Programmation Fonctionnelle

IO Monad

Michael Lampis

2023-2024

Functor, Applicative, Monad – Rappel

- On a vu les classes (paramétriques) Functor, Applicative, Monad

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

Rappel – Monad

- Un Monad est un container (au sens large) qui nous permet de faire `>>=`
- La fonction `>>=` extrait le contenu d'un monad et le passe comme argument à une fonction.
- On retourne le résultat de cette fonction, qui est aussi un monad.
- Par conséquent : on peut enchaîner les applications de `>>=`

Exemple : (rappel : `Maybe` est un Monad)

```
f1 :: Int -> Maybe Int
f2 :: Int -> Maybe Int
f3 :: Int -> Maybe Int
f3 2 >>= f2 >>= f1 :: Maybe Int
-- Just f1 (f2 (f3 2)) ou Nothing
```

Rappel – Do

- On a vu qu'enchaîner les `>>=` est souvent utile.
- Haskell nous donne une notation pratique pour le faire avec le mot–clé `Do` et l'opérateur `<-`.
- Stucture : `do` suivi par des instruction alignées de la forme `variable <- expression` où
- `variable` est une ou plusieurs variables (on peut faire de pattern matching)
- `expression` a un type de monad.

Exemple : (variation du programme du transparent précédent)

```
comp :: Int -> Maybe Int
comp x = do
  x1 <- f1 x
  x2 <- f2 x1
  x3 <- f3 x2
  return x3
```

Rappel – Do

```
comp :: Int -> Maybe Int
comp x = do
  x1 <- f1 x
  x2 <- f2 x1
  x3 <- f3 x2
  return x3
```

Correspond à

```
comp :: Int -> Maybe Int
comp x =
  f1 x  >>= \x1 ->
  f2 x1 >>= \x2 ->
  f3 x2 >>= \x3 ->
  return x3
```

- Quand on écrit `x1<- f1 x`, le type de `x1` est `Int`
- Explication : `<-` extrait la valeur du Monad `f1 x`
- Pattern Matching : si le type de `f1 x` était p.ex. `(Int,Char)` on pourrait aussi écrire `(x,c)<-f1 x`
- Si le pattern matching échoue, `fail` est appelée.
- Écrire `_<- blabla` équivaut à écrire simplement `blabla` (dans sa propre ligne)
- `return` ne fonctionne pas comme en C ! C'est juste une fonction.
- Conséquence : `return` ne termine pas un do–block

IO Monad

Vive la Transparence !

- Une propriété clé de la programmation fonctionnelle (et de Haskell en particulier) et la **transparence référentielle** (referential transparency).
  - Pour chaque fonction `f` et argument `v` l'application `f v` retourne toujours le même résultat et ne change rien d'autre dans notre programme.
  - Si `f v` retourne `x`, on peut remplacer `f v` partout dans notre programme par `x` ça ne devrait pas changer le résultat.

Exemple : les programmes suivants sont équivalents

```
fact n = if n==0 then 1 else n*fact (n-1)
fact n = if n==0 then 1 else
      n*(if (n-1)==0 then 1 else fact ((n-1)-1))

fact n = if n==0 then 1 else
      n*(if (n-1)==0 then 1 else
       (n-1)*(if ((n-1)-1)==0 then 1 else
        ((n-1)-1)*fact (((n-1)-1)-1)))
```

Vive la Transparence !

- Pour Haskell, le calcul n'est qu'un processus de traduction/remplacement.
  - Pour évaluer l'expression `f arg` on prend la définition de `f`
  - La définition dit `f x = expr` où `expr` utilise la variable `x`.
  - L'évaluation est simplement le remplacement de `x` par `arg` dans `expr`.
  - On continue comme ça jusqu'au moment où on arrive à une expression irréductible.
- La transparence référentielle nous garantit que tous ces remplacements sont corrects et ne changent pas le résultat.
- Elle nous permet également de raisonner et prouver de bonnes propriétés de nos programmes en prenant en compte seulement une fonction par fois.

Manque de Transparence ?

- Pour comprendre à quoi sert la transparence référentielle, regardons un exemple dans un langage impératif.
- Qu'affiche ce programme ?

```
#include <stdio.h>

int newval(int *p)
{
        *p = p[*p];
        return p[*p];
}

int main()
{
        int a[5] = {1,2,1,3,0};
        for(int i=0; i<5; i++)
                printf("%d\n",newval(a));
}
```

Manque de Transparence ?

- Le programme du transparent précédent appelle la fonction `newval` 5 fois avec le même argument.
- Or, il affiche `1,2,1,2,1`.
- Problème, le comportement de cette fonction ne peut pas être analysé, sans prendre en compte l'état global de la mémoire.
- Raison : la fonction `newval` a le droit d'utiliser et modifier des valeurs qui ne lui "appartiennent" pas.

En Haskell :

- `newval a` retournera toujours la même valeurs si on utilise la même valeur pour `a` (loi de Transparence Référentielle)
- Autrement dit, aucune fonction ne peut modifier l'"état" du reste du programme.
- Cela nous permet d'analyser la fonction de manière indépendante et garantir (par induction mathématique) qu'elle retourne le bon résultat.

Un problème de philosophie

OK, on a bien compris

- Qu'est-ce que la transparence référentielle.
- À quoi ça sert.

Regardons maintenant un exemple où cette loi nous impose de contraintes.

```
#include <stdio.h>
int getnum(int max)
{
  int r;
  do{
    printf("Give me a number up to %d\n",max);
    scanf("%d",&r);
  }while(r>max);
  return r;
}
```

Un problème de philosophie

- La fonction `getnum` prend une valeur max, et demande à l'utilisateur de saisir une valeur sous cette borne.
- Quelle serait le type d'une fonction équivalente en Haskell ?
- Il semble naturel de dire :

```
getnum :: Int -> Int
```

- Or, ça pose le problème fondamentale que `getnume 20` devrait être une valeur constante
  - Transparence référentielle $\rightarrow$ ce qui est retourné par une fonction ne dépend que de ses arguments.
- Évidemment, quand on écrit `getnum 20` on aimerait avoir comme résultat parfois $12, 15, 7, \ldots$, selon le choix de l'utilisateur.

Leçon :

- Le système d'entrée/sortie de notre ordinateur est intrinsèquement difficile à modéliser dans un langage fonctionnel.
- Les fonctions qui interagissent avec l'utilisateur ne peuvent pas être garanties de respecter la transparence !

La solution de Haskell

- Le système de types de Haskell n'est pas facilement compatible avec des fonctions comme `printf,scanf,getchar...`
- Problème inhérent : ces fonctions ont des effets secondaires et/ou leur valeur de retour dépend des informations extérieurs, autres que leurs arguments.
- Donc, c'est impossible de programmer le système d'entrée/sortie en Haskell ?

La solution de Haskell

- Le système de types de Haskell n'est pas facilement compatible avec des fonctions comme `printf,scanf,getchar...`
- Problème inhérent : ces fonctions ont des effets secondaires et/ou leur valeur de retour dépend des informations extérieurs, autres que leurs arguments.
- Donc, c'est impossible de programmer le système d'entrée/sortie en Haskell ?

Pas du tout !

- La solution utilisée par Haskell est le `Monad IO`.
- L'idée est que le type d'une fonction comme `getChar` ne devrait pas être un `Char` (pour les raisons qu'on a vues), mais un `Char` contenu dans un `IO Monad`.
- Le `Monad` modélise l'état de tout le système d'entrée sortie.

IO Monad

- Le IO Monad modélise toutes les interactions de notre programme avec le monde extérieur.
  - Écran, clavier, fichier, ...
- `IO` est un Monad, donc son kind est `*->*`
- On a les types `IO Int`, `IO Char`, `IO Double`,...
- On a aussi le type `IO ()` où `()` signifie le type vide.
- Lecture informelle : `IO a` veut dire un objet de type `a` dont l'obtention a entrainé une interaction avec le monde extérieur. (Et donc, dont l'obtention peut avoir modifié l'état du système).

Différence :

- `f:Int->Int` la fonction `f` prend un argument `n` et retourne une valeur `m` qui ne dépend que de `n`
- `f:Int->IO Int` la fonction `f` prend un argument `n` et interagit avec le système I/O. Elle retourne le nouvel état du système IO et le `Int` qui en résulte.

IO Monad – Décryptage

- Le IO Monad est une manière pour Haskell de modéliser de procédures intrinsèquement non–fonctionnelles.
- Techniquement, déclarer qu'une fonction a type `f:Int->IO Int` nous permet de rester dans les règles du jeu.
- Pratiquement, si on est obligés de dire que la fonction retourne l'état du système pour rester dans les règles, c'est difficile de dire qu'on a vraiment conservé la transparence.
- Par conséquent :
  - On essaie d'écrire nos programmes en sorte que la partie `IO` soit aussi limitée que possible.
  - On considère la partie `IO` la partie "sale" du programme.
  - On écrit la plus grande partie en utilisant les types sans `IO`.
  - Avantage : le système de type peut détecter quand une valeur dépend de l'extérieur (son type inféré implique `IO`). Ça nous permet de garantir qu'on évite les erreurs.
  - $\rightarrow$ Si le compilateur accepte que `f::Int->Int` la transparence référentielle est garantie.

IO Monad – Pratique

- Bon, c'est intéressant (ou pas), mais comment utiliser le IO Monad en pratique ?
- Jusqu'ici on n'a utilisé que le ghci (ghc interpreter). Or, Haskell nous permet de compiler un programme qui contient une fonction `main`, comme C.
- Considérez le programme suivant :

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n*fact (n-1)

main = do
  xs <- getLine
  putStrLn $ show $ fact (read xs)
```

Compilation

- On sauvegarde ce programme dans un fichier `printfac.hs`
- Pour compiler (dans un terminal on écrit) :

```
> ghc printfac.hs
[1 of 1] Compiling Main
( printfac.hs, printfac.o )
Linking printfac ...
```

- (Rappel : comme on écrit `gcc prog.c` pour compiler en C)
- Résultat (si compilation réussie) : un nouveau fichier `printfac`

```
> ./printfac
 -- (Le programme bloque et attend une valeur)
5
120
```

- On peut aussi se passer de la compilation et écrire sur la ligne de commande `runhaskell printfac.hs`

IO – fonctions de base

Nom	Type	Description
	I/O au terminal	
`getChar`	`IO Char`	lit un caractère au clavier
`getLine`	`IO String`	lit une ligne saisie au clavier
`getContents`	`IO String`	lit jusqu'à EOF (Ctrl+D)
`putChar`	`Char -> IO ()`	affiche un caractère au terminal
`putStr`	`String -> IO ()`	affiche une chaîne de caractères
`putStrLn`	`String -> IO ()`	comme `putStr` mais ajoute un `\n`
`print`	`Show a=>a->IO ()`	`putStrLn . show`

Exemples

Programme qui demande deux entiers, les additionne, affiche le résultat.

```
main = do
 l1 <- getLine
 x1 <- read l1
 l2 <- getLine
 x2 <- read l2
 print (x1+x2)
```

- Est-ce que ce programme compile ?

Exemples – Corrigé

```
main = do
 putStrLn "Give me two numbers"
 l1 <- getLine
 let x1 = read l1
 l2 <- getLine
 let x2 = read l2
 print (x1+x2)
```

- **Attn:** la notation `<-` ressemble à une affectation classique dans un langage impératif.
- Or, c'est en réalité une application de `>>=` (comme on l'a vu)
- Ce qui est à droite du `<-` doit être un `Monad`
- `<-` extrait la valeur contenue dans le Monad et la met dans la variable à gauche.
- Donc, `l1::String`
- `let var=expr` est une affectation classique : en ce qui suit `var` est remplacé par `expr`

Exemples

- L'exemple précédent ne lit que deux entiers, qui doivent être séparés par un `\n`.
- Version améliorée :

```
main = do
  putStrLn "Give me many numbers"
  c <- getContents
  print $ sum $ map read $ words c
```

- La fonction `words :: String -> [String]` prend une chaîne de caractères et la découpe où elle trouve des espaces vides.
- Exemple :

```
> words "Four score and \n\n seven \n  years ago "
["Four","score","and","seven","years","ago"]
```

- Le programme termine quand on donne Ctrl+d au terminal (EOF – End-of-file)

Exemples

`main` est une fonction, comme une autre.

```
main = do
 putStrLn "Give me a number"
 c <- getLine
 let x = read c
 putStrLn "The double of your number is"
 print (2*x)
 putStrLn "Continue? (y/n)"
 c' <- getLine
 if c'=="y" then main else return ()
```

- La branche `else` est obligatoire (comme toujours)
- Il faut que ce qui est dans `else` ait le même type que ce qui est dans `then`
- `main::IO ()`

Return ne fait pas return

- **Attn:** la notation `do` est une illusion. Haskell reste un langage non–impératif.
- Notamment : `return` est juste une fonction (de la classe `Monad`)
- `return` n'a pas l'effet de terminer la fonction ! (comme en C)

```
main = do
 putStrLn "What's your name?"
 l1 <- getLine
 putStrLn $ "Hello " ++l1
 return ()
 putStrLn "Will this happen?"
 return "blabla"
 putStrLn "What now?"
 return 27
 putStrLn "Goodbye!"
```

- Qu'affiche ce programme ?

Utiliser les fichiers

- L'utilisation des fichiers ressemble au système I/O du terminal.
- Pour utiliser des fichiers on fait au début de notre programme
  `import System.IO`
- On peut ouvrir un fichier avec la fonction
  `openFile :: FilePath->IOMode->IO Handle`
  - `FilePath == String` est le nom du fichier
  - `IOMode` est un des
    `ReadMode | WriteMode | AppendMode | ReadWriteMode`
  - La fonction retourne un `IO Handle` qu'on peut utiliser pour accéder au fichier.
- Il y a des versions des fonctions qu'on a vues pour les fichiers :
  - `hGetLine :: Handle -> IO String` : comme `getLine` mais prend un `Handle` qui représente le fichier
  - `hPutStrLn :: Handle -> String -> IO ()` idem pour `putStrLn`
- À la fin il faut fermer le fichier avec `hClose`

Utilisation des fichiers – Exemple

```
import System.IO

main = do
 putStrLn "Input file?"
 fin <- getLine
 putStrLn "Output file?"
 fout <- getLine
 hin <- openFile fin ReadMode
 hout <- openFile fout WriteMode
 c <- hGetContents hin
 map (hPutStrLn hout) $ map show $
              map (2*) $ map read $ words c
 hClose hin
 hClose hout
```

Utilisation des fichiers – Exemple

```
import System.IO

main = do
 putStrLn "Input file?"
 fin <- getLine
 putStrLn "Output file?"
 fout <- getLine
 hin <- openFile fin ReadMode
 hout <- openFile fout WriteMode
 c <- hGetContents hin
 map (hPutStrLn hout) $ map show $
              map (2*) $ map read $ words c
 hClose hin
 hClose hout
```

Ne compile pas !? (pourquoi ?)

Correction

- Le problème de notre programme est la ligne
  `map (hPutStrLn hout) ..`
- Une telle ligne n'appartient pas dans ce contexte :
  - On est dans un do-block
  - Chaque ligne a la forme `var <- action` ou `action`
  - où `action` est de type `IO a`
  - or `map ...` a type `[a]`...
- Plus simplement, on a mélangé la partie fonctionnelle et "impérative" du programme.
- Comment corriger cette ligne pour que le programme compile ?

Utilisation des fichiers – Exemple

```
import System.IO

main = do
 putStrLn "Input file?"
 fin <- getLine
 putStrLn "Output file?"
 fout <- getLine
 hin <- openFile fin ReadMode
 hout <- openFile fout WriteMode
 c <- hGetContents hin
 let ff = map (hPutStrLn hout) $ map show $ map (2*)
                               $ map read $ words c
 hClose hin
 hClose hout
```

Compile bien !

Utilisation des fichiers – Exemple

```
import System.IO

main = do
 putStrLn "Input file?"
 fin <- getLine
 putStrLn "Output file?"
 fout <- getLine
 hin <- openFile fin ReadMode
 hout <- openFile fout WriteMode
 c <- hGetContents hin
 let ff = map (hPutStrLn hout) $ map show $ map (2*)
                               $ map read $ words c
 hClose hin
 hClose hout
```

Compile bien !
Mais ne fonctionne pas !!!!

Correction 2

- Problème : évaluation paresseuse.
- La ligne `let ff = ...` n'a en réalité aucun effet, sauf si la variable `ff` est utilisée quelque part (dans ce cas on devra l'évaluer).
- Il faut donc dire explicitement au programme qu'il faut exécuter toutes les actions de `ff`
- D'ailleurs, quel est le type de `ff` ?
  - `map show $ ... :: [String]`
  - `hPutStrLn hout :: String -> IO ()`
  - Donc, `ff :: [ IO () ]`
- C'est normale que la première version ne marchait pas, car cette expression n'est pas une action IO, mais une liste d'actions IO...
- Pour convertir une liste d'actions IO en action IO on peut utiliser la fonction `sequence`
  - Cf. la fonction `sequenceA` qu'on a vu avec les applicatives.
- `sequence :: Monad m => [m a] = m [a]`

Utilisation des fichiers – Exemple

```
import System.IO

main = do
 putStrLn "Input file?"
 fin <- getLine
 putStrLn "Output file?"
 fout <- getLine
 hin <- openFile fin ReadMode
 hout <- openFile fout WriteMode
 c <- hGetContents hin
 let ff = map (hPutStrLn hout) $ map show $ map (2*)
                               $ map read $ words c
 sequence ff
 hClose hin
 hClose hout
```

Enfin ça fonctionne !