même type) contenant les résultats.

```
Prelude> fmap (+2) (Just 3)
Just 5
Prelude> (+2) <$> [4,5]
[6,7]
```

Applicative – Rappel

- `Applicative a` est un Functor avec des fonctionnalités extra.
  - Exemple : `[a]`, `Maybe a`
- Applicative offre les fonctions `pure` et `<*>`

```
pure :: Applicative f => a -> f a
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

- `pure` initialise un container trivial avec l'élément donné
- `<*>` prend un container de fonctions et un container d'arguments
- ... elle applique chaque fonction sur chaque argument pour donner un container d'éléments

```
Prelude> pure (+) <*> Just 2 <*> Just 3
Just 5
Prelude> (*) <$> [2,3] <*> [7,8,9]
[14,16,18,21,24,27]
```

Monads – Motivation

- Cas d'usage :
  - Functor : un container où on veut traiter chaque élément de façon indépendante
  - Applicative : nous permet aussi de combiner les éléments – appliquer sur eux une fonction de plusieurs variables.

```
Prelude> (+2) <$> (Just 3) -- Functor
Just 5
Prelude> (+) <$> (Just 3) <*> (Just 5) -- Applicative
Just 8
```

- Aujourd'hui : `Monad` est une sous-classe des Applicatives
- Nouvelle Fonctionnalité : la capacité d'enchaîner les applications de fonctions sur un `Monad`.

Monads – Définition

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
  {-# MINIMAL (>>=) #-}
```

- Fonction principale : `(>>=)` (appelée “bind”)
- Que fait–elle ?
  - Version plus sophistiquée de `fmap`
  - Prend un container (monad) de `a`, et une fonction qui prend un argument de type `a`
  - Contrairement à `fmap`, cette fonction retourne un container de résultats (`m b`) et pas forcément un seul résultat.
  - `(>>=)` applique cette fonction sur chaque élément et retourne la collection de tous les résultats.
- Note : `return` est juste un autre nom pour `pure`

Monads de base

- L'intérêt d'utiliser l'abstraction des Monads est que ça nous permet d'écrire un algorithme une fois pour le type `Monad` et le réutiliser pour plusieurs instances, mêmes pour des types qui ne semblent pas être des containers (mais sont des instances de la classe `Monad`).
- Pour comprendre les Monads, il faut voir quelques exemples.
- La bibliothèque de Haskell contient notamment les Monads suivants :
  - `Maybe` : représente un calcul qui peut avoir un résultat ou pas.
  - `[]` : représente un calcul qui donne une collection de résultats.
  - `IO` : représente un calcul qui interagit avec le système d'entrée/sortie.

Maybe Monad

- Rappel : `Maybe` représente la possibilité qu'une valeur n'ait pas pu être calculée (par exemple à cause d'une erreur).
- Le but des monads est de nous permettre d'enchaîner les calculs sur les valeurs contenues dans le monad.
- `Maybe` est un monad pour nous permettre de faire facilement de calculs qui peuvent échouer, sans devoir s'inquiéter de la possibilité d'erreur.
- Pour le type `Maybe` on a

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
-- devient
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

- Comment se servir de la fonction `(>>=)` ?

Maybe Monad – exemple

Considérez le programme suivant :

```
f1unsafe :: Int -> Int
f1unsafe x = 10 `div` x

f2unsafe :: Int -> Int
f2unsafe x = 10 `div` (x-1)

f3unsafe :: Int -> Int
f3unsafe x = 10 `div` (x-2)
--
*Main> f3unsafe $ f2unsafe $ f1unsafe 3
3
*Main> f3unsafe $ f2unsafe $ f1unsafe 2
*** Exception: divide by zero
```

Maybe Monad – exemple

- `f3unsafe $ f2unsafe $ f1unsafe x` représente la fonction

$$\frac{10}{\lfloor \frac{10}{\lfloor \frac{10}{x} \rfloor - 1} \rfloor - 2}$$

- Pour $x = 3 \rightarrow 3$
- Pour $x = 2 \rightarrow ??$ (division par zéro)
- Est-ce qu'on peut utiliser la fonctionnalité de `Maybe` pour gérer ce type d'erreur ?
- Objectifs :
  - Fonction qui retourne le bon résultat si possible
  - `Nothing` si division par zéro
  - Fonction facile à écrire (pas de pattern matching pour `Just, Nothing...`)

Maybe Monad – exemple

Versions “safe” des fonctions

```
f1 :: Int -> Maybe Int
f1 0 = Nothing
f1 x = Just (10 `div` x)

f2 :: Int -> Maybe Int
f2 1 = Nothing
f2 x = Just (10 `div` (x-1))

f3 :: Int -> Maybe Int
f3 2 = Nothing
f3 x = Just (10 `div` (x-2))
```

On aimerait que :

```
f3 $ f2 $ f1 3 --> Just 3
f3 $ f2 $ f1 2 --> Nothing
```

Mais ça ne marche pas !!

Maybe Monad – exemple

On aimerait que :

```
f3 $ f2 $ f1 3 --> Just 3
f3 $ f2 $ f1 2 --> Nothing
```

- Problème : `f1 :: Int -> Maybe Int`, mais `f2` prend comme argument un `Int`, pas un `Maybe Int`.
- Solution déjà vue : réécrire `f1, f2, f3` pour qu'elles prennent `Maybe Int` comme argument
  - Ça marche, mais pénible à écrire...
- Meilleure solution : utiliser `(>>=)`

```
*Main> f1 3 >>= f2 >>= f3
Just 3
*Main> f1 2 >>= f2 >>= f3
Nothing
```

Ça marche !

Maybe Monad – exemple

```
*Main> f1 3 >>= f2 >>= f3
Just 3
*Main> f1 2 >>= f2 >>= f3
Nothing
```

Explication :

- Association : `((f1 2) >>= f2) >>= f3`
- Rappel :
  `(>>=) :: Maybe Int -> (Int->Maybe Int) -> Maybe Int`
- `f1 2` retourne un `Maybe Int`
- `f2` prend un `Int` et retourne un `Maybe Int`
- Lecture informelle : l'opérateur `>>=` nous permet d'enchaîner les fonctions (faire une "composition") qui stocke les résultats successifs dans des `Maybe`.
- À chaque étape l'opérateur `>>=` extrait la valeur dans le `Maybe` (premier argument) et la donne comme argument à la fonction (deuxième argument).
- Si l'extraction échoue (`Nothing`), l'opération retourne `Nothing`.

Maybe Monad – définition

Pour confirmer notre lecture informelle, regardons la définition de `Maybe` dans la bibliothèque de Haskell :

```
instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing
```

- L'opérateur `>>=` prend deux arguments
  - Si le premier est `Just x`, on applique le deuxième argument (qui est une fonction) sur `x`.
  - Sinon, on retourne `Nothing`.

Do Notation

- L'enchaînement des utilisations de l'opérateur `>>=` est un phénomène très répandu. On a très souvent besoin d'écrire un programme qui dit :
  - J'appelle la fonction `f1` sur `x`. Soit `x1` le résultat.
  - J'appelle `f2` sur `x1`. Soit `x2` le résultat.
  - J'appelle `f3` sur `x2`. ...

La manière directe d'écrire ça est :

```
composef x = f1 x >>= f2 >>= f3
```

- Or, le désavantage de cette méthode est qu'elle nous permet seulement de transférer chaque valeur directement (et rien d'autre).
- Comment faire si on veut calculer (p.ex.) `f3 ((f2 (f1 x)) + (f1 x))` ?

Do Notation

- Comment faire si on veut calculer (p.ex.)
  `f3 ((f2 (f1 x)) + (f1 x))` ?

Une solution :

```
composef' x = f1 x >>= \y ->
              f2 y >>= \z ->
              f3 (z+y)
```

- Ça marche, mais assez pénible !
- Dans un langage impératif on aurait écrit :

```
composef x:
  y = f1 x
  z = f2 y
  return f3 (z+y)
```

Do Notation

- La **Do Notation** de Haskell nous permet d'écrire dans un style pseudo–impératif.

```
composef x = do
  y <- f1 x
  z <- f2 y
  f3 (y+z)
```

Règles :

- les lignes qui suivent le `do` doivent être alignées.
- chaque ligne a la forme `var <- expression`
  - qui correspond à `expression >>= \var ->`
- Le programme ci-dessus est équivalent à :

```
composefl x =
  f1 x >>= \y ->
  f2 y >>= \z ->
  f3 (y+z)
```

List Monad

- Les listes font aussi partie de la classe `Monad`

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

- Idée générale :
  - `Maybe` représente un calcul qui peut échouer
  - `[]` représente un calcul qui peut donner plusieurs résultats (un calcul non–déterministe).
- Dans ce contexte, enchaîner les listes représente quoi ?
- Quel est le sens de `>>=` pour les listes ?

List Monad

Jouons avec les listes et `>>=`

```
*Main> [1,2,3,4] >>= (\x -> [x+7,x*5])
[8,5,9,10,10,15,11,20]
*Main> closeto x = [x-1,x,x+1]
*Main> [1] >>= closeto >>= closeto
[-1,0,1,0,1,2,1,2,3]
*Main> filter (\x -> x`mod`5==0) [0..10] >>= closeto
[-1,0,1,4,5,6,9,10,11]
*Main> [] >>= closeto
[]
```

Lecture informelle :

- `xs >>= f` applique la fonction `f` sur chaque élément de la liste (comme `fmap`)
- Contrairement à `fmap`, la fonction `f` ici retourne une liste
- Le résultat de `xs >>= f` est la concaténation de toutes les applications de `f` sur les éléments de `xs`

List Monad

Pour confirmer notre lecture informelle, regardons la définition de `[]` dans la bibliothèque de Haskell :

```
instance Monad []  where
    xs >>= f                 = [y | x <- xs, y <- f x]
```

Explication :

- Pour chaque élément `x` de `xs`
- On applique `f` sur `x`. Cela retourne une liste.
- Pour chaque élément `y` de cette liste on collectionne `y` dans la liste qu'on retourne.
- Équivalent à

```
xs >>= f                 = concat [ f x | x <- xs ]
```

Wait a minute!

- Regardons de plus près la notation de compréhension des listes :

```
ls = [ x+y | x<-[1..3], y<-[5..7] ]
```

- Maintenant, ça devrait vous rappeler quelque chose !
- Effectivement, cette notation n'est qu'une variation syntactique de la notation `do`
- Version équivalente :

```
ls = do
       x <- [1..3]
       y <- [5..7]
       return (x+y)
```

Compréhension de listes == Monads

- La notation de listes en compréhension qu'on a utilisé se base sur le fait que les listes sont des Monads.
- `x<-[1..3]` équivaut à `[1..3] >>= \x ->`
- donc les trois programmes suivants sont équivalents.

```
ls = [ x+y | x<-[1..3], y<-[5..7] ]
```

```
ls = do
       x <- [1..3]
       y <- [5..7]
       return (x+y)
```

```
ls' = [1..3] >>= \x -> [5..7] >>= \y -> return (x+y)
```

Compréhension de listes + Pattern Matching

- Quid des expressions comme :
  `ls = [ x+y | x<-[1..3], x`mod`2==1, y<-[5..7] ]` ?
- Une manière de représenter ce type de compréhension (qui utilise de filtering) est comme

```
ls = do
       x <- [1..3]
       True <- [x`mod`2==1]
       y <- [5..7]
       return (x+y)
```

- Ici la ligne `True <- [x`mod`2==`]` correspond à un pattern matching implicite.
- Quand on utilise le pattern matching dans ce contexte, si la matching échoue, la fonction `fail` du Monad correspondant est appelée.
- Pour les listes, cette fonction retourne simplement `[]`.

Les Lois des Monads

- Comme pour les Functors et Applicatives, les bons Monads doivent satisfaire quelques conditions, qui ne peuvent pas être vérifiées par le compilateur.

1. `m >>= return = m`
2. `return x >>= f = f x`
3. `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Explications :

- La première loi dit que la fonction `return` ne change pas les contenus du monad `m`
- La deuxième loi dit que le monad trivial construit par `return x` ne contient que `x`
- La troisième loi est la loi associative. Enchaîner les action `m,f,g` équivaut à faire d'abord `m` et puis la composition de `f,g`.

Autres Monads

- Les Monads sont une notion basique pour Haskell qui permettent de regrouper des structures différentes.
- Ils nous permettent aussi de modéliser dans le monde fonctionnel des processus impératifs, en respectant les lois du système de type de Haskell.
- Ceci est particulièrement important quand on veut utiliser un système de nature non–fonctionnelle.
- C'est notamment le cas du système d'entrée et sortie qui, à cause de l'interaction imprévisible avec l'utilisateur, ne peut pas être facilement modélisé dans un langage fonctionnel.
- En Haskell ce système est modelisé grâce à l'`IO Monad`