Laboratori de PL: Haskell. Sessió 4

1 Les Mònades

Les mònades s'utilitzen en els llenguatges funcionals per modelar característiques com ara: l'indeterminisme, les excepcions, la noció d'estat, la concurrència, l'entrada/sortida, etc.

La idea és encapsular un còmput d'un cert tipus `a` mitjançant un constructor. En Haskell es defineixen tres classes de mònades: `Functor`, `Monad` i `MonadPlus`.

Totes les mònades de Haskell es defineixen com a instàncies d'una o varies d'aquestes tres classes. Ara bé, les classes monàdiques no és poden derivar.

2 La classe Functor

La definició de la classe és la següent

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

Els (contructors de) tipus llista ([]), IO i Maybe són instàncies de la classe Functor

```
instance Functor Maybe where

    fmap g Nothing    =  Nothing
    fmap g (Just x)   =  Just (g x)


instance Functor IO where

   fmap g x           =  x >>= (return . g)


instance Functor [] where

    fmap = map
```

Exercici. Feu que els tipus Arbre i ArbreGen fets en la sessió 1 siguin instàncies de la classe Functor, definint previament la funció de mapping corresponent.

3 La classe Monad

La definició de la classe és la següent

```
class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

      -- Definition mínima completa inclou:
      --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

L'operador `(>>=)` representa la composició de còmputs i ens permet simular els efectes laterals present en els llenguatges imperatius. Mentre que `return x` representa un còmput que produeix `x` mateix. És a dir, és `x` vist com a còmput.

Els (contructors de) tipus llista ([]), IO i Maybe són instàncies de la classe Monad

```
instance Monad Maybe where
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing
    return           =  Just
    fail s           =  Nothing

instance Monad [] where
    m >>= k          = concat (map k m)
    return x         = [x]
    fail s           = []
```

Noteu que `concat` retorna la concatenació d'una llista de llistes. La mònada list s'utilitza per expressar l'indeterminisme (com a llista de possibles resutats).

El IO és més complicat i simplement veurem com s'usa més endavant.

És pot fer un `import Monad`, per tenir accés a moltes més operacions predefinides sobre instàncies de la classe `Monad`.

Exercici. Definiu el tipus Pila amb constructor per la pila buida i per empilar (i definiu la resta d'operacions estàndar) i feu que sigui una instància de la classe `Monad`, seguint l'esquema de les llistes.

3.1 La notació `do`

La notació `do`, ens dona una forma més natural des del punt de vista computacional d'usar la concatenació de còmputs. Aquesta notació ens introdueix el punt i coma (`;`) per concatenar i l'operador `<-` per guardar resultats intemitjos. Així tenim que

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

Si suposem que `p` és una variable. Si és una expressió la traducció és una mica més complicada.

A més, si usem línies separades i indentem, bé no cal posar el `;` entre els còmputs. Per exemple

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

Si no indentem bé s'han d'usar les claus { i } a més del `;` com ara a

```
main = do {l <- getLine;
print (factorial (read input))}
```

3.2 Les lleis de les Mònades

Les mònades és regeixen per tres regles, que s'hauria de comprovar que es compleixen. Tingueu en compte que el compilador no ho fa.

1. Identitat per l'esquerra: `return a >>= f` $\equiv$ `f a`
2. Identitat per la dreta: `m >>= return` $\equiv$ `m`
3. Associativitat: `(m >>= f) >>= g` $\equiv$ `m >>= (\x -> f x >>= g)`

4 La Mònada IO

En Haskell usem el constructor de tipus `IO` per gestional l'entrada/sortida. Com hem dit `IO` és una instància de la classe `Monad`, que usarem normalment amb notacio `do`.

Les operacions bàsiques del tipus són

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

Així `getChar` no té paràmetres i ens torna un element del tipus monàdic `IO` de `Char`. Mentre que `putChar` té un paràmetre `Char` i ens torna un element del tipus monàdic `IO` de "res" (anomenat en Haskell *unit* type, que és com el void dels llengutges de la família del C).

Com a exemple del seu ús, la implementació del getLine és la següent:

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

4.1 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. Per usar aquetes funcions i definicions heu de fer un `import IO`

Per exemple tenim que

```
import IO

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

On stdin és el canal d'entrada estàndar. També ho podriem fer amb un fitxer. De fet, podem generalitzar la nostra funció de lectura per a que treballi amb qualsevol `Handle`.

```
import IO

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.

5 Generador de nombres aleatoris

Per generar nombres aleatoris no cal una mònada directament si triem nosaltres la llavor (o el generador). Ara bé si volem que la llavor s'obtingui automàticament llavors usem una mònada `IO`. Cal fer

```
import System.Random
```

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.
```

Considerem ara `StdGen` un tipus instància de la classe `RandomGen`, que és el que usarem per generar els nombres. Per aquest tipus tenim una funció que ens dona un generador a partir d'un enter, i una serie d'operacions que retornen elements de la mònada IO, la llavor (el generador) és l'estat de la mònada que representa una llavor global.

```
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.

setStdGen :: StdGen -> IO ()
-- estableix com a generador de la mònada el genrador 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.
```

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 pero sense interval

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

randoms :: RandomGen g => g -> [a]
-- el mateix pero torna una llista infinita de nombres aleatoris

randomRs :: RandomGen g => (a, a) -> g -> [a]

randoms :: RandomGen g => (a, a) -> 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`:

```
Random Bool
Random Char
Random Double
Random Float
Random Int
Random Integer
```

Feu proves generant llistes de nombres aleatoris dins d'un rang donat a partir d'un enter llavor.

Apliqueu aquest generadors a la segona pràctica. Noteu que això obliga a afegir a algunes de les funcions un paràmetre de tipus StdGen i que el resultat sigui un parell amb un StdGen com a segona component.

Exercici Feu una comprovació de la qualitat dels nombres generats. Per això feu per exemple un experiment generant una llista prou gran de nombres aleatoris entre 1 i 6, i calculeu el nombre d'aparicions de cada un. Feu proves amb diferents mides de la llista (1000, 10000, 100000, ...).

Feu dues versions. Una que usi un enter com a llavor i un altre que usi la mònada IO amb la llavor global. El cas amb IO el podeu fer usant el `getStdGen` per generar la llavor i després procediu com l'altra cas, però es recomana fer-lo usant `getStdRandom` de manera que la llavor no es vegi, ja que forma part de la mònada IO.