

Llenguatges Funcionals: Haskell

Albert Rubio

Llenguatges de Programació, FIB, UPC

Primavera 2016

Monads

- 1 Introducció a les Mònades
- 2 Class Functor
- 3 Class Monad
- 4 La Monad IO
- 5 Nombres aleatoris en Haskell

Continguts

- 1 Introducció a les Mònades
- 2 Class Functor
- 3 Class Monad
- 4 La Monad IO
- 5 Nombres aleatoris en Haskell

Introducció a les Mònades

Les Mònades s'utilitzen en Haskell per modelar característiques no massa funcionals com ara:

- indeterminisme
- excepcions
- noció d'estat (efectes laterals)
- seqüenciació
- concurrència
- entrada/sortida
- etc.

Introducció a les Mònades

Les Monads no són *impures*. Només ho és l'entrada/sortida.

Objectiu:

Encapsular un còmput d'un cert tipus a mitjançant un contenidor `m`.

Només considerarem dos classes de Mònades:

- Functor
- Monad

Les Mònades les definirem com a instàncies d'una d'aquestes classes.

NO és poden derivar

Continguts

- 1 Introducció a les Mònades
- 2 **Class Functor**
- 3 Class Monad
- 4 La Monad IO
- 5 Nombres aleatoris en Haskell

Class Functor

És el cas més simple.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Les llista `[]` o el `Maybe` són instàncies de la classe `Functor`:

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

```
instance Functor [] where
    fmap = map
```

Class Functor

La funció `fmap` associada al Functor:

- Aplica la funció a les dades d'un contenidor.
- Pot canviar el tipus del contingut, però NO el contenidor.

Altres exemples que podem instanciar:

- `Arbre` i `ArbreGen`.
- `Set` o qualsevol contenidor predefinit.

Noteu que no podem compondre `fmaps`, només podem fer el `fmap` de la composició de funcions.

Això és el que introdueix la classe `Monad`, per representar *còmput*s

Continguts

- 1 Introducció a les Mònades
- 2 Class Functor
- 3 Class Monad**
- 4 La Monad IO
- 5 Nombres aleatoris en Haskell

Class Monad

Conté dues funcions principals:

```
return :: a -> m a
```

```
(>>=)  :: m a -> (a -> m b) -> m b
```

- (>>=) representa la composició de còmputs i ens permet simular els efectes laterals present en els llenguatges imperatius.
A l'operador (>>=) se l'anomena *bind*.
- (return x) representa un còmput que produeix x mateix. És a dir, és x vist com a còmput.

A banda d'un canvi d'ordre en els paràmetres només hi ha una diferència entre `fmap` i (>>=):

La funció que rep retorna el contenidor amb el contingut nou
(no només el contingut nou)

Class Monad

També tenim l'operació (\gg), que és defineix com:

```
(\gg)    :: m a -> m b -> m b  
r \gg k  =  r \gg= \_ -> k
```

Els tipus llista `[]` i `Maybe` són instàncies de la classe `Monad`

```
instance Monad Maybe where
```

```
(Just x) \gg= k    =  k x  
Nothing  \gg= k    =  Nothing  
return   = Just
```

```
instance Monad [] where
```

```
m \gg= k          = concat (map k m)  
return x          = [x]
```

Les lleis de les Monad

Les mònades és regeixen per tres regles, que s'hauria de comprovar que es compleixen.

- 1 Identitat per l'esquerra: $\text{return } a \gg= f \equiv f \ a$
- 2 Identitat per la dreta: $m \gg= \text{return} \equiv m$
- 3 Associativitat: $(m \gg= f) \gg= g \equiv m \gg= (\lambda x \rightarrow f \ x \gg= g)$

Noteu que el compilador no ho comprova. Ho hem de fer nosaltres!

Extensions de la classe Monad

Moltes funcions predefinides tenen una versió per la classe Monad, com ara `mapM`, `filterM`, `foldM`, `zipWithM`, ...

També disposem d'operacions per estendre ("lift") operacions per a que treballin amb element de la classe Monad.

```
import Control.Monad
```

```
liftM    :: Monad m => (a -> b) -> m a -> m b
```

```
liftM2   :: Monad m => (a -> b -> c) -> m a -> m b -> m c
```

```
sumaMaybe :: Num a => Maybe a -> Maybe a -> Maybe a
```

```
sumaMaybe = liftM2 (+)
```

Extensions de la classe Monad

o directament

```
> liftM2 (+) (Just 3) (Just 2)
(Just 5)
> liftM2 (+) (Just 3) Nothing
Nothing
```

similarment liftM3, liftM4, liftM5, o bé

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

```
myliftM f x = ap (return f) x
```

```
myliftM2 f x y = ap (ap (return f) x) y
```

```
myliftM2 f x y = (return f) 'ap' x 'ap' y
```

La notació do

Ens proporciona una sintaxi natural per composar còmputs.

- Tenim el punt i coma (;) per concatenar còmputs i
- l'operador <- per guardar resultats intermedis.

```
do { e1 ; e2 }      =  e1 >> e2
do { p <- e1; e2 } =  e1 >>= \p -> e2
```

Si usem línies separades i indenteu, bé no cal posar el ; entre els còmputs.
Per exemple

```
main = do input <- getLine
         print (factorial (read input))
```

Continguts

- 1 Introducció a les Mònades
- 2 Class Functor
- 3 Class Monad
- 4 La Monad IO**
- 5 Nombres aleatoris en Haskell

La Monad IO

- Usem el constructor de tipus IO per gestionar l'entrada/sortida.
- IO és una instància de la classe Functor i Monad.
- La usarem normalment amb notació `do`.

Les operacions bàsiques del tipus són

```
getChar  :: IO Char
putChar  :: Char -> IO ()
getLine  :: IO String
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
```

En Haskell `()` s'anomena el tipus *unit* i representa el “res” (com ara el `void` dels llenguatges de la família del C).

La Monad IO

Com a exemple d'ús, la implementació del `getLine` és la següent:

```
getLine = do c <- getChar
            if c == '\n'
            then return ""
            else do l <- getLine
                    return (c:l)
```

La Monad IO

Degut a la definició del ($\gg=$), el `where` pot donar problemes.

```
main = do
    x <- getLine
    print f
    where f = factorial (read x)::Integer
```

És incorrecte.

```
.....: Not in scope: 'x'
```

La raó és que si ho escrivim amb ($\gg=$), tenim

```
main = getLine >>= \x -> print f
    where f = factorial (read x)::Integer
```

Que no pot ser, ja que a les definicions del `where` no podem usar la variable abstracta `x`.

La Monad IO

En canvi amb el `do` podem usar el `let` (sense `in`) de forma molt natural usant aquestes variables abstrètes:

```
main = do
    x <- getLine
    let f = factorial (read x)::Integer
    print f
```

Si posem el `in` hem de posar un altre `do`

La Monad IO

Encara que més lleig, també podem fer

```
main = do
    x <- getLine
    f <- return (factorial ((read x)::Integer))
    print f
```

Ja que primer el convertim el convertim en IO Integer i després extraïem l'Integer

Operacions amb arxius o canals

Per treballar amb arxius i canals considereu les següents definicions i funcions:

```
type FilePath = String
data IOMode = ReadMode|WriteMode|AppendMode|ReadWriteMode

openFile :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()
getContents :: Handle -> IO String
```

On `Handle` és el tipus que usa Haskell per els arxius i els canals.

Operacions amb arxius o canals

Per usar aquestes funcions i definicions heu de fer un `import System.IO`.

Per exemple:

```
import System.IO
mgetLine :: IO String
mgetLine = do c <- hGetChar stdin
              if c == '\n'
              then return ""
              else do l <- mgetLine
                     return (c:l)
```

On `stdin` és el canal d'entrada estàndard.

Operacions amb arxius o canals

Ho podríem fer amb un fitxer i generalitzar la nostra funció de lectura

```
import System.IO

mgetLine :: Handle -> IO String
mgetLine ha = do c <- hGetChar ha
                  if c == '\n'
                  then return ""
                  else do l <- mgetLine ha
                           return (c:l)

main = do name <- getLine
          ha <- openFile name ReadMode
          s <- mgetLine ha
          putStrLn s
          hClose ha
```

Per a més informació sobre operacions mireu la documentació del `System.IO` de Haskell.

Continguts

- 1 Introducció a les Mònades
- 2 Class Functor
- 3 Class Monad
- 4 La Monad IO
- 5 Nombres aleatoris en Haskell

Nombres aleatoris

Cal usar la llibreria `System.Random` (segons la versió s'ha d'instal·lar).

Per a generar nombres aleatoris cal:

- generar una *llavor* inicial
- generar una seqüència de nombre a partir d'aquesta llavor (modificant la llavor després de generar cada nou nombre).

La llavor s'ha d'anar modificant.

- Podem treballar amb nombres aleatoris sense mònades, si generem la llavor i la passem i rebem explícitament en totes les funcions que l'usin.
- Podem treballar amb nombres aleatoris amb IO i la gestió de la llavor serà transparent.

Class RandomGen

Presentem primer la classe de tipus per generadors aleatoris:

```
class RandomGen g where

next :: g -> (Int, g)
-- a partir d'un generador ens dona un nou Int aleatori dins
-- del rang del generador i el següent generador

split :: g -> (g, g)
-- d'un generador n'obté dos

genRange :: g -> (Int, Int)
-- indica el rang de valors associat al generador.
```

StdGen

Considerem ara StdGen un tipus instància de la classe RandomGen.

```
data StdGen
    deriving (Read, Show, RandomGen)

mkStdGen :: Int -> StdGen
-- obté un generador a partir d'un enter.

getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
-- donada una funció generadora de nombres aleatoris retorna
-- un nou nombre aleatori dins de la mònada i manté el
-- generador de la mònada.

getStdGen :: IO StdGen
-- obte un generador.
```

StdGen

```
setStdGen :: StdGen -> IO ()  
-- estableix com a generador de la mònada el generador que  
-- rebem com a paràmetre  
  
newStdGen :: IO StdGen  
-- aplica split i es queda un coma nou generador de la nostra  
-- mònada i retorna una nova mònada amb l'altra generador.
```

StdGen

Finalment, mostrem la classe de tipus que tenen funcions generadores de nombres aleatoris.

```
class Random a where
```

```
randomR :: RandomGen g => (a, a) -> g -> (a, g)
-- (a,a) és l'interval de valors
-- g és un generador de nombres aleatoris
-- retorna el nombre aleatori i un nou generador
```

```
random :: RandomGen g => g -> (a, g)
-- el mateix però sense interval
```

```
randomRs :: RandomGen g => (a, a) -> g -> [a]
-- el mateix però torna una llista infinita de nombres
-- aleatoris
```

StdGen

```
randoms :: RandomGen g => g -> [a]
```

```
randomRIO :: (a, a) -> IO a
```

```
-- com el randomR però usant el generador global de la mònada
```

```
randomIO :: IO a
```

```
-- com el random però usant el generador global de la mònada
```

Són instància de la classe Random:

Bool, Char, Double, Float, Int, Integer

Random sense IO

```
genera :: RandomGen s => s -> Int -> Int -> Int -> ([Int],s)
genera s 0 _ _ = ([],s)
genera s n lo hi = (x:l,s2)
  where (x,s1) = randomR (lo,hi) s
        (l,s2) = genera s1 (n-1) lo hi

main = do
  std <- newStdGen
  let (l1,s1) = genera std 6 7 14
  print l1
  let (l2,s2) = genera s1 6 7 14
  print l2
```


Random amb IO

```
generaIO :: Int -> Int -> Int -> IO [Int]
generaIO 0 _ _ = return []
generaIO n lo hi = do
    x<-randomRIO (lo,hi)
    l<-generaIO (n-1) lo hi
    return (x:l)

main = do
    setStdGen (mkStdGen 0)
    l1 <- generaIO 6 7 14
    print l1
    l2 <- generaIO 6 7 14
    print l2
```

Random amb IO

```
generaIO :: Int -> Int -> Int -> IO [Int]
generaIO 0 _ _ = return []
generaIO n lo hi = do
    x<-randomRIO (lo,hi)
    l<-generaIO (n-1) lo hi
    return (x:l)

main = do
    l1 <- generaIO 6 7 14
    print l1
    l2 <- generaIO 6 7 14
    print l2
```